GRACE TECHNICAL REPORTS

Proceedings of the First International Workshop on Formal Methods Education and Training

Jim Davies, Jeremy Gibbons, Mike Hinchey and Kenji Taguchi (editors)

GRACE-TR 2008–03 2008 October 28

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Foreword

Formal methods have an important role to play in the development of complex computing systems — a role acknowledged in industrial standards such as IEC 61508 and ISO/IEC 15408, and in the increasing use of precise modeling notations, semantic markup languages, and model-driven techniques.

There is a growing need for software engineers who can work effectively with simple, mathematical abstractions, and with practical notions of inference and proof. However, there is little clear guidance — for educators, for managers, or for the engineers themselves — as to what might comprise a basic education in formal methods. The present IEEE/ACM Software Engineering Body of Knowledge (SWEBOK), in particular, lacks the kind of specific information that teachers and practitioners need to establish an adequate, balanced programme of learning in formal methods.

The workshop on *Formal Methods Education and Training* provided a forum for the discussion of the key issues in formal methods education, with a particular focus upon the development and advocacy of a Formal Methods Body of Knowledge (FMBOK), analogous to the Institute of Project Management's PMBOK. This BOK would facilitate the design of appropriate programmes of education and training — undergraduate, graduate, and professional — for modern software engineers, as well as promoting the sharing of teaching approaches, educational tools, and teaching materials.

Contributions were invited on this theme, and on a number of related aspects of education and training in formal methods: teaching experience, both academically and industrially; curriculum issues, and the relationship with computer science and software engineering; teaching methodologies; and the use of tools.

The extended abstracts that were submitted were reviewed for relevance. A mixture of these extended abstracts and longer draft papers was presented at the workshop on 28th October 2008, co-located with the *International Conference on Formal Engineering Methods* at the Kitakyushu International Conference Center, Kitakyushu, Japan. This National Institute of Informatics technical report represents the workshop proceedings. After the workshop, full papers will be solicited for submission and rigorously peer-reviewed for a special journal issue to be published by ACM.

The Programme Chairs would like to thank the authors for their contributions to the workshop, the Programme Committee for their efforts in reviewing, and the organizers of ICFEM for handling all the logistical arrangements.

Jim Davies

Jeremy Gibbons
Mike Hinchey
Kenji Taguchi

October 2008

Organizers

Programme Chairs

Jim Davies
University of Oxford, UK
Jeremy Gibbons
University of Oxford, UK
Mike Hinchey
Lero, University of Limerick, Ireland
Kenji Taguchi
National Institute of Informatics, Japan

Programme Committee Members

Toshiaki Aoki
Japan Advanced Institute of Science and Technology, Japan
Cyrille Artho
Advanced Industrial Science and Technology (AIST)/RCIS, Japan
Raymond Boute
Ghent University, Belgium
David Duce
Oxford Brookes University, UK
Lars-Henrik Eriksson
Uppsala University, Sweden
Chris George
UNU/IIST, China
J. Paul Gibson
Telecom SudParis, France
Shaoying Liu
Hosei University, Japan
Hideaki Nishihara
Advanced Industrial Science and Technology (AIST)/CVS, Japan
Wolfgang Reif
University of Augsburg, Germany
Steve Schneider
University of Surrey, UK
Jing Sun
University of Auckland, New Zealand

Table of Contents

FORMAL METHODS versus ENGINEERING 1
Tom Maibaum

Teaching Formal Methods in the Context of Software Engineering 2
Shaoying Liu, Kazuhiro Takahashi, Toshinori Hayashi, Toshihiro Nakayama

Formal Methods: Teaching and Practicing Computer Science at the University Level 11
Raymond Boute

An Introductory Course on Programming based on Formal Specification and Program Calculation 26
Nazareno Aguirre, Javier Blanco, María Marta Novaira, Sonia Permigiani, Gastón Scilingo

Evolution of a Course on Model Checking for Practical Applications 33
Yasuyuki Tahara, Nobukazu Yoshioka, Kenji Taguchi, Toshiaki Aoki, Shinichi Honiden

Model Checking Education for Software Engineers in Japan 49
Hideaki Nishihara, Koichi Shinozaki, Koji Hayamizu, Toshiaki Aoki, Kenji Taguchi, Fumihiro Kumeno

A simple refinement-based method for constructing algorithms 59
Dominique Méry

Formal Methods for Electronic Government 74
Jim Davies and Jeremy Gibbons

FORMAL METHODS versus ENGINEERING

Tom Maibaum

Department of Computing and Software
McMaster University
1280 Main Street West, Hamilton ON, Canada
tom@maibaum.org

Abstract. Classical engineering is based on solid scientific and mathematical foundations, but neither the science, nor the mathematics, is simply borrowed from the scientists or the mathematicians. Engineers develop their own formulations of the relevant science and mathematics, adapted to support the engineering knowledge used in design of artifacts. There are many formulations of the same science and mathematics, as classical engineering is highly domain specific. It may well be argued that it is this high degree of domain specificity that makes engineering effective. This scientific and mathematical knowledge is organised by engineers into engineering methods, sometimes encoded in ”cookbooks”. These methods make engineering a normative discipline: follow the rules and you will have some guarantees based on past success and failure. Formal Methods, so called, are not really methods at all, as engineers understand the term. Firstly, there is generally very little of ”engineering maths” involved. Most computer science or software engineering courses with mathematical foci are theoretical courses in the sense that they focus on mathematics and not engineering. ”Mathematicians prove, engineers calculate.” Secondly, there is very little method in Formal Methods. There is no serious attempt at defining the methods that are the organisational principle of all engineering. Thirdly, there is very little evidence of domain specificity in software engineering. Until these shortcomings are addressed, there is very little point in talking about Formal Methods education. Unfortunately, there is not all that much engineering material to communicate to students.

Teaching Formal Methods in the Context of Software Engineering

Shaoying Liu[1], Kazuhiro Takahashi[2], Toshinori Hayashi[2], Toshihiro Nakayama[1,2]

[1] Hosei University
[2] The Nippon Signal Co., Ltd., Japan

Abstract. Formal methods were developed to provide systematic and rigorous techniques for software development, and they must be taught in the context of software engineering. In this paper, we discuss the importance of such a teaching paradigm and describe several specific techniques for teaching formal methods. These techniques have been tested over the last fifteen years in our formal methods education programs for undergraduate and graduate students at universities as well as practitioners at companies. Our experience shows that students can gain confidence in formal methods only when they learn their clear benefits in the context of software engineering.

1 Introduction

Despite more than forty years of effort to develop various theories, languages, methods, and tool supports, practical software engineering is still like a "desert", lacking directions and effective ways of finding the way out of the software crisis. Formal methods were developed to address this problem by providing mathematically-based techniques, including formal specification, refinement, and verification. In theory, we now know how to use formal notations to write specifications, use refinement calculus to gradually transform a specification into a correct implementation, and use Hoare or Dijkstra's logics to prove programs correct with the same degree of the rigor that we apply to mathematical theorems. However, none of these techniques is easy to use by ordinary practitioners to deal with real software projects. The problem is the complexity of formal methods and the difficulty in manipulating mathematical formulas.

Having said the above challenges in directly applying formal methods, we do not mean that formal methods are useless. In fact, they are more necessary than ever when more and more software systems are embedded into systems deployed in many places of our society, but their role is different from other software techniques. The role of formal methods is *education*, and their power can be transferred to software engineering projects through the developers who have learned and mastered them. The way to use formal methods in practice is formal engineering methods [1], not formal methods. For example, the SOFL formal engineering method provides a three-step approach to constructing formal specifications to help requirements analysis and system design, and specification-based review and testing for detecting bugs in both specifications and programs [2]. Software projects are human activities; they must be completed by required time and within specified budget, and they often face the instability of development teams. In such a situation, completely applying formal methods is rarely practical, but the improvement of software quality can be realized by equipping the developers with a disciplined manner and rigorous way of thinking through formal methods education.

In order to encourage more software developers to learn formal methods, we must first build up their motivation by demonstrating the clear benefits of formal methods in improving current software engineering practice. In fact, this is rather challenging and even more difficult when more and more young people become less interested in mathematics, especially in Japan. Nevertheless, this seems the only way we can possibly move forward in formal methods education. In this paper, we describe several techniques for teaching students formal methods. The fundamental idea is to put the formal methods education in the context of software engineering as far as we are concerned. Of course, as Parnas pointed out [3], formal methods should not be restricted to software engineering, but linked to and integrated in general engineering mathematics.

2 Teaching Techniques

In this section, we introduce some specific techniques for teaching formal methods. These techniques have been tested by the first author over the last fifteen years of teaching VDM [4], SOFL [1], and Morgan's refinement calculus [5] at universities and companies.

2.1 Starting with Examples

Learning formal methods is similar to learning other theories or techniques, students like to start with simple examples. These examples must come from the daily life and must be able to link the problem in practice to a potential formal methods solution. This way of teaching will motivate students and build up their interests in formal methods. For example, when explaining the ambiguity problem in informal specifications and the fact that it can be resolved by formalization, we often use an operation for searching for an integer in an integer list as an example. After explaining the impreciseness of the informal requirement statements, we present a formal specification which is both precise and concise. This example helps students understand the potential power of formalization.

2.2 Gradual Introduction to Important Concepts

The fundamental concepts are the key to understand the spirit of formal methods. It is quite effective to help students understand the essential principle of formal methods if sufficient efforts are made to teach the concepts. For example, when introducing formal specifications, we focus on the illustration of pre- and post-conditions. An effective way to teach the pre-post concept is by comparing them with the corresponding algorithm and let students understand the real difference and relation between a specification and an algorithm. The comparison can be made on the basis of simple scientific computation. For example, we often use the operation for yielding the square root of an integer as an example. The pre-condition of the operation is $x \geq 0$ and the post-condition of the operation can be $y^2 = x$, where x is input and y is output. But the corresponding algorithm would be something like $y = Math.sqrt(x)$. This example gives rise to a problem that output y produced by the algorithm may not satisfies the post-condition of the operation because the algorithm obtains only an approximation of the real square root of some positive integers. In this circumstance, it is useful to tell the students the importance of noticing this inconsistency between the specification and the implementation. This is also a good example to show the need for using or building proper theories in the application domain.

Furthermore, an operation specified using pre- and post-conditions defines a way of transforming an initial state to a final state. In order to let students understand this essential idea, making great efforts on the explanation of the fundamental concepts, such as state, type, and variable declaration, is helpful. The explanation can be given from different angles, for example, from the views of both mathematics and software. After students understand the basic concepts, we can then show them, with simple examples, how an algorithm transforms an initial state to a final state step by step through its statements, and how such a transformation can be abstracted into a pre- and post-conditions. This way of teaching helps students build an association between programs and specifications, which paves the way for teaching specification-based verification techniques, such as formal verification, review, testing, or their combinations late on.

2.3 Massive Exercises on Basics

Efficiently writing accurate formal specifications requires the developer to have a good understanding of features of various data types and high skills in applying the well-defined operators on the data types, such as boolean, set, sequence, and map types. Therefore, massive exercises on the basic operators must be done by students. The most effective way to incorporate exercises into the teaching program is to let students do exercises immediately after a data type is introduced. For example, after the introduction of the set types, students must learn the meaning of the operators, such as union, intersection, cardinality, membership, subset, proper subset, and so on by applying them to specific set values. If time allows, a public discussion on students' results is helpful. According to our experience, such a discussion can help capable students find out the reason for their mistakes and ordinary students find out the correct way of thinking. This training is similar to the basic training in sports. To be an excellent football player, for example, one must run fast and have a strong body. To build up these qualities, he or she must spend much time and make great efforts in the basic training. Anybody who ignores the basic training will fail to perform satisfactorily in matches.

The same principle is applicable to the teaching and study of formal refinement and verification techniques. To master the refinement calculus, students must be required to do many small exercises on applying every refinement law in the calculus. To be skillful for formal verification, students must be required to do the same in understanding the meaning of each axiom and inference rule in the corresponding logics and their applications to small programs. The important point here is to let them understand the underlying principle and skills of these techniques so that they will be possibly apply them, even informally, in practice.

2.4 Teaching Specification Patterns for Abstraction Skills

Effectively using a formal method requires the developer to have high skills and ability in mathematical abstraction, especially in the context of software development. How to help students strengthen their abstraction skills and ability therefore becomes an important issue in formal methods education. While this has been recognized widely as the most difficult thing in teaching, we have gained sufficient knowledge and understanding through our long time teaching experience. Considering the fact that the basic operations required in a software system usually include searching, sorting, merging of two collections of objects, adding some elements to a collection of objects, eliminating some elements from an existing collection of objects, updating some elements from an existing collection of objects, mathematical computation, and their combinations, we put the emphasis on the teaching of how to express all of the above functions using appropriate data types and their related operators. Each of such expressions will form a specification pattern that will remain in students mind and available for application in real software development. For example, what are possible specification patterns for a function which tests that a collection of integers is empty? To answer this question, we first define a collection of integers as a *set* and a *sequence* in SOFL (or VDM), respectively, such as int_set: $set\ of\ int$ and int_seq: $seq\ of\ int$. We then discuss the most commonly used specification patterns for each of the data abstractions. For example, for the set of integers, we can use the following patterns to express the fact that the set is empty: $int\_set = \{\}$ and $card(int\_set) = 0$. Of course, we could have more patterns to express the same meaning, but those would be much more complex and no good for readability. For instance, a possible pattern can be: $forall[x$: $int] \mid x\ notin\ int\_set$. It is up to the teacher to decide whether to discuss such a complicated pattern within the required teaching time. In the case of a sequence of integers, we can use the following patterns to express the fact that the sequence is empty: $int\_set = []$ and $len(int\_set) = 0$.

After each basic specification pattern is mastered by students, we can then go further to explain how such basic patterns can be applied in a more complicated situation. Let us take an operation to search for an integer in a collection of integers as an example. To explain how such an operation is specified, we take the same approach as the one to teaching the basic patterns by first defining the collection of integers as *a set of integers* and *a sequence of integers*, respectively, and then explaining how the operation can be specified by combining the basic patterns for each of the data abstractions.

2.5 Practice through Small Projects

While the basic training is important in teaching and studying formal methods, we should never forget to give students opportunities for linking formal methods to software engineering. In other words, they need to be taught how formal methods will possibly help them in software development practice; otherwise, students (perhaps with some exceptions) will likely to lose the motivation of learning or applying formal methods in practice. The most effective way for this is to let students conduct small projects. For example, after the introduction of VDM-SL and massive exercises on the basics, we can ask students to do one or two small projects. One project can be the construction of a formal specification for a small library system, and another possibility is to let students complete a formal specification for an ATM software. Through such small projects, students can really feel how formal specifications can be built and organized in real software development projects. Of course, such a practice may also give students an opportunity to find the weakness of the specification language they are using. For example, lacking an intuitive mechanism for structuring a whole system in a structured manner in VDM could be found by students. The answer to this problem is to introduce the SOFL specification language to them, since SOFL has solved this problem by using intuitive and formalized data flow diagrams and process decompositions. In fact, many existing formal notations focus only on one aspect of the problem in software engineering and ignore the others, but a real software project needs to take care of all possible aspects. If a method or technique merely helps solve one problem but create more other problems in the context of software engineering, it is unlikely to be popular among practitioners and to be applied in real projects. In this regard, the SOFL method has shown to be the exception, because it provides a systematic and rigorous process to integrating formal techniques into existing software engineering practices and creates no more problems.

2.6 Teaching Formal Methods Using Formal Engineering Methods

The ultimate goal of teaching formal methods (FM) is to create possibility of students applying them in practice. Formal engineering methods (FEM) show *how* FM can be applied in real projects. One of the very important aspects of FEM is the emphasis of combining diagrams, formal notation, and natural language in a coherent and systematic manner for writing specifications [1]. The purpose of this is to help the developer easily understand the specifications they are writing and the specifications written by others. Visualization is intuitive and suitable for describing the overall idea and system architecture; formal notation has a strength to achieve preciseness of statements in specifications; and natural language can be used to provide a friendly interpretation of formal expressions. In general, FEM differs from FM in that FM tries to answer the question "what should we do and why?" in software development, but FEM tries to answer the question "what can we do and how?". To this end, FEM focuses on techniques and methods for integrating formal methods into the entire process of software development so that the strength of formal methods can be utilized in practice and their weakness of being complex can be avoided. FEM offers how software systems, including all level documents, are actually created and expressed formally, not just a simple mixture of formal notations with pictures. Since a detailed introduction to FEM is beyond the scope of this paper, we refer the reader to the SOFL book [1] for a comprehensive description of FEM.

In fact, the same principle of FEM can also be effectively applied to the teaching of formal methods courses, since teaching is actually a kind of software project whose product is educated students. For example, when explaining a mathematical expression, such as $Z = X$ **union** Y, we can use a graphical representation (e.g., Venn diagrams) to illustrate the union operation, and at the same time use English, for instance, to explain the meaning of the operation. When introducing an operation in VDM, we can draw a process as we do in the SOFL language to show the input, output, and external variables, but the details of the function of the operation are defined using pre- and post-conditions. With informal explanations in English, the meaning of the whole operation specification can be easily digested by students.

2.7 Tool Support in Education

Almost all of us may have experienced using tools in teaching programming languages, such as Java and C, and found that it is effective to help students write, execute, and test programs (they need many pre-defined packages). Many of formal methods educators apply this idea to the teaching of formal methods courses as well. However, our experience in teaching both VDM and SOFL courses, which focus on formal specification techniques, suggest that using tools in teaching formal methods is not necessarily effective; perhaps less effective than not using tools in some circumstances. There are two reasons. One is that learning formal methods requires students to learn both syntax and semantics of the related specification language. The most effective way for students to remember them is to write formal specifications by hand, as they learn English as a foreign language. It is feasible, because exercises assigned to students in classes are of small scale. It is also effective in strengthening students' memory of the syntax and in deepening their understanding of the abstraction techniques, because students would have no chance to "copy and paste" without thinking by themselves, as we often do on a computer. Another reason is that the purpose of writing a specification is not for a computer to directly run it, but for people (including himself or herself) to read and to understand. Therefore, letting themselves write a good style of formal specifications by hand is much more helpful for learning than by using a tool to automatically improve the style and format of their specifications. In the case of programming, without a tool, such as a compiler, we cannot run the program. But in the case of writing a specification, there is no need to run it, so without a tool support will not create any significant inconvenience. Instead, for some students who do not want to study formal methods, tool support will create chances for them to "copy and paste" without thinking.

Having said the above, it does not mean that tool support is not necessary for using formal methods in practice. On the contrary, tool support is crucial for improving productivity and reducing chances of creating mistakes in practical developments. For this reason, we let students use a supporting tool, such as IFAD VDMTools [6] or SOFL GUI editor, when they carry out a small project, after a systematic learning of formal specification techniques in classes. This way also has an effect that students feel extremely happy with the tool offering high automation in both writing and analyzing specifications. They have this kind of feeling because they have gone through a hard time in learning formal methods by hand. This is similar to the situation where a person feels happy when he or she has a chance to eat delicious food after a long time starving.

