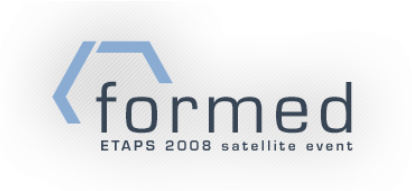
Formal Methods in Computer Science Education

FORMED2008

Budapest, Hungary
March 29, 2008

PROCEEDINGS



Satellite workshop of ETAPS 2008

Editor
Zoltán Istenes

Contents

Preface

This volume contains the proceedings of the *First Workshop on Formal Methods in Computer Science Education* (FORMED2008). FORMED2008 workshop is a satellite event of the European Joint Conferences on Theory and Practice of Software (ETAPS 2008), held in Budapest, Hungary on the 29th of March 2008.

Nowdays formal methods have an important role in the curricula of higher education. Teaching formal methods is a challenge for both students and teachers. As the student's motivation and knowledge background varie, it is difficult to choose the adequate formal method and it's opportune place in the curriculum. One must decide about the formal method and software tool, the specification language and the relations between the formal and semi-formal methods. Learning and teaching programming languages seem to be easier tasks, however the development of rigorous and high level abstraction skills is also crucial and imperative. We would like to provide a forum for the lecturers, teachers, industrial partners to discuss and present their experiences, their pedagogical methodoloy and their best practices.

The papers were refereed by the program committee, whose help is gratefully acknowledged.

Many thanks to the invited speaker —Jean-Raymond Abrial—, to the authors and the attendees who allow the workshop to be a fruitful occasion of discussion and exchange of new ideas.

These proceedings will be published as a volume in the series of Electronic Notes in Theoretical Computer Science (ENTCS).

Finally, special thanks to Viktória Zsók and Zoltán Porkoláb for the valuable discussions about the organization...

Budapest,
February 18, 2008

Zoltán Istenes
organiser of the FORMED2008 workshop

The program committee of FORMED2008 workshop:

- András Benczúr, Eötvös Loránd University, Hungary
- Steve Dunne, University of Teesside, U.K.
- Ákos Fóthi, Eötvös Loránd University, Hungary
- Henri Habrias, University of Nantes, France
- Zoltán Horváth, Eötvös Loránd University, Hungary
- Zoltán Istenes, Eötvös Loránd University, Hungary
- David Lightfoot, Oxford Brookes University, U.K.
- Ken Robinson, University of New South Wales, Australia
- Andrew Simpson, Oxford University, U.K.

The invited speaker of FORMED2008 workshop:

- Jean-Raymond Abrial, Swiss Federal Institute of Technology Zurich

Teaching Formal Methods: an Experience with Event-B (Invited Speaker's extended abstract)

Jean-Raymond Abrial[1]

Swiss Federal Institute of Technology Zurich

Event-B. It is the name of a mathematical approach used to develop *discrete system models*, be they computerized or not [2]. It is the successor of the B Method [1], which has been used in various operational industrial projects [8] [7]. Event-B has been strongly influenced by the work of the Finnish School on Action System [6] [9]. It is a parent approach to that of TLA+ [11].

It has been found over the years that a *unique paradigm*, that of discrete transition systems, was the common denominator to apparently very different computation mechanisms: sequential, distributed, concurrent, parallel, etc. Such a paradigm is the heart of Event-B.

Teaching Event-B. The Event-B approach has been chosen to introduce formal methods to students at ETH. There are three main reasons for this: (1) Event-B can be used to specify and develop systems following very different models of computation as said above, (2) I was one of the creators of B and Event-B, and (3) a set of tools for Event-B, the Rodin Platform [13], was developed at ETH Zurich during the years 2004-2007 within the framework of the European Project Rodin [14].

Event-B Courses. I gave 2 courses each year on this matter during my three years stay as a Guest Professor in ETH Zurich from 2004 to 2007: an *Introductory Course* for Undergraduate (2nd year) students, and an *Advanced Course* for Master and PhD students. After I left, another Advanced Course was taught at ETH during the winter 2007/2008. Other Event-B course are taught in France (Metz, Nancy, Bordeaux), in England (Southampton, Manchester), and in Finland (Turku). Some are intended to be taught in Australia (Sydney) and in Canada (Mac Master, York).

Nature of the Introductory Course. The Introductory Course was mandatory. I had more than one hundred students. It lasted 7 weeks. Each lecture in a week

[1] Email: jabrial@inf.ethz.ch

was made of 3 sub-lectures (in the same morning) lasting 45 minutes each (it is too much in only one morning). Besides these lectures, there were one tutorial session per week. The tutorial sessions were not obligatory, nor were graded the home work exercises given at each tutorial. The grading was performed via a single final written examination.

Nature of the Advanced Course. The Advanced Course was optional. I had 15 students on the average. It lasted 14 weeks. Each lecture in a week was made of 2 sub-lectures (in the same morning) lasting 45 minutes each. Besides these lectures there were one tutorial session per week. As for the Introductory Course, the tutorial sessions were not obligatory, nor were graded the home work exercises given at each tutorial. The grading was performed via: (1) a project made by groups of two students (60%), and (2) a final written examination (40%).

Contents of each Lecture. Whatever the course (introductory or advanced), the pedagogy was based on an inductive approach. That is, each lecture in a week was focused around a single example. The purpose of that example was to practically introduce some general notions which were summarized at the end of the lecture and thus accumulated as the course proceed.

Course Material. Before each lecture, the following material was distributed to the students: (1) a written text (in fact a draft chapter of my coming book [2]), and (2) a set of electronic slides. Besides this, the formal proofs needed to validate the studied example were automatically or interactively performed with some tools: *Click'n'Prove* [5] first and then the *Rodin Platform* [13]. Finally, each example was ultimately executed according to the outcome of the last refinement step. The construction of this material represents a considerable effort, but I think it is indispensable in order for the students to understand that formal methods in general and Event-B in particular can be made practical.

Outcome of the Introductory Course. The Introductory Course evolved over the years. The first one was far too heavy, the students were completely overwhelmed with too much material. As a consequence, that first course was not well received by the students although they understood the importance of the discipline. But they objected that it was not well taught, that they could not swallow the material in such a short period of time. Subsequent courses were better received as I removed lots of material. The last one had far better student reviews than the two previous ones. The grading curve was one with two hills: one very good one and one just average. It shows that a group of good students were able to understand the material in depth while most of the other students were just able to get it, but without mastering it very deeply.

Lessons Learned from the Introductory Course. I found this course quite difficult to teach. I think that *seven weeks is far too short*: students can only enter into the material in a gradual fashion, they only start to see what it is all about by the end of the course, which is a bit late. One of the main problems is that the students are not at all used to make a practical use of logic and set theory. Although they attended previously a course on logic and discrete mathematics, they had nevertheless great difficulties to perform practical formal proofs. I think that the logic and discrete mathematics course should insist on the concept of proofs. Students also have difficulties with the concepts of abstraction and refinement. From

their course in programming they always think of a program in operational terms. The key lesson I learned is that one should *not try to be exhaustive*: one should insist on a few messages, such as refinement steps development and formal proofs. These messages have to be delivered by means of a few simple examples.

Outcome of the Advanced Course. The Advanced Course was far better received than the Introductory Course and also easier to teach: the course being optional, the students were also far more motivated. It was possible to cover almost the entire material (development of sequential, distributed, concurrent, or parallel "programs", and also complete development of embedded systems including a formalization of the environment). The grading curve show that almost all students followed the material extremely well. In the latest Advanced Course taught at ETH during the winter 2007/2008, almost all students had already attended the previous Introductory Course.

Lessons Learned from the Advanced Course. Over the three years, the tools evolved considerably: on the last course, the students were using the Rodin Platform [13] to develop their projects. The main lesson to learn here is that good tool support accelerate significantly the learning and mastering of formal methods. We also found that the project done along the entire 14 weeks was very successful: the students learned how to develop a system which "worked first time".

Using the Tool. The Rodin platform is an open tool, so it is easy for the students to load it and use it freely. It is absolutely clear that this tool has to be used in the Advanced Course. Concerning its usage in the Introductory Course, we have no experience so far, although the tool was shown to the students in the tutorial sessions. The fact that the tool is implemented under Eclipse presents a certain advantage as most of the students are familiar with Eclipse. It is probably the case that further Introductory Courses will use the tool. Again, we found that it is clearly far more "fun" for the students to use the tool than to do their homework with paper and pencil only, and in doing so, they learn more. However, it is important that the tool is not distributed too early: the students have to figure out first what the tool is supposed to do, simply by doing an initial development manually.

More on The Project. The project was an important part of the Advanced Course. It was not possible to have it in the Introductory Course: not enough time, too many students, and moreover the tool was not mature enough. But this might be envisageable in the future with the new Rodin Platform in its present state. The three projects of the Advanced Course were a lift system, a cooling system, and a CD player. In all three cases, a bad initial requirement document was distributed to the students. It was "bad" in that it described what the system was doing in operational terms (a kind of pseudo-implementation, as is often the case) rather than defining on which grounds it could be said to be correct. So, their first task (and first milestone) was to perform a complete rewrite of it according to the guidelines given in the course on that matter (I insisted a lot in the course on the importance of the requirement document). Then they had to define their refinement strategy (second milestone). This was followed by the development of an initial model and then the various formal refinement steps (several milestones here). At each step, they had to do all the formal proofs. We insisted on the very strong connection that must exist between modelling and proving: we wanted the students

to understand that difficulties in doing the proofs are an important sign that the corresponding model is either faulty, insufficient, or too complicated. Some of the students figured that out quite well and consequently decided to have, in the middle of their project development, an heavy reformulation of their refinement strategy and refinement steps.

Concluding Remarks. I enjoyed a lot giving these courses. It is clear to me that, contrarily to what is said in lots of circles, the discrete mathematics framework is not difficult to master for the students. What is far more difficult however is the mastering of the construction of significant models. This is the reason why I decided to base my pedagogy on examples and on the project. But I still think that more has to be done in this area.

Acknowledgements: I would like to thank a lot D.Basin, L.Voisin, S.Hallerstede, Thai Son Hoang, F.Mehta, D.Cansell, D. Mery, and P.Casteran who helped in the courses I gave in Zurich or themselves developed Event-B courses in other places.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modelling in Event-B: System and Software Engineering.* To be published by Cambridge University Press, 2008.

[3] J.-R. Abrial. *Tools for Constructing Large Systems (a proposal).* In *Rigorous Development of Complex Fault-Tolerant Systems.* M. Butler, etc. (Eds). LNCS 4157 Springer, 2006

[4] J.-R. Abrial, M. Butler, S. Hallerstede, L. Voisin. *An Open Extensible Tool Environment for Event-B.* ICFEM 2006

[5] J.-R. Abrial, D. Cansell *Click'n'Prove: Interactive Proofs within Set Theory* TPHOL, 2003

[6] R. Back. *Decentralization of process nets with centralized control.* 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 1983.

[7] F. Badeau. *Using B as a high level programming language in an industrial project: Roissy val.* Proceedings of ZB'05, 2005.

[8] P. Behm. *Meteor: A successful application of B in a large project.* Proceedings of FM'99, 1999.

[9] M.J. Butler. *Stepwise Refinement of Communicating Systems.* Science of Computer Programming, 1996

[10] M.J. Butler and S. Hallerstede *The Rodin Formal Modelling tool.* BCS-FACS Christmas 2007 Meeting Formal methods in Industry London, 2007

[11] L. Lamport *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2003.

[12] *Eclipse.* http://www.eclipse.org

[13] *Rodin Platform.* http://www.event-b.org

[14] *Rodin Project.* http://rodin.cs.ncl.ac.uk

Teaching specifications, hands on

Henri Habrias[1]

I.U.T, département informatique
Université de Nantes
Nantes, France

Abstract

In this paper, we expose some exercises that we use to teach formal specification for 1^{st} year students.

Keywords: Formal specifications, Teaching, Hands on, B method

"En informatique, la tentation d'imprimer beaucoup et de réfléchir peu est inquiétante." Pierre-Gilles de Gennes, Nobel prize

1 Introduction

The students I teach to all possess the French scientific Baccalaureat. They succeeded that national exam passed at the end of secondary school. These students received in my Institute a Mathematics course of 10 hours about sets and relations, along with 40 hours of "directed works". Then, I receive these students for a first course and practical works on specifications. I use the B method [1], [2], re-using of what the students have been taught (sets and relations - not understood, as we will see) I noticed that the students with "good" marks at the mathematics exam made in fact confusion between $\in$ and $\subseteq$. They don't just confuse the symbols, they are even not able to apply what they learnt. They even don't comprehend the concept of set of atoms.

After a lot of exercises, I decide to stop and to practice *La main à la pâte*. *La main à la pâte* (Hands on) is the name of a programe launched in 1995 by the Nobel Prize winner Georges Charpak (after L.M. Lederman in Chicago), intended to revitalize the teaching of sciences in primary school in France (www.lamap.fr/, www.elearningeuropa.info/).

[1] Email: henri.habrias@univ-nantes.fr

[2] We want to thank Thibaud Poncin, student in first year of the IT department of the IUT de Nantes, who checked and corrected the English of this document.

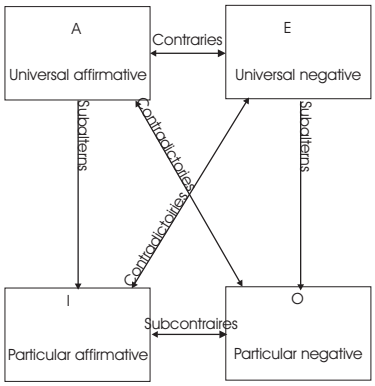


Fig. 1. Aristotelian Square

In the first section we present some exercices of translation from natural language to the set language and the first order language. Then we practice *La main à la pâte* with the example of the wallet.

2 Set language and the first order language

We use these two exercises to show the power of set notation *vs* logic language, and also to show the ambiguity of natural language.

2.1 The Aristotelian Square of Opposition

Are distinguished :

- Particular *vs* Universal *Callas is a man, every student is a man*
- Affirmative *vs* Negative *Every man is mortal, No professor is stupid*

The cartesian product {universal, particular} $\times$ {affirmative, negative}, gives four ordered pairs which are the four corners of the famous *Aristotelian Square*.

2.2 The Aristotelian Square in B, in predicate language and in set language

The four propositions are named with the vowels (A, E, I, O) taken from the latin words *AffIrmo et nEgO*. We will write these propositions considering a SET T and two SETS A and B. We give first the expression in the first order logic language then in set language.