2.8 Dealing with Time Constraint

Mathematical concepts and expressions usually require students to take time to digest, the teaching of them should take slow pace with many examples. However, a course is like a software project: it also has time constraint. As a teacher, we often face a dilemma. On the one hand, we want to teach more contents which are all important for studying formal methods, but on the other hand, we do not have enough time. To tackle this problem, our experience suggests that each course should not be too ambitious; instead, it should be focused. For example, we can teach formal specification, refinement, and formal verification in three different courses, and it would be effective to focus the teaching in each of them on the most fundamental but important parts and give students sufficient time for them to apply the learned techniques. For example, when teaching SOFL, particularly techniques for writing formal specifications using pre- and post-conditions, to students, we usually take the interleaving approach: teaching concepts and asking students to practice using them. After finishing the whole course, we ask students to carry out a small project in which all knowledge learned is required to use. Such a way provides students with many opportunities to learn how theoretical results can be effectively applied in practice.

2.9 Continuing Education

Formal methods education is necessary, but it does not necessarily mean that it is popular among students. According to our experience in teaching VDM to the employees of the Nippon Signal Co., Ltd. and SOFL to university students, people with certain working experience usually find formal methods, particularly formal specification techniques, easy to learn and use, but this may not be true for students without working experience. The important reasons include that the students usually do not deeply understand the importance of the role of formal methods in software quality assurance and the contents of formal methods are quite complex and detailed. Since formal methods education is necessary, a possible solution to this problem is to arrange formal methods as compulsory rather than optional courses. Thus, every student will be forced to learn formal methods. In addition to this assurance, by applying effective teaching methods such as those mentioned above and appropriate requirements for different level students, it would be highly possible to let more and more students learn formal methods. However, even if this possibility becomes reality at present or in future, it will not guarantee that formal methods will become attractive to students. Our experience has suggested that to be attractive, formal methods must achieve a good balance among the three qualities: simplicity, visualization, and preciseness, and must also demonstrate its benefits in ensuring software quality and reducing the cost of software projects as well as providing fun for students, such as computer graphics or animation, but unfortunately, few of existing formal methods have satisfied these criteria and it is hard to imagine that any teaching method would significantly improve this situation. Since software development needs mathematical way of thinking, we believe that no matter whether formal methods are attractive or not, education in formal methods must continue at university and hopefully in industry as well. Only formal methods education can make the application of formal methods, either directly or indirectly, in software engineering possible.

3 Conclusions

Education is the necessary and most effective way to transfer formal methods to software industry. The most important influence factor for the success of formal methods education is whether the education is put in the context of software engineering. In this paper, we have described several techniques for teaching formal methods in the context of software engineering to both experienced and inexperienced students, each of which has been tested in practice. We believe that no matter whether formal methods can be used as an effective software engineering technique in practice, their education will definitely benefit software engineering practice. The only way to effectively transfer formal methods to industry is: education, education, and education.

References

1. S. Liu, *Formal Engineering for Industrial Software Development Using the SOFL Method*, Springer-Verlag, ISBN 3-540-20602-7, 2004.
2. S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba, *SOFL: A Formal Engineering Methodology for Industrial Applications*, IEEE Transactions on Software Engineering, 24(1):337–344, January 1998. Special Issue on Formal Methods.
3. D. L. Parnas, *Education for Computing Professionals*, Computer, 23(1):17-22, 1990.
4. C. B. Jones, *Systematic Software Development Using VDM*, 2nd edition, Prentice Hall, 1990.
5. C. Morgan, *Programming from Specifications*, 2nd edition, Prentice-Hall, 1994.
6. The VDM-SL Tool Group, *Users Manual for the IFAD VDM-SL tools*, The Institute of Applied Computer Science, February 1994.

Formal Methods: Teaching and Practicing Computer Science at the University Level

Raymond Boute

INTEC — Ghent University, Belgium,
boute@intec.UGent.be http://www.funmath.be

Abstract. At too many universities, CS curricula are not taught at the university level. This causes stagnation in professional practices. The missing element is the pervasive presence of mathematical modeling throughout the curriculum. This is the role of formal methods (FM) in its original sense. Mathematical fundamentals and concepts are crucial, software tools are secondary and even misleading without the former.

Social, professional, educational and local influencing factors are discussed. Recommendations are given for for curriculum structure, for specific key cources and for attitudes to instill in students and educators. As a conclusion, FM should break outside the limitations caused by the conservatism of policy makers but also the self-imposed ones.

1 Observations

Conventions Drawing boundaries between Computer Science and Engineering is more in the interest of departmental power games than epistemologically useful, so we take CS in a wide sense, including (even adopting) the engineering view.

Parnas [21] observes that “Professional engineers can often be distinguished from other designers by the engineers’ ability to use mathematical models to describe and analyze their products”. The central role of mathematics is a fortiori evident to those viewing CS as a more theroretical activity than engineering.

Hence the use of mathematical modeling is a proper working criterion for what constitutes “university level” in teaching as well as in professional practice.

For completeness, some earlier material [5] will be recalled in passing.

1.1 The level of Computer Science curricula

In classical pure science and engineering disciplines, the pervasive use of mathematics throughout the curriculum has become self-evident since centuries.

Also, quite a large number of universities do teach Computer Science at the university level, and hence give no cause for concern.

Still, too many universities teach CS at the level of pre-Newtonian mechanics. When colleagues who teach classical engineering courses (say, electronics) look at the computing courses in such curricula, they often remark that these courses are merely descriptive, lacking in intellectual content, and altogether little more than inflated programming courses—an opinion reinforced by the kind of assignments: writing programs, more programs and reports, and doing uninstructive projects [20]. Unfortunately, this remark is all too often justified.

Indeed, many CS curricula seem to be designed as a refuge for mathphobic students just to increase the student count. The study by Tucker et al. [28] uses American data, yet reflects European trends equally well (see section 3.3). Instead of devoting the scarce teaching resources to solid fundamentals [20], they are wasted on trendy topics that students could easily pick up on their own. Courses like "Introductory Programming" usually teach some language (which is better learned in passing via the assignments) rather than program design.

Formal Methods, if taught at all, have a small place in the curriculum, looking like an afterthought. Students see this as the best "proof" of their uselesness: if FM were af any value, wouldn't all professors use them? Very few students are mature enough to see that curricula reflect the faculty [12] rather than relevance.

1.2 The level of professional practice

It is eery to see how little software practice has changed over the past 50 years.

Some younger software professionals suggest that things were better in the "old times" [24]. For instance, Spolsky also notes that "programmers seem to have stopped reading books" [25]. However, a 1975 paper [19] already commented on the computer illiteracy of professional programmers!

Google yields ample comments on the phrase "Programmers don't read", indicating that many programmers find the books worthless anyway. However, the titles quoted in the rejection are on programming languages, not program analysis or design. This just reflects the old misconception that the main capability of a programmer is the language.

Lethbridge surveyed "what knowledge is important to a software professional" [17]. The use of the preposition "to" rather than "for", also in the cited paper's abstract, is significant: it indicates that the survey is about subjective views. Circumstantial evidence leaves little doubt that many topics score low with the "average programmer" because they are not sufficiently known or mastered to be applied, rather than because they are truly less relevant or useful.

Hence, whereas Lethbridge advocates that "efforts to develop licensing requirements, curricula, or training programs for software professionals should consider the experience of the practitioners who actually perform the work", it may be wise to use this information to detect the gaps, and educating professionals in what they need for advancing their professionalism rather than what they want, which is just more of the same in what they already know.

An important question is: how could the extreme conservatism that has been dominating software practices over the past 50 years survive, whereas practice in other engineering disciplines has kept abreast with technology? The answer is manifold, but the common element is complacency.

– Software thrives on the spectacular advances in hardware technology. Increased circuit speed and density often compensate for the stagnation in software design practices.
– Every few years some "method" is promoted that relies on acronyms rather than intellectual content, yet promises panacea with little effort. By contrast: formal methods promise considerably less and require more effort. In other words, adapting Gresham's law somewhat, "continuous injection of bad commodities keeps the good ones out of circulation".
– In the economic situation of the recent decades, computer and information technology have seen continued growth with only rare periods of faltering. Due to this fast pace, there always are plenty of urgent development tasks to be done at all levels. Even people with no computing background who invest a minor effort in self-study can contribute usefully, and many well-paid tasks are easily learned "on the job". With the urgent demand for quantity, and industry still uncertain about the exact qualifications software designers need, level and scientific basis are easily neglected.

The main problem is that many universities cater for the latter category, and actually advertise this attitude as "being responsive to the demands of industry". Demands maybe, needs certainly not. Students do deserve better.

Students are also the key. Indeed, Max Planck is reputed to have said

> Eine neue wissenschaftliche Wahrheit pflegt sich nicht in der Weise durchzusetzen, daß ihre Gegner überzeugt werden und sich als belehrt erklären, sondern vielmehr dadurch, daß ihre Gegner allmählich aussterben und daßß die heranwachsende Generation von vornherein mit der Wahrheit vertraut geworden ist.

Translated: "A new scientific truth usually does not break through in such a manner that its opponents are convinced and declare themselves informed, but rather because its opponents gradually die out, and the emerging generation has become familiar with it beforehand." The task of the universities is clear.

The Formal Methods community should also heed Planck's advice. All too often, the failure of FM is attributed to insufficient arguments for convincing practitioners. This has lead to considerable effort in producing successes in industrial applications that are impressive by themselves, but whose effects essentially remain limited to the industries who benefit and the participanting researchers.

The effective lever for FM is not convincing the practitioners but educating the emerging generations.

2 Realizing the full potential of Formal Methods

2.1 Formal Methods

As observed by Gopalakrishnan [10], the term “formal methods” seems to suggest the sudden discovery by computer scientists of the use of mathematics—so evident in other branches of engineering that a separate appellation is redundant.

Still, the specific term could be justified by the fact that CS/ECE requires a more formal kind of mathematics than the classical mathematical/engineering disciplines, a point also emphasized in Lamport’s book on specifying systems [14].

Here “formal” means that expressions are manipulated on the basis of their form (syntax) following precise calculation rules. This contrasts with traditional practice in math/engineering, where expressions are often manipulated on the basis of their interpretation (semantics) in some application domain. The benrefit is in letting the symbols do the work, which is especially useful in areas where intuition for the application domain is still nascent. It also develops a “parallel intuition” for handling symbols that supports domain-oriented intuition [7].

Understood in this wider (actually, more original) sense, Formal Methods point the way not only to teaching CS at the university level, but also provides a refreshing new style to mathematics [9]. The realization has been demonstrated for discrete mathematics [11] and essentially all of engineering mathematics [4, 6].

Hence the non-CS engineering mathematics can equally benefit, and the entire curriculum considerably streamlined by unification.

2.2 The role of tools

To realize the aforementioned potential of FM, the use of software tools has to be led into the proper channels.

For the sake of completeness, we first mention that some tool developers consider that tools are essential to FM or even “the only important thing” [23]. Taken literally, this statement is self-refuting by reductio ad absurdum. Still, it reflects the acceptance by industry of formal methods for solving specific kinds of problems economically. Epistemologically, however, it is far off the mark.

In mathematics, classical engineering and CS/ECE, software tools are meant to alleviate the burden of handling tedious details in calculations and proofs, and to reduce mistakes or avoiding pitfalls. Yet, this works only for users with ample mathematical maturity: for novices it can only lead to sorcerer's apprentice attitudes and even ruin [13]. Indeed, current tools are far from mature:

– No single tool has sufficient scope; even small systems require multiple tools;
– Implementation restrictions cause a very narrow view on mathematics;
– The many irrelevant syntax details and even semantic errors confuse novices.

For instance, tools like Maple and Mathematica seem designed fairly well for calculus and algebra, but in nearly every use for discrete mathematics the author found calculations going astray in unexpected and educationally hazardous ways. Here is an example, arising from number representation. Consider the functions

– `digits` such that, for any natural number n, the base 10 representation is `digits(`n`)`, defined as a function such that `digits(`n`)(`i`)` is the i-th digit; in Maple: `digits := n -> i -> floor (n/10^i) mod 10;`
– `decnum` such that `decnum(`f`)(`k`)` is the number represented by the k lowest digits in representation (function) f;
in Maple: `decnum := f -> k -> sum(f(i)*10^i, i = 0..k-1);`.

One expects `decnum(digits(`n`))(`k`)` $= n$ if k digits suffice to represent n. Yet `decnum(digits(210))(3);` yields 620 while `decnum(i -> i)(3);` yields 210. The idea that "tools help students discover their errors" thus gets a new meaning!

In mathematics education, the trap of thinking that tool use can obviate solid mathematical reasoning abilities seems to have been largely avoided. Some calculus textbooks [27] even wisely contain examples and assignments involving tool use especially to foster awareness of the pitfalls. Likewise, insofar as Ralston [22] advocates tools over pencil-and-paper simulation of computational algorithms (e.g., "long division") in elementary school, he does not do so at the cost of reduced awareness for numbers and mathematics. To the contrary: he proposes more emphasis on head calculation as an antidote to rote and the errors typical for using calculators without numeric awareness.

Electronics engineers routinely use tools like Maple, Matlab etc. by relying on a good mathematical background which, more than the tools themselves, explains the successful results. Tool vendors do not boast or encourage mathphobia; to the contrary: an announcement for a textbook on Simulink [8] states:

"students should have the appropriate mathematical preparation [such as] calculus and differential equations".

Unfortunately, in the FM area tools are often advertised as "hiding the math" for professionals (thus depriving them of the most powerful intellectual tool) and depicted by some lecturers as an aid to learn math for students (instead of mathematics preparing for tools). Tools are popular because they cater for the affinity of some students for video games, while seemingly "realistic" tool-based projects give them the illusion of becoming "real engineers" quickly and easily: the "sics munce ago i coodnt evun spel enguneer, now i are won" syndrome.

Proper FM courses are essentially mathematical, do not teach tools, but contain assignments where students learn the use of tools and their defects.

3 Curriculum and course design

3.1 The mathematics content of engineering curricula

Learning by analogy from classical engineering disciplines and also considering the huge body of mathematical knowledge generated by CS in recent decades [10], we propose the mathematical content for a computing engineering curriculum.

Table 1. Fundamental engineering mathematics at various levels

(level)	EM. Engineering Math.	CM. Computer Engineering Mathematics
Basic (general)	Analysis, Linear algebra Probability and Statistics Discrete mathematics (combinatorics, graphs)	Formal proposition and predicate calculus Function(al)s, relations, orderings Lambda calculus (basics) Lattice theory, induction principles, ...
Targeted (modeling)	Physics, Circuit theory, Control theory Stochastic processes Information Theory, ...	Formal languages and automata Formal language semantics Concurrency (parallel, mobile calculi etc.) Type theory, ...
"Advanced"	Functional analysis Distribution Theory Hilbert and Banach spaces Measure theory, ...	Category theory Unified algebra Modal logic Co-algebras and co-induction, ...

Table 1 draws an epistemological parallel in terms of topics and levels, not course names. Here "basic" and "targeted" are proper for undergraduates; "basic" is domain-independent, "targeted" is more domain-oriented. We put "advanced" in quotes because it is debatable: given today's state of the art, some topics are arguably valuable for undergraduates as well.

EM is mathematics for classical engineering, e.g., electrical (EE). CM is the mathematics for Computing Engineering (CE). In view of curriculum design, the columns are meant to complement, not replace each other.

Indeed, considering just the first two rows, the EM topics are generally (and rightly) considered crucial in the formation of every engineer, and there is no reason to start making exceptions for CE's, to the contrary [20]: CE's can learn from the rich variety of examples and styles in mathematical modeling.

Conversely, the new mathematical style that emerged from Computing Science [9] is relevant to all exact sciences, as amply demonstrated for discrete mathematics [11] and for engineering mathematics in general [6]. For instance, [6] shows that it is useful for electrical and computer engineers (ECEs) to be as fluent in calculating with quantifiers ($\forall$, $\exists$) as with derivatives and integrals.

3.2 A Bachelor program in Computer Engineering

A program following the principles set out thus far is outlined in table 2. Note that it is a concept program, intended as an archetype for curriculum design by tailoring it to the local goals and possibilities. It can also be extended by either a 4th Bachelor year or a 2-year Master program.

Whereas most of the topic titles are familiar, the added value is in the unified mathematical modeling throughout all courses. In other words, the use of Formal Methods is ubiquitous, obviating a separate course named "Formal Methods".

The basis is a first-semester course in logic in a form that is useful in the spirit of Gries [12], not only for CS but for all of engineering mathematics, starting with analysis. To reflect this, we called it Application-Oriented Formal Logic. The style is calculational. We briefly outline one actual realization. Apart from the usual proposition calculus (e.g., [11]), the two main elements provided are

a. Generic functionals [4]: using higher order functions supporting systems modeling and mathematical reasoning in a point-free style, and smooth conversion between point-free and the more common pointwise style.
b. Functional predicate calculus [6], supporting the aforementioned fluency in calculating with quantifiers in the context of applications.

This unifying framework forms the bridge between the continuous world of classical engineering and the discrete world of computing. By providing a reference frame, it also facilitates introducing greater diversity in the formalisms

Table 2. A concept BCE Program (3 years, extensible to 4)

	First semester
6	Application-Oriented Formal Logic
6	Mathematical Analysis A
3	Algebra
3	Geometry
6	Physics A (particle & wave mechan.)
6	Introductory Programming

	Second semester
6	Physics B (electricity & magnetism)
6	Mathematical Analysis B
3	Languages and Automata
3	Disctete Math (combinatorics, graphs)
6	Classical Mechanics
6	Algorithms and Data Structures

	Third semester
6	Probability and Statistics
6	Complexity
6	Signals and Systems
3	Elements of Quantum Mechanics
3	Basic Electronics
6	Computer Architecture

	Fourth semester
6	Programming Languages
6	Thermodynamics, heat & mass transfer
6	Database & Information Systems
3	Elements of Quantum Computing
3	Electrical Networks
6	Operating Systems

	Fifth semester
6	Chemistry
3	Digital Systems
3	Information Theory
6	Concurrency
6	Software Engineering A
6	Embedded Systems

	Sixth semester
3	Properties of Materials
3	Elements of Relativity
6	Communications Systems
6	Communication Networks & Protocols
6	Software Engineering B
6	Hybrid Systems

(Legend: the numbers are a measure for the size of the course — see text)

and tools in the various other courses. No valuable time is wasted on "teaching" tool use or language features: these are picked up via the assignments. The courses themselves are fundamental and the appropriate body of knowledge can be structured around mathematical modeling (analysis, specification, design).

This point characterizes the curriculum, rather than a detailed listing of the topics in each course, which would be beyond the scope of a paper anyway. Yet, some additional observations help convey the guiding idea of "unity in diversity".