SETS T
CONSTANTS $A; B$
PROPERTIES $A \subseteq T \wedge B \subseteq T$

- Universal affirmative (A):
  Every A is B, *Everyone dies someday*
  $\forall x \bullet x \in A \Rightarrow x \in B$

  $A \subseteq B$

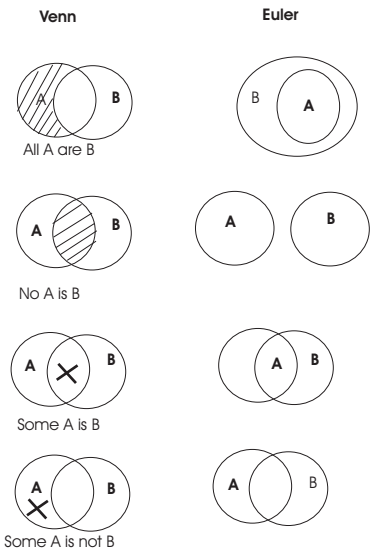


Fig. 2. Aristotelian Square with Euler and Venn Diagrams

- Particular affirmative (I):
  Some A is B, *Some men are philosophists*
  $\exists x \bullet x \in A \wedge x \in B$

  $A \cap B \neq \emptyset$
- Universal negative (E):
  No A is B, *No man lives forever*
  $\forall x \bullet (x \in A \Rightarrow \neg\, (x \in B))$ or

  $\neg\, (\exists x \bullet x \in A \wedge x \in B)$
  $A \cap B = \emptyset$
- Particular negative (O):
  Some A is not B, *Some men are not philosophists*
  $\exists x \bullet (x \in A \wedge \neg\, x \in B)$

  $A - B \neq \emptyset$

Exercise :
Represent the four above propositions with Euler diagrams or Venn diagrams.

2.3 Dudule loves Huguette. An exercise on logic language and set notation

This example can be found in lots of logic lessons. It likely originates from S.C. Kleene book, "Mathematical logic" [5]. We completed Kleene's statements list.

MACHINE *ourLoves*
SETS $PERSON$
VARIABLES *loves*
INVARIANT $loves \in PERSON \leftrightarrow PERSON$
END

It's all about translating every statement in predicate logic language (with the B notation) and in set notation.

- *Someone loves Huguette*
  - $\exists p \bullet (p \in PERSON \land (p \mapsto Huguette) \in loves)$
  - $Huguette \in ran(loves)$
- *Nobody loves Huguette*
  - $\neg\ (\exists p \bullet (p \in PERSON \land (p \mapsto Huguette) \in loves)$
  - $\forall p \bullet (p \in PERSON \Rightarrow (p \mapsto Huguette) \notin loves)$
  - $card(loves \rhd \{Huguette\}) = 0$
  - $Huguette \notin ran(loves)$
  - $loves \rhd \{Huguette\} = \emptyset$
- *Everybody loves Huguette*
  - $\forall p \bullet (p \in PERSON \Rightarrow (p \mapsto Huguette) \in loves)$
  - $dom(loves \rhd \{Huguette\}) = PERSON$
  - $(loves \rhd \{Huguette\}) \in PERSON \rightarrow \{Huguette\}$
- *Everybody loves somebody*
  - $\forall p \bullet (p \in PERSON \Rightarrow (\exists q \bullet (q \in PERSON \land (p \mapsto q) \in loves)))$
  - $dom(loves) = PERSON$
- *Everybody loves everybody*
  - $\forall p \bullet (p \in PERSON \Rightarrow loves[\{p\}] = PERSON)$
  - $dom(loves) = ran(loves) \land ran(loves) = PERSON$
  - $card(loves) = card(PERSON) \times card(PERSON)$
- *There is someone loved by everyone*
  - $\exists q \bullet (q \in PERSON \land \forall p \bullet (p \in PERSON \Rightarrow (p \mapsto q) \in loves))$
  - $\{p | p \in PERSON \land loves^{-1}[\{p\}] = PERSON\} \neq \emptyset$
- *Everybody loves oneself*
  - $\forall p \bullet (p \in PERSON \Rightarrow (p \mapsto p) \in loves)$
  - $id(PERSON) \subseteq loves$
- *Nobody loves himself*
  - $\neg\ \exists p \bullet (p \in PERSON \land ((p \mapsto p) \in loves))$
  - $id(PERSON) \cap loves = \emptyset$
- *Everyone loves who loves him*
  - $\forall p, q \bullet (p \in PERSON \land q \in PERSON \land (p \mapsto q) \in loves \Rightarrow (q \mapsto p) \in loves)$
  - $loves = loves^{-1}$
- *Everyone loves who his beloved ones love, another version of My friends' friends are my friends*
  - $\forall p, q, r \bullet (p \in PERSON \land q \in PERSON \land r \in PERSON \land (p \mapsto q) \in loves \land (q \mapsto r) \in loves \Rightarrow (p \mapsto r) \in loves)$
  - $loves \fatsemi loves \subseteq loves$

2.3.1 Love loving not itself none other can.
Duchess of York: O king, beleive not this hard-hearted man! ***Love loving not itself none other can.*** Richard II, Act V, scene III, W. Shakespeare
$dom(loves) - dom(loves \cap id(loves)) = \emptyset$

3 The wallet

I asked a group of students: "Who's got a wallet?" Happily, one of them raised her hand. And now, I wrote on the white board the following:

MACHINE $Wallet$ END

The student was surprised.
Student: *A machine! For my wallet!*
Professor: *Do we need a variable?*
Student: *We don't*
Professor: *Why you was using your wallet?*
Student: *To put coins in and to take off coins from the wallet.*
Professor: *So do the content of your wallet change?*
Student: *Yes, it does.*
Professor: *So the content of your wallet is variable! What does that content contents?* Several students answered $COINS$.
Professor: *But what do you mean by coins? Can you give me the SET in extension?*
Here's what they found out: $COINS = \{0,01; 0,02; 0,05; 0,10; 0,20; 0,50; 1; 2; 5\}$
Professor: *Do you respect the B notation for sets? I have never written a set this way. I used commas between elements!* Student: *But we want to distinguish the commas for cents and the ones between elements.*
Professor: *Please! Don't change notations! You would not change Ada vocabulary and language when you are programming in Ada! Would you? Then, why not to give the names of the coins in cents?*
$COINS = \{1, 2, 5, 20, 50, 100, 200, 500\}$

Then, I let the group (25 students) alone for a while, having for task to write the invariant all together. Coming back, here's what I found on the white board:
$wallet\_content \subseteq COINS$
Professor: *Do you all agree with that?*
One student proposes: SETS
$0,01; 0,01; 0,02; 0,05; 0,10; 0,20; 0,50; 1; 2; 5$. Seeing how surprised I was, he starts specifying:

VARIABLES $1c, 2c, 5c, 10c, 20c, 50c, 1e, 2e5e$ INVARIANT $1c \in NAT \wedge 2c \in NAT \wedge ...$

Then he proposes to change the variables names by $numberOf1c, numberOf2c$, etc. All the students seemingly agree.
Professor: *Good. We will come back to this specification later.* I then ask them to consider the first one.
Professor: *Is $COINS$ a SET, a basic set? The answer is "no". $COINS$ is a subset of NAT. So it is not a basic set. What can we do about that? We can write* SETS $COINS = \{1c, 2c, 5c, 10c, 20c, 50c, 1e, 2e, 5e\}$
So now your invariant is: $wallet\_content \subseteq COINS$. Now could you please give me the content of your wallet?
$wallet\_content = \{1c, 1c, 5c, 5c, 5c, 2e\}$
Now observe what is written. Do you really think everything is fine? Luckily, one student notices that if the same element is written twice, it is the same one. So $wallet\_content = \{1c, 5c, 2e\}$. I now propose the student to give her back what is in the *wallet_content*. She does not agree. She wants her money back! So I propose to look at the second specification.
Professor: *Could you find a solution with less variables?* No answer of the students.
Professor: *Could you literally say what is in your wallet?*
The student : *Two coins of one cent, three of five cents and one of two euros*
Professor: *Do you think it looks like a set of ordered pairs? Why not a relation?*
I propose the following specification.

MACHINE $Wallet$
SETS
$FACIAL\_VALUE = \{1, 2, 5, 20, 50, 100, 200, 500\}$
VARIABLES $wallet\_content$
INVARIANT $wallet\_content \in FACIAL\_VALUE \nrightarrow NAT$
END

I ask if a better name could be found to the SET. One student says $VALUE$. I remark that this term is too much generic. Two days ago French press informs us EADS and other big companies would leave Europe. Airbus sells his jets in dollars and pays his workers in euros. Then what is the value of 1 euro? In our institute, our students have received more economics and management classes than specification. I am going to reuse them!So I propose to name the SET "facial value".
$wallet\_content \in FACIAL\_VALUE \nrightarrow NAT$
Professor: *Why a partial function? Because the student wallet does not include all the values. But we will come back to this point again. Is NAT a good idea? Do you think you could put enough money in your wallet to reach MAXINT?*

It is time to consider the practical utility of our exercise (outside didactic utility). I explain to the students that lots of machines use "wallets". It is a basic component of vending machines. It would be interesting to specify, prove and implement a wallet. And we also want to use our wallet in different countries, and for different sizes of wallets. It is an occasion to specify a definition of the "size" ($Maximum = max(ran(wallet\_content))$) in terms of elements of $FACIAL\_VALUE$. We will have a machine with two parameters: $FACIAL\_VALUE$ and $Maximum$. It is

also the right time to introduce the term *bag* or *multiset*. I give the students the Z [7] symbol for *bag union*: $\uplus$ and ask the students why the $+$ inside the $\cup$? I think we have to avoid these terms. The Occam razor is a good didactic tool. We were been able to avoid these terms. So now we can tell that we defined a bag as "Le Bourgeois Gentihomme" was writing prose without knowing he was doing so. I ask the student to write the INITIALISATION and some operations: to add a coin in the wallet, to take off a coin from the wallet, to add multiple coins in the wallet, to add a wallet to another one (we will call this operation, "union of bags"). We used the term "union of bags", why not writing the specification of an intersection of bags?

I do not give the parameters of these operations. It is a part of the specification to find these parameters. I ask the students to specify first minimal operations.

MACHINE *example*
SETS AA
VARIABLES ss, tt, rr
INVARIANT
$ss \subseteq AA \wedge tt \subseteq AA \wedge rr \in ss \rightarrow tt$
...
OPERATIONS
$Op(el1) \mathrel{\widehat{=}}$ PRE $el1 \in AA \wedge el2 \in AA$
THEN $ss := ss \cup \{el1\}$ END;

$Op(el1)$ cannot be a minimal operation. It is necessary to have the following substitution $rr := rr \cup \{(el1 \mapsto el2)\} || ss := ss \cup \{el1\}$ to respect the invariant. If we modify $rr \in ss \rightarrow tt$ by $rr \in ss \nrightarrow tt$ in the invariant, we could specify: $Op(el1) \mathrel{\widehat{=}}$ PRE $el1 \notin dom(ss)$ $THEN$ $rr := rr \cup \{el1\}$ END;

A student: *I proposes to use a total function instead a partial one. Like this, it will not be necessary to verify that when we want to apply the function to x that x is in the domain of the function.*
Professor: *Could you specify the initialisation?*
The same student: *I have difficulties to write it.*
An other student: *I proposes:* $ran(wallet\_content) := \{0\}$
The first student: *I do not agree:* $ran(wallet\_content)$ *is not a variable.*
Not other student was disturbed by such a substitution!
The first student: *I wanted to write:* $wallet\_content := FACIAL\_VALUE \rightarrow \{0\}$
but I think it is not correct.
Professor: *Why? Always my same question! instanciate! Consider* $FACIAL\_VALUE = \{1euro, 2euros\}$.
$wallet\_content := \{1euro, 2euros\} \rightarrow \{0\}$
$\{\{(1euro \mapsto 0), (2euros \mapsto 0)\}\}$
So, is such an initialisation correct?
The first student: *No. The typing is not correct. We have a set of sets.*
Professor: *So, what is the good initialisation?*

The first student: *I don't know how to do.*
Professor: *Please, look at your notes on my first course, or look at my printed course!*
nota: The very large majority of students do not write notes during the course (normal : they do not read it!), do not read the printed course and when they open my blog, they spend an average of 1 second on a page! The maximum visit length is 28 minutes. Thanks *sitemeter*!).
Professor: *Here what I wrote on the black-board during the first course and several times since:* $wallet\_content \in \{1euro, 2euros\} \rightarrow \{0\}$ or $wallet\_content \in \{1euro, 2euros\} \times \{0\}$. Here we will choose the union of two wallets. Since the begining of our teaching we ask again and again to the students to give an example before to write some B text. Since some years, the large majority of the students refuse to use papygrams or examples and counter-examples.
$\{(a \mapsto 3), (k \mapsto 2), (z \mapsto 1)\} \uplus \{(a \mapsto 2), (b \mapsto 5)\} = \{(a \mapsto 5), (k \mapsto 2), (z \mapsto 1)\}$
We note that: $\{(a \mapsto 3), (k \mapsto 2), (z \mapsto 1)\} \cup \{(a \mapsto 2), (a \mapsto 2), (b \mapsto 5)\} = \{(a \mapsto 3), (k \mapsto 2), (z \mapsto 1), (a \mapsto 2), (b \mapsto 5)\}$.
An example being obtained, we ask the students to express in french what has been done. And then to write the specification of the operation. This operation is interesting because the students try to use a loop to specify. It is a good occasion to make the difference between programming and specifying. I suggest to define a lambda expression to specify a function addition. The students are now free to use the Atelier B. Here is the first solution obtained by a student. He choose to keep the state of the two wallet_contents, filling a third wallet with the two first wallets.

MACHINE $Wallet$
SETS $FACIAL\_VALUE$
VARIABLES $wallet\_content1, wallet\_content2$
INVARIANT
$wallet\_content1 \in FACIAL\_VALUE \rightarrow NAT \wedge$
$wallet\_content2 \in FACIAL\_VALUE \rightarrow NAT$
DEFINITIONS
$union\_bags(yy, zz) \mathrel{\widehat{=}} \lambda xx.(xx : FACIAL\_VALUE | yy(xx) + zz(xx))$
INITIALISATION
$wallet\_content1, wallet\_content2 := FACIAL\_VALUE \times \{0\},$
$FACIAL\_VALUE \times \{0\}$

OPERATIONS
$add\_coin(fv) \mathrel{\widehat=}$ PRE
$fv \in FACIAL\_VALUE \wedge wallet\_content1(fv) + 1 < MAXINT$
THEN
$wallet\_content1 := wallet\_content1 <+ (fv \mid wallet\_content1(fv) + 1)$
END;
$take\_off\_a\_coin(fv) \mathrel{\widehat=}$ PRE
$fv \in FACIAL\_VALUE \wedge wallet\_content1(fv)) > 0$
THEN
$wallet\_content1 := wallet\_content1 <+ \{(fv \mapsto wallet\_content1(fv) - 1)\}$
END;
$add\_a\_wallet\_to\_another\_one(yours, mine) \mathrel{\widehat=}$ PRE
$mine = wallet\_content1 \wedge yours = wallet\_content2$
THEN
$wallet\_content1 := mine <+ union\_bags(mine, yours)$
END
END

3.1 Intersection of bags

A student proposes the following example: $\{(a \mapsto 3), (k \mapsto 2), (z \mapsto 1)\}$ "intersection" $\{(a \mapsto 2), (b \mapsto 5)\} = \{(a \mapsto 2)\}$. Why not? We say him that in Z, exists the *Union of bags* but not the Intersection of bags. So there is no confusion with an other "official" definition. It seems for us very important to show to the students that they have to learn how to define. In school they have been accustomed to learn definitions by heart. Too often the professors do not construct the definitions with the students.

4 Conclusions

These conclusions are my conclusions after 40 years of teaching specifications in France and written the day where the newspaper Le Monde publishes pages about the OECD PISA (Programme for International Student Assessment ranking). France has lost 9 places in 3 years. Edouard Brézin (French Academy Of Sciences) writes: "Setting up an educative system leaving place to observation [...] I fear an abuse of computers which give an instant answer driving to remove from children every investigation step [...] Prolong the success of "La main à la pâte" in primary and secondary schools sounds as the only answer to me."

If you want your students to use maths, teach maths in your specification course. Pay attention! Concepts of sets and relations are very often misunderstood. With students more and more accustomed to their "virtual reality", the concept of model is non-sense. The definition of Minsky (*"To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A"*) [6] is out of their scope. All is virtual ... So use physical objects. I used an abacus to implement an integer [4]. I used the wallet in the practical work

exposed in the present paper. The teaching method "La main à la pâte" is good to teach formal specification. Observation, abstraction, manipulation are considered as complementary to calculus and formal verification.

References

[1] Abrial J.R. (1996) The B-Book, Assigning Programs to Meanings. Cambridge University Press.

[2] Habrias H. (2001) Spécification formelle avec B. Hermes Lavoisier.

[3] Habrias H., Faucou S. (2004) *Linking Paradigms, Semi-formal and Formal Notations*, TFM'04, LNCS, Ghent

[4] Habrias H. (2006) *"La main à la pâte", An Abacus to Teach Formal Specifications, TFM'06, The British Computer Society, London*

[5] Kleene S.C. (1967), Mathematical Logic, Wiley

[6] Minsky M. L. (1968), Matter, mind and models, *Semantic Information Processing*. MIT Press

[7] Spivey M. (1988), The Z Notation. Prentice Hall

Reflecting on the Future: Objectives, Strategies and Experiences

Ken Robinson

School of Computer Science & Engineering
The University of New South Wales
Sydney NSW 2052, Australia

Abstract

This paper discusses the future of the teaching of rigorous software engineering methods, aimed at achieving full strength engineering of high reliability software systems. The discussion considers objectives and strategies and presents a critical assessment of past experience in planning to achieve essentially the same objectives in modified courses. The paper gives an assessment of current attitudes to *formal methods* both from within and without the formal methods community, and also tries to detail the essential understanding and skills that need to be developed.

Keywords: modelling, refinement, decomposition, formal, rigorous, software engineering, classical-B, event-B, fault-free designs

1 Introduction

This paper is a non-technical paper that discusses the author's experiences in teaching rigorous methods, largely in the context of software engineering teaching at *The University of New South Wales* (*UNSW*). The paper is partly prompted by the FORMED call for papers, which opened (*with my emphasis*):

> Nowadays formal methods have an important role in the computer science curricula of higher education. Given the wide variation in students' motivation and background knowledge in this area, formal methods certainly represent a challenge for both students and teachers. Difficulties arise in choosing an appropriate formal method and finding its most opportune place in the curricula. Deciding the formal method with its underlying specification language and its supporting software tool is a complex task. *The development of rigorous and high-level abstraction skills is indisputably a crucial and essential objective of modern computing curricula.* These skills have a wider area of applicability, even beyond software engineering. On the other hand, developing abstraction skills through

[1] Email: kenr@cse.unsw.edu.au

> formal methods is more challenging than the traditional education of programming. The purpose of the workshop is to provide a forum for teachers and industrial partners to discuss pedagogical methodologies and share experiences of teaching formal methods.

This statement contains much with which this author can agree from an aspirational or motivational viewpoint, but also much that is stated as fact with which there can be strong disagreement.

There can be little disagreement —from anyone who has tried— about the difficulty in choosing an appropriate formal method and tools for the support of the method, but saying that *"nowadays, formal methods have an important role in the computer science curricula of higher education"* is not consistent with my experience. It may be that I live in the wrong part of the world, but locally I do not observe any strengthening of desire to use formal methods; indeed, if anything, a weakening.

As a point of clarification, it is assumed that for the purpose of this workshop "Computer Science" is used as a generic term for any computing discipline. Personally, I draw a distinction between Computer Science and, say, Software Engineering.

At an FME symposium [6] on *Teaching Formal Methods*, held in Ghent in 2004, this author presented a paper [8] detailing observed failures of formal methods. Points from that paper will provide a starting point for this paper.

One of the early points made in that paper:

> As a general observation the use of formal methods in system development appears to have had very little penetration into practice. One reason we submit is because *Formal Methods* has developed as though it is a separate discipline sitting in glorious isolation from the rest of Computer Science, Software Engineering and Computer Engineering. While members of the Formal Methods community have a common concern, namely to use rigorous and formal techniques in computing, they are motivated by a variety of problems and there are no foci. For fundamental research in formal methods this is not a disadvantage, but it is a serious obstacle to the transfer of formal techniques to application areas.

is unfortunately still true. This makes the claim from the call for papers for this workshop, that *formal methods have an important role in the computer science curricula of higher education*, seem more like wishful thinking. It could be worse: it could be a case of self-deception. Or it could be that formal methods are indeed strong in *Computer Science* (*CS*) departments with a strong theoretical emphasis.

In discussing *objectives*, *strategies* and *experiences* this paper is concerned with teaching in a software engineering engineering environment to software engineers. It may be perfectly reasonable for computer science centres to be concerned with formal methods as a pure science pursuit.

The remainder of this paper presents a number of issues associated with teaching rigorous methods to software engineers:

(i) requirements: what we want to achieve;

(ii) objectives: a more detailed statement of objectives;

(iii) strategies: how we might achieve those objectives;

(iv) problems: some basic problems in achieving the objectives;

(v) experiences at UNSW;

(vi) changes and planning for the future based on those experiences.

2 Objectives and Strategies

A very simple example of the problem with curricula in our area of concern appears in the recent IEEE/ACM Software Engineering Curriculum [4], which contains a core course called *Formal Methods in Software Engineering*. This looks like an advance —and it is to some degree— but it's an isolated course that deals with formal methods in the small and without context. Some students with a strong theoretical bent will pursue it further, but its impact on many students will be minimal.

The problem is that much formal methods hype sees itself as the solution, but it does not demonstrate that it has anything specific to offer. Indeed it often offers promises like "use formal methods to prove that your xyz will not fail", a promise that is not sustainable.

2.1 What is Required?

Quite apart from the more rarefied levels of formal methods, it should be clear to those who teach any computing discipline, but especially software engineering, that what is desperately required is the ability to *reason* about system designs. Programming encourages empirical reasoning about software designs, but the discrete nature of computing renders this a reasonably weak form of validation. This is well known, but it is still the most common form of validation.

I find generally that students —and probably others also— cannot reason why their software does what they think it does. If challenged, they will jump straight to testing.

What is required is an increase in rigour and an increasing ability to reason about abstract designs, that is, *modelling*. This requirement is more of an *engineering* requirement than a *science* requirement, and is at the root of the concerns of this paper.

2.2 Objectives

In setting objectives, there are some questions that can be asked:

(i) Why is the course being given?

(ii) What are the expected outcomes?

If the teaching of formal or rigorous methods are to have a lasting effect on students the following seem to be required:

- the exposure must develop *motivation*;
- there must be exposure to problems that possess some aspect of non-triviality;
- development of an appreciation of abstract modelling in which the key design components are highlighted;

- development of an appreciation of the concept and practice of verification (proof);
- the experience with rigorous methods must integrate with the remainder of the students' experience;
- there must be an effective toolkit to aid modelling and reasoning;
- there should be some visible future for the use of the methods.

These objectives are not easy to achieve, especially if the exposure is within a single course (subject), after which students return to more conventional courses. However a significant amount can be achieved from such a course:

- a better understanding of the incremental introduction of detail, *refinement*;
- a better appreciation of the role of modelling;
- a better appreciation of the role of design;
- a better understanding of data-structures and algorithms;
- an appreciation of the role of an invariant in the expression of the integrity of an operation (procedure, method) or even a whole class;
- a better appreciation of the concepts of specification, design and implementation.

2.3 What are the Problems?

Short and limited historical experience: students being taught in an undergraduate program will be around 20 years of age. Their computing experience will be around 10 years maximum. They will have a limited view of the computing landscape. For example, objected-oriented programming has existed for all their computing lifetime; they have probably never seen any formal method, that is, they don't exist.

Programming is experimental: students have only ever experienced programming as a *program, test, fix* cycle; they have never been asked to reason about any program they have written, and they wouldn't know how to do it if requested.

2.4 Motivation

Motivating rigorous approaches to software development is difficult. One of the big problems with *formal methods* has been that many in the formal methods community thought that motivation was obvious, hence the common assertions that are probably familiar:

(i) If you are developing safety critical systems, then you must use formal methods.

(ii) We are using formal methods to develop a system that will never fail.

(iii) If formal methods had been used the Ariane-5 disaster would never have happened.

All of the above assertions are false, and demonstrate a very limited understanding of the role and possible outcomes of the use of formal methods.

2.5 Our Experience

At UNSW, over a number of years, we have been attempting to develop courses for software engineering to satisfy the above requirements. We have been teaching some form of rigorous software development since the mid 1990s. Initially we used Z and moved to *the B Method* (*B*) in 2001

2.5.1 How We Currently Use B

We chose B as a method that covers the complete software development life-cycle and is clearly aimed at system development. It also comes with a toolkit (two toolkits in fact). We have mostly used the BToolkit from B-Core [5], although we also have AtelierB from Steria [12].

We integrate B via three courses:

COMP2111 *System Modelling and Design* [2]: a course in which students learn to develop specifications and refinements in B. The exercises and assignments in this course are significant, but relatively small.

SENG2010 *Software Engineering Workshop 2A* [9]: a practical software engineering workshop course in which students work typically in teams of four on the requirements and specification of some small, but realistic system. This course runs in parallel with COMP2111 and the final output is a specification of the system expressed in B.

SENG2020 *Software Engineering Workshop 2B* [10]: another software engineering workshop course that follows on from SENG2010. In this course the teams take their B specification and implement a prototype via a process of embedding formal developments.

2.5.2 Embedding B Developments

The object of the embedding exercise is to produce a 4-layer architecture like that shown in fig1 consisting of

B layer: some kernel of the B specification. This will be implemented via a refinement to an implementation, from which the BToolkit produces C code.

JNI layer: a layer that provides a *Java Native Interface* (*JNI*), which allows Java methods to call C code translation of the B operations in the B layer.

Control layer: a Java layer that possibly provides some extra functionality and controls communication from the GUI to the B layer.

GUI layer: the graphical user interface.

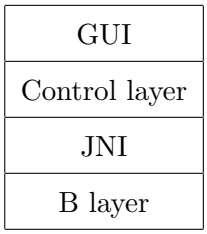


Fig. 1. 4-layer Architecture

The JNI is automatically generated by a tool that was developed at UNSW by Thai Son Hoang and is an extension of the BToolkit. The JNI tool produces its own GUI, so it is possible to collapse the above architecture to essentially two layers.

Other architectures are of course possible.

The purpose of the above architecture is to provide a framework wherein a formally developed, critical component of a system is embedded into less critical interface code. This also provides closure for the students who are able to take a B specification of a non-trivial system and produce an executable prototype.

To produce a prototype of the system the students first map their complete B specification onto a class diagram. This is reasonably straightforward to do as most of the machines are specifying classes, although B doesn't recognise them as such.

The next step is to decide how much of the class diagram will be implemented in B and how much will be implemented as control classes in, say, Java. This results in a class diagram that contains a B class and other classes whose methods can call the operations of the B class. The B class must be an API machine with robust operations.

To simplify B implementation we have used the BToolkit's base generator technology.

2.5.3 Assessment of Experiences

The scheme outlined in the previous sections has been followed for a number of years, so at one level the exercise could be regarded as a success. But student perceptions suggest a number of shortcomings:

no perception of future: it is difficult for students to take a long view of a very different technique like B. Future use of B would be encouraged by the availability of a free toolkit. Absence of availability of a freely available toolkit is a serious disincentive to continue use of the method.

Lack of visibility of industrial use of the method: although some very significant use of B can be cited it is very sparse.

Difficulty discharging proof obligations: experience in discharging of proof obligations, especially for the proof obligations generated from implementations produced using machines produced by the base generator.

Base machines seen as obscure: while convenient in many ways, the base generator is seen as obscure.

Problems with toolkit diminish confidence: the BToolkit has a number of problems, which while not too serious for an experienced user, do not encourage confidence in students who expect, and are used to, something better.

The above problems are largely based on perceptions, but these have a very strong influence on students at this stage in their development.

3 The Future

Having invested a considerable amount of time and effort into applying B to achieve our objectives —with mixed results— there is now a new variant of B available,

Event B (*eventB*) [1], and we have to decide whether to move with that or remain —perhaps becalmed— with what is becoming known as *Classical B* (*classicalB*).

There are significant differences between classicalB and eventB, some of which mean that our existing methods will not work without modification, but there are also significant advantages and the following is a description of our plans, some of which are yet to be developed, on how we will proceed.

3.1 Rodin Platform

In 2004 the RODIN project [3] was establisehd to develop a toolkit for eventB. This project was partly funded by the European Union and consisted of researchers from many European countries. The project finished at the end of 2007 and has produced a toolkit [7] implemented as an Eclipse platform. The toolkit is open source and supports Linux, Windows and Mac OS X. It also supports the possibility of independently produced plug-ins to extend functionality of the basic toolkit.

Before enumerating the disadvantages and advantages of eventB over classicalB, it might be worth making a general observation. While classicalB and eventB share almost-identical mathematical toolkits there is a considerable difference in *feel* to the two methods. Largely because of the different way refinement paths are used in eventB, the latter can develop a more abstract approach to modelling. This can take a bit of getting used to, but in my view leads, more naturally, to more satisfactory finer-grained modelling.

In the following discussion of eventB, eventB tends to be identified with the language supported by the Rodin toolkit, although there is a separate language definition for eventB itself. In practice, anything that is missing from the Rodin toolkit is missing from the language as the toolkit plays an important role in system modelling.

3.2 Disadvantages

More abstract: as mentioned above eventB tends to lead, more naturally, to more abstract modelling developments. This will also be listed as an advantage, but it may make it more challenging for students. At the moment there is no experience.

No decomposition: while decomposition, the ability to break a model up into sub-machines, is part of the eventB language this has not yet been implemented in the Rodin toolkit. It is possible to develop separate machines, but it is not possible to show state composition using *includes* or *extends*. Unfortunately, support for decomposition is not scheduled for the near future.

Constructs and mathematics missing: the sequence type (*seq(X)*) is not supported and consequently sequence operations (*head*, *tail*, *concatenation*, *take*, *drop*, *etc*) are not supported. The absence of *seq* itself is not serious as it is modelled by a total function. It is possible to define appropriate constant functions to operate on sequences of a specific base type, but not generic sequences. It is not yet known how serious this omission is. The mathematical operators Σ, *iterate* and *closure* are not supported. Again, they can be defined as constant functions for specific types, but not generically.

Cannot refine to implementation: at the moment the Rodin toolkit does not support refinement to implementation, that is, to B0 in classicalB. In time, there will undoubtedly be a plug-in to enable this.

3.3 Advantages

More abstract: the natural tendency to more abstractness for eventB, while possibly having disadvantages, can lead to modelling in which detail is distributed across refinements. This can lead to less cluttered, clearer models. Apart from better modelling, this might enable students to see the advantages of modelling over rushing into implementation.