Many non-CS courses are included, such as physics and engineering, for two reasons: (a) to enhance professionalism, as many students will need some classical engineering in their later career [20], (b) to broaden the intellectual horizon and bring students in contact with a wide variety of modeling techniques.

As in any science curriculum, courses form streams, with dependencies and prerequisites, which restricts ordering somewhat but also provides opportunities.

For instance, arguably the best language choice for Introductory Programming is Scheme, with Abelson and Sussman [1] as a reference textbook. Of course, programming assignments should use Haskell as well. This background in computing meets with Physics A in Classical Mechanics (Sussman and Wisdom [26]). The Lagrange-Hamilton approach used to be covered in classical engineering curricula and has been neglected for some years (perhaps for being less intuitive initially), but the time is ripe for reinstating it. This also paves the way for the later courses on quantum mechanics and quantum computing.

Algorithms and Data Structures emphasizes formal specification and derivation of algorithms (rather than just listing them) and elementary type theory. The main criterion is correctness; other issues are covered in Complexity.

The Signals and Systems course is another crucial link between continuous and discrete modeling. The book by Lee and Varaiya [16] is one of the rare textbooks so far that, in addition to providing a good overview of the field, identify and correct most of the numerous notational defects in classical mathematical notations. Such defects have always hampered clean formal reasoning.

Traditional CS topics like Computer Architecture, Operating Systems and Information and Database Systems may have become less central to curriculum design in the past decades, but they are included as yet another opportunity for exercising and consolidating mathematical modeling techniques. This assumes the traditional content is revamped in this spirit.

Needless to say, Programming Languages is not meant to introduce languages, but to provide an introduction to language modeling, e.g., lattices and fixpoints, denotational semantics and more advanced type theory: language theory is the materials science for software. Application examples are the languages already encountered via the assignments in other courses (say, Scheme, Haskell, Maple, TLA+, Matlab, Simulink, LabVIEW, VHDL, Java, C++, . . .).

The sequence starting from Physics A to Elements of Quantum Computing expands the students' horizon with nonstandard computational models.

Software Engineering is left most open to interpretation; the only assumption being that it meets the standards of the remainder of the curriculum. The Formal Methods community has generated a wealth of suitable material. Lecturers may opt for staying within a single series with a wide scope, such as Bjørner's Software Engineering trilogy [2], or devote a full course to a specific approach (such as refinements) using diverse sources, or any other combination or variant.

In a 4-year BCE, table 2 can be extended with Software Engineering C and D, Computer Security, with more advanced topics such as category theory and refinements, and with "integrating" courses combining mathematical modeling techniques. Examples are: Global timekeeping, positioning and navigation and Embedded Systems in the Automotive Domain. The Automotive -, Train - and Avionics Domain are typical examples of domains where a great variety of modeling techniques must be used in harmony. Hence, although such titles may sound specialistic, the content can be given a wide scope. The educational value of topics with this characteristic is well-known in classical engineering, but the domains idea has been given a new impetus for (E)CE by Dines Bjørner [3].

Course metrics The numbers in table 2 indicate course size in units, assuming a 60 units per year system. The design is adaptable to various local situations.

For instance, regulations at our own School of Engineering stipulate per semester 30 units involving 900 hours of study (classes, guided or lab exercises, self-study), amounting to 30 hours per unit. The smallest useful size of a course is 3 units, with classes over 12 weeks at $1\frac{1}{4}$ h/week. The exam period is 4 weeks. Arithmetic shows that this does not favor activity during the 12 weeks of classes (and that the 900 hours are administrative fiction).

A better option is spreading the courses over 15 weeks, assuming per 3 units weekly 1 hour of classes plus $3\frac{1}{2}$ hours of self-study and lab exercises, if any. Own experience at various universities indicates that homework is considerably more effective than guided exercises in stimulating activity[1], but puts heavier demands on the staff. Anyhow, the total load of 45 hours/week is reasonable, and low in comparison to what is expected at world-class universities.

3.3 Local and international context

Effects of the local context Thus far, too many universities have been unable to integrate mathematical modeling throughout the CS/CE curriculum or approached the university level considered normal in classical engineering fields.

Still, a well-documented design of an actual curriculum integrating mathematics education with software engineering is the BESEME (BEtter Software Engineering through Mathematics Education) project [18].

This report also outlines the impediments to be expected [18, p. 6]:

- Students find it demanding.
- Most instructors must revise notes.

In reality, “revising notes” is a euphemism for acquiring essential background [12]. Whereas students are young and (hence supposedly) flexible, many lecturers do not embrace lifelong learning, especially if it entails novel ways of thinking [9].

Here are two more small “case studies” illustrating other concrete situations.

[1] Regular homework also better prepares students, making 1 week for exams sufficient.

(i) At our School of Engineering, the effects just mentioned have proved quite manifest. As a result, weaving mathematical modeling in the CE curriculum remains a faraway target supported by a minority, but felt as a threat by others.

At one stage, our CE program included a course designed like Application-Oriented Formal Logic and a sequel on formal modeling, both non-optional.

The positive effects were most apparent in how students who had taken these courses applied the acquired abilities in their MSc and PhD dissertations under various supervisors who were themselves not even involved in FM.

Still, courses with a strong mathematical content taught by various lecturers, not only in FM but also in classical engineering, got negative evaluations from a few CE students. This was used as an argument by a certain faction among the faculty to clamp down on the classical engineering course involved, and to relegate FM to just one course, for a few last-year students. The fact that foundations are most efective when laid early in the program was well-known.

(ii) In the design of their CS program, our School of Science made a more constructive use of this fact, and placed Application-Oriented Formal Logic with the contents described earlier in the first semester. The professor teaching the course is one of the author's best former students, although scarcity of qualified teaching assistants may have eroded the application-oriented elements somewhat.

These experiences confirm the observation by many authors regarding the overwhelming importance of general acceptance by the faculty, or at least tolerance from those not wanting to invest effort in new ways of thinking [12, 15, 29]. Other major obstacles against teaching CS/CE at the university level are inbreeding and the idea that CS-related courses can be entrusted to lecturers from a different engineering area (e.g., EE) with just some programming experience.

In principle, students pose fewer problems. Of course, in curricula where formal methods courses are isolated, their mathematical content and lack of applications in other courses demotivates some students. This is why, judging from the literature on teaching FM, so much effort is spent on student motivation, and positive results are strongly highlighted (although difficult to measure).

However, consider for the sake of comparison the student acceptance of Mathematical Analysis by the less mathematically inclined students (the math enthusiasts are not at issue here). Much of the acceptance stems from this topic being an unavoidable part of any first and second year engineering program. Choosing engineering means choosing Mathematical Analysis, even for CE majors. Ubiquitous use in other courses confirms relevance in the classical areas.

If a basic course like Application-Oriented Formal Logic is also made a fixed part of the first-year program (as in case study (ii) above), students tend to accept it more easily as characteristic for university-level (E)CE education. The basis given can then be assumed without further ado in all other courses, providing further consolidation by various applications, which motivates students by confirming relevance. Whether other courses pick up the thread depends on the lecturers, but at least the basis is there and the other courses can evolve gradually. Moreover, with this background the better students will be more demanding with respect to methodology, and subconsciously or consciously exert gentle pressure on the lecturers if necessary. The attitude of faculty towards this prospect is a decisive factor (and a quality measure), which closes the circle.

In this context, course evaluations must be used wisely. The comments of the students always provide very valuable information for the instructors. However, if students are given the impression (or confirmation) that their feedback is used uncritically for making ad hoc curriculum reforms, their answers will become heavily biased and only serve hidden agendas incompatible with quality.

International context Traditions regarding the mathematical content of engineering curricula vary greatly between countries, and are subject to oscillations and mutual feedback.

For instance, Belgian engineering schools used to have very strong mathematical requirements, enforced by an entrance examination. The high standards were set since the early 19th century by the Grandes Ecoles and in particular the Ecole Polytechnique in France.

Recent politically-inspired European reforms such as the Sorbonne-Bologna declaration have reshuffled the landscape, increasing the differences they pretended to reduce.

For instance, the entrance examination has been cancelled in Flanders, not in Wallonia. The impact has been downplayed, since at that time Flanders scored first among Western countries in the TIMSS (Trends in International Mathematics and Science Study) surveys. However, as a result the mathematically strong programs in secondary school experienced less interest, and declined. As the effects became noticeable in the TIMSS scores, attention shifted to the PISA (Programme for International Student Assessment) criteria. These critera are based on RME (Realistic Mathematics Education), which has strongly influenced math education for young children in some Western European countries. As expected, these countries now tend to get a better place in the PISA rankings than they formerly did under TIMSS, creating the illusion of improvement.

The catch is that RME focuses on concrete “everyday life” problem situations, for which actually just a modicum of common sense suffices. Now common sense is laudable, but genuine mathematics involves considerably more, such as developing abilities for logic and symbolic reasoning and for making abstraction. These abilities are precisely the most important ones for engineering, classical as well as computer-oriented.

The mathphobia deplored in [28] is certainly not limited to the U.S., where even ample effort is devoted to countering it, but has become a matter of fashion among certain groups of children and adults in Western Europe, even in countries that used to have a strong mathematical tradition. There are indications, needing further study, that some Eastern European countries have escaped this fashion.

Due to these various factors, the mathematical level of students entering engineering schools has decreased in various European countries. Since technology does not comply by lowering its complexity, the first year of the curriculum is important for helping the students to bridge the gap. A framework unifying the mathematics for classical and for computer engineering can be instrumental.

The top scores in the TIMSS are achieved by Asian countries. This indicates considerable potential for maintaining universities at a suitable level to meet the challenges of the future. The names in the literature constitute ample evidence. Judging by the faculty lists of universities in various regions, the U.S. appear to have tapped these intellectual resources more effectively than Europe.

The emphasis on mathematics that was consciously maintained throughout this discussion may appear excessive if one sees mathematics just as a collection of tricks and facts, which is the view held by many laypersons. However, the main educational value of mathematics resides in the development of effective problem solving, reasoning and abstraction abilities and the attitudes it imparts.

4 Conclusion

Referring to the quotation by Max Planck, we have argued that the most effective way for making Formal Methods an evident part of everyday practice is not convincing the current practitioners but investing in the education of future generations. Formal Methods, in the sense of mathematical modeling, can be the lever to lift the entire computing curriculum to the scientific and professional level that would be considered acceptable in classical university-level engineering. It goes without saying that this strategy is not directly needed, yet can still be inspiring, for universities where (E)CE education is already world class.

We have outlined the design principles and to some degree the content of a no-nonsense program that can serve as an archetype for various curriculum designs by tailoring it to the locally available structure and human potential. Graduates from such a program will consider formal methods as evident in their professional practice as classical engineering math in, say, electronics or mechanical engineering.

On the other hand, we have seen that the road to an integrated curriculum is fraught with many impediments, the most important obstacle perhaps being lack of intellectual flexibility and curiosity and, as Gries notes [12], sometimes even mathphobia on the part of the lecturers for the computing-related courses.

The last part of the paper addressed some local and international factors affecting the availability and effective use of intellectual resources in various parts of the world.

One final observation: throughout our argumentations, we have hesitated to invoke one of the primary tasks of a university, because the focus on direct utility in recent years has made even its mentioning suspect. This task is conveying to our students a rich intellectual heritage and stimulating curiosity, and ultimately contributing to their cultural development, not just their professionalism.

Such goals may appear overly ambitious and perhaps lofty today, but the least one can do is avoiding a curriculum ridden with shortcuts.

Acknowledgement The author thanks the anonymous reviewers for their many helpful comments and observations, some of which appear in the text this paper.

References

1. Harold Abelson and Gerald Jay Sussman with Julie Sussman, Structure and Interpretation of Computer Programs. The MIT Press (1996).
2. Dines Bjørner, Software Engineering (3 volumes). Springer (2006).
3. Dines Bjørner, "What do I mean by Domain?". `http://www2.imm.dtu.dk/~db/`
4. Raymond Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons, Johan Jeuring, eds., Generic Programming, pp. 89–119, Kluwer (2003).
5. Raymond Boute, "Can lightweight formal methods carry the weight?", in: David Duce et al., eds., Teaching Formal Methods 2003, Oxford Brookes University (2003). Web: `http://cms.brookes.ac.uk/tfm2003/papers/boute.pdf`
6. Raymond Boute, "Functional declarative language design and predicate calculus: a practical approach", ACM Trans. Prog. Lang. Syst. 27, 5, pp. 988–1047 (2005).
7. Raymond Boute, "Calculational semantics: deriving programming theories from equations by functional predicate calculus", ACM Trans. Prog. Lang. Syst. 28, 4, pp. 747–793 (Jul. 2006).
8. James B. Dabney and Thomas L. Harman, Mastering Simulink 4 (2nd ed.). Prentice Hall (2001).
9. Edsger W. Dijkstra, "How Computing Science created a new mathematical style", EWD 1073. University of Texas at Austin (Mar. 1990).
   Web: `http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1073.PDF`
10. Ganesh Gopalakrishnan, Computation Engineering: Formal Specification and Verification Methods (Aug. 2003).
    Web: `http://www.cs.utah.edu/classes/cs6110/lectures/CH1/ch1.pdf`
11. David Gries and Fred Schneider, A Logical Approach to Discrete Math. Springer (1993).
12. David Gries, "The need for education in useful formal logic", IEEE Computer 29, 4, pp. 29–30 (April 1996).

13. Henri Habrias and Sébastien Faucou, “Linking Paradigms, Semi-formal and Formal Notations”, in: C. Neville Dean and Raymond T. Boute, eds., Teaching Formal Methods, pp. 166–184, Springer LNCS 3294 (Nov. 2004).
14. Leslie Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002).
Web: `http://research.microsoft.com/users/lamport/tla/book.html`
15. Edward A. Lee and Pravin Varaiya, “Introducing Signals and Systems — the Berkeley Approach”. First Signal Processing Education Workshop, (Oct. 2000).
Web: `http://ptolemy.eecs.berkeley.edu/publications/papers/00/spe1/`
16. Edward A. Lee and Pravin Varaiya, Structure and Interpretation of Signals and Systems. Addison-Wesley (2003).
17. Timothy C. Lethbridge, “What knowledge is important to a software professional?”, IEEE Computer 33 5, pp. 44–50 (May 2000).
18. Rex L. Page, “Software is discrete mathematics”.
Web: `http//www.cs.ou.edu/˜beseme/besemePres.pdf`
19. A. Parkin, “Professional Programmers – Do They Read?”, Computer Bulletin, Series II, 4, p. 23 (1975).
20. David L. Parnas, “Education for computing professionals”, IEEE Computer 23, 1, pp. 17–22 (Jan. 1990).
21. David L. Parnas, “Predicate Logic for Software Engineering”, IEEE Trans. SWE 19, 9, pp. 856–862 (Sep. 1993).
22. Anthony Ralston, “Let’s Abolish Pencil-and-Paper Arithmetic”. Journal of Computers in Mathematics and Science Teaching, Vol. 18, No. 2, pp. 173–194 (1999).
Web: `http://www.doc.ic.ac.uk/˜ar9/abolpub.htm`
23. John Rushby and Natarajan Shankar, “Theorem Proving and Model Checking for Software”, Tutorial, Fourth Symposium on the Foundations of Software Engineering (Oct. 1996).
Web: `http://www.csl.sri.com/users/rushby/slides/fse4tut.ps.gz`
24. Joel Spolsky, “The Perils of Java Schools”, in: Joel on Software (Dec. 2005).
Web: `http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html`
25. Joel Spolsky, “Stackoverflow.com”, in: Joel on Software (Apr. 2008).
Web: `http://www.joelonsoftware.com/items/2008/04/16.html`
26. Gerald Jay Sussman and Jack Wisdom with Meinhard E. Mayer, Structure and Interpretation of Classical Mechanics. The MIT Press (2001).
27. George B. Thomas, Maurice D. Weir, Joel Hass, Frank R. Giordano, Thomas’s Calculus, 11th ed. Addison Wesley (2004).
28. Allen B. Tucker, Charles F. Kelemen, Kim B. Bruce, “Our Curriculum Has Become Math-Phobic!”, ACM SIGCSEB, SIGCSE Bulletin 33 (2001).
Web: `http://citeseer.ist.psu.edu/tucker01our.html`
29. Jeannette M. Wing, “Weaving Formal Methods into the Undergraduate Curriculum”, Proc. 8th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST) pp. 2–7 (May 2000). Web:
`http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/amast00.html`

An Introductory Course on Programming based on Formal Specification and Program Calculation

Nazareno Aguirre[1], Javier Blanco[2], María Marta Novaira[1], Sonia Permigiani[1], Gastón Scilingo[1]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Argentina, {naguirre,mnovaira,spermigiani,gaston}@dc.exa.unrc.edu.ar
[2] Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, Argentina, blanco@mate.uncor.edu

Abstract. In this paper, we report on our experience in teaching introductory courses on programming based on formal specification and program calculation, in two different Computer Science programmes. We favour the use of logic as a tool for software development, the notion of program as a formal entity, as well as some issues associated with efficiency. We also review and use in practical cases some important program transformation strategies, such as generalisation, tupling and modularisation.
We will describe our approach, as well as the advantages and drawbacks that we have observed over the years teaching these courses.

1 Introduction

It is generally agreed that teaching introductory courses on programming is a very difficult task. Often, such courses have various different aims in Computer Science curricula, besides providing students with the basics of programming. Some of these "extra" aims are training students in some of the necessary technologies they will need in later courses, and provide a glance of a bigger picture in software development, and the many challenges associated with it. It is not surprising then that many of the current approaches to a first course on programming are related to what is thought to be more useful to students in their later programming practices, typically strongly based on modern and sophisticated programming languages such as Java, and including small to medium size programming projects where students can experience, to some extent, some typical activities in software development (e.g., separated phases for analysis, design and implementation, the importance of modularity, testing, etc). This situation generally leaves lecturers with limited time to teach students the complexities associated with "programming in the small", and concentrating on reasoning about small programs (it is not surprising then the growing belief amongst practitioners that dealing with programming in the small is easy).

Some approaches related to the use of formal methods in introductory courses to programming have an emphasis on program verification. Often, these courses are based on structured programming, and tend to treat program verification as a task to be performed after one constructs a program. In our opinion, based on our own experience, and discussions with students and colleagues, this generally leaves students the feeling that verification is an additional "burden". Furthermore, if this is combined with the fact that usually one picks relatively simple problems for teaching programming and verification in introductory courses, students are typically left with the feeling that verification is not only additional burden, but also optional burden (students get the idea that they have to verify programs that they already know to be correct).