Modelling is not programming: despite being abstract, classicalB's *Abstract Machine Notation* (*AMN*) has a strong resemblance to a programming language and this can encourage students to try to "program". This is far less likely in eventB.

Ability to refine to code fragments: despite not having (at the moment) the capability to refine to an implementation (B0 in classicalB), it is very straightforward to refine to code fragments that are naturally implemented by recursion, or could be easily assembled into a code loop, for example. This makes eventB excellent for modelling algorithms.

Simpler structure: eventB is much leaner than classicalB. It does not have a separate conditional construct, only guards on events. While this may initially appears limiting, it makes for very spare event models, and effectively mitigates against complex specifications of events.

Better toolkit: the type of editing used within the Eclipse platform, while having some irritating aspects for more experienced users, does place greater control and assistance on editing the components of a machine, and this can help to reduce errors, especially for novices learning the syntax of eventB.

The toolkit responds to the commitment of changes of any part of a development by analysing and running the proof obligation generator and auto-provers, thus keeping the state of the development up-to-date.

Better provers: except for some cases the provers are much stronger and easier to use than the provers in classicalB toolkits, especially the BToolkit. While proof obligation discharge will remain a challenging activity, it is likely that students will make quicker progress using the proof facilities provided with this toolkit.

3.4 Adaptation to Event B

At the time of writing, we are planning to move to eventB in 2008. Clearly the strategy described in sec 2.5.2 will have to be modified given the differences discussed in sec 3.2. This is *work in progress*, but the current plan is as follows:

Conversion of eventB refinement to a class diagram: this will be done if possible with tools. The tool UML-B [11] may be able to be used, but this is not determined at the time of writing. There is a UML-B plug-in for the Rodin toolkit, but no experience has been obtained in its use in the required strategy.

Conversion to Java implementation: the class diagram will be converted to

Java, using as much automation as possible.

3.5 Expectations

Although the adaptation is still in an early stage of planning the following seem reasonable outcomes:

An improved experience: the Rodin toolkit will provide an improved experience, especially in the area of proof obligation discharge.

Improved understanding of modelling: considerable emphasis will be placed on using eventB itself to provide an improved experience of modelling.

Improved understanding of the formal to informal transition: whether this is done informally or otherwise students should have a better understanding of the mapping from eventB to, say, Java.

Greater confidence in continued use: the improved experience provided by the Rodin toolkit and the fact that the toolkit is multi-platform and freely available should instil a greater opportunity and willingness to use the method in the future.

4 Conclusion

We have discussed the integrated formal/informal approach we take to rigorous development and we discuss merits and shortcomings of the experience up until now using classicalB. We discuss our plans to adapt our scheme to eventB commencing in 2008 and conjecture on possible improvements from that move.

5 Acknowledgements

Many cohorts of students at UNSW have provided valuable experience with the implementation of the course strategies mentioned in this paper. Their contribution is gratefully acknowledged.

I am grateful to the anonymous reviewers, whose comments helped improve the discussion in this paper. All remaining shortcomings are the responsibility of the author.

References

[1] Jean-Raymond Abrial. $B^\sharp$: Towards a Synthesis between Z and B. In ZB2003 [13], pages 168–177.

[2] COMP2111. System Modelling and Design. http://www.cse.unsw.edu.au/~cs2111.

[3] Alexander Romanovsky et al. RODIN: Rigorous Open Development Environment for Complex Systems. Technical report, University of Newcastle upon Tyne, UK, 2004–2007. http://www.rodin.cs.ncl.ac.uk.

[4] The Joint Task force on Computing Curricula IEEE Computer Society and ACM. Computing curriculum — software engineering. Technical report, IEEE Computer Society & ACM, 2004. http://sites.computer.org/ccse/volume/Draft3.1-2-6-04.pdf.

[5] B-Core(UK) Ltd. B Toolkit. http://www.b-core.com.

[6] C Neville Dean and Raymond T Boute, editors. *Teaching Formal Methods*, volume 3294 of *Lecture Notes in Computer Science*. CoLogNET/FME, Springer, November 2004.

[7] Rodin platform. http://rodin-b-sharp.sourceforge.net/.

[8] Ken Robinson. Embedding Formal Methods in Software Engineering. In Neville Dean and Raymond T Boute [6], pages 203–213.

[9] SENG2010. Software Engineering Workshop 2A. http://www.cse.unsw.edu.au/~se2010.

[10] SENG2020. Software Engineering Workshop 2B. http://www.cse.unsw.edu.au/~se2020.

[11] Colin Snook and Michael Butler. UML-B: Formal Modeling and Design Aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[12] Steria. Atelier B. http://www.atelierb.societe.com/.

[13] *3rd International Conference of B and Z Users*, volume 2651 of *Lecture Notes in Computer Science*. Springer, 2003.

EB: A constructive approach for the teaching of data structures

M. Guyomard[1,2]

Enssat/Irisa, Université de Rennes 1, BP 80518, 6 rue de Kerampont
22305 Lannion, France

Abstract

We present an innovative approach for the teaching of data structures. This approach, based on: 1) a set-oriented specification of abstract and concrete versions of data structures; 2) a functional specification of abstract and concrete operations and 3) a functional refinement, allows the derivation of the functional representation of the operations.

Keywords: Formal methods, data structures, algorithms, specification, refinement, homomorphism, B method, sets, functional design, derivation of programs, induction, AVL trees.

1 Introduction

The present situation of teaching formal methods at the heart of degree courses in computer science is strange. In practice, when such a teaching module exists it is put side by side with traditional modules, which include operating systems, data structures, databases, and algorithms, etc. whereas, by virtue of its status, such a course should enrich these other branches of computer science by the transfer of ideas and tools suitable for raising them to the level of real scientific disciplines.

In this article we try to show that there is at least one area where the collision with the world of formal methods can already be fruitful, namely that of teaching data structures.

Studying data structures and the algorithms attached has always been part of the basic training of the computer scientist. All the same, even in the best known or most frequently cited books ([3] for example) and whatever their other merits, the authors often limit themselves to listing the usual representations, having one chapter for stacks, another for linked lists, a third for queues and so on. This type of structure can be criticised on several levels:

[1] Thanks to Henry Hicks

[2] Email: Marc.Guyomard@enssat.fr

- It maintains the confusion between abstract type, or specification, and concrete type. For us it is necessary to distinguish between at least two levels: firstly the abstract types in which are of interest (for example subsets of scalar quantities, stacks, queues, priority queues, flexible arrays); secondly their implementation using lower level types (linked lists, binary search trees, tries, B-trees, position trees, heaps, etc.).
- These authors are forced to do without the refinement relation which links a concrete type to an abstract one which acts as a reference, while ideally this relation should be used to guide the discovery and even the derivation of the operations, and at the same time to prove that these operations fulfil their specifications.

Moreover, those authors who busy themselves showing that the concrete type, considered as an invariant, remains satisfied after the execution of an updating operation are rare.

Yet solutions exist which allow us to overcome these three drawbacks. One of them succeeds in a particularly elegant way in as much as these problems emerge naturally during the course of development and are resolved interdependently. It is a question of EB approach whose principles are presented in this article. This approach results from a synthesis of two schools of thought which are concerned with formal methods: that pioneered by V.J. Dielissen and A. Kaldewaij and by R. Hoogerwoord at Eindhoven on the one hand [4,6], and that which is advocated by J.-R. Abrial [1] (namely B method) on the other.

The second chapter presents the Dutch method as well as B method. Chapter three gives an example of the data structure which serves as a link with the rest of the article: the finite subsets of naturals, defined by extension. A first implementation is developed which allows drawing up the principles of the EB approach [3]. Chapter four suggests a second, more ambitious implementation using AVL trees as concrete structures. Finally our conclusions are in chapter five.

2 Functional refinement and set-oriented specification

The necessary parts of the EB approach come from the Dutch school and, in particular from [4,6] and for the smaller part, from B method [1].

2.1 The Dutch School

In [6], R. Hoogerwoord considers the implementation of flexible arrays by Braun trees. In an introductory example the author shows how it is possible to implement bags insertion by Braun trees. He emphasises particularly the following points:

- The distinction between abstract structures (the bags) and concrete structures (the Braun trees).
- The definition of a function between the two structures. This function explains how a Braun tree represents a bag. We will see in the following chapter that this function behaves like a refinement function and defines a form of homomorphism between the two structures.

[3] EB: stands for Eindhoven-B method, in honour of these two schools of thought.

Table 1
Some B notations

$(\ ,\ )$	:	ordered pair	$\{\ldots\}$	:	set defined by extension
$\rightarrow$	:	total function	$\leftarrow$	:	overriding a relation by another
$\nrightarrow$	:	partial function	$\lhd$	:	domain restriction of a relation
$\rightarrowtail$	:	total injection	$[\ldots]$	:	image of a set given by a relation
$\twoheadrightarrow$	:	total surjection	$\mathbb{F}(e)$	:	set of finite subsets of e

- The functional nature of the operations defined on the structures.

The author shows how, by calculation, it is possible to derive the functional representation of the operations. The guarded command conditional statements [5] are used systematically.

The article of V.J. Dielissen and A. Kaldewaij [4] takes up the same principles which are applied equally to the implementation of flexible arrays, but this time from the leaf trees.

2.2 B method

B method [1] allows, from the specification of a "machine", to reach, by successive refinements of the data and control structures, an implementation which is consistent with the specification. We emphasises the following points:

- The specification includes an invariant which each operation must preserve.
- The set theory, extended to the arithmetic, allows the specification of the data, and of the relationships that hold among them (invariants, parameters, etc.).
- The refinement of the data is relational (and not functional as above).

The EB approach uses the set notations of B method. Those notations which are used in this article are listed in table 1.

3 Principles of EB approach – example

The example which is the linking thread of our presentation is that of finite subsets of $\mathbb{N}$ defined by extension, on which three operations are allowed. As a first stage we specify the type SET in question. Figure 1 shows this specification. The main heading gives the name of the type (SET), the name of the set support (*set*, which represents the set of the various values which can be taken by an entity of type SET), the list of internal operations (that is the list of operations which return a value of the type in question): *clear* and *insert*, and finally the list of external operations (reduced here to $isIn$). The heading **uses** lists the auxiliary types necessary for the specification of type SET ($\mathbb{N}$ and *bool* here). The heading **support** gives a definition of the support. Here we assert that "to say that e is a finite subset of naturals is equivalent to saying that e belongs to the set *set*". The heading **operations** takes the operations listed in the main heading giving their names and their parameters'name and the category of the function represented (thus *insert*

is a partial function, while $isIn$ is a total function). Also given are the possible precondition, including the type of the parameters ($insert$ demands that the value v to be inserted be absent from the set e) and finally the specification itself, expressed by a "functional equation" of which the right-hand side is a formula of the type returned by the function.

abstractType SET= $(set, (clear, insert), (isIn))$
uses $bool, \mathbb{N}$
support
$e \in \mathbb{F}(\mathbb{N}) \quad \Leftrightarrow \quad e \in set$
operations
1) **name:** $clear() \rightarrow set$
spec: $clear() \quad = \quad \varnothing$
2) **name:** $insert(v, e) \nrightarrow set$
pre: $v, e \in \mathbb{N} \times set \wedge v \notin e$
spec: $insert(v, e) \quad = \quad e \cup \{v\}$
3) **name:** $isIn(v, e) \rightarrow bool$
pre: $v, e \in \mathbb{N} \times set$
spec: $isIn(v, e) \quad = \quad (v \in e)$
end

Fig. 1. Definition of abstract type SET

Once the specification has been completed, we can envisage one or several implementations. In our example we offer two types of implementation. The first, which is going to be used to present the principles of EB approach, is founded on the representation of a subset by an array holding the values belonging to the subset. The second, more ambitious, considers AVL trees as the concrete support (cf. chap. 4). The array implementation is shown in figure 2 (except the representation of the operations whose calculations are detailed below).

The main heading includes information similar to that in figure 1. m, parameter of type SETA, corresponding to the physical length of the array, is declared here. The heading **support** needs more explanation. It states that all [4] ordered pairs (t, n) such that:

- t is a integer array defined on the interval $1..m$,
- n is an integer belonging to the interval $0..m$ (n is going to represent the logical length of t),
- the sub-array of t restricted to the interval $1..n$ is without duplicate (this is captured by the fact that it is an injective function, symbolised by $\rightarrowtail$),

is an element of $setA$ and vice versa. A new heading appears here, it is called **refinementFunction**. This defines the refinement function rf specifying how each element of $setA$ represents an element of set. rf is a total function, from $setA$ to set [5]. The definition of function rf states that the subset represented by an ordered

[4] Variables are implicitly quantified universally.

[5] It is total: every element of $setA$ represents a subset set, non surjective: subsets exist which can not be represented by this refinement (the array only represents the subsets the cardinal of which is $\leq m$) and non injective: a given subset can be represented by several different arrays.

type SETA$(m \in \mathbb{N}^*) = (setA, (clear_a, insert_a), (isIn_a))$
refines SET
uses $bool, \mathbb{N}$
support
$t \in 1..m \ \rightarrow \ \mathbb{N} \ \wedge \ n \in 0..m \ \wedge \ (1..n) \lhd t \in 1..n \ \rightarrowtail \mathbb{N} \ \Leftrightarrow \ (t, n) \in setA$
refinementFunction
$rf \in setA \ \rightarrow \ set$
$rf((t, n)) = t[1..n]$
operations
1) **name:** $clear\_a() \rightarrow setA$
spec: $rf(clear\_a()) \ = \ clear()$
2) **name:** $insert\_a(v, (t, n)) \nrightarrow setA$
pre: $v, (t, n) \in \mathbb{N} \times setA \wedge v \notin rf((t, n)) \wedge n \in 0..m - 1$
spec: $rf(insert\_a(v, (t, n))) \ = \ insert(v, rf((t, n)))$
3) **name:** $isIn\_a(v, (t, n)) \rightarrow bool$
pre: $v, (t, n) \in \mathbb{N} \times setA$
spec: $isIn\_a(v, (t, n)) \ = \ isIn(v, rf((t, n)))$
end

Fig. 2. Array implementation of abstract type SET

pair (t, n) is the image of the interval $1..n$ given by t.

The heading **operations** takes up the different operations appearing in the main heading. The keyword **pre,** when it exists, allows the formulation of the precondition expressed through the refinement function. Turning to the heading **spec,** we must distinguish the external operations from the internal. For a concrete internal operation $io\_c(\ldots)$ corresponding to the abstract internal operation $io(\ldots)$, if rf is the refinement function, the specification takes the form $rf(io\_c(\ldots)) = io(\ldots)$ where the arguments of the concrete type are converted into abstract type by the application of the refinement function (cf. $insert/insert\_a$). Thus the refinement function behaves as a sort of homomorphism. The case of external operations is simpler since only the arguments are possibly converted by the refinement function.

3.1 Calculation of concrete operations

The concrete type SETA shown in figure 2 is, however, incomplete: the *concrete representation* of operations is missing. This can, in general, be obtained by calculation from 1) the definition of abstract and concrete support, 2) the refinement function and 3) the specification of abstract and concrete operations. In general this calculation boils down to a set of guarded equations of the form $[guard \rightarrow]co\_c(\ldots) = exp$, where co_c is a concrete operation and exp an expression of the type returned by the operation co_c. The actual calculation consists, for internal operations such as io_c, in searching for, under the hypothesis of a possible guard, an expression exp such that $rf(io\_c(\ldots)) = rf(exp)$. Once this result is obtained, Leibniz's axiom (if f is a function which can be applied to a and b, then $a = b \Rightarrow f(a) = f(b)$) allows it to end by $[guard \rightarrow]io\_c(\ldots) = exp$.

External operations eo_c are treated in an analogous manner, but simply by

searching for an expression exp such that $eo\_c(\ldots) = exp$.

We are applying these principles to two operations $insert\_a$ and $isIn\_a$. Let us start with $insert\_a(v,(t,n))$. Here are the calculation details:

$$
\begin{array}{ll}
 & rf(insert\_a(v,(t,n))) \\
= & \quad \text{- - specification of } insert\_a \\
 & insert(v, rf((t,n))) \\
= & \quad \text{- - definition of } rf \\
 & insert(v, t[1..n]) \\
= & \quad \text{- - specification of } insert \\
 & t[1..n] \cup \{v\} \\
= & \quad \text{- - set theory, definition of } \lhd, \; n \in 0..m-1 \\
 & (t \lhd \{(n+1, v)\})[1..n] \\
= & \quad \text{- - definition of } rf \\
 & rf((t \lhd \{(n+1, v)\}, n+1))
\end{array}
$$

From where, by Leibniz, we obtain the equation:

$$insert\_a(v,(t,n)) = (t \lhd \{(n+1, v)\}, n+1)$$

We are now able to provide a much traditional version of the operation:

function $insert\_a(v,(t,n)) \nrightarrow setA \mathrel{\widehat{=}}$
pre
$v \in \mathbb{N} \quad \wedge \quad (t,n) \in setA \quad \wedge \quad n \in 0..m-1 \quad \wedge \quad v \notin t[1..n]$
then
$(t \lhd \{(n+1, v)\}, n+1)$
end

which can be paraphrased as: the result of the insertion of the value v in the set represented by the ordered pair (t,n), under the precondition hypothesis, is a subset represented by an ordered pair identical to the first except that the position $n+1$ of the array is significant and indicates the value v. Now let us look at the functional representation of the operation $isIn\_a$:

$$
\begin{array}{ll}
 & isIn\_a(v,(t,n)) \\
= & \quad \text{- - specification of the operation } isIn\_a \\
 & isIn(v, rf((t,n))) \\
= & \quad \text{- - specification of } isIn \text{ and definition of } rf \\
 & v \in t[1..n]
\end{array}
$$

To continue the development requires the formulation of hypotheses on the value of n ($n \in 0..m \Rightarrow (n = 0 \vee n \in 1..m)$ $(m \in \mathbb{N}^*)$). Let us consider the first case $n = 0$:

$$
\begin{array}{ll}
 & v \in t[1..n] \\
= & \quad \text{- - by hypothesis } n = 0 \\
 & v \in t[1..0] \\
= & \quad \text{- - set theory} \\
 & v \in \varnothing \\
= & \quad \text{- - set theory} \\
 & \textbf{false}
\end{array}
$$

From where we obtain the first guarded equation $n = 0 \rightarrow isIn\_a(v,(t,n)) =$

false. The second case makes clear the fact that under the hypothesis $n \neq 0$, $1..n \;=\; 1..n-1 \;\cup\; n..n$:

$$
\begin{array}{ll}
 & v \in t[1..n] \\
= & \quad \text{- - by hypothesis } n \neq 0 \\
 & v \in t[1..n-1] \;\; \vee \;\; v = t(n)
\end{array}
$$

We carry on performing a case analysis, starting with $v = t(n)$:

$$
\begin{array}{ll}
 & v \in t[1..n-1] \;\; \vee \;\; v = t(n) \\
= & \quad \text{- - case } v = t(n) \\
 & \textbf{true}
\end{array}
$$

From where we obtain the guarded equation:

$$
\begin{array}{l}
n \neq 0 \rightarrow \\
\quad v = t(n) \rightarrow isIn\_a(v,(t,n)) = \textbf{true}
\end{array}
$$

And if $v \neq t(n)$:

$$
\begin{array}{ll}
 & v \in t[1..n-1] \;\; \vee \;\; v = t(n) \\
= & \quad \text{- - case } v \neq t(n) \\
 & v \in t[1..n-1] \\
= & \quad \text{- - definition of } rf \\
 & v \in rf((t,n-1)) \\
= & \quad \text{- - specification of } isIn \\
 & isIn(v, rf((t,n-1))) \\
= & \quad \text{- - specification of } isIn\_a \\
 & isIn\_a(v,(t,n-1))
\end{array}
$$

From where we obtain the guarded equation:

$$
\begin{array}{l}
n \neq 0 \rightarrow \\
\quad v \neq t(n) \rightarrow isIn\_a(v,(t,n)) = isIn\_a(v,(t,n-1))
\end{array}
$$

We are now able to bring together the results of the calculation, so as to provide a traditional representation of $isIn\_a$:

function $isIn\_a(v,(t,n)) \rightarrow setA \;\widehat{=}$
pre
  $v \in \mathbb{N} \wedge (t,n) \in setA$
then
  if $n = 0 \rightarrow$
    false
  | $n \neq 0 \rightarrow$
    if $v = t(n) \rightarrow$
      true
    | $v \neq t(n) \rightarrow$
      $isIn\_a(v,(t,n-1))$
    fi
  fi
end

We can now produce a balance sheet, listing the criticisms along with the way

we overcome them:

- Confusion between specification and implementation: we have a clear separation between these two levels.
- Doubt that the concrete operation fulfils its specification: this disappears here since the starting point of the calculation is the specification.
- Doubt that the concrete types, considered as an invariant for the internal operations, are preserved. The approach manages naturally to maintain the type of the result since the calculations are typed.

3.2 Towards an approach to design and refine data structures

The approach that we have adopted below is sufficiently general to be presented to students as a list of points to be tackled successively in order to obtain a concrete implementation of an abstract data structure. A first suggestion could be: 1) specify in a formal way the abstract type under consideration; 2) choose a concrete support and define it formally (this stage is as crucial for the success of the undertaking as for the efficiency of the result); 3) define the refinement function; 4) specify formally the operations; and 5) calculate the functional representation of the operations (this is a key stage in which a large part of the calculations is trivial but which demands critical decisions).

At the end of such a development we have available a concrete type in which each operation is represented by a function. We have therefore several possibilities. We can implement the operations directly in a functional language. The discipline to be followed is naturally that of "never modifying an existing structure", which lends itself well to linked structures, but which is often too expensive if, as in the example above, it is concerned with arrays. At the cost of a final stage of program transformation (cf. [6]), a discipline of "modification *in situ*" can be applied. Another type of transformation can sometimes be easily used: if the algorithm is tail-recursive, we can substitute a loop for the recursion (cf. [2,6]).

In the following chapter we apply the EB approach to the refinement of abstract data type SET (cf. fig. 1) by AVL trees.

4 From sets to Avl trees

Discovered in 1962 by G.M. Adelson-Velskii and E.M. Landis, the form of the binary search trees known by the name AVL trees has been considered since then as the archetype of balanced trees. Although other balanced types compete (B-trees, semi-balanced trees, etc.), the AVL tree is often found in the catalogue of data structures given to students of computer science. The basic principle of the AVL tree is to keep the following balance principle invariant under all the nodes of the tree: "the difference in heights between the left sub-tree and the right sub-tree must never exceed 1 in absolute value".

Traditionally, the implementation of update operations in AVL trees is carried out according to the rule "modification *in situ*" in integrating in each node not the height but the *difference* in heights. At the time of an insertion, one output parameter indicates whether or not an increase in height has occurred. If this is

the case and if this has led to a violation of the balance property, the tree must be re-balanced by simple or double rotation.

Here we are using a different approach, which avoids making premature choices. This is particularly true for the way the height is calculated [4]. The result is an insertion algorithm, *insert_avl*, different from the traditional one[6], in so far as, thanks to its functional nature, it is possible to know simultaneously the height of a tree before and after insertion (the output parameter mentioned above becoming unnecessary).

4.1 Choice of concrete support

The definition of concrete support is done in three stages. An Avl tree being a binary search tree satisfying a constraint linked to its height, we are going to define successively 1) the binary search trees (*bst* set), 2) the height of such trees (function h) and 3) the Avl trees[7] (*avl* set).

1) $\langle\rangle \in bst$
2) $r \in \mathbb{N} \ \wedge \ g \in bst \ \wedge \ d \in bst \ \wedge \ \mathsf{max}(rf(g)) < r \ \wedge \ \mathsf{min}(rf(d)) > r \ \Rightarrow$
   $\langle g, r, d\rangle \in bst$
3) law of closure for *bst*[8]

rf is the refinement function defined below. The height function (h) is defined by:

$h \in bst \rightarrow \mathbb{N}$
1) $h(\langle\rangle) = 0$
2) $h(\langle g, r, d\rangle) = \mathsf{max}(\{h(g), h(d)\}) + 1$

Finally an Avl tree is defined inductively by:

1) $\langle\rangle \in avl$
2) $\langle g, r, d\rangle \in bst \wedge g \in avl \wedge d \in avl \wedge h(g) - h(d) \in -1..1 \ \Rightarrow \langle g, r, d\rangle \in avl$
3) law of closure for *avl*

4.2 Definition of the refinement function

As a result of an update, a tree may not be an Avl tree, consequently the refinement function rf does not have as its definition domain *avl* but *bst*:

$rf \in bst \ \twoheadrightarrow set$
1) $rf(\langle\rangle) \ = \ \varnothing$
2) $rf(\langle g, r, d\rangle) \ = \ rf(g) \cup \{r\} \cup rf(d)$

4.3 Specification of the operation insert_avl

The description of the different headings of the operation *insert_avl* are:

name: $insert\_avl(v, a) \nrightarrow avl$
pre: $v, a \in \mathbb{N} \times avl \wedge v \notin rf(a)$

---

[6] However, of the same running time ($O(\log(n))$).

[7] These definitions assume the availability of the operations min and max which return, respectively, the smallest and the biggest of a possibly empty set (in particular $\mathsf{min}(\varnothing) = +\infty$ and $\mathsf{max}(\varnothing) = -\infty$).

[8] This rule stipulates that every element of the set *bst* is constructed with the aid of rules 1) and 2) applied a finite number of times.

spec: $rf(insert\_avl(v,a)) \quad = \quad insert(v, rf(a))$

4.4 Calculation of the functional representation of insert_avl

The calculation of the operation $insert\_avl(a,v)$ is done by induction on the structure of a. A first attempt should lead to the realisation that two extra inductive hypotheses must be introduced in order to achieve the development.

Theorem 4.1 *Inductive hypothesis ($Aug(a)$). Inserting an element v into an* AVL *tree a produces an increase in height of at most one:*

$$Aug(a) \;\widehat{=}\; h(insert\_avl(v,a)) - h(a) \in \{0,1\}$$

Theorem 4.2 *Induction hypothesis ($Des(a)$). If the insertion of a value v into a non-empty* AVL *tree a produces an increase in height, then the new balance is either -1 or 1:*

$$\begin{array}{l} Des(a) \;\widehat{=}\; \\ \quad a \neq \langle\rangle \;\wedge\; h(insert\_avl(v,a)) - h(a) = 1 \\ \Rightarrow \\ \quad \textit{Letting } insert\_avl(v,a) = \langle g', r', d' \rangle \\ \quad h(g') - h(d') \in \{-1, 1\} \end{array}$$

These two hypotheses must be proved *a posteriori* for each stage of calculation. The rest of the calculation of the representation of *insert_avl* is simply sketched. There are two cases to consider: $a = \langle\rangle$ and $a \neq \langle\rangle$.

In the case where $a = \langle\rangle$ it can easily be shown that $insert\_avl(v,a)) = \langle\langle\rangle, v, \langle\rangle\rangle$; and that $Aug(\langle\rangle)$ and $Des(\langle\rangle)$ obtains. Which gives us the first rule (l. 5-6, p. 35).

For the case where $a \neq \langle\rangle$, once the formula $rf(g) \cup \{r\} \cup rf(d) \cup \{v\}$ has been obtained, the two cases $v > r$ and $v < r$ must be considered. It is possible to produce only one calculation ($v < r$ for example) and then to take advantage of the symmetry of the situations to deduce the second. Once the formula $rf(insert\_avl(v,g)) \cup \{r\} \cup rf(d)$, has been obtained, the induction hypothesis $Aug(g)$ can be used to conclude that we only need to consider the two following cases:

(i) $h(insert\_avl(v,g)) - h(g) = 0$ (the height of the left sub-tree does not rise)

(ii) $h(insert\_avl(v,g)) - h(g) = 1$ (the height of the left sub-tree grows by 1).

In the first case it is easy to derive the following rule (l. 7-11, p. 35):

$$\begin{array}{l} a \neq \langle\rangle \;\rightarrow \text{Letting } a = \langle g, r, d \rangle \\ \quad v < r \;\rightarrow \text{- - left insertion} \\ \quad\quad h(insert\_avl(v,g)) - h(g) = 0 \;\rightarrow \\ \quad\quad\quad insert\_avl(v,a) \;=\; \langle insert\_avl(v,g), r, d \rangle \end{array}$$

and to prove $Aug(a)$ and $Des(a)$. In the second case (the height of the left sub-tree grows by 1), we have three cases of balance to consider: $h(g) = h(d), h(g) = h(d) - 1$ and $h(g) = h(d) + 1$. The first two cases are treated in a similar way while the last is based on the induction hypothesis $Des(\langle g_g, r_g, d_g \rangle)$ to limit the analysis to two cases (l. 18, l. 20) which is treated respectively by a simple rotation (l. 19) and by a double rotation (l. 21). In all cases it is easy to prove $Aug(a)$ and $Des(a)$. Finally

we obtain the following monolithic version:

```
01  function insert_avl(v, a) ↛ avl ≙
02  pre
03      v ∈ ℕ ∧ a ∈ avl ∧ v ∉ rf(a)
04  then
05      if a = ⟨⟩ →
06          ⟨⟨⟩, v, ⟨⟩⟩
07      | a ≠ ⟨⟩ →
08          Letting a = ⟨g, r, d⟩
09          if v < r → - - left insertion
10              if h(insert_avl(v, g)) − h(g) = 0 → - - the height does not rise
11                  ⟨insert_avl(v, g), r, d⟩
12              | h(insert_avl(v, g)) − h(g) = 1 → - - the height grows by 1
13                  if h(g) = h(d) ∨ h(g) = h(d) − 1 →
14                      ⟨insert_avl(v, g), r, d⟩
15                  | h(g) = h(d) + 1 →
16                      Letting insert_avl(v, g) = ⟨g_g, r_g, d_g⟩
17                      Letting d_g = ⟨g_{g_d}, r_{g_d}, d_{g_d}⟩
18                      if h(g_g) = h(d_g) + 1 → - - right rotation
19                          ⟨g_g, r_g, ⟨d_g, r, d⟩⟩
20                      | h(g_g) = h(d_g) − 1 → - - left-right rotation
21                          ⟨⟨g_g, r_g, g_{g_d}⟩, r_{g_d}, ⟨d_{g_d}, r, d⟩⟩
22                      fi
23                  fi
24              fi
25          | v > r → - - right insertion
⋮               ⋮
41          fi
42      fi
43  end
```

At this stage we have still not taken any decision about the calculation of function h. The following stage would consist, in particular, of adding a field h' to the representation of nodes in such that the value of the function h can be put there, which is, if necessary, calculated into field h' from the two sons. It is also a good idea to merge the multiple identical calls to *insert_avl* into a single one.