Partly motivated by the drawbacks of the above mentioned approaches, which we experienced in the past in our programmes when following them, we decided to try a formal approach to introducing students to programming based on formal specification and program calculation. The idea of this course is to attempt to make students feel the use of logic as a necessary tool, during the initial stages of programming, in the formal specification of problems, and as part of the body of rules for transforming these specifications into programs. We report on our experience in teaching introductory courses on programming based on this approach, in two different Computer Science programmes, at the National Universities of Córdoba and Río Cuarto, in Argentina. As we mentioned, the approach is based on program calculation. In order to make the calculations smooth, and not having to deal with the complexities of imperative languages, most of our approach is based on functional programming. The connection to imperative programming, although limited, is based on program transformation. As it will be described later on, we skip many of the important "programming in the large" issues, as well as technicalities associated with imperative (or object oriented) languages. We favour instead the use of logic as a tool, the notion of program as a formal entity, as well as some issues associated with efficiency. We also review and use in practical cases some important program transformation strategies, such as generalisation, tupling and modularisation. Also, as it will be explained more throroughly later on, we try to carefully choose the exercises, trying to emphasise the cases in which problems would be extremely difficult to solve if they are not formally manipulated. We describe our approach via one of these cases, the segment of minimum sum problem. We will also discuss about some of the effects of our courses in the corresponding Computer Science programmes, and how these have influenced later courses.

2 Description of the Course

As explained before, the course we are describing here is an introductory course on programming taught at two different Universities. In the National University of Córdoba, the course is taught during the first two semesters of a 5-year Computer Science programme. In the National University of Río Cuarto, on the other hand, the course is taught during the third and fourth semesters, again of a 5-year Computer Science programme. In the former, students have no previous courses on programming or logic, but they take simultaneously with this course one on discrete mathematics; In the latter case, when students start the course they had already taken a 2-semester introduction to imperative programming course, and a basic course on mathematical logic.

Although the course is taught in different contexts in the two Universities, we seek achieving the same general goals. Essentially, we want the students to accomplish the following:

- Develop the ability to formalise problems, using logic as a tool.
- View specifications and programs as formal entities, and consider programming as the manipulation of these formal entities.
- Obtain considerable training on using recursion as a powerful mechanism for defining functions/programs.
- Understand that reasoning about functions can be exploited not only in functional programming, but also in imperative programming (particularly via transformation schemata).
- Get acquainted with a very elementary theory of abstract data types (and its relevance in program development).

The contents of the course is composed of the following three main modules:

- Logic and specifications. We employ an equational version of predicate logic with generalised quantifiers [7] (see also [9, 6]). The main goal is to have a suitable tool for reasoning with large formulae (mainly programs). This logic is used for both the functional and the imperative paradigm.
- Construction of functional programs. The students learn how to formally construct functional programs from specifications, with their corresponding inductive proof of correctness. We make an induction guided use of the *fold/unfold* rules, in order to guarantee termination in the calculated programs, as in [10] . Operational reasoning appears only as a motivation for the axioms of the calculus, and for efficiency considerations. In this paradigm, programs and specifications are written in the same formalism (programs are a subset of the possible formulae) [10].

– Construction of imperative programs. This module is rather traditional (see for example [5, 8, 6, 11, 4]). We try, however, to use what was learned in the previous module to help in this process. The main tool is to translate functional programs into the imperative formalism by using tail recursion, which not only allows us to translate the program itself, but also its proof of correctness. Although the underlying computational models are different in the functional and imperative paradigms, the students can get the feeling that proofs in the two contexts share similar ideas (e.g., invariants can be seen as a restricted way to use induction). Also when introducing imperative programming, students are faced with the notion of abstract data type. We employ the case of lists (which are somehow inherent to functional programming) and an array based implementation of lists in imperative programming, to show how functional programs handling lists are transformed into imperative programs manipulating arrays. The usual formal concepts of abstraction function, representation invariant and the like are used superficially.

3 A Sample Exercise

In order to better illustrate our approach, and the notation used, let us provide an example, which is an exercise used in the course. Consider the problem of, given a list `xs` of integers, finding the sum of the elements in the segment of minimum sum of `xs`. A segment of `xs` is simply any sublist of `xs`. So, for instance, if list `xs` is `[1,-4,-2,1,-5,8,-7]`, then the minimum sum segment of `xs` is `[-4,-2,1,-5]`, and its sum equals `-10`; if we consider the list `[1,2]`, then its minimum sum segment is `[]`. This problem has been discussed by various researchers (particularly in the context of program transformation), for example in [2, 3, 10].

A first step is to formally specify the problem. For this task, the specification language we use provides generalised quantified expressions (with general rules for dealing with these), which can be built out of any binary operator, as long as it admits a neutral element, and is associative and commutative. This style is similar to that used in [9] The generalised expression corresponds to applying the operator under consideration to a range of values. The operator Min satisfies the above conditions, and therefore can be used in a quantified expression, allowing us to straightforwardly specify the problem in the following way:

$$\mathit{minSum}.xs = \langle \mathsf{Min}\ as, bs, cs : xs = as{+}{+}bs{+}{+}cs : \mathit{sum}.bs \rangle$$

where *sum* is a function that computes the sum of the elements of a list, for which we already have an operational version. Deriving a recursive function from the above specification is done via induction on the length of `xs`. A detailed calculation of function *minSum* can be found in [10], page 147. The resulting recursive function is the following:

$$\begin{aligned} \mathit{minSum}.[] &= 0 \\ \mathit{minSum}.(x \triangleright xs) &= g.(x \triangleright xs) \ \mathsf{min} \ \mathit{minSum}.xs \end{aligned}$$

where g is defined as follows:

$$\begin{aligned} g.[] &= 0 \\ g.(x \triangleright xs) &= 0 \ \mathsf{min} \ (x + g.xs) \end{aligned}$$

From these functions, we can do various things. For instance, we can employ schemata for transforming the above functions to tail recursive versions, and from the resulting functions straightforwardly obtain imperative programs. Also, we could attempt to derive a more efficient version of *minSum*, using transformation strategies (in this case, tupling is a suitable one). The resulting more efficient version of *minSum*, obtainable using tupling on *minSum* and g, is the following:

$$\begin{aligned} h.[] &= (0, 0) \\ h.(x \triangleright xs) &= ((x + b) \ \mathsf{min} \ a, 0 \ \mathsf{min} \ (x + b)) \\ & \qquad\qquad\qquad |[(a, b) = h.xs]| \end{aligned}$$

4 Experiences in the two Programmes

We have been using the described approach for the past 10 years in the National University of Córdoba, and for the past 6 years in the National University of Río Cuarto. The experience we gained along these years enabled us to improve the course in various respects, in particular, collect better exercises, with interesting non trivial solutions, illustrating some of the benefits of program calculation. This task has benefited from newer mature bibliography, such as [1], and including bibliography in Spanish (e.g., [12, 13]).

We have also observed various advantages and drawbacks associated with the course. As advantages, we can say that students who get to assimilate the principles taught in the course have demonstrated to incorporate these in their programming practices, in particular in later courses. Also, although they generally do not use formal approaches in later programming courses, the acquired skills in logic and program manipulation leads to producing better programs (with fewer bugs and clearer), and to a more careful reasoning when programming. This is noticed in later courses on data structures and algorithms; the students who assimilated the concepts taught in our course tend to specify routines (e.g., provide pre- and post- conditions for methods in object oriented implementations of abstract data types), write well structured programs, and exploit recursion when solving problems in imperative and object oriented languages. In general, good students tend to appreciate formal methods and their associated benefits, which is evidenced in the optional courses they choose later on in the programmes, and the topics they choose for their final projects. We also detected a number of drawbacks. First, about 50% of the students fail the course. This seems to be a very critical point against the approach; however, previous versions of the introductory courses (more traditional ones) had roughly the same failure rate. This is the case in both Universities. In the case of the University of Córdoba, where the course is taught in the first year of the Computer Science programme, a great part of the students failing the course switch to a different programme (this is normal in the first year of most programmes in Argentina). In the case of the University of Río Cuarto, on the other hand, the 50% failure rate in the second year of a programme is not normal (although it has been the usual for this course, even for previous, more traditional, versions of it). In both Universities, students failing the course can take it again the following year (i.e., they are not forced to abandon the corresponding programmes).

Another important drawback is that the average student usually becomes too "syntactic" (or "mechanical") in reasoning about programs, which has a negative impact in abstraction. In particular, it is usually rather difficult for students to "jump" in and out from the calculus (i.e., take perspective on the situation of the derivation at hand, and decide accordingly). In order to overcome this problem, we are currently seeking for an integration between the described course and a later course on data abstraction and the implementation of abstract datatypes. We hope that as a result of this integration the students will have a chance to exercise the role of abstraction in problem solving, which requires them to combine thinking outside the calculus, for designing data representations, and using the calculus, for deriving correct implementations for the operations associated with the data abstractions.

5 Conclusions

We have described an approach to a first course on programming strongly based on formal specification and program calculation, that we have been using in the National Universities of Río Cuarto and Córdoba in Argentina. We have enumerated some of the advantages that we have observed, as well as a number of important drawbacks of the approach. Overall, we have been satisfied with the results. For the particular case of the course in Río Cuarto, where the course is taught in the second year, surprisingly the results have generally been worse than in Córdoba. We believe this might be due to the case that, since the students have already learned the basics of programming in an informal setting, most tend to feel that the rigour associated with the program calculus used in the course is not necessary for programming. Solving more complicated problems, compared to those in their first course, and developing more sophisticated solutions, helps in making students appreciate the benefits of a calculational approach. Students in Río Cuarto also exhibit a greater resistance, compared to students in Córdoba, to learning and applying logic for specification.

As work in progress, we are starting to conduct a more thorough study of the consequences of our approach, mostly based on qualitative analysis.

References

1. R. Backhouse, Program Construction, Calculating Implementations from Specifications, Wiley, 2003.
2. J. Bentley, Programming Pearls, second edition, Addison-Wesley, 2000.
3. R. Bird, Algebraic Identities for Program Calculation, The Computer Journal, 32(2), Oxford University Press, 1989.
4. E. Cohen, Programming in the 1990s: An Introduction to the Calculation of Programs, Springer-Verlag, 1990
5. E. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.
6. E. Dijkstra and W. Feijen, A Method of Programming, Addison-Wesley, 1988.
7. E. Dijkstra and C. Scholten, Predicate Calculus and Program Semantics, Monographs in Computer Science, Springer-Verlag, 1990.
8. D. Gries, The Science of Programming, Monographs in Computer Science, Springer-Verlag, 1981.
9. D. Gries and F. Schneider, A Logical Approach to Discrete Math, Monographs in Computer Science, Springer-Verlag, 1993.
10. R. Hoogerwoord, The Design of Functional Programs: A Calculational Approach, PhD Thesis, Eindhoven University of Technology, The Netherlands, 1989.
11. A. Kaldewaij, Programming: The Derivation of Algorithms, Prentice Hall, 1990.
12. N. Martí-Oliet, Y. Ortega-Mallén and J. Verdejo-López, Estructuras de Datos y Métodos Algorítmicos, Ejercicios Resueltos, Pearson Educación, 2004.
13. N. Martí-Oliet, C. Segura-Díaz and J. Verdejo-López, Especificación, Derivación y Análisis de Algoritmos, Ejercicios Resueltos, Pearson Educación, 2006.

Evolution of a Course on Model Checking for Practical Applications

Yasuyuki Tahara[1], Nobukazu Yoshioka[2], Kenji Taguchi[2], Toshiaki Aoki[3], and Shinichi Honiden[4]

[1] The University of Electro-Communications, Japan
[2] National Institute of Informatics, Japan
[3] Japan Advanced Institute of Science and Technology
[4] National Institute of Informatics / The University of Tokyo, Japan

Abstract. Although model checking is expected as a practical formal verification approach for its automatic nature, it still suffers from difficulties in writing the formal descriptions to be verified and applying model checking tools to them effectively. The difficulties are found mainly in grasping the exact system behaviors, representing them in formal languages, and using model checking tools that fit the best to the verification problems. Even capable software developers need extensive education to overcome the difficulties. In this paper, we report our education course of practical applications of model checking in our education program called Top SE. Our approach consists of the following two features. First, we adopt UML as the design specification language and create the descriptions for each specific model checking tool from the UML diagrams, to enable easy practical application of model checking. Second, we build taxonomies of system behaviors, in particular behaviors of concurrent systems that are main targets of model checking. We can organize the knowledge and the techniques of practical model checking according to the taxonomies. The taxonomies are based on several aspects of system behaviors such as synchronization of transitions, synchronization of communications, and modeling of system environments. In addition, we make clear which model checking tools fit which types of systems. We treat the three different model checking tools: SPIN, SMV, and LTSA. Each tool has its specific features that make the tool easier or more difficult to be applied to specific problems than others. In our education course, we explain the taxonomies, the knowledge, and the techniques using very simple examples. We also assign the students exercises to apply the knowledge and the techniques to more complicated problems such as the dining philosopher problem, data copying between a DVD recorder and a hard disk recorder, and the alternating bit protocol.

1 Introduction

This paper describes one of our courses on model checking given in the Top SE program [1]. This project was established to bridge the gap between the software industry and software engineering education in Japan. The model checking technique typically addresses such a gap between education focusing on the fundamental theories, basic techniques and toy problems, and industry demanding techniques applicable to practical problems of decreasing faults in large-scale complex software products. Thus our model checking courses should aim addressing practical applications of the technology.

In our previous work [1], we explained two model checking courses as sample courses of Top SE, called "Verification of Design Models". The main features of the courses consisted of the following four: (1) it treated UML, (2) it arranged practical knowledge and techniques according to the verification process, (3) it treated three model checking tools (SPIN [2], SMV [3], and LTSA [4]), and (4) it provided practical project work using evaluation boards from prototypical digital home appliances (hard disk (HD)/DVD recorders). The two courses were originally given as one course. After giving these original courses to the first year students, as many as nineteen, and examining their achievements and their responses to the course, we noticed the following two issues. First, the students felt the gap between the basic usage of the tools and their application to the practical problem of verifying HD/DVD recorder software. Second, students did not succeed in understanding the differences of the three tools and how to choose one from them that is the most appropriate to the specific problem they were tackling.

In this paper, we describe the newly organized advanced course as the solutions to the above issues. We described the Verification of Design Models (Foundations) courses in [1]. The model checking courses are currently called "Design Verification". Thus this paper treats the "Design Verification (Advanced)" course. In this course, we built taxonomies of system behaviors and made the students apply the model checkers to various design models which represents the different types of the behaviors. The taxonomies are based on several aspects of system behaviors such as synchronization of transitions, synchronization of communications, and modeling of system environments. According to the taxonomies, we provided various types of examples. For example, we can model the dining philosopher problem using the different synchronization mode of the transitions. This means that we can assume each philosopher behaves synchronously or asynchronously. We expected that the students can well understand how to choose the most appropriate tools to their specific problems. It is ideal if the students can find a new tool or customize existing tools for themselves and judge if the tool is more appropriate to the problems.

This paper is organized as follows. Section 2 presents a brief overview of the Top SE program. Section 3 describes the details of our courses devoted to model checking of system behavior designs. In particular, we focus on the features of our new Advanced course that makes use of the taxonomies. Section 4 compares our course with existing similar ones. Section 5 gives some concluding remarks and the future work.

2 Overview of Top SE

In this section, we will briefly introduce our education program Top SE. Top SE is a non-accredited program at Masters level fully funded by the Japanese Government, and is operated through a close collaboration between industry and academia at the National Institute of Informatics. The whole curriculum, which consists of five lecture series (Requirements Analysis, System Architecture, Formal Specifications, Model Checking and Implementation) is shown in Table 1. The overview and the curriculum design of the program are discussed in [1].

The program has the following distinguishing features:

- The treated topics include only advanced software engineering technologies that have theoretical backgrounds and supporting tools that are easily available. For this characteristic, the students are required to have sufficient basic knowledge about software engineering in advance.
- Top SE has currently seventeen courses shown in Table 1. Each course consists of twelve lectures for three months. Students needs to finish at least eight courses for each one to start their final year projects. Completing the entire curriculum requires at least one year and a half.
- Each course includes a considerable amount of exercises based on practical problems usually carried out by groups. Each course usually treats multiple techniques and tools to give the students skills and knowledge to choose techniques or tools appropriate to the practical development situations. Some courses use even actual devices to practice on the developed software embedded and operated in the devices. Figure 1 shows a set of evaluation boards as an example of the devices.
- Course materials are developed under close coordination of the academia and the industry. The academia provides their advanced knowledge about software engineering, while the industry provides practical problems they are tackling in the actual working situations.

That is, it covers a wide range of cutting-edge software engineering technologies, e.g., software patterns, aspect-oriented development and model checking. We regard our education program as a medium to transfer these new technologies to industry. In order to do so, we tailor our program to meet educational needs from industry and create case studies for laboratory work with industrial partners (Hitachi, NEC, Toshiba, NTT Data, Fujitsu, and Nihon Unisys, among

Table 1. Curriculum

Series	Courses
Requirements Analysis	Requirements Analysis
	Security Requirements Analysis
System Architecture	Component-based Development
	Software Patterns
	Aspect-Oriented Development
Formal Specifications	Formal Specifications (Foundation)
	Formal Specifications (Advanced)
	Formal Specifications (Security)
Model Checking	Design Verification (Foundation)
	Design Verification (Advanced)
	Real-time Model Checking
	Modelling and Verifying Concurrent Systems
Implementation	Testing
	Program Analysis
	Software Model Checking
Foundations	Basic Theory
	Software Engineering Practice

others). We also have close collaborative relationships with academic partners (Shinshu Univ. , Tsukuba Univ. , etc) to develop and deliver the courses.

As Software Engineering is a practical engineering discipline, we teach good practices rather than theories in all courses. We specifically emphasize the use of tools and all courses are based on laboratory works in which several tools are used.

3 Design Verification Courses

This section describes our courses devoted to model checking of system behavior designs.