Thus we obtain a new version, based on calculation as well as on motivated decisions, of a non trivial algorithm. We must, however, warn students by telling them that in industry, the strategy "never modify an existing structure" must be backed up, in order to be viable, by an efficient garbage-collector.

5 Conclusion

We have presented the EB approach which, applied to the teaching of data structures, give the advantage of allowing the calculation of operations of a data structure incrementally from its specifications while justifying each decision. The example of

the abstract data structure "finite subset of naturals" is stated in two different ways (arrays and Avl trees). This approach is the result of a synthesis between, on the one hand, the set notations offered by B method and on the other hand the Dutch approach. This is shown to be more constructive than the first in this context, and, thanks to the elegance and power of the B set notations, its field of application is not limited to inductive structures, unlike the second.

Our work in this area is recent and we do not have any teaching results to present. We are, however, convinced that the eb approach can contribute to a renewing of the teaching of data structures. The next steps will consist of introducing the eb approach into the teaching of data structures at Enssat, and to study how far it is possible to make use of a prover to facilitate the task of the developer.

References

[1] Abrial, J.-R., "The B-Book," Cambridge University Press, 1996.

[2] Cohen, E., "Programming in the 1990s, an Introduction to the Calculation of Programs," Springer-Verlag, New York Inc., 1990.

[3] Cormen, T., Ch. Leiserson, R. Rivest, C. Stein, "Introduction to Algorithms," 2nd Ed., MIT Press, McGraw-Hill Book Company. 2003.

[4] Dielissen, V.J., A. Kaldewaij, *A simple, efficient, and flexible implementation of flexible arrays,* Lecture Notes in Computer Science, Mathematics of program Constructions. **947** (1995). 232-241.

[5] Dijkstra, E.W., W.H.J. Feijen, "A Method of Programming," Addison-Wesley, 1988.

[6] Hoogerwoord, R., *A Logarithmic Implementation of flexible Arrays.* Mathematics of Program Construction: Second International Conference, Oxford U.K. June 29-July 3. (1992). Proceedings: R.S. Bird, C.C. Morgan, J.C.P. Woodcock, Eds. LCNS 669, Springer-Verlag, Berlin Heidelberg (1993). 191-207.

Assisted Calculational Proofs and Proof Checking Based on Partial Orders

Jaime Bohórquez[1]

Departamento de Ingeniería de Sistemas
Escuela Colombiana de Ingeniería
Bogotá, Colombia

Camilo Rocha[2]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61820, USA

Abstract

The ability to effectively prove theorems, by both human and mechanical means, is crucial to formal methods. An open problem in the *Computer Mathematics* community is the feasibility to fully formalize mathematical proofs. Here, feasibility is understood as the capability to generate correct formal mathematics with an effort comparable to that of writing a mathematical paper in, say, LaTeX. In this paper we investigate a way to solve this problem. Dijkstra's *calculational style*, a deduction method based on formula manipulation, is a promise for easy and tractable formalization and mechanization of proofs in which the use of equational reasoning (understood as mainly based on the preeminence of logical equivalence and equalities) is preferred over the traditional one based on logical implication. We have used *Calculational Logic*, the logic following the calculational style, in discrete mathematics and algorithms design courses for the past eight years. In these courses, effective heuristics to calculate and check proofs are emphasized with rewarding results. We briefly present Calculational Logic and illustrate its use, through the concepts of indirect equality and inequality, with calculi for abstract relation algebra and number theory. We also present *Proof Star*, the first version of an experimental interactive proof assistant and proof checker for Calculational Logic. We corroborate that Proof Star is a reliable proof assistant and proof checker of computer mathematics by comparing its main characteristics with the ones required by H. Barendregt [2].

Keywords: Calculational Logic, assisted proofs, proof checking, partial orders, proof theory, Proof Star

1 Introduction

The problems concerning formal methods, that is, the correctness and modularity of programs, appeal in a deep way to formula manipulation, namely, for calculating and checking proofs. As a matter of fact, formal mathematics and its view of proof is largely a 20^{th} century invention having the computer as its best ally: it consists of definitions, statements and proofs having such a complete level of detail, that

[1] Email: jbohorqu@escuelaing.edu.co
[2] Email: hrochan2@cs.uiuc.edu

its correctness (relative to a context in which the primitive notions and axioms are introduced) can be mechanically obtained or verified by a computer.

In recent years, researchers have gained considerable experience with computer systems for checking mathematics in general. These systems are mainly based on type theory [29,35], set theory [28,38], and higher order logic [21,30,27]. In many modern proof-development environments, proofs are obtained in an interactive fashion between user and proof-checker systems. Therefore, one refers to *interactive mathematical assistants* or *proof assistants*. It is important to distinguish the activity of *theorem proving* from the one of *proof checking*, the latter is not expected to invent proofs, but rather to verify whether some input is correct or not, according to the underlying mathematics [11]. Since at present it is mathematically not appealing to construct formal proofs, a crucial open problem is the feasibility to fully formalize mathematical proofs. In this context, feasibility is meant in a sense more strict than that of computer science: a mathematician should be able to generate correct formal mathematics with an effort comparable to writing a mathematical paper in, say, LaTeX [3].

Logicians predict that computer databases of mathematical knowledge will contain, organize, and retrieve most of the known mathematical literature by 2030 ± 10 years, as a prospect for the twenty-first century [9]. We might not have to wait that long. For more than twenty years now, a revolution on the way of reasoning and proving in mathematics (largely unnoticed by pure logicians) has gained a substantial community of enthusiastic practitioners (see, for example, `http://www.mathmeth.com/`). Dijkstra's *calculational style* is a deduction method based on formula manipulation, originally devised as an informal but rigorous and practical theorem proving discipline, in which the use of equational reasoning (understood as mainly based on the preeminence of logical equivalence and equalities) is preferred over the traditional one based on logical implication [14,12]. This calculational style has led to an extensive array of techniques for elegant proof constructions that we believe are formalizable and mechanizable. As a matter of fact, outstanding members of the Computer Mathematics community have expressed disappointment to the calculational community for not having considered mechanical theorem proving seriously, exploiting the close connection between calculational and mechanical proofs [23].

Some progress in the direction of formalization and mechanization has been made in the calculational community. Calculational proof methods were formalized for classical predicate logic by Gries and Schneider [18,17] and, subsequently, streamlined in [19] and also by Tourlakis [36,37] and Lifschitz [22]. R. Boute [8] has extended it to 'functional predicate calculus' smoothly conjoining discrete and continuous applied mathematics. The authors have also accomplished related results: an analogous approach for the case of the Intuitionistic Predicate Logic has been developed by the first author, while the second author has successfully mechanized in Rewriting Logic [25,5] decision procedures for the Propositional Logic and the Syllogistic Logic with Complements, and a semi-decision procedure for the first order case [32], vía the so-called 'theorem proving modulo' approach [15]. We give a brief presentation of *Calculational Logic*, the logic inspired by the calculational approach, by emphasizing its proof system and proof format, in Section 2.

In this paper we draw particular attention to the teaching of proof theory within Calculational Logic. In order to illustrate some of its effective proof methods and elegant results, we develop a case study based on *partial orders*. These have been used extensively by the authors for the past eight years with rewarding results in discrete mathematics and in algorithm design and verification courses. By means of a partial order, we characterize the concepts of *indirect inequality* and *indirect equality* in Section 3. We present calculi for *Abstract Relation Algebra* and *Number Theory* in Section 4, as further evidence of the power of calculational deduction within the setting of partial orders. In Section 5, we present *Proof Star*, an experimental interactive proof assistant for Calculational Logic that provides the user with an approachable interface for calculational proofs, definitions, and computations in a way inspired by modern document processors. We corroborate that Proof Star is a reliable proof assistant and proof checker of computer mathematics by an ample fulfillment of the list of properties required by H. Barendregt [2] for formal support systems. Finally, in Section 6, we present some concluding remarks and future work.

We believe that altogether the methods, approaches, and Proof Star take a step further in the road of mechanizing Calculational Logic and, at the same time, provide encouraging evidence of the great importance of this logic as an effective tool for formal methods practice.

2 Calculational Logic

The computing scientist and the working mathematician demand simplicity and conciseness of expression. This is the basic principle of E.W.Dijkstra's development of *Calculational Logic* [14] which emphasizes the use of equational reasoning (in the sense explained above) rather than the traditional emphasis on logical implication.

Calculational Logic becomes an alternative formalization of classical predicate logic based on logical equivalence rather than on implication, and on textual substitution of logically equivalent subformulas rather than on modus ponens. The inference rules for calculational predicate logic are the *Leibniz Rules* (LR):

$$\text{LR} \quad \frac{E(p/A) \qquad A \equiv B}{E(p/B)} \qquad \frac{E(p/B) \qquad A \equiv B}{E(p/A)}$$

where A, B and P are formulas, and $\equiv$ is the Boolean equivalence operator. These are actually the only two (mutually symmetric) rules needed to derive arbitrary valid formulas from a small set of axiom schemas.

The proof format we use for this logic is formalized as a chain of derivations, encoding a non-null sequence of applications of Leibniz rules. Each item of the chain is a triple of formulas: the *initial member*, the *final member* and the *hint*. For every pair of consecutive items in a deductive chain, we have that the final element of the first coincides with the initial element of the second. Every hint is an equivalence. Two formulas in a deductive chain have special names: the initial member of the first triple is the *source* of the deductive chain, and the final member of the last triple is the *destination*. The hints from different instances of the axiom schemas are together called the *hypothesis* of the chain.

Deductive chains are represented as follows: the horizontal bar in a Leibniz rule is replaced by $\equiv$[3], and the hint $A \equiv B$ is written to the right of the equivalence within angular braces, as in $\langle A \equiv B \rangle$. The general format for a 'link' of the deductive chain is of the form:

$$\begin{array}{l} E(P/A) \\ \equiv \quad \langle\ A \equiv B\ \rangle \\ E(P/B) \end{array}$$

Accordingly, a deductive chain has a representation that looks like:

$$\begin{array}{l} P_1 \\ \equiv \quad \langle\ E_1\ \rangle \\ P_2 \\ \equiv \quad \langle\ E_2\ \rangle \\ \equiv \quad \cdots \\ P_n \end{array} \quad \langle\ E_{n-1}\ \rangle$$

If there exists a deductive chain with source P, destination Q, and hypothesis Γ, then $\Gamma \cup \{P\} \vdash Q$. Taking advantage of the symmetry among the Leibniz rules, and using the same arguments, it is also the case that $\Gamma \cup \{Q\} \vdash P$. Therefore, $\Gamma \vdash P \equiv Q$ and $\Gamma \vdash Q \equiv P$ by the Deduction Metatheorem, which justifies the abuse of notation mentioned above.

The deductive system can be enhanced with the Modus Ponens (MP) rule, extending the proof format by representing in deductive chains the horizontal bar of the rule with $\Rightarrow$. It is routine to show that MP can be derived from the original axioms and rules of Calculational Logic. If there exists an extended deductive chain with source P, destination Q and hypothesis Γ, where MP is used in the 'from P to Q' direction any non-null number of times, then $\Gamma \cup \{P\} \vdash Q$ and also $\Gamma \vdash P \Rightarrow Q$.

Other well known metatheorems, such as the Generalization Metatheorem, may be applied as a complementary methods for even a more streamlined proof system.

3 Indirect Inequality and Indirect Equality

E. W. Dijkstra, in many of his famous EWDs [13] (see also EWD1257, 1276, 1272, 1279, 1292 and 1299), hinted at the importance of lattice theory (and more generally, at the theory of preorders and partial orders) in terms of theorem proving techniques on topics of discrete mathematics. These include set theory, combinatorics, Boolean algebra, number theory, relation algebra as well as propositional and predicate logic. In the same way that his Calculational Logic made evident the fact that the congruence properties of logical equivalence had not been fully exploited, he also rightly pointed out that lattice theory was 'not half as well known as it deserved' [13]. In fact, the order-based rather than equational character of many aspects of the calculational style, including Calculational Logic, provides simple and yet powerful theorem proving methods. The lattice-theoretical concept of a *Galois connection* allows a useful characterization of the equality relation through what R. Backhouse [1] has called *indirect equality*:

Proposition 3.1 (Indirect inequality [13,7,1]) *Let $\sqsubseteq$ be a dyadic relation. The following propositions are equivalent:*

[3] Actually, an abuse of notation not causing confusion.

(i) $\sqsubseteq$ *is a preorder (reflexive and transitive relation)*

(ii) $a \sqsubseteq b \equiv \forall x.(x \sqsubseteq a \Rightarrow x \sqsubseteq b)$

(iii) $a \sqsubseteq b \equiv \forall x.(b \sqsubseteq x \Rightarrow a \sqsubseteq x)$

In fact, this theorem characterizes preorder relations. Partial orders (i.e. anti-symmetric preorders) have a stronger property as they characterize the associated equality relation.

Proposition 3.2 (Indirect equality [7]) *Let $\sqsubseteq$ be a partial order. The following are valid propositions:*

(i) $a = b \equiv \forall x.(a \sqsubseteq x \equiv b \sqsubseteq x)$

(ii) $a = b \equiv \forall x.(x \sqsubseteq a \equiv x \sqsubseteq b)$

In any structure endowed with a partial order, two elements are equal if and only if the corresponding sets of successors or predecessors coincide. At first glance, this 'transformation' seems to make things more complex. However, for many theories this transformation facilitates straightforward proofs. This is the case of homogeneous dyadic relations, as we present in the following section, where the common and traditional reasoning by elements is completely avoided. Moreover, the proofs obtained by this method are human readable, and can easily be mechanized and checked by a machine.

A more in depth treatment on the importance and use of preorders and partial orders, and its lattice counterpart, in proof theory for Calculational Logic, is illustrated in with more applications [7].

4 Lattice Theories and the Calculational Style

We proceed to illustrate the importance and use of preorders and partial orders for proof theory in Calculational Logic with two examples, one on dyadic homogeneous relations, and the other on Gödel codes.

The algebra of relations has played an important role in the specification and correctness of programs [20,26]. We use the expressive power of partial orders to illustrate how to prove basic relation algebra theorems using lattice-theoretical methods in a fluid and elegant way. Specifically, we give an axiomatization for the calculus of homogeneous dyadic relations. One of the main objectives of this section is to put in evidence the proximity of the calculus to both humans and machines in terms of readability and mechanization, respectively. We also aim for the effective use of the calculus to teach proof theory, not only in the relational setting but in the more abstract one of an arbitrary structure endowed with partial orders. A more in depth treatment is given in [6].

Capital letters Q, R, S and X will denote (dyadic homogeneous) relations. $\overline{R}$ and $R^{\smile}$ respectively denote complement and inverse of R. $R \circ S$ (often abbreviated by RS) denotes composition, $R \cup S$ union, $R \cap S$ intersection of R and S, and $R \subseteq S$ inclusion of R in S. Notations for the special relations are: $\mathbf{I}$ for the identity relation, $\mathbf{O}$ for the empty relation, and $\mathbf{L}$ for the universal relation. The set-theoretic operations union, intersection, and complement, the inclusion relation

and the constants **O** and **L** obey the axioms of a Boolean lattice. The operation of composition and the constant **I** obey the axioms of a monoid. Besides these axioms, we have the Schröder rules:

$$QR \subseteq S \;\equiv\; Q^{\smile}\overline{S} \subseteq \overline{R} \qquad \text{and} \qquad QR \subseteq S \;\equiv\; \overline{S}R^{\smile} \subseteq \overline{Q}\;.$$

This is basically Tarski's calculus of relations, where the membership operator is avoided [34,33]. We show the proof of a basic property on the above axiomatization.

Proposition 4.1 (Inverse is involutive) $(R^{\smile})^{\smile} = R$

Proof. By indirect equality is enough to prove $R \subseteq S \;\equiv\; (R^{\smile})^{\smile} \subseteq S$ for any S.

$$\begin{array}{ll} & R \subseteq S \\ \equiv & \quad \langle \text{ Identity } \rangle \\ & R\mathbf{I} \subseteq S \\ \equiv & \quad \langle \text{ Shröder rules } \rangle \\ & R^{\smile}\overline{S} \subseteq \overline{\mathbf{I}} \\ \equiv & \quad \langle \text{ Shröder rules } \rangle \\ & (R^{\smile})^{\smile}\overline{\overline{\mathbf{I}}} \subseteq \overline{\overline{S}} \\ \equiv & \quad \langle \text{ Complementation is involutive } \rangle \\ & (R^{\smile})^{\smile}\mathbf{I} \subseteq S \\ \equiv & \quad \langle \text{ Identity } \rangle \\ & (R^{\smile})^{\smile} \subseteq S \end{array}$$

□

Calculational Logic and its proof format are easy to teach and to follow, respectively, as illustrated with the proof of Proposition 4.1. The appropriate combination of indirect equality with the Shröder rules, led to a clean, short and straightforward proof. Observe that by providing a 'tag' for the property used in each proof step, not only a human can easily verify that the proof is sound, but also a mechanical procedure in the form of proof checker.

There is an interesting connection in theory of integers between two different lattices: one associated with the usual linear order relation of the integers together with the maximum and minimum operators ($\uparrow$ and $\downarrow$) [16], and the other associated with divisibility as an order relation, the greatest common divisor (gcd) and the least common multiple (lcm) operators (noted respectively $\Downarrow$ and $\Uparrow$).

In order to stablish a lattice-theoretical connection between multiplicative and additive arithmetic we proceed to define the concept of *Gödel code* of a positive integer which, with the help of indirect methods and the use of the Fundamental Theorem of arithmetic, will allow us to state a very interesting connection between these two theories.

Definition 4.2 [Primes and Gödel Codes] For a positive integer n, we define:

(i) The *Gödel number* of n (written $\overline{n}$) is a natural valued function defined on the set of prime numbers $\mathbb{P}$, such that for all $p \in \mathbb{P}$:

$$\overline{n}(p) = \langle \uparrow k \in \mathbb{N} \;:\; p^k \cdot | \, n \;:\; k \rangle$$

(i.e. the exponent of the maximum power of p that divides n).

(ii) Since the Gödel codes are functions, by a convenient abuse of notation we can provide them with a lattice structure based on their values. For n and m positive integers we define:

$(a)\ \overline{n} = \overline{m} \ \equiv\ \forall p.(p{\in}\mathbb{P} \Rightarrow \overline{n}.p = \overline{m}.p)$ $(b)\ \overline{n} \leq \overline{m} \ \equiv\ \forall p.(p{\in}\mathbb{P} \Rightarrow \overline{n}.p \leq \overline{m}.p)$
$(c)\ \overline{n} + \overline{m} = \lambda p{\in}\mathbb{P}.(\overline{n}.p + \overline{m}.p)$ $(d)\ \overline{n}\uparrow\overline{m} = \lambda p{\in}\mathbb{P}.(\overline{n}.p \uparrow \overline{m}.p)$
$(e)\ \overline{n}\downarrow\overline{m} = \lambda p{\in}\mathbb{P}.(\overline{n}.p \downarrow \overline{m}.p)$

The Fundamental Theorem of Arithmetic allows us to show that the lattice of Gödel codes is isomorphic to the one of multiplicative arithmetic.

Proposition 4.3 *For positive integers n and m,*

$(a)\ n = m \ \equiv\ \overline{n} = \overline{m}$ $(b)\ n\cdot|\ m \ \equiv\ \overline{n} \leq \overline{m}$ $(c)\ \overline{n \cdot m} = \overline{n} + \overline{m}$
$(d)\ \overline{n \Downarrow m} = \overline{n} \downarrow \overline{m}$ $(e)\ \overline{n \Uparrow m} = \overline{n} \uparrow \overline{m}$

Proof. Again 'Indirect Equality' properties are useful to prove these statements. We show how to prove (d). First observe that:

$\overline{n \Downarrow m} = \overline{n} \downarrow \overline{m}$
$\equiv$ ⟨ Function equality; definition of $\overline{n \downarrow m}$ ⟩
$\forall p \in \mathbb{P}.(\overline{n \Downarrow m}.p = \overline{n}.p \downarrow \overline{m}.p)$
$\equiv$ ⟨ Indirect equality with $\leq$ as preorder ⟩
$\forall p \in \mathbb{P}, z \in \mathbb{N}.(z \leq \overline{n \Downarrow m}.p \ \equiv\ z \leq \overline{n}.p \downarrow \overline{m}.p)$

By the Generalization Metatheorem, is enough to show for all $n, z \in \mathbb{N}$ and $p \in \mathbb{P}$:

$z \leq \overline{n \Downarrow m}.p$
$\equiv$ ⟨ Definition of Gödel code ⟩
$p^z \cdot|\ n \Downarrow m$
$\equiv$ ⟨ Defining axiom of $\Downarrow$ ⟩
$p^z \cdot|\ n \ \wedge\ p^z \cdot|\ m$
$\equiv$ ⟨ Definition of Gödel code ⟩
$z \leq \overline{n}.p \ \wedge\ z \leq \overline{m}.p$
$\equiv$ ⟨ Defining axiom of $\downarrow$ ⟩
$z \leq \overline{n}.p \downarrow \overline{m}.p$

□

In fact, indirect methods allow us to restate several propositions on multiplicative arithmetic in terms of stronger but easier to prove corresponding propositions on additive arithmetic. For instance,

Theorem 4.4 *For positive integers p, q and r we have that*

$(ia)\ m \Uparrow (n \Downarrow r) = (m \Uparrow n) \Downarrow (m \Uparrow r)$ $(iia)\ m \cdot (n \Downarrow r) = (m \cdot n) \Downarrow (m \cdot r)$
$(iiia)\ m \cdot n = (m \Downarrow n) \cdot (m \Uparrow n)$ $(iva)\ m \Downarrow n = 1 \ \wedge\ r \cdot|\ m \ \Rightarrow\ r \Downarrow n = 1$
$(va)\ m \Downarrow n = 1 \ \Rightarrow\ m \Downarrow (n \cdot r) = m \Downarrow r$

are respectively consequences of the following statements being true for all natural numbers a, b and c:

$(ib)\ a\uparrow(b\downarrow c) = (a\uparrow b)\downarrow(a\uparrow c)$ $\qquad$ $(iib)\ a+(b\downarrow c) = (a+b)\downarrow(a+c)$

$(iiib)\ a+b = (a\downarrow b)+(a\uparrow b)$ $\qquad$ $(ivb)\ a\downarrow b = 0\ \wedge\ c\leq a\ \Rightarrow\ c\downarrow b = 0$

$(vb)\ a\downarrow b = 0\ \Rightarrow\ a\downarrow(b+c) = a\downarrow c$

Calculational proofs are also convenient for human readability where 'obvious' facts are left implicit in some proof steps. In the next section, we present a tool that assists in the task of interactive theorem proving and proof checking modulo built-in axioms as associativity, commutativity and identity.

5 Proof Star

Manolios and Moore have advised the calculational community to 'implement a program to check your proofs' [23]. Proof Star is our first effort to mechanize Calculational Logic in that sense: it is an experimental interactive proof assistant and proof checker for Calculational Logic that provides the user with an approachable interface for calculational proofs, definitions, and computations.

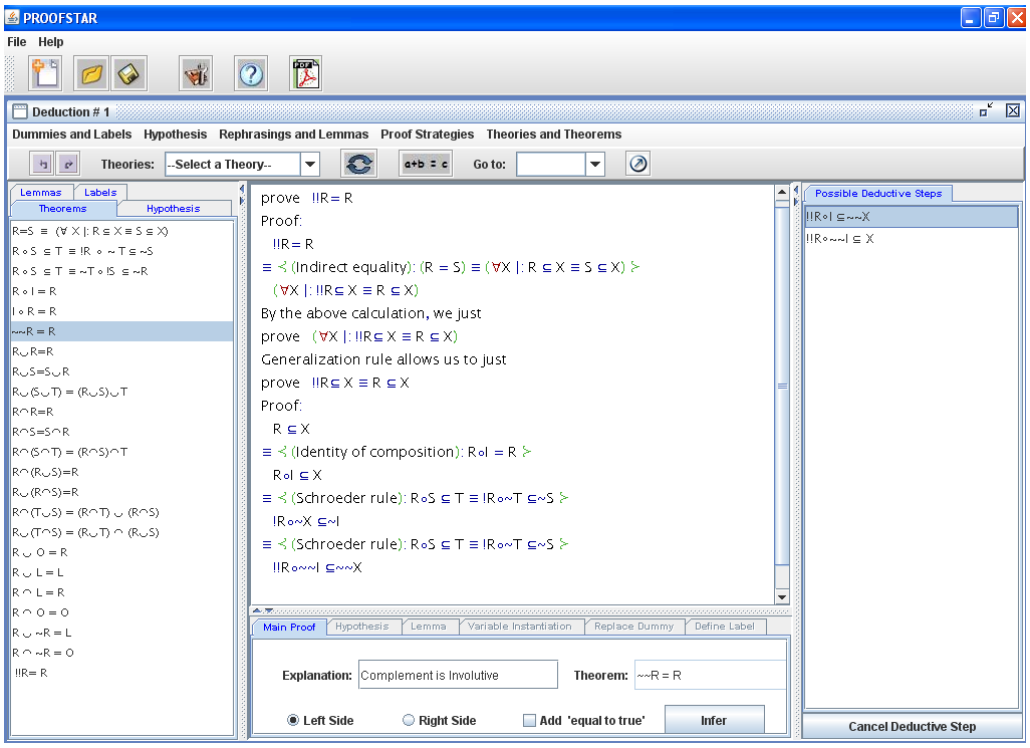


Fig. 1. Proof of Proposition 4.1 in Proof Star, where complement and inverse are denoted with the prefix symbols ˜ and !, respectively.

Proof Star was first devised as a document processor for proofs. Its main goal was to serve as an environment where students could recreate the theories and the proofs appearing in [18]. Currently, and due to the intrinsic power of Calculational Logic, Proof Star has become a special rewrite system exploiting the order-based

properties of logic and math. Particularly, it provides an user-definable library of first-order theories where the axioms are typed order-based formulas, namely, equalities or inequalities of the associated lattice. For example, for the theory of Peano arithmetic, the formulas $a + Sb = S(a + b)$ and $0 < Sa$ can be defined as part of its set of axioms. Both equalities and inequalities are used in any direction by Proof Star's core to assist the user with appropriate matches for terms in the ongoing proof. When applying substitutions, Proof Star ensures that each step in a proof is valid by restricting substitutions based on typing information on terms, and not allowing the use of inequalities in a way that make a deductive chain order-unsound. A proven formula and its proof can be saved in the theorem database for further use.

Proof Star has built-in support for metatheorems, including the Deduction Metatheorem and the Generalization Metatheorem, among others. For instance, recalling the Theorem 4.1, it is proved similarly with Proof Star as depicted in Figure 1. Support for metatheorems plays a key role in what we call proofs with *goal rephrasing*. As its name may suggest, goal rephrasing refers to the fact that in order to prove a formula one can prove another which is logically equivalent, logically stronger or equi-provable (via metatheorems). Proof Star supports deduction by goal rephrasing natively. For example, in order to prove $A \wedge B \Rightarrow \forall x(C \equiv D)$, for A, B and C formulas in some theory, the tool assists (1) in applying the Deduction Metatheorem, rephrasing the proof to $A \wedge B \vdash \forall x(C \equiv D)$ where all the free variables of A and B are now fixed or frozen, (2) using the Conjunction Introduction Metatheorem to separate 'conjunctive hypothesis' and get $A, B \vdash \forall x(C \equiv D)$, and (3) using the Generalization Metatheorem to obtain $A, B \vdash (C \equiv D)[x := \hat{x}]$, where $\hat{x}$ is a fresh variable in the proof. Hence, the rephrased goal is $C[x := \hat{x}] \equiv D[x := \hat{x}]$ under the assumptions of A and B, which is equi-provable to the initial goal.

Among other additional features, Proof Star supports *localized matching*, namely, the user can select by using the mouse any subterm of a formula and obtain a set of matching substitutions (corresponding, mainly, to Leibniz rules applications of a theorem) particular to that term. It also comes packed with LaTeX support, so users can export proofs following the calculation proof format to a LaTeX document.

After the experience of implementing the first version of the tool and using it in the classroom, we feel confident to claim (with some caution, due to its subjective character) that Proof Star complies with the list of properties stated by H. Barendregt for computer mathematic systems, in terms of *adequacy*, *faithfulness*, *efficiency* and *practicality* [2,4]. We summarize this comparison in the following paragraphs.

Adequacy. Calculational Logic supported by Proof Star is naturally based on order sorted logic, and just like a programming language, allows variable and metavariable names, thus providing a powerful definitional framework for a wide range of requirements. Accordingly, reasoning by means of logical deductions is easily formalized by deductive chains where the equational nature of the logic allow substitution of 'equals by equals'.

Faithfulness. If Δ is a proof of a statement S (in a context Γ), then the intuitive statement 'S is provable' is true in ordinary mathematics relative to the corresponding context.

Efficiency. The current implementation of Proof Star is coded in Haskell and Java. The core of Proof Star is where 'proofs' are proof checked, the matching algorithms modulo theories are implemented (e.g., modulo associativity and commutativity), and the support for defining theories is coded. We used Java to endow Proof Star with an approachable graphical user interface.

Practicality. We have used the tool for enriching students experience in discrete mathematics and in algorithms design courses. Students and enthusiastic practitioners of calculational mathematics face relative absence of difficulty in the use of this versatile and flexible tool. On the other hand, the freedom and effectiveness with which the calculational community chooses its notation and formalism to obtain the appropriate level of 'expressive granularity' thus avoiding excessive amounts of detail, is also supported in Proof Star by means of libraries and the ability to state conjectures.