3.1 Verification Process

As described in [1], these courses focus on efficient use and proper application of model checking tools that use the automata theory, specifically SPIN, SMV, and LTSA. The reorganized courses still share these aims and the four features to achieve them described in Section 1. Therefore we teach the following verification process (Figure 2 [1]). The verification process consists of six steps and five kinds of verification models and descriptions: the design model (DM), verification requirements (VR), model checker independent model (MCIM),

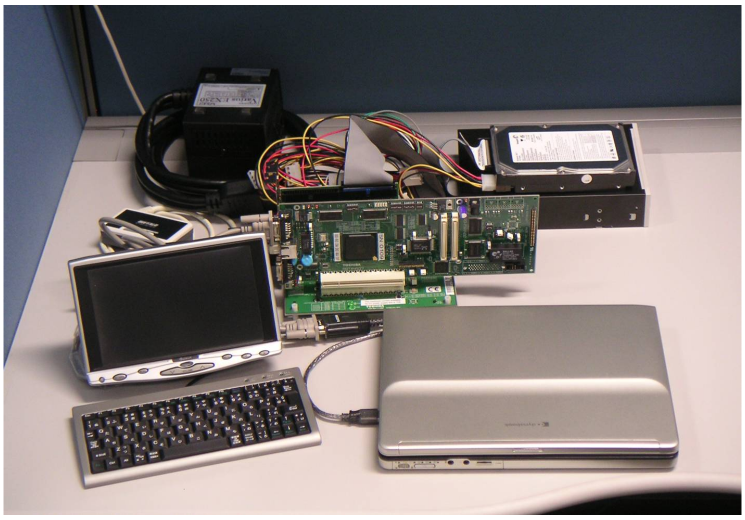
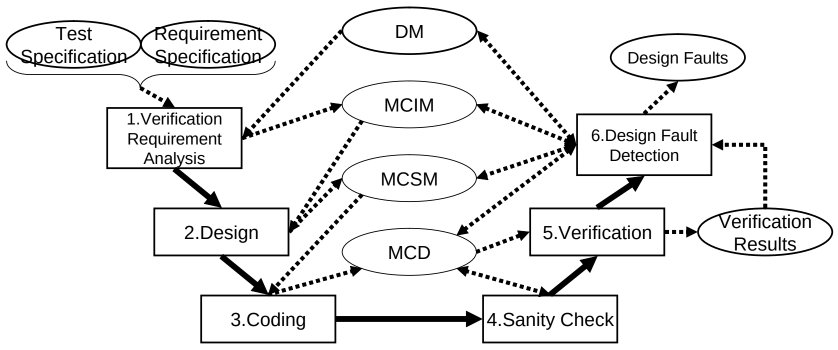


Fig. 1. Evaluation Board Used in Top SE Courses

model checker specific model (MCSM), and model checker description (MCD) as shown in Figure 2. Figure 3 explains why so many models are used. Once an MCIM is created, the developer needs to know only the methods of transformation from the MCIM to the MCSM if the developer uses a new model checker. Therefore this process enables the developer to choose appropriate model checkers easily. The process presupposes that the developer has finished requirements analysis and abstract design for the target system without considering the verification process. Thus the process takes as inputs three kinds of information: (1) a DM described in UML, (2) a requirement specification representing the required behaviors of the target system which the DM should realize, and (3) a test specification representing the constraints on system behaviors that the DM should satisfy. All of them are given from the development process. The verification process should produce two kinds of information: (1) possible design faults and (2) the verification results including verification succeeded or not and the counterexample if verification did not succeed. The design faults and counterexamples are then fed back to the development process.

The Foundation Course treats only SPIN and teaches the verification process using toy problems and practical problems. The practical ones are about the HD/DVD recorder problem. On the other hand, the Advanced course treats some other problems with intermediate complexity. The details are explained below.



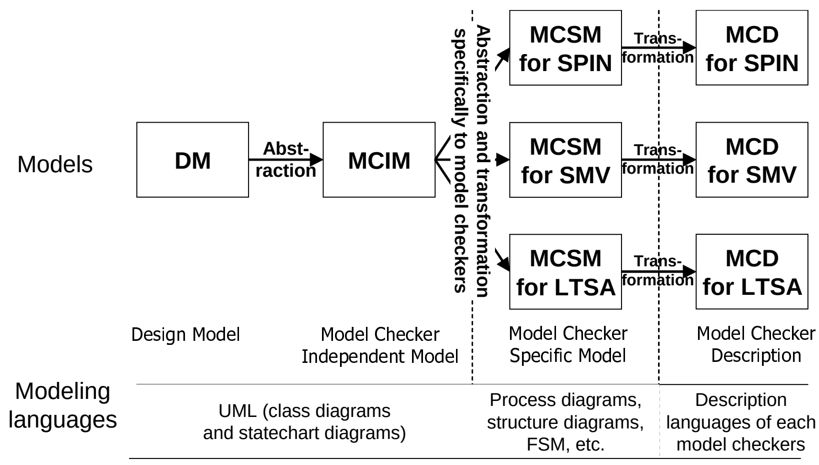
Inputs of the process: DM, requirement specification, and test specification
Outputs of the process: Design faults, and verification results.

Fig. 2. Overview of the Verification Process

3.2 Taxonomies of System Behaviors

The aim of the Advanced course is to enable students to choose the most appropriate model checker for the specific problems they are tackling and to make effective use of the tool. For this purpose, we first introduce taxonomies of system behaviors as the criteria of choosing the tools and the modeling policies. The taxonomies are based on the following characteristics of system behaviors.

– Synchronization of transitions
  In concurrent systems, the processes may execute their actions in a variety of timings. In our Design Verification courses, we use state machine diagrams to specify the system behaviors. Therefore the executions of the actions are represented by transitions. We classified system behaviors into the following two types from the viewpoint of transitions. One is synchronous transition model and the other is asynchronous transition model. In the synchronous transition model, all the processes should trigger their transitions in each execution step. If even only one of them can trigger no transitions at some time, the entire system falls into deadlock. On the other hand, the asynchronous transitions model forces no constraints on triggering the transitions. The latter model is usually represented by the interleaving semantics formally.
– Synchronization of communications
  Even asynchronous transition model usually requires synchronization mechanisms for the processes. We treat communication among such mechanisms in particular because it is easily represented by UML. It is natural to classify



Models
DM
Abst-raction
MCIM
Abstraction and transformation specifically to model checkers
MCSM for SPIN
MCSM for SMV
MCSM for LTSA
Trans-formation
MCD for SPIN
MCD for SMV
MCD for LTSA
Design Model
Model Checker Independent Model
Model Checker Specific Model
Model Checker Description
Modeling languages
UML (class diagrams and statechart diagrams)
Process diagrams, structure diagrams, FSM, etc.
Description languages of each model checkers

Fig. 3. Use of Models in Verification Process

the mechanisms into synchronous ones and asynchronous ones. Each type is rigorously defined by the semantics of UML state machine diagrams.

– Environment modeling
  Practical systems interact with the external environments in most cases. We need to consider such interactions when verifying the system behaviors. In this course, we treat the behaviors of communication media as an example of such environments. We classify the behaviors according to the following types of the characteristics of the media: (1) the communication is stable: it always carries the signals correctly, (2) the signals may be altered, and (3) the signals may vanish.

Among these taxonomies, we explain the details of the above two. In theory, we have two times two, that is, four types of behaviors by combining these two classifications. However, we do not treat the synchronous transition model with communication because it is not easy and not so useful. Therefore we treat only the three types.

We teach to the students the taxonomy according to the synchronization of communication by using the simple example shown in Figure 4. Figure 4 represents a system composed by the two processes P and Q.

1. The first action is common to both of the synchronization modes. It makes P send the event e1 and change its state to state2, and Q receive e1 and change its state to state2 (Figure 5).
2. Next, both of the synchronization modes allow one of the following behaviors.

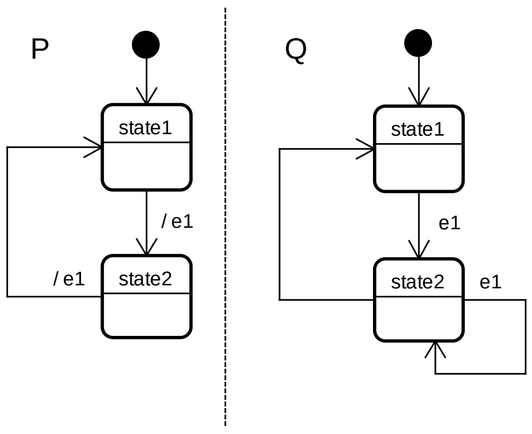


Fig. 4. Example of State Machine Diagram

- After only Q changes its state to state1, P and Q exchanges e1 again. The state of P is now state1 and that of Q is state2 (Figure 6).
- P and Q exchanges e1 again. The state of P is now state1 and that of Q is still state2 (Figure 7).

3. Only the asynchronous communication model allows the following action. Though P sends e1 and changes its state to state1, Q does not receive e1. Q also changes its state to state1 (Figure 8). The synchronization communication model does not allow this action because this model forces a sender or a receiver of an event to wait until its counterpart appears.

After that, we show the students the following summary of the above explanations. The asynchronous communication model allows all the behaviors (1), (2), and (3) of Figure 9. On the other hand, the synchronous communication model does not allow (1).

As for the synchronous transition model, we avoid interprocess communications and use the state machine diagram shown in Figure 10 This diagram allows only the behavior shown in Figure 11.

Next, we teach the students how to create input descriptions for each model checker. The details of the creation method are as follows.

1. Create preamble parts of the descriptions such as definitions of variables, symbols, and communication channels.
2. Create descriptions of a module for each process. Such a module is represented by a proctype for SPIN, a module for SMV, and a process for LTSA.

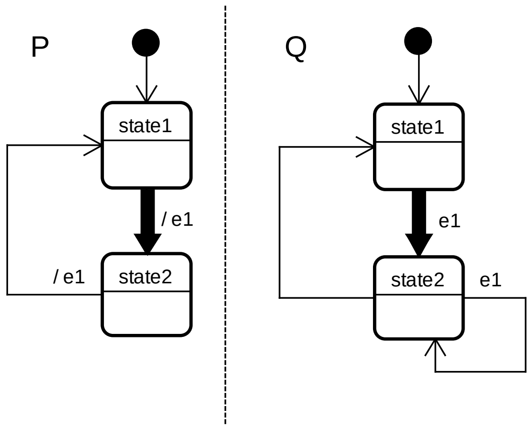


Fig. 5. First Action

3. Transform each transition to the corresponding descriptions. Such descriptions are, for example, changes of state variable values and message exchanges for SPIN.

After that, we make the students work on the small exercises described afterwards.

3.3 Course Organization

The Advanced course is organized as Table 2.

In the week 1, after giving a brief review of the Foundation course, we show the difficulties in modeling concurrent distributed systems. In detail, we make the students understand the difficulties involved in the synchronization modes of transitions and communications and the difficulties in modeling environments. We use the following problems in the explanations

- The dining philosopher problem
  This problem is appropriate for the issues about the synchronization modes because we can design the behaviors using both communications and shared variables. We can also use both synchronous and asynchronous communications.
- The HD/DVD recorder problem We use a version of this problem simpler than the one used in the Foundation course as follows.
  - The problem is about copying data between one DVD recorder and one HD recorder.

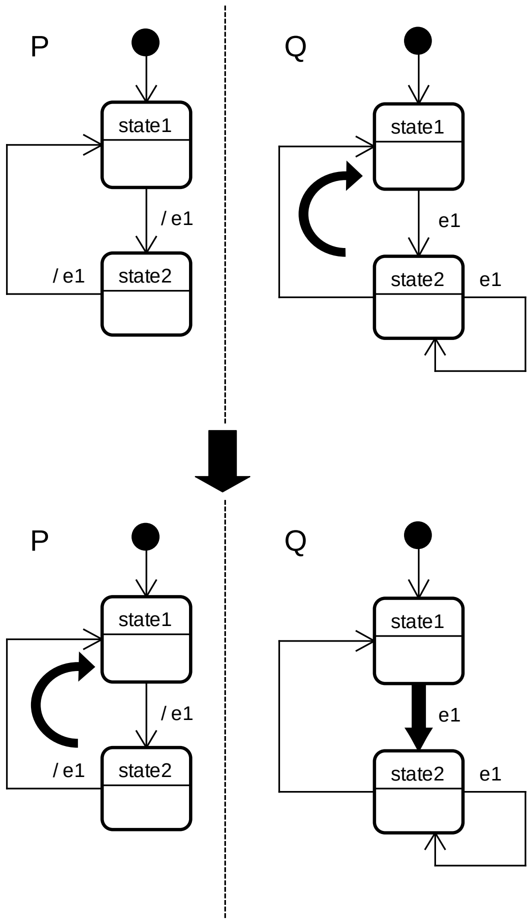


Fig. 6. Second and Third Actions: Pattern 1

- The DVD recorder the HD recorder an event requesting permission to record the data stored in the HD recorder to DVD media. The HD recorder sends back an acknowledge event as the response.
- The students need to verify if the copying functionality works correctly.

This problem is also concerned with the difficulties of the synchronization modes. The aim of using this problem is to make the students understand that the difficulties also arise in practical systems.

– The alternating bit protocol [5] problem
This protocol is used for unstable communication media that may alter or lose the data. This problem is used to make the students understand how to model such environments and carry out verification considering the environ-

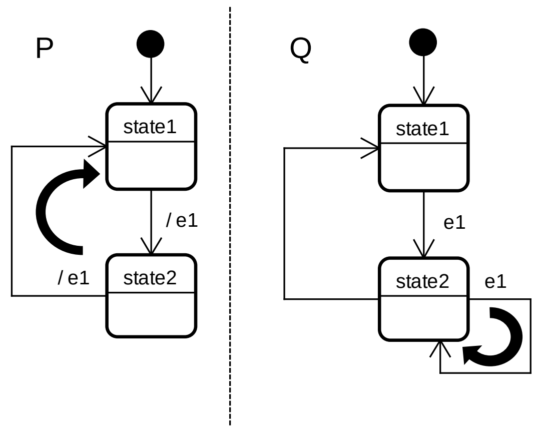


Fig. 7. Second Action: Pattern 2

ments. The students need to verify if the data are correctly transferred from the sender to the receiver in each type of environment.

Finally, we order the students to examine the problems to work on in the final group work. It is desirable if the students choose problems they are facing at their workplaces.

The weeks 2, 5, and 7 treat the issues of the synchronization modes of transitions and communications. As exercises The students work on the simple examples shown in 3.2, the dining philosopher problem, and the HD/DVD recorder problem. The weeks 3, 6, and 8 treat the issues of environment modeling using the alternating bit protocol problem. The students apply the model checkers to the problems as follows.

1. They examine how the systems behave using the simulation functionalities of the tools.
2. They verify the properties such as deadlock-freedom, livelock freedom, and progress. They use the options and the functionalities to verify LTL formulae of the tools.

The week 9 is devoted to teaching tips and techniques of model checking such as abstraction, fairness constraints, optimization of verification, and the property specification patterns described in [6].

In the weeks 10 and 11, the students are divided into groups and carry out the group work. Each group consists of three to five people. Each group chooses a problem one of the members has been examining since the first week. They are instructed to make explicit the following data.

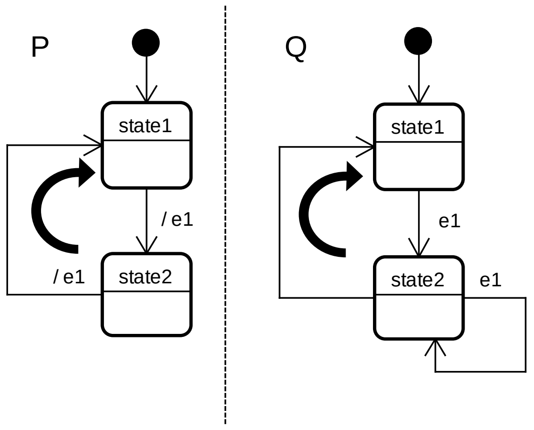


Fig. 8. Second Action: Pattern 3

- The problem descriptions including:
  - The requirements specifications described in natural languages and use cases,
  - The prerequisites and the assumptions of the system described in natural languages and UML diagrams, and
  - The extents of the system to be designed including exceptional cases described in natural languages and use case descriptions (for exceptional scenarios).

  It is desirable if the problem can be modeled using as many synchronization modes as possible.
- Property specifications to be verified
- Chosen tools and the reason of the choice
- Design specifications to be verified (MCIM in 3.1) including the system itself, the environments, and the considered abstractions
- Verification activities including the used functionalities of the tools, the order of the use of the functionalities, and the adopted tips and techniques
- Verification results including the counterexamples, the detected faults in the design specifications, and the corrections of the faults

Finally, in the week 12, representatives of each group make presentations of the group work results. The students and we discuss the results.

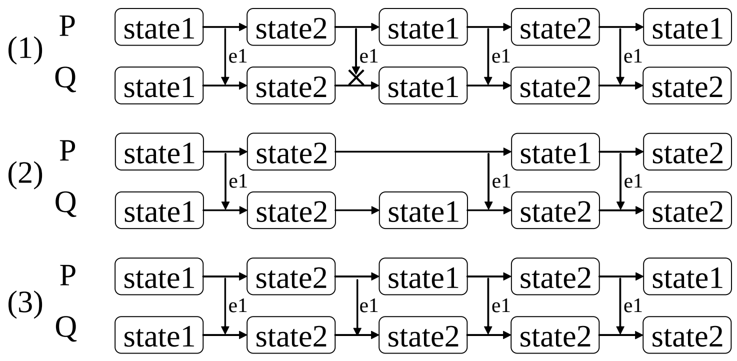


Fig. 9. Behaviors Allowed by Asynchronous Communication Model

Table 2. Course Organization

Week 1	Introduction, Review of the Foundation Course, Issues to be handled in this course (difficulties in modeling concurrent distributed systems)
Week 2	Verification of synchronous and asynchronous system behaviors using SPIN
Week 4	Introduction to SMV
Week 5	Verification of synchronous and asynchronous system behaviors using SMV
Week 6	Verification considering environment models using SMV
Week 7	Introduction to LTSA and verification of synchronous and asynchronous system behaviors using LTSA
Week 8	Verification considering environment models using LTSA
Week 9	Tips and techniques for model checking
Week 10, 12	Group work for practical problems
Week 12	Group presentations and conclusions

3.4 Current Status

At the time this paper is written (the end of November 2007), the first operation of the Advanced course has just finished. Therefore we cannot much evaluate the achievements. Seven students participated in the group work. They are divided into two groups. Each group chose the following problems.

- Verification of the RTS/CTS protocol for the hidden node problem
  This protocol is used in the IEEE 802.11 standard for wireless LAN. The group tried to verify the deadlock-freedom and the correctness of the functionalities of the protocol.
- Verification of a shared resource management system

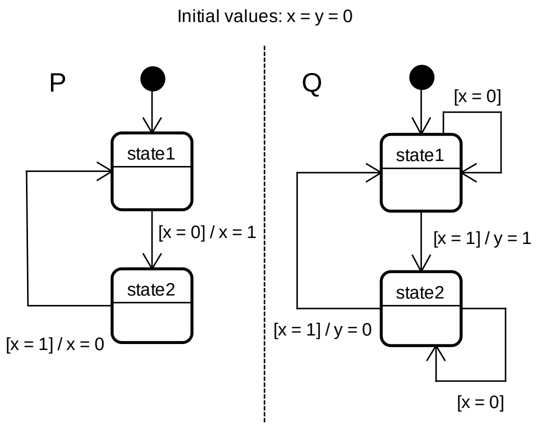


Fig. 10. Example of Synchronous Transition Model

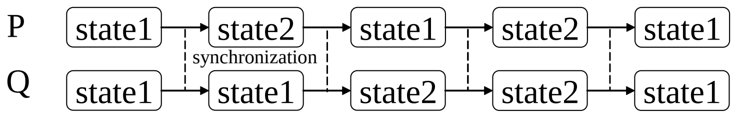


Fig. 11. Behavior of Synchronous Transition Model

The group tried to verify the same properties as above for an abstract design of a realistic system. A member of the group is working on development of the system in his company in practice.

Only a few students completed all the coursework reports so far. Seeing this fact and the presentations of the group work, the students understood the difficulties and the outlines of the solutions to the difficulties. We need more time to judge if the students can use the model checkers efficiently in practice.

4 Related Work

There are a huge number of courses treating model checking easily found by WWW search engines such as Google with keywords such as "model checking", "course" and "lecture". There are many courses sharing features similar to ours. The following courses are just a few examples.

– Treating multiple model checkers

Atlee [7] treats SMV, LTSA, and Alloy (called Alcoa at that time). Chechik [8] treats SMV, SPIN, and some software model checking tools. There are many coursed treating two model checkers.

- Treating synchronization modes of communications

  The chapter 10 of [9] describes how to verify systems with synchronous and asynchronous communications using LTSA.
- Including project work on practical problems

  Atlee [7] also provides a project work on a problem of an elevator system. Mitchell [10] treats several analysis problems of practical security protocol including model checking.

However, we can find no courses treating various types of problems that require considerations on various types of modeling strategies including synchronization modes of not only communications but also transitions and various types of environment modeling strategies. We incorporated such treatments into our course through our experiences of providing the previous model checking course. By these treatments, we expect our course succeeds in making the students choose tools appropriate to specific problems.

5 Conclusions

In this paper, we described one of our courses on model checking given in the Top SE program. In particular, we focused on the newly organized "Design Verification (Advanced)" course. In this course, we built taxonomies of system behaviors and made the students apply the model checkers to various design models each one of which represents each type of the behaviors. The taxonomies are based on several aspects of system behaviors such as synchronization of transitions, synchronization of communications, and modeling of system environments. In addition, we made the students work on exercises using the various types of modeling strategies in the taxonomies. The exercises treat various types of problems, too. We expected that the students can finally find or create a new tool for themselves and judge if the tool is more appropriate to the problems.

As described in 3.4, we have not yet completed the evaluations of the achievements of this course. We will carry out them by analyzing the coursework reports and the questionnaires given to the students. Next, we will improve the course on the basis of the analysis as well as the other related courses of Top SE including those in the Model Checking course series.

References

1. Honiden, S., Tahara, Y., Yoshioka, N., Taguchi, K., H.Washizaki: Top SE: Educating super-architects who can apply software engineering tools to practical development in Japan. In: Proc. of ICSE 2007. (2007) 708–717
2. Holzmann, G.J.: The SPIN model checker: Primer and reference manual. Addison Wesley (2004)
3. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Norwell, MA, USA (1993)
4. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs, Second Edition. John Wiley & Sons (2006)
5. Carver, R.H., Tai, K.C.: Modern Multithreading : Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs. Wiley-Interscience (2006)
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. of ICSE'99. (1999) 411–420
7. Atlee, J.: Automated analysis of specifications. `http://se.uwaterloo.ca/~jmatlee/746/` (2000) Lectures in University of Waterloo, School of Computer Science.
8. Chechik, M.: Automated verification. `http://www.cs.toronto.edu/~chechik/courses07/csc2108/index.html` (2007) Lectures in University of Toronto, Department of Computer Science.
9. Magee, J., Kramer, J.: Concurrency: State models & java programs slides. `http://www.doc.ic.ac.uk/~jnm/book/slides.html` (2006) A set of lecture notes for Chapters 1-10 of the book.
10. Mitchell, J.: Security protocols. `http://www.stanford.edu/class/cs259/index.html` (2006) Lectures in Stanford University, Department of Computer Science.

Model Checking Education for Software Engineers in Japan

Hideaki Nishihara[1], Koichi Shinozaki[2], Koji Hayamizu[3], Toshiaki Aoki[4], Kenji Taguchi[5], Fumihiro Kumeno[5,6]

[1] Research Center for Verification and Semantics
National Institute of Advanced Industrial Science and Technology(AIST)
[2] The Kansai Electric Power Co., Inc.
[3] Melco Power Systems Co., Ltd.
[4] Research Center for Trustworthy e-Society
Japan Advanced Institute of Science and Technology
[5] Grace Center, Information Systems Architecture Research Division,
National Institute of Informatics
[6] Mitsubishi Research Institute, Inc.

Abstract. This paper is the preliminary report of a joint research project on advocacy of a Body of Knowledge on Model Checking being carried out by six organizations which deliver model checking courses to software engineers in Japan. In this paper we will explain the main objective of the project and report the evaluation results of our model checking programs.

1 Introduction

Model checking has been used as a verification methodology for hardware and software systems and taught in computer science and software engineering curricula mostly at higher education. On the other hand, our organizations, National Institute of Advanced Industrial Science and Technology (AIST), Melco Power Systems Co., Inc. (MPS), Japan Advanced Institute of Science and Technology (JAIST) and National Institute of Informatics (TOPSE) have been delivering various courses on model checking to software engineers over the years. Our programs have very strong focus on practical aspects on model checking, which make ours different from those in academia.

Formal methods have been recognized as a rigorous software development methodology for safety critical systems and it is now recommended to be used in safety and security areas such as functional safety (IEC 61508) [2] and security assurance (ISO/IEC 15408) [1]. Particularly model checking is attracting software engineers as a rigorous verification methodology for system development and we have been meeting their demands by providing education courses, consultancy works and system developments using this technology.

We started our joint research project to standardize courses on model checking and soon realized that model checking education was still not mature yet due to lack of curriculum guidance based on a description of knowledge, which covers the whole area of model checking. SWEBOK standardized by IEEE CS and ACM for software engineering education [3] is too broad and model checking is only referred in Validation and Verification so that it is largely insufficient for our purpose. This observation motivated us to work on the Body of Knowledge on Model Checking (MCBOK). A precursor of this kind of BOK in Formal methods could be found in a work by Oliveira [8]. He presented a survey on the undergraduate curricula on formal methods as a part of FME-SoE (Formal Methods Europe, subgroup on Education). The paper shows a wide variety of formal methods courses in Europe, but does not present a well structured body of knowledge on formal methods. It is unfortunate that there has not been any follow-up activity from this group since then, even the demand for education of formal methods is becoming more important than it has ever been.

In this paper, we will report the first result of our joint project, i. e., evaluation of each courses based on student feedback. The paper is organized as follows. The next section briefly explains each course delivered by AIST, MPS, JAIST and TOPSE. Section 3 presents evaluation results based on questionnaires taken by each course, and in Section 4, we will conclude the paper.

2 Model Checking Courses

This section briefly explains four courses on model checking offered by each organization. Although these organizations deliver their courses independently, several commonalities can be found in them.

– There are many common topics (summarized in Table 1). In particular fundamental theories are dealt with in each course.
– Each course aims at software engineers, who have experience in software development.
– A special emphasis is placed on practical skills and knowledge. Each course mainly teaches laboratory works, tool support, and good practices.

2.1 AIST

CVS/AIST (Research Center for Verification and Semantics, National Institute of Advanced Industrial Science and Technology) was founded to promote Formal Methods as standard verification methods, and CVS/AIST has had experiences in applying model checking in industries. As a direct way to support the

Table 1. Courses Summary

Common topics	Different topics
Tools	AIST:
· Spin	· Differences between LTL and CTL
· SMV	· Composition and Shared variables
Concurrency	· Kripke structure transformation
Logics	· Empty word problem
· LTL	· Bounded model checking
· CTL	NII:
Property	· Timed automaton
· Safety	· LTSA
· Liveness	MPS
· Fairness	· Verification of practical systems
Abstraction	
· Abstraction map	
· Preservation theorem	
· Data abstraction	
· Predicate abstraction	
Verification process	
Theory	
· Finite/Buchi automata	
· Labeling(CTL ModelChecking)	

activity, we have been developing a model checking training course series for engineers for technology transfer and promotion.

Courses in the series aim to teach principles and experiences of model checking. Many parts of courses are based on examples, and some of them are sanitized materials taken from our experiences in industries. In the courses tool-dependent descriptions and knowledge are avoided, thus every example in the courses is checked by more than one tool (precisely they are NuSMV [7] and SPIN [4]).

The series consists of the following three courses: the elementary course, the intermediate course, and the advanced course. The elementary course gives an overview of model checking and a skill to exercise model checking procedures. The intermediate course gives some typical techniques in model checking: behaviours in some kinds of products of models and abstraction of models. The advanced course deals with some search algorithms to give knowledge about efficient verifications.

These courses take three or four days to teach appropriate materials to students at each level.

2.2 JAIST

Recently, keywords 'formal methods and model checking' are attracting engineering in Japanese industry. However the details of those technologies are not known well, and the keywords alone are spreading in the industry. JAIST recognizes that providing such information about advanced technologies to them is one of its mission. This is our motivation to hold seminars. Accordingly, the objective of the seminar is that its participants become to identify whether those technologies are useful in their fields or not.

Though there are many model checking tools, we focus on one of them, SPIN model checker [4], in the seminar. This is because concurrent processes of SPIN are similar to multi-tasks of RTOS(Real-Time Operating Systems) which is used in embedded software. Moreover, the syntax of Promela, which is the input language of SPIN is similar to that of imperative languages like C. We think that those facts make it easier to learn model checking technologies for engineers in the industry.

Target participants of our seminar are engineers in the field of embedded systems. We think that it is important to show successful examples using technical terms appearing in embedded system developments. Thus, we show examples from the fields of system programming and protocol, for instance, mutual exclusion, scheduling and alternating bit protocol. These ones are already known examples where model checking effectively works. The number of the examples used is about 90, and their total lines of code is about 4,000.

2.3 TOPSE

The Top SE program is a non-accredited course at Masters level fully funded by the government and is operated through a close collaboration between industry and academia at the National Institute of Informatics. The overview and the curriculum design of the whole program are discussed in [5]. We deliver the following five courses on model checking: *Model Checking Foundations*, *Model Checking Applications*, *Real-time Model checking*, *Software Model checking*, and *Modelling and verification of concurrent models*. These courses focus on efficient use and proper application of model checking tools that use automata theory, specifically SPIN [4], SMV [7], LTSA [6] and FDR. The key learning objectives of the Foundations and Applications courses are to learn how to detect and correct faults in design specifications in UML state machine models. More detailed explanation was presented in [5].

2.4 MPS

MPS runs an in-house curriculum which aims to teach practical techniques of model checking. We especially focus on the practical side of the technology, and thus we put a higher priority on the practical techniques than on the theory in the curriculum. The objective of the curriculum is to give such a skill to engineers so that they can apply model checking to actual software development immediately.

The curriculum consists of the following three courses: the basic course, the application course, and the practice course. In the basic course, temporal logic and state transition systems are explained with lectures and exercises. The application course and the practice course are mainly composed of laboratory works in which verifications of flow charts and source codes are taught. Especially, in the practice course, the course materials are taken from softwares developed by the course participants themselves. We use NuSMV [7] in our curriculum. In addition, in the application course and the practice course, we use a GUI tool for NuSMV "Support Software for Model Checking" which was jointly developed by The Kansai Electric Power Co. and MPS. The aim of the tool is to make the hurdle of applying model checking lower as much as possible when it is applied to actual software development, by reducing burdens in using the bare tool.

3 Evaluation and Observation of Questionnaires

In this section, we will summarize the results of questionnaires carried out by four organizations in terms of understandability, usefulness and feasibility. Detailed results are shown in Appendix, and Table 2 shows their summary.

Questionnaires were taken before we started working on this joint project so that we used different questionnaire forms. However some questions are shared and fundamental by them. We pick from practical issues: the degree of understanding of the tools and techniques taught, the usefulness of the tools and techniques to students' own problems, and feasibility of the techniques to be used in the industrial context. In summarizing the results of questionnaires we recognize necessities for standardizing questionnaires as well as the curriculum, and thus we notice that we will make a common questionnaire form.

Table 2 shows the result of their shared and fundamental questions. Each question has some choices which represent the degree of goodness and badness, and the number of the *positive* answers among them are only shown, that is, sums of numbers presented in the boldface in Table 3. The understandability, usefulness and feasibility stand for the number of the participants who could understand the contents of each course, the participants who feel that the model

Table 2. Questionnaire Results

Courses	Students	Understandability	Usefulness	Feasibility
AIST (elementary)	75	59(79%)	70 (93%)	N/A
AIST (intermediate)	24	7 (29%)	24 (100%)	18 (75%)
TOPSE (foundations)	36	22 (61%)	N/A	26 (72%)
TOPSE (applications)	7	5 (71%)	N/A	5 (71%)
JAIST	61	58 (95%)	54 (89%)	43 (70%)
MPS (in-house training)	13	10 (77%)	N/A	10 (77%)
Total	216	161/216 (74%)	148/160 (93%)	102/141 (72%)

checking is useful in their fields, and the participants who feel that the model checking can be practically feasible in their fields respectively.

On understandability, most of the courses mark high scores. the AIST intermediate course takes lower points since the contents were rather theoretical compared with other courses. In addition, it must be noted that the results obtained in the TOPSE are somewhat different from other courses due to the fact that participants of the program come to learn not only model checking courses but also some other courses such as software architecture and requirements engineering. Even taking these backgrounds of each course into account, we can observe that model checking is not so difficult for engineers to learn. The scores of the usefulness and the feasibility have also high similarly to the understandability. These results show that the Japanese industry has a potential to appreciate model checking, and that model checking is accepted as a practical method in Japanese industry.

We would like to quote some notable comments from the free description of the questionnaires. Many of the participants were surprised at the analysis power of the model checking tools as they know that concurrent and non-deterministic behaviour of the systems is very hard to analyze by hand. On the other hand, some participants were curious about how the model checking tools are integrated into their own system development processes. Unfortunately, we do not have suitable answers for this comment right now because the software processes to apply the model checking tools are not established yet. One way to meet this request is that we show successful examples to them. Making such examples for the education is one of our future works.

These results imply that teaching the model checking technologies to the engineers are fruitful. They were convinced of the relevance to learn the usage and application of the tools. On the other hand, they did not feel that to learn the theory of the model checking even though the theory exists behind the tools.

They might be just interested in how the model checking tools are taken into their daily works. However, we still believe that teaching the tools to the engineers is important. Though, right now, their interest is the usage of the tools, the interest will be extended to advanced topics like the theory and principle around the tools. The model checking has the limitations in some senses such as state explosion problems and descriptive power. Those limitations would be the motivation to learn technologies which are complement with the model checking and more advanced issues.

In summary, the overall score of feedback from the participants are very positive. We can conclude that our courses are well accepted by our participants. We hope that both the industries and academia tackle practical problems not only by taking engineering practices but also scientific approaches into their education programs.

4 Future Direction

As described the previous section, our past activities show participants' interests and their understanding for model checking. Our courses explained in Section 2 have no or a few vacant seats every time, and indeed we have over 200 respondents of questionnaires. We can consider that model checking is a noteworthy technology in Japan. It had not been thought that applying model checking to actual software developments was realistic, but questionnaire results show us it is not true now. There are many participants who comment that they want to apply model checking to actual software development with enough knowledges. We can see from the fact that many people in Japanese industry are understanding applicability of model checking and want to introduce it to software development processes.

On the other hand, appropriately considered education programs are necessary to learn model checking. Such programs should include theoretical backgrounds, since model checking is based on mathematics and logics. Moreover they should include how to use tools efficiently and successful examples in which model checking is applied to software development processes, since Japanese industry is sure to want them. But now every educational organization develops curricula and materials in its own purpose. A sufficiently adaptable and tidy curriculum is needed that considers both of industry's wants about usage and applications, and theories dealt with at appropriate levels.

In the state described above, in order to let model checking spread in industry, it is necessary to prepare an education program for software engineers, working as a reference. As a collaboration with industry and academia, we plan the following projects for the program:

1. making an MCBOK,
2. making a reference curriculum of model checking, and

in this financial year, and further

– improving our MCBOK and curriculum by reviews in public, and
– making a system to certifying the skill

in future.

Good curricula should be based on wide knowledge and practices in the subject, including theories, applications, and successful examples. These knowledge have not been summarized yet, and thus we will make a model checking body of knowledge (MCBOK) at first. The next is to make a curriculum based on the MCBOK. With MCBOK, one can construct various appropriate curricular on model checking and teach it as to various needs and requirements in industry. Our curriculum will be for software engineers, and it will be a reference. It will make possible to compare several curricula (including our own ones) strictly, by mapping them to the reference curriculum. It will also make it possible to cooperate in educations or training in model checking among educational organizations. The MCBOK and the reference curriculum will make a skill in model checking clear, and moreover engineers having a skill in model checking will be certified by them.This certification will encourage software engineers to learn about model checking, and will contribute to standardization of model checking for industry.

5 Acknowledgments

The Top SE program is fully supported by the Special Coordination Funds for Promoting Science and Technology, Fostering talent in emerging research field by Ministry of Education, Culture, Sports, Science and Technology, Japan.

We thank Hiroki Takamura and Satoru Yoshida for valuable comments.

References

1. ISO/IEC 15408. *Information technology - Security techniques - Evaluation criteria for IT security - Part1, Part2 and Part3.* ISO/IEC, 2005.
2. IEC 61508. *Functional safety of electrical/electronic/programmable electronic safety-related systems.* Bureau Central de la Commission Electrotechnique International, 2000.
3. A. Abran, J. W. Moore, P. Bourque, and R. Dupuis. *Guide to the Software Engineering Body of Knowledge 2004 Version SWEBOK.* IEEE, 2004.
4. G. J. Holzmann. *The SPIN model checker: Primer and reference manual.* Addison Wesley, 2004.

5. Shinichi Honiden, Yasuyuki Tahara, Nobukazu Yoshioka, Kenji Taguchi, and Hironori Washizaki. Top se: Educating superarchitects who can apply software engineering tools to practical development in japan. In *ICSE*, pages 708–718. IEEE Computer Society, 2007.
6. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs, Second Edition*. John Wiley & Sons, 2006.
7. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
8. José Nuno Oliveira. A survey of formal methods courses in european higher education. In C. Neville Dean and Raymond T. Boute, editors, *Teaching Formal Methods*, volume 3294 of *Lecture Notes in Computer Science*, pages 235–248. Springer, 2004.

A Questionnaire in detail

In this section we show detailed informations of our questionnaires(they are summarized in Section 3): concrete questions, concrete choices, results, and the ratios of *positive* answers.

In Table 3, AIST(E), AIST(I), TOPSE(F), and TOPSE(A) mean the courses AIST(elementary), AIST(intermediate), TOPSE(fundamental), and TOPSE(application) respectively. The column marked by (A) means the number of answers, and every datum which we regard as a positive answer is presented in the boldface. The column marked by (B) means the ratio of the positive answers to the population. Notice that the numbers of vacant answers are not shown.

Table 3. Detailed Questionnaire Results

Understandability				
	Question	Choices	(A)	(B)
AIST(E)	How was the course level?	too easy/easy	**4**	
		appropriate	**55**	
		difficult/too difficult	16	59/75
AIST(I)	How was the course level?	too easy/easy/bit easy	**7**	
		bit difficult/difficult/too difficult	16	7/24
TOPSE(F)	Could you catch all the contents?	yes/mostly yes	**22**	
		neither yes nor no	7	
		mostly no/no	6	22/36
TOPSE(A)	Could you catch all the contents?	yes/mostly yes	**5**	
		neither yes nor no	2	
		mostly no/no	0	5/7
JAIST	How much could you understand?	very well/mostly all	**58**	
		could follow	1	
		little/couldn't understand	2	58/61
MPS	How much do you understand?	very well/mostly all	**10**	
		average	3	
		little/couldn't understand	0	10/13
Usefulness				
AIST(E)	How do you feel about the course?	very useful/useful	**70**	
		not useful	0	70/75
AIST(I)	How do you feel about the course?	very useful/useful/bit useful	**24**	
		bit useless/useless/very useless	0	24/24
TOPSE(F)	(N/A)		-	-
TOPSE(A)	(N/A)		-	-
JAIST	How do you feel about the course?	very useful/useful	**54**	
		average	2	
		useless/very useless	5	54/61
MPS	(N/A)		-	-
Feasibility				
AIST(E)	(N/A)		-	-
AIST(I)	Do you consider applying MC to your job?	yes/maybe	**18**	
		maybe not/no	1	18/24
TOPSE(F)	Can you apply the course to your job?	totally yes/mostly yes	**5**	
		partially yes	**21**	
		partially no/totally no	4	26/36
TOPSE(A)	Can you apply the course to your job?	totally yes/mostly yes	**1**	
		partially yes	**4**	
		partially no/totally no	2	5/7
JAIST	How do you feel about feasibility?	directly feasible/indirectly feasible	**43**	
		feasible in future	6	
		less feasible/not feasible	12	43/61
MPS	How do you feel about feasibility?	very feasible/maybe feasible	**10**	
		not feasible, but significant personally	2	
		less feasible/not feasible	1	10/13

A simple refinement-based method for constructing algorithms

Dominique Méry

LORIA Nancy Université
mery@loria.fr

A Introduction

Overview. Event B is supported by the RODIN platform and provides a framework for teaching programming methodology based on the famous pre/post-specifications, together with the refinement. We illustrate a methodology based on Event B and the refinement by developing algorithms like for instance computing the shortest distances of a graph, sorting an array by insertion, . . . Floyd's algorithm is redeveloped and we add comments on the complexity of proofs and on the discovery of invariant; it should be considered as an illustration of a technique introduced in a joint paper with D. Cansell[8]. The development is based on a paradigm identifying a non-deterministic event with a procedure call and by introducing control states. We discuss points related to our lectures at different levels of the university, mainly master. It is also a way to introduce a pattern used for developing sequential structured programs. The complete development of Floyd's algorithm can be found in the document [14] and we will illustrate our paper with comments of students. The paper will intend to focus on both Formal Method, namely Event B, and, Education and Training, namely Master Degree Lectures.

Progamming methodology. The development of structured programs is carried out either using bottom-up techniques, or top-down techniques; we show how refinement and proof can be used to help in the top-down development of structured imperative programs. When a problem is stated, the incremental proof-based methodology of event B[7] starts by stating a very abstract model and further refinements lead to finer-grain event-based models which are used to derive an algorithm[3]. The main idea is to consider each *procedure call* as an *abstract event* of a model corresponding to the development of the *procedure*; generally, a procedure is specified by a pre/post specification and then the refinement process leads to a set of events, which are finally combined to obtain the *body of the procedure*. The refinement process can be considered as an *unfolding* of *calls* statements under preservation of invariants. It means that the abstraction corresponds to the procedure call and the body is derived using the refinement process. The refinement process may also use recursive procedures and supports the top-down refinement. The procedure call simulates the abstract event and the refinement guarantees the correctness of the resulting algorithm. A preliminary version[8] introduces ideas on a case study and provides an extended partial abstract of the current paper.

Proof-based Development. Proof-based development methods[5, 1, 13] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete events. The relationship between two successive models in this sequence is that of *refinement*[5, 1]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination. A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine[4]. At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations. The goal of a event B development is to obtain a *proved model* and to implement the correctness-by-construction[12] paradigm. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity.

Education and training. The best way to convince your students that formal methods are something good for you, and then for them, is to apply the medicine to yourself. However, I will be provocative and base my teaching method on the famous quastion of the donkey who is not thirsty and the problem is to convince him to drink. According to the classical leitmotiv, we do not give water to a donkey who is not thirsty. Now, you understand that students may be consider as donkeys and you should warn that donkeys are very strong and very helpful. If a farmer wants to give water to a donkey, and if the donkey does not want to drink, the best way is to find another donkey who is drinking water, because he enjoys drinking water. The drinking donkey will make the environment full of happiness and the non-drinking donkey will finally drink, because he understands that his neighbour is happy and mer(r)y. Now, back to the paper, and we apply the principle to teaching formal methods. The drinking donkey is the author of the paper and the not-drinking donkey is a student. The water is using a formal method Event B based on sets theory and generalized substitution. The main question is to equip workstations with the software supporting Event B, namely RODIN and it is quit easy. First, after some clicks on the web, students can get the RODIN platform. Second, I give lectures with a video projector and students can check themselves that what I am saying and doing is also doable by them.

The teacher should be able to illustrate concepts using the tool that students will use. The use of set theory is very convenient, since it provides another way to model data. Students should model and not only program. The challenge is clearly to teach them how abstract should be a model and to have realistic case studies. If you use a theorem prover, you should be a demonstration of the feasibility of this approach and not only preaching that theorem provers are useful. Using Event B, two concepts emerge during the first lecture: abstraction and refinement. The most difficult question is to state when a model is a good abstraction and what is a model. Now, we have to consider case studies that appear to be tractable and which are sufficiently complex to require a formal development.

We consider case studies which are mainly sequential algorithms, when completely developed. We have improved the expression of the methodology introduced by Cansell and Méry [8] by formalizing the different step in a global schema or pattern [14]. We introduce the general view of the pattern as it was expressed in [14].

B The modelling framework

This paper is based on concepts of the Event B modelling language developed by J.-R. Abrial[2, 7]; it us the purpose of thos paper to outline the general methodology we are applying. The ingredients for describing the modelling process based on events and model refinement can be found in [2, 7]. We assume that the goal is to solve a given problem described by a semi-formal mathematical text and we assume that the problem is defined by a precondition and a postcondition[13]. The modelling process starts by identifying the domain of the problem and it is expressed using the concept of **CONTEXT**. A **CONTEXT** PB (see Figure 1) states the theoretical notions required to be able to express the problem statement in a formal way. The **CONTEXT** PB declares

– a domain D which is the global set of possible values of the current system.
– a list of constants x, which is specifying the input of the system under development, P, which is the set of values for x defining the precondition, and Q, which is a binary relation over D defining the postcondition of the problem.
– a list of axioms assigns types to constants and adds knowledges to the RODIN environment; for instance, the axiom 5 states that there is always a solution y, when the input value x satisfies the precondition P.

☹ *It appears that the set-theoretical notation 1 is well known by french students; they have used notations like sets, relations, bijections ... The main problem is that they are often willing to program rather than to model. The first exercises helps them to learn the notation and the link between the keyboard and the screen in the RODIN platform. The ECLIPSE-like interface is not a problem for the student who can get the software through the web. Moreover, we maintain a MOODLE chapter for lectures and students can find lectures notes, exercises and archives containing Event B models. On the mathematical and logical concepts, students have lectures on computability and logical theories. They have already heard of axioms and theorems; moreover, the sequent calculus [10] is taught by colleagues and the $\vdash$ symbol appears to be friendly. They have not derived many proofs and RODIN provides a very nice interface for deriving theorems from axioms. The cartsian product of two sets is denoted $A \times B$ and a member of the set is denoted $x \mapsto y$ with $x \in A$ and $y \in B$; ussually, we use the notation (x, y) instead of $x \mapsto y$. This point is not a real problem for computer scientists, since they used to learn new languages very often. The cartsian **CONTEXT** PB is a simple set-theoretical reformulation of the pre-and-post-specification.* ☺

A **CONTEXT** may include a clause THEOREMS containing properties derivable in the theory defined by sets, contants and axioms; theorems are discharged using the proof assistant of the tool RODIN. The underlying language is a (typed) set-theoretical language partially given in Table 1. When an expression E is given, a well-definedness condition is generated by the tool; this point allows us to check that some side conditions are true. For instance, the expression $f(x)$ generates a condition as $x \in dom(f)$.

☹ *The well-definedness condition is a crucial issue in computer science. This pôint provides a way to understand the termination of programs. It is also the opportunity to separate the generous programming from the defensive programming. We can also refered to the programming-by-contract methodology*

CONTEXT PB

SETS
D

CONSTANTS
x, P, Q

AXIOMS
$axm1 : x \in D$ $/*$ x belongs to a general set of the problem domain $*/$
$axm2 : P \subseteq D$ $/*$ P is a set defining the precondition $*/$
$axm3 : Q \subseteq D \times D$ $/*$ Q is a binary relation over S defining the postcondition $*/$
$axm4 : x \in P$ $/*$ x is supposed to satisfy the precondition P $*/$
$axm5 : \forall a \cdot a \in P \Rightarrow (\exists b \cdot a \mapsto b \in Q)$ $/*$ there is at least one solution for each data x satisfying the precondition P $*/$

END

Fig. 1. Context for modelling the problem PB

*and have a real instantiation of the methodology. The **CONTEXT** is used to provide a general expression of the pre/post-specification and it is also very usefull for teaching the link between the expressiveness of a language with respect to a set-theoretical language. Skolem functions are not mentionned but in research master students, I am refering to this concept.* ☺

The first model provides the declaration of the procedure call. Variables y are *call-by-reference* parameters, constants x are *call-by-value* parameters and carrier sets s are used to type informations and also for defining a generic procedure:

procedure call(**x; var** y)
precondition $y = y_0 \wedge Init(y_0, x, D) \wedge \tilde{P}(x)$
postcondition $\tilde{Q}(x, y)$

☹ *The main question is to link programming concepts to logical concepts; the approach is simply a formulation of the translation and the call is considered as an event. In fact, it is the instance of the call which is an event. The expression of the contract is given in an algorithmic language. We state that the contract can be expressed either in Spec♯ or in JML. Students get the feeling that the contract is not only a simple statement but we should check conditions called proof obligations. Static checking and dynamic checking are now expressed.*

Name	Syntax	Definition
Binary relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Composition of relations	$r_1; r_2$	$\{x, y \mid x \in a \;\wedge\; y \in b \;\wedge$
		$\exists z.(z \in c \;\wedge\; x, z \in r_1 \;\wedge\; z, y \in r_2)\}$
Inverse relation	r^{-1}	$\{x, y \mid x \in \mathcal{P}(a) \;\wedge\; y \in \mathcal{P}(b) \;\wedge\; y, x \in r\}$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \;\wedge\; \exists b.(b \in t \;\wedge\; a \mapsto b \in r)\}$
Range	$\mathsf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Identity	$\mathsf{id}(s)$	$\{x, y \mid x \in s \;\wedge\; y \in s \;\wedge\; x = y\}$
Restriction	$s \lhd r$	$\mathsf{id}(s); r$
Co-restriction	$r \rhd s$	$r;\ \mathsf{id}(s)$
Anti-restriction	$s \mathbin{\lhd\mkern-14mu-} r$	$(\mathsf{dom}(r) - s) \lhd r$
Anti-co-restriction	$r \mathbin{\rhd\mkern-14mu-} s$	$r \rhd (\mathsf{ran}(r) - s)$
Image	$r[w]$	$\mathsf{ran}(w \lhd r)$
Overriding	$q \mathbin{\lhd\mkern-9mu-} r$	$(\mathsf{dom}(r) \mathbin{\lhd\mkern-14mu-} q) \cup r$
Partial Function	$s \mathrel{+\mkern-14mu\to} t$	$\{r \mid r \in s \leftrightarrow t \;\wedge\; (r^{-1}; r) \subseteq \mathsf{id}(t)\}$

Table 1. Set-theoretical notation for event B models

Parameter passing methods are chosen as simple as possible; we have given a way to separate pass-by-value and pass-by-reference. The role of constant in Event B is clearly played by pass-by-value parameters. ☺

Figure 2 describes the complete model for the problem PB; it is expressd by a generic procedure stating the pre/post-specification. The term *procedure* can be substituted by the term *method*. The current status of the development can be represented as follow:

$$\begin{tikzcd}
\texttt{call(x,y)} \arrow[r, "call-as-event"] & \text{PREPOST} \arrow[r, "\text{SEES}"] & \text{PB}
\end{tikzcd}$$

The statement of a given problem in the Event B modelling language is relatively direct, as long as we are able to express the mathematical underlying theory using the mechanism of contexts. The existence of a solution y for each value x is assumed to be an axiom; however, it would be better to derive the property as a theorem and it means that we should develop a way to validate axioms to ensure the consistency of the underlying theory.

☹ *The translation from a programming/algorithmic language into Event B expressions is not a very difficult task. However, the expression $x : |(P(x, x')$ is very powerfull and we can produce events corresponding to instances in a mecahnical way. We separate sets from assertions using the $\overset{\sim}{.}$ operation. It is also an opportunity to rember computability results and to ask questions on the possibility to provide a program from the specification. The generalized substitution helps us to warn students on the frame problem.* ☺

HOARE logic[11] provides a very interesting framework for dealing with specifications an development and our work shows how the ingredients of HOARE logic can be used to provide a general framework for developing sequential programs correct by construction. Event B and the RODIN plateform can be used to teach basic notions like pre and postconditions, invariant, verification and finally design-by-contract.

☹ *At this point, we can recall the essence of the HOARE logic, which can be seen as a design logic for deriving programs. The design-by-contract approach is clearly supported by our framework. Students have got some lectures on semantics and logics of programs; however, they have not really related the algorithmics lectures to the semantics lectures. Moreover, concepts are becoming useful for understanding our approach.* ☺

```
MACHINE PREPOST

SEES PB

VARIABLES
    $y$

INVARIANTS
    $inv1 : y \in D$

EVENTS

INITIALISATION
    BEGIN
        $act1 : y :\in D$
    END

EVENT call
    BEGIN
        $act1 : y : |(x \in P \wedge x \mapsto y' \in Q)$
    END

END
```

Fig. 2. Machine defining the model for modelling the problem PB

The call-as-event pattern is then derived and we give some short explanations on its structure and components. The call instance `call(x,y)` is considered as an event which is then refined and the next diagram gives the global process of the methodology. If an event is deterministic, it is translated into a code and, if it is non-deterministic, it is considered as a call of another procedure. We should start a new development following the same schema. The SEES relation provides a logical and mathematical framework for the definition of the event modelling the call. The REFINEMENT relation is inherited from Event B and the *mapping* operator is simply a reformulation of the REFINEMENT model, which contains a control variable. The control variable provides a way to derive an algorithmic version of the REFINEMENT model. The diagram is divided into two parts:

– the left part is the world of programming with pre/post-specification
– the right part is the world of modelling in Event B

$$\begin{tikzcd}
\texttt{call(x,y)} \arrow[r, "call-as-event"] \arrow[d, "call"'] & \mathrm{PREPOST} \arrow[r, "\mathrm{SEES}"] \arrow[d, "\texttt{REFINEMENT}"'] & \mathrm{PB} \\
\texttt{call} & \mathrm{MODEL} \arrow[l, "mapping"'] \arrow[ur, "\mathrm{SEES}"]
\end{tikzcd}$$

☹ *The diagram was introduced on a simple case study, namely the computation of binomial coefficients. It is a dynamic programming problem and it is easy to explain how the function is defined using Pascal's triangle. You should remember that we want to give the good water to donkeys, who do not know the fantastic quality of the water. Hence, a picture is a very good way to communicate. Main difficulties are in the definition of the* PBCONTEXT *context and the discovery of a computation principle translated into the* REFINEMENT *model. The call-as-event operator and the mapping operator are defined in a systematic way. We are developing tools for implementing both operators.* ☺

On the next section, we will illustrate the application, of the pattern to several examples of algorithmics.

C Constructing algorithms with the *call as event* pattern

☹ *The pattern is explained to students who have got 8 hours lectures on Event B. The problem is to show that we can solve problems which are complex. The classical sorting problem or the classical graph-theoretical problems are good candidates. However, we should recall that we prove every model and we should reach a totally proved model using the RODIN platform. We have developed three algorithms using the pattern and we describe the technical details. We were surprised that students were discovering a bug in one model: we were using an old version and we did not correct it.* ☺

C.1 Example of the binomial coefficients

The computation of binomial coefficients is based on Pascal's triangle and we define it as a partial function c. Data n and k are defined in the context called $BN0$. The call $bn(n, k, vcoef)$ is translated as an event which is simply setting $vcoef$ to $c(n \mapsto k)$. We have got three elements of the diagrams.

☹ *Students can be driven carefully from the call statement to the expression of the function c. Pascal's triangle is very usefull and provides a graphical guide for writing d's definition into $BN0$. We have introduced another graphical structure for supporting the case analysis related to the values of n and k and the introduction of control flow.* ☺

The refinement $BN2$ produces a collection of events analysing the different steps of the computations required for computing the value of $c(n \mapsto k)$.

☹ *We tell to students that the world of mathematics is defining a value $c(n \mapsto k)$ and the world of computing will derive a process using c and its definition for producing the same value. It is also the time to talk on questions of run-time errors, overflow, …* ☺

$$\begin{tikzcd}
\texttt{bn(n,k,vcoef)} \arrow[r, "call{-}as{-}event"] \arrow[d, "call"'] & \mathrm{BN1} \arrow[r, "\mathrm{SEES}"] \arrow[d, "\texttt{REFINEMENT}"'] & \mathrm{BN0} \\
\texttt{bn} & \mathrm{BN2} \arrow[l, "mapping"'] \arrow[ur, "\mathrm{SEES}"] &
\end{tikzcd}$$

The refinement introduces three cases for the call instances. Either k is 0, or k is n, or is neither 0, nor n. Let us consider the difficult case: $k \neq 0$ and $k \neq n$:

$$\forall k \in \{1, \dots n-1\}. \binom{k}{n} = \binom{k-1}{n-1} + \binom{k-1}{n-1} \tag{1}$$

Hence, the main idea is to use the same event for computing $\binom{k-1}{n-1}$ and $\binom{k-1}{n-1}$. These events are trasnlated into a recursive call by the mapping; the final computing event is computing the value of the sum of the two values. Control states are simply organizing the computations in a sound order. The invariant gives the different steps and the different intermediate values namely $vtcoefy$ and $vtcoefx$:

INVARIANTS

$inv1: \ l \in LOC$
$inv3: \ vtcoefx \in \mathbb{N}$
$inv4: \ vtcoefy \in \mathbb{N}$
$inv5: \ l \in \{callx, cally, endcalling\} \Rightarrow k \neq 0 \land n \neq 0 \land k < n$
$inv6: \ l = cally \Rightarrow vtcoefx = c(n-1 \mapsto k-1)$
$inv7: \ l = endcalling \Rightarrow vtcoefy = c(n-1 \mapsto k) \land vtcoefx = c(n-1 \mapsto k-1)$
$inv8: \ l = end \Rightarrow vcoef = c(n \mapsto k)$

LOC contains the control states which are containing two values at least $start$ and end. The refinement produces 42 proof obligations and 2 were manual. The other proof obligations are automatically discharged.

☹ *The example is simple and the function c is easy to define. The invariant is built by analysing the expression of the computed value.* ☺

C.2 Example of the insertion sort

Sorting by insertion is developed using several times the pattern and it is based on the following principle. The problem is to sort an array t between 1 and m, where $dom(t) = 1..n$ and $m \leq n$. The sorting can be done by sorting the array from 1 to $m-1$ and then to insert the value $t(m)$ at the right position in $1..m$.

$$\begin{tikzcd}
\texttt{insert-sort(m,t,st)} \arrow[r, "call-as-event"] \arrow[d, "call"'] & \text{IS1} \arrow[r, "\text{SEES}"] \arrow[d, "\texttt{REFINEMENT}"'] & \text{IS0} \\
\texttt{insert-sort} & \text{IS2} \arrow[l, "mapping"'] \arrow[ur, "\text{SEES}"] &
\end{tikzcd}$$

$IS0$ is the context defining the array t and the size of the array, namely n. $IS1$ is a model containing only one event, which is sorting the array between 1 and m in one shot:

sorting

any
  pi
where
  $grd1: \ pi \in 1..m \rightarrowtail 1..m$
  $grd2: \ \forall i \cdot i \in 1..m-1 \Rightarrow t(pi(i)) \leq t(pi(i+1))$
then
  $act1: \ st := t; pi$
end

☹ *Students understand the role of the event **sorting**, which is defining a pre/post-specification of a sorting algorithm. When we organize the lectures, we add tutorials and machine sessions for learning notations and learning to specify an algorithm. For instance, both modules $IS0$ and $IS1$ are reused for previous exercises.* ☺

Now, we express in the refinement machine $IS2$, the essence of the sorting by insertion. We introduce control points and analyse the problem by cases.

☹ *We draw the two cases on a blackboard and we add the steps for the effective insertion. During our lectures, we understand that charts, pictures or diagrams are very good for giving the relationship among control points. These diagrams are not yet formalised and we will bring this feature in the tool.* ☺

Either m is 1 when the control is at start and the next control state is end, or m is greater than 1 and we introduce a control point for calling other procedures. We use a variable st for storing the sorted array. To make easier the expression, we use the following predicates:

- $sorted(s)$ means that s os a sorted array.
- $permutation(a,b)$ means that the array a is equal to the array b upto a permutation of values.
- $a(i..j)$ is the subarray of t from i to j.
- $a.b$ is the concatenation of both arrays a and b.

$$\begin{tikzcd}
 & \begin{array}{l} c = start \\ \wedge m \in \mathbb{N} \\ \wedge t = st \end{array} \arrow[dl, "m\neq 1\ sorting\ call"'] \arrow[dr, "m=1"] & \\
c = start \wedge m \neq 1 \wedge t = st \arrow[d, "tosortingcall"'] & & \begin{array}{l} c = start \\ \wedge m = 1 \\ \wedge t = st \end{array} \arrow[ddd, "sortingm=1"] \\
c = sortingcall \wedge m \neq 1 \wedge t = st \arrow[d, "sortingcall"'] & & \\
\begin{array}{l} c = insertioncall \wedge m \neq 1 \wedge \\ \exists a. \left(\begin{array}{l} st = a.t(m) \\ \wedge permutation(s(1..m-1), a) \\ \wedge sorted(a) \end{array} \right) \end{array} \arrow[d, "insertioncall"'] & & \\
\begin{array}{l} c = end \wedge m \neq 1 \wedge \\ permutation(t, st) \wedge sorted(st) \end{array} \arrow[dr, "m\neq 1"] & & c = end \wedge m = 1 \wedge sorted(st(1..m)) \arrow[dl, "m=1"'] \\
 & \begin{array}{l} c = end \\ \wedge sorted(st(1..m)) \\ \wedge permutation(st, t) \end{array} &
\end{tikzcd}$$

The diagram gives the different events of the refinement model $IS2$; it contains an event called **sortingcall**, which is sorting the array t between the value 1 to $m-1$ nd the event **insertioncall** which is inserting the value $t(m)$ at the right position in the array sorted between 1 and $m-1$. This last event can not be translated into an algorithmic expression and should be considered as defining a new problem which is the insertion of a value in a sorted array. We re-apply the pattern by starting a new development for solving the insertion problem.

☹ *The reapplication of the pattern shows that the pattern is really central and is very usefull. Students can practise the top-down development without toil. We use a diagram for illustrating the insertion of* $t(m)$ *into the values of* $st(1..m-1)$. *It is an illustration of the reapplication of the pattern.* ☺

C.3 Example of Floyd's algorithm

The sp procedure can be derived from the list of events of the model SP2 and we structure events into conventional programming structures like `while` or `if` statements. J.-R. Abrial[3] has proposed several rules for producing algorithmic statements. The next diagram gives the complete description of the process we have followed:

$$\begin{tikzcd}
\texttt{sp(1,a,b,G,D,FD)} \arrow[r, "call-as-event"] \arrow[d, "call"'] & \mathsf{SP1} \arrow[r, "\text{SEES}"] \arrow[d, "\texttt{REFINEMENT}"'] & \mathsf{SP0} \\
\texttt{floyd} & \mathsf{SP2} \arrow[l, "mapping"'] \arrow[ur, "\text{SEES}"] &
\end{tikzcd}$$

We do not give more details and we refer to the paper [14] containing the development of Floyd's algorithm.

D Concluding Remarks and Perspectives

The main objective of the paper is to show how we can develop a sequential structured algorithm using a one-step refinement strategy. We have illustrated the technique introduced by Cansell and Méry in [8] and partially formalized by Méry [14]. The paper gives hints to use the technique for teaching *correct-by-construction* algorithmics using a tool which is clearly a very good mate for controling the development. You may have questions on the treatment of arithmetics. The technique of developmment is a top/down approach, which is clearly well known in earlier works of Dijkstra[9, 13], and to use the refinement for controlling the correctness of the resulting algorithm. It relies on a more fundamental question related to the notion of *problem to solve*. It is also an illustration of the application of the *call-as-event* pattern.

What we have learnt from the case study is summarized as follows:

1. Developing a first abstract one-shot model using pre/post-condition. It provides the declarations part of the procedure (method) related to the one-shot model. The basic structure to express is the function d which the key of the problem. Constants of the model are defined as *call-by-value* parameters and variable of the model are *call-by-reference* parameters,
2. Refining the abstract model to obtain the body of procedure. New variables are defined as *local variables*. The refinement introduces control states which provide a way to structure the body of the procedure. We have clearly the first control point namely *start* and the last control point namely *end*. The diagram helps to decompose the procedure into steps of the call and a special control point called *call* is introduced. The main question is to obtain a deterministic transition system in the new refinement model.
3. If there are still remaining non-deterministic events, we can eliminate the non-deterministic events by developing each non-deterministic event in a specific B development starting by the statement of a new problem expressed by the non-deterministic event itself.
4. Proof obligations are relatively easy to check because the invariant is written by a list of properties according to d. Even if the number of *manual* proof obligations is high, it was very easy to discharge them using the prover and to reuse former interactive ones.
5. The translation of Event B model into a C program was carried out by hand and we did a mistake. We forgot that C arrays are starting the index by 0 and it leads to a bad call. We should mechanize this step to avoid this mistake.

Now, if we have to teach concepts, it is easier to teach how to write concepts and definitions using notations provided by Event B. You will get a way to check definitions and the type checker is sometimes cruel, since it recalls that Event B is typed. We can discuss on many questions using this methodology: coding of numbers, preconditions, postconditions, invariant, assertions, proofs, ... and questions can lead to replies which are pertinent because of the proof tool.

Future works will provide more case studies and tools for supporting the link between models and codes. We aim to enrich the RODIN tools[15] by specific plug-ins managing libraries of models and implementing new proof obligations.

References

1. J.-R. Abrial. *The B book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

2. J.-R. Abrial. $B^{\#}$: Toward a synthesis between z and b. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer, June 2003.
3. J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME 2003*, pages 51–74, 2003.
4. J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOL 2003*, pages 1–24, 2003.
5. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.
6. Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages*. EATCS Textbook in Computer Science. Springer, 2007.
7. Dominique Cansell and Dominique Méry. *The event-B Modelling Method: Concepts and Case Studies*, pages 33–140. Springer, 2007. See [6].
8. Dominique Cansell and Dominique Méry. Proved-patterns-based development for structured programs. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *CSR*, volume 4649 of *Lecture Notes in Computer Science*, pages 104–114. Springer, 2007.
9. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
10. G. Gentzen. *Untersuchungen Uber das Logische Schliessen ou Recherches sur la dé duction loqique*. 1955. Traduction de Feys et Ladrière.
11. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
12. Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, October 2006.
13. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.
14. Dominique Méry. Teaching programming methodology using event b. In H. Habrias, editor, *The B Method: from Research to Teaching*, June 2008.
15. project RODIN. Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/, 2004. 2004–2007.

Formal Methods for Electronic Government

Jim Davies and Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
http://www.comlab.ox.ac.uk/people/{Jim.Davies, Jeremy.Gibbons}/

Abstract. Electronic government is a challenging domain for software engineering, with complex requirements involving agility, transparency, accuracy, and accessibility. The techniques of semantic frameworks—metadata-based, model-driven development—may help to address these challenges. Data semantics and model transformations are prime application areas for formal methods, and so electronic government is an exciting new domain for education and training in formal methods.

A Introduction

Increasing reliance upon electronic communication, together with the ambitions and demands of a global information society, means that electronic government is becoming the expected means of implementation for government policies, activities, and initiatives. Although considerable progress has been made, the reputation of public sector information technology remains poor. Most people can quote at least one high-profile disaster, in which a large electronic government project singularly failed to deliver.

The challenges in developing information technology for public sector applications are in principle no different from those encountered in other large, enterprise computing initiatives. They are, however, exacerbated by three main factors: the likelihood of conflict and misunderstanding between different stakeholder groups; the fact that requirements are linked to changes in policy and legislation; and the expectation that data and processes should be accessible, and also compatible with those in other initiatives.

We believe that a big step towards addressing these challenges can be made by integrating ideas from data semantics and model-driven development, an integration we call a semantic framework. Moreover, we claim that semantic frameworks both provide an interesting new domain for, and can derive great benefit from, work in formal methods. This paper sets out our position.

A.1 Electronic government

The term electronic government means more than a literal translation of existing government services and processes into electronic form: it carries expectations of transformation, often in connection with hopes for a better society. Issues such as access, transparency, change, democracy, and interaction, suggest that there may be specific domain challenges in electronic government, with significant implications for software design and development. In particular, electronic government requires a significant degree of formalisation and computerisation of semantics. The size of the community, the rate of evolution, and the importance of documentation make it essential that the semantics can be accessed, maintained, and incorporated into delivered systems without the need for extensive, error-prone manual intervention.

A.2 Challenges

The requirements of electronic government systems are more complex than those of their commercial counterparts; they are also more subject to change. Policy reforms or shifts in public opinion may require substantial changes to the design of a system, changes that may be expensive to make once implementation is under way. The government of a developed country may be able to afford such costs, but the government of a developing country cannot. In a commercial context, it is quite common to find that information system design is shaping business processes; in electronic government, this is less likely to be acceptable.

It is also more important that these requirements are correctly reflected in the behaviour of the system. In electronic government, computing systems will do more than facilitate policy—they will serve as its principal, and perhaps its only, implementation. This has significant implications not only for the criticality of development processes, but also for the design of the systems themselves.

In a commercial system, the information pertaining to an individual may define and constrain that individual as a customer; in a government system, it may define and constrain that individual as a citizen. The data may be driving the processes of government as they act upon the individual: there is a greater responsibility to maintain its correctness and availability over time. After all, most commercial organisations have competitors, and a dissatisfied customer may always change provider; that option is not nearly as straightforward when the provider is the customer's national government, with a monopoly on their relationship.

In electronic government, the stakeholders, including the end users, have a particular relationship to the processes of development and operation: this system is being procured, designed, developed, and operated on their behalf, and at their expense. We might consider there to be an implicit contract, reflected in the system requirements, similar to that which exists between government representatives and the people they represent. This means that the extent to which requirements are 'owned by the users' is far greater, and thus the system must be a better fit for the social processes that it is intended to support, than is often the case in ordinary enterprise computing. Furthermore, stakeholders may require more in the way of evidence that the system is in fact doing what is expected—the implicit contract applies in operation as well as in development.

B Formalisation

The large-scale sharing and integration of data from dynamic, heterogeneous sources requires computable representations of the semantics of data, and it is here that a significant part of the challenge lies. Natural language or informal understanding is sufficient for such a semantics only when the concepts are straightforward, the community is small or homogeneous, and the period of time over which understanding must be maintained is short. For complex problems, heterogeneous communities, or long-running initiatives—all characteristics of electronic government—a more formal approach is required. The semantics has to be amenable to automatic processing, and this processing has to be automatically linked to the processing of the data itself. This entails the faithful representation of data semantics in constructing models, and the application of model-driven development in generating system artifacts—queries, scripts, programs, services, forms, and interfaces—from these models.

B.1 Metadata-based

Robust, trustworthy, and transparent information systems require the careful consideration and representation of the semantics of the information they record; a structured, computable representation is essential if we wish to adopt and maintain rich terminologies across multiple initiatives. Conflicts and misunderstandings about the semantics of data can be resolved, or at least identified at an earlier stage, if aspects of structure, functionality, and interpretation are conveyed through the use of models. This is standard practice in software engineering; however, the audience for the model is usually quite restricted, and thus much of the detail, or semantic metadata, may be left implicit. For electronic government, it is a requirement that models may be validated, so that public servants can be held accountable; it is therefore more important that the models are comprehensive, and that metadata is properly recorded.

B.2 Model-driven

The dynamically evolving context of policy and legislation, the greater requirements for accountability and transparency, and the sheer scale of many electronic government initiatives, all encourage the automatic generation of a system implementation from more abstract models. Information systems have always been modelled, but often only informally, using fragments of specification, written in natural language, and presented as reports, spreadsheets, and diagrams. These are partial descriptions, often containing apparent contradictions, and there is no prospect of using these to generate a system automatically. Yet these are the documents that inform decisions such as those on whether to proceed, on project scope, on supplier selection, and on contract fulfilment, and it is here that a semantic framework can start to produce real benefit. Reports and spreadsheets in which key terms are annotated with a link to agreed terminology, and data elements are annotated with a link to detailed semantics in a metadata registry, can be concise and unambiguous, while making explicit a shared understanding of exactly what is required.

In development, more formal models—typically, object models and service descriptions—can present precise descriptions of structure and functionality in which data attributes have an accessible, computable semantics, and terms have an agreed meaning. It may then be possible to determine programmatically—at the design stage, or after deployment—whether two systems are holding data that has exactly the same semantics. This is an essential prerequisite to the systems integration required for 'joined-up government', in which central and local government, different departments and agencies, work together to tackle social problems.

One way to represent the semantic information required, and to facilitate programmatic access, is to represent the various aspects of semantics using models of usage. We can identify three particularly useful kinds of model: ontologies, models which explain the meaning of a metadata item in terms of named relationships to other elements; applications, models in which the item appears in context: for example, in the context of a design document, or a form template; and transformations, models which explain how data collected against one set of elements can be transformed to fit another. Although only the first of these is usually seen as defining or recording meaning, the others also have semantic import: meanings are sometimes best expressed, and will evolve, through usage.

B.3 Semantic frameworks

The ideas of metadata-based and model-driven development together make what we call a semantic framework. A practical semantic framework can be defined in terms of constructs at three different levels: terminology services, metadata registries, and model repositories. The first level presents a collection of defined terms, structured in a way that suits one or more possible applications. For example, a terminology for education might include terms such as 'institution' and ‘qualification’, record that the terms ‘university’ and ‘high school’ denote particular kinds of institution, and record also that the terms ‘master’s degree’ and ‘international baccalaureate’ are related in some way to the notion of institution.

The second level presents a collection of metadata elements, each of which describes a measurement or observation. A metadata registry for education might include elements such as institution attended, full title of degree awarded, and result obtained. Each element may be related to one or more terms in the underlying terminology, and additional semantic information is provided by informal explanations of intended purpose and an association with a domain of possible values. The registry also records relationships between elements, such as equivalence, specialisation, and versioning.

The third level presents re-usable models for the definition of information artifacts, such as database schemas, service descriptions, forms, queries, and reports. A model repository for education might include models of admissions forms, study transcripts, and spreadsheets for reporting registration and progress data to national agencies. The fields on the forms, the entries on the transcripts, and the columns on the spreadsheets may be described, and given computable semantics, by linking them to the metadata elements in a metadata registry.

In the Software Engineering group at Oxford, we have explored these ideas in the domain of clinical trial informatics. The CancerGrid project is a consortium involving the universities of Oxford, Cambridge, Birmingham, Belfast and London, funded by the Medical Research Council with additional support from Microsoft Research. For the last three years, the consortium has been developing a common vision for semantic frameworks and model-driven software engineering, focussed upon software support for the design and operation of cancer clinical trials. We believe that the ideas are more widely applicable than clinical trials, or even than health informatics; indeed, we believe that they are a close fit for the challenges of electronic government.

C Education and training

Formal methods have traditionally been seen as most applicable in limited domains: typically high-integrity, safety-critical, embedded systems. Electronic government represents an exciting and important new application domain, and an opportunity to widen the impact of formal methods: the challenges of representing data semantics — precisely enough to support the automatic generation of the information systems that manipulate that data — call for the leverage that only formal methods can apply.

Electronic government exemplifies what is becoming known as the digital economy [2] — the transformative effects of technology upon society. Successful developments in such domains necessarily entail a multidisciplinary approach, taking into account issues of management, user engagement, ethics, security, and society, as well as the more obvious technical matters of computer science. The leaders of the digital economy must be broadly educated: it is more important that they have some appreciation of all of these issues, than that they are a specialist in one. We see the digital economy as a promising initiative for widening the scope of education and training in formal methods.

D Acknowledgements

This position paper draws on an earlier paper [1], with contributions from Aadya Shukla and Steve Harris, and also on long and detailed discussions on the Digital Economy with Marina Jirotka, Janet Smart, and others at Oxford.

References

1. Charles Crichton, Jim Davies, Jeremy Gibbons, Steve Harris, and Aadya Shukla. Semantics frameworks for e-government. In Theresa Pardo and Tomasz Janowski, editors, *International Conference on e-Government*. ACM, December 2007.
2. EPSRC. Centres for doctoral training in the digital economy. `http://www.epsrc.ac.uk/PostgraduateTraining/NewCentres/DigitalEconomy.htm`, March 2008.