We believe that Proof Star with its rather intuitive user interface, along with the interesting contributions of the Calculational community in terms of notation and elegance for reasoning, make Calculational Logic attractive for formal methods.

6 Conclusions and Future Work

The teaching experience gained during these years has shown that Calculational Logic is an attractive tool for formal methods practitioners. In this paper, we showed how partial orders are used with the calculational style to effectively calculate and check proofs by means of indirect equalities and inequalities. We illustrated the application of these techniques within the setting of a calculus for abstract relation algebra and number theory; besides this, we introduced Proof Star, a versatile calculational proof assistant and proof checker.

Proof Star is our first effort to support mechanical formalization of the calculational style. We have also explored Rewriting Logic as a logical framework [24]. The order-based (even more than equational) nature of calculational theorem proving makes amenable the easy application of rewriting, obtaining different techniques in forms of *decision* and *semi-decision procedures*. We have already obtained decision procedures in the propositional case and in the syllogistic case [31], and a semi-decision procedure in the first order case [32]. One of the following steps is to complement Proof Star with rewriting decision and semi-decision procedures. Moreover, the next version of Proof Star is likely to have its core fully specified in Rewriting Logic, and implemented in Maude [10], a rewrite engine that efficiently implements Rewriting Logic. This will allow further scalability and extensibility possibilities, while exploiting a very efficient implementation of rewriting modulo theories. This will greatly enhance Proof Star not only as a proof assistant and proof checker, but will also enable it as mechanical theorem prover for the Calculational Logic.

References

[1] Backhouse, R., "Program Construction: Calculating Implementations from Specifications," Willey, 2003.

[2] Barendregt, H., *The quest for correctness* (1996).
URL http://citeseer.ist.psu.edu/222204.html

[3] Barendregt, H., *Towards an interactive mathematical proof* (2003).
URL ftp://ftp.cs.kun.nl/pub/CompMath.Found/mathmode.pdf

[4] Barendregt, H. and F. Wiedijk, *The challenge of computer mathematics*, Transactions A of the Royal Society **363** (2005), pp. 2351–2375.

[5] Basin, D., M. Clavel and J. Meseguer, *Rewriting logic as a metalogical framework*, ACM Transactions on Computational Logic **5** (2004), pp. 528–576.

[6] Bohórquez, J., *An inductive theorem on the correctness of general recursive programs*, Logic Journal of IGPL (2007).
URL doi:10.1093/jigpal/jzm053

[7] Bohórquez, J. and C. Rocha, *Towards the effective use of formal logic in the teaching of discrete math*, Information Technology Based Higher Education and Training, 2005. ITHET 2005. 6th International Conference on (7-9 July 2005), pp. S3C/1–S3C/8.

[8] Boute, R., *Functional declarative language design and predicate calculus: a practical approach*, ACM Transactions on Programming Languages and Systems **27** (2005), pp. 988–1047.

[9] Buss, S. R., A. S. Kechris, A. Pillay and R. A. Shore, *The prospects for mathematical logic in the twenty-first century*, Bulletin of Symbolic Logic **7** (2001), pp. 169–196.
URL http://www.math.ucla.edu/~asl/bsl/0702/0702-001.ps

[10] Clavel, M., F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet and C. Talcott, "All About Maude - A High-Performance Logical Framework," Springer LNCS Vol. 4350, 2007, 1st edition.

[11] de Bruijn, *Checking mathematics with computer assistance*, NOTICES: Notices of the American Mathematical Society **38** (1991).

[12] Dijkstra, E. W., *How computing science created a new mathematical style*, EWD 1073 in The writings of Edsger W. Dijkstra, 2000. URL http://www.cs.utexas.edu/users/EWD (1994).

[13] Dijkstra, E. W., *A little bit of lattice theory*, EWD 1240a in The writings of Edsger W. Dijkstra, 2000. URL http://www.cs.utexas.edu/users/EWD (1998).

[14] Dijkstra, E. W. and C. S. Scholten, "Predicate Calculus and Program Semantics," Springer-Verlag, 1990.

[15] Dowek, G., T. Hardin and C. Kirchner, *Theorem proving modulo*, J. Autom. Reasoning **31** (2003), pp. 33–72.
URL http://dx.doi.org/10.1023/A:1027357912519

[16] Feijen, W. H. J., *Exercises in formula manipulation*, in: E. W. Dijkstra, editor, *Formal Development of Programs and Proofs* (1990), pp. 139–158.

[17] Gries, D., *Teaching calculational logic*, in: D. Gries and W. P. de Roever, editors, *PROCOMET*, IFIP Conference Proceedings **125** (1998), pp. 9–10.

[18] Gries, D. and F. B. Schneider, "A Logical Approach to Discrete Math," Texts and Monographs in Computer Science, Springer Verlag, 1993.

[19] Gries, D. and F. B. Schneider, *Equational propositional logic.*, Inf. Process. Lett. **53** (1995), pp. 145–152.

[20] He Jifeng and C. A. R. Hoare, *Weakest prespecification*, Information Processing Letters **24** (1987).

[21] L. Paulson, "Isabelle: A Generic Theorem Prover," Lecture Notes in Computer Science **828**, Springer Verlag, 1994.

[22] Lifschitz, V., *On calculational proofs.*, Ann. Pure Appl. Logic **113** (2001), pp. 207–224.

[23] Manolios, P. and J. S. Moore, *On the desirability of mechanizing calculational proofs*, Inf. Process. Lett. **77** (2001), pp. 173–179.

[24] Martí-Oliet, N. and J. Meseguer, *Rewriting logic as a logical and semantic framework*, in: D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd. Edition*, Kluwer Academic Publishers, 2002 pp. 1–87, first published as SRI Tech. Report SRI-CSL-93-05, August 1993.

[25] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[26] Mili, A., J. Desharnais and F. Mili, "Computer Program Construction," Oxford University Press, New York, NY, 1994.

[27] M.J.C. Gordon and T.F. Melham, "Introduction to HOL: A Theorem Proving Environment for Higher Order Logic," Cambridge University Press, 1993.

[28] Muzalewski, M., *An outline of PC mizar* (1993).
URL http://web.cs.ualberta.ca/~piotr/Mizar/Doc/MM-manual.ps

[29] Nederpelt, R. P., J. H. Geuvers and R. C. de Vrijer, editors, "Selected Papers on Automath," Studies in Logic and the Foundations of Mathematics **133**, North-Holland, 1994.

[30] Paulson, L. C., *The isabelle reference manual.*
URL http://isabelle.in.tum.de/doc/ref.pdf

[31] Rocha, C. and J. Meseguer, *A rewriting decision procedure for Dijkstra-Scholten's syllogistic logic with complements*, Revista Colombiana de Computación **8** (2007).

[32] Rocha, C. and J. Meseguer, *Theorem proving modulo based on boolean equational procedures*, Technical Report 2007-2922, University of Illinois at Urbana-Champaign (2007).

[33] Schmidt, G. and T. Ströhlein, "Relations and Graphs - Discrete Mathematics for Computer Scientists," EATCS Monographs on Theoretical Computer Science, Springer, 1993.

[34] Tarski, A., *On the calculus of relations*, J. Symbolic Logic **6** (1941), pp. 73–89.

[35] Team, T. C. D., *The coq proof assistant reference manual.*
URL ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual-all.ps.gz

[36] Tourlakis, G., *A basic formal equational predicate logic*, Technical Report CS1998-09, York University, Computer Science (1998).
URL http://citeseer.ist.psu.edu/365157.html,http://www.cs.yorku.ca/General/techreports/1998/CS-1998-09.pdf

[37] Tourlakis, G., *On the soundness and completeness of equational predicate logics*, JLC: Journal of Logic and Computation **11** (2001).

[38] Wiedijk, F., *Mizar: An impression* (1999).
URL http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz

A Hoare-Style Calculus with Explicit State Updates

Reiner Hähnle and Richard Bubel[1]

Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University

Abstract

We present a verification system for a variant of Hoare-logic that supports proving by forward symbolic execution. In addition, no explicit weakening rules are needed and first-order reasoning is automated. The system is suitable for teaching program verification, because the student can concentrate on reasoning about programs following their natural control flow and proofs are machine-checked.

Keywords: Hoare-logic, program verification, symbolic execution

1 Introduction

An introduction to formal program verification is part of many courses and text books about Formal Methods (for example, [23,14]). Most of these use a variant of Hoare logic [13] or weakest precondition calculus [7] for a small imperative programming language. Teaching formal program verification on this basis comes with a number of challenges:

- Because of the assignment rule, one needs to compute explicitly weakest preconditions and, therefore, reasons backward through the target program. This is unnatural.
- Even small proofs are tedious to do by hand and one tends to forget "trivial" assumptions such as upper/lower bounds. We found several by-hand proofs in lecture notes on program verification that could not be machine-checked, because of too weak preconditions or invariants.
- Checking first-order conditions is a distraction and requires to introduce first-order inference rules.

The last two points could be easily addressed by a verification tool containing a sufficiently powerful first-order reasoner as an "oracle" to be invoked whenever

[1] Email: {reiner,bubel}@chalmers.se

program-free verification conditions are reached. Surprisingly, there seems to be no easy-to-use Hoare-style verification tool on the market serving that purpose. There are a number of verification systems for imperative languages used in research [24,17,16,22,2,4,3], but none of them is suitable for teaching purposes.

In this paper we present a verification system for a version of Hoare-calculus that addresses the problems described above: it is usable with minimal effort, it contains a clean separation between program and first-order rules, and it features a first-order reasoner tailored to verification tasks that can be presented as an oracle. We also address the first point: our program logic enables *forward* symbolic execution while still being based on a weakest precondition calculus. While the calculus is based on weakest precondition, the chosen presentation form are Hoare triples as in our experience the latter one seems to be used more frequently in education. The technical device used to achieve this is an explicit notion of symbolic program states. We show that this introduces only minimal overhead, but has substantial advantages from a pedagogical view. The system is freely available and easy to install. [2] Our implementation is based on the KeY tool[3], one of the most powerful verification systems for Java.

2 Background

2.1 Target Programming Language

We use a simple imperative **while** programming language with only four kinds of *statements*:

Program ::= (*Statement*)?
Statement ::= *AssignmentStatement* | *CompoundStatement* |
ConditionalStatement | *LoopStatement*
AssignmentStatement ::= *Location* `=` *Expression*`;`
CompoundStatement ::= *Statement Statement*
ConditionalStatement ::= `if '('`*BooleanExp*`')'`
`'{'`*Statement*`'}' else '{'`*Statement*`'}'`
LoopStatement ::= `while'('`*BooleanExp*`')' '{'`*Statement*`'}'`
Expression ::= *BooleanExp* | *IntExp*
BooleanExp ::= *IntExp ComparisonOp IntExp* | *IntExp* `==` *IntExp* |
BooleanExp BooleanOp BooleanExp | `!`*BooleanExp* |
Location | **true** | **false**
IntExp ::= *IntExp IntOp IntExp* | $\mathbb{Z}$ | *Location*
ComparisonOp ::= `<` | `<=` | `>=` | `>`
BooleanOp ::= `&` | `|` | `==` *IntOp* ::= `*` | `/` | `%` | `+` | `-`

Locations are simply program variables. In general, they could be more complex structures, such as array or field accesses, but we will not discuss this here. [3] Locations and expressions are *typed*. There are two incomparable types called **boolean** and **int**. The type **int** denotes the mathematical integers $\mathbb{Z}$, not a finite integer

[2] From http://www.key-project.org/download/hoare/, including all examples discussed in this article.
[3] The definitions given here are unsound in the presence of aliasing. General definitions of the concepts involved are found in [3].

type like in most real world languages like Java. Note that equality is overloaded. The grammar above is simplified in the sense that the real grammar uses common precedence rules for the different operators and allows of course parenthesised expressions. Obviously, the programming language defined here is a syntactic subset of the imperative fragment of Java [10].

2.2 First-Order Logic

In order to specify programs we use typed first-order logic. The only types allowed are **`boolean`** and **`int`**. Terms and formulas of first-order logic are defined as usual, with one notable exception: expressions of the programming language are also permitted as terms. This is ok, because expressions are side-effect free. Atomic formulas either have the form

- $P(t_1, \ldots, t_n)$ with P standing for an arbitrary user-defined predicate symbol of arity n and terms $t_1, \ldots, t_n$ of appropriate type or
- $s \doteq t$ with the reserved *equality* symbol $\doteq$ taking arbitrary terms as arguments.

Program variables are (despite their name) not modelled as first-order variables but as constants (0-ary functions). Therefore, it is not possible to quantify over program variables. Further, we distinguish between *rigid* and *non-rigid* (or *flexible*) symbols. The difference is that rigid symbols are evaluated by a classical interpretation function and variable assignment. Their value is fixed and cannot be changed by a program. Uninterpreted rigid constants are often used to specify initial and final values of program variables. The availability of rigid functions and constants makes it easy to capture and refer to earlier program states and initial values. In addition, built-in symbols with a fixed semantics, such as equality $\doteq$ and the operators occurring in expressions of the programming language, are rigid.

In contrast, the value of *non-rigid* symbols depends on the current state in which they are evaluated. *Non-rigid* symbols can be changed by programs. In the presented logic the only *non-rigid* symbols are program variables.

We decided against modelling program variables as logical variables, mainly for usability reasons: in order to get by with logical variables alone one needs introduce *primed* variables as in [6]. Each state change during symbolic execution necessitates introduction of fresh primed variables. These increases the number of symbols required to specify and to prove a problem which in turn compromises readability of proofs. Readability, however, is an important issue when user interaction is required–not only for students on the beginner level. A further point in favour of the introduction of non-rigid symbols is to avoid confusion for students who may just have gotten used to the static viewpoint of first-order logic. In addition, the provided tool is built-up on an inference engine tailored to dynamic logic where non-rigid modelling is natural.

Some useful *conventions*: program variables are typeset in typewriter font, logical variables in italic. When we specify a program π we assume that all program variables of π are contained in the first-order signature with their correct type. The *semantics* of first-order formulas is interpreted over fixed domain models. Specifically, all boolean terms are interpreted over {**`true`**,**`false`**} and all integer terms over $\mathbb{Z}$. There are built-in function symbols for arithmetic including `+`, `-`, `*`, `/` and `%`

and integer comparison operators `<=`, `<`, `>` and `>=` with their obvious meaning. See the reference manual [11] for concrete formula syntax. Apart from that, all semantic notions such as satisfiability, model, validity, etc., are completely standard, see, for example, [8].

2.3 Hoare Calculus

Before we define our own version we present a standard version of Hoare calculus [13]. As usual, the behaviour of programs is specified with *Hoare triples*:

$$\{P\}\,\pi\,\{Q\} \tag{1}$$

Here, P and Q are closed first-order formulas and π is a program over locations $L = \{l_1, \ldots, l_m\}$. The meaning of a Hoare triple is as follows: *for each model $\mathcal{M}$ of P, if π is started with initial values $i_k = \mathcal{M}(l_k)$ $(1 \leq k \leq m)$ and if π terminates with final values f_k, then $\mathcal{M}^{f_1,\ldots,f_m}_{l_1,\ldots,l_m}$ is a model of Q.*

We can paraphrase this in a slightly more informal, but more intuitive, manner: for a given program π over locations $\{l_1, \ldots, l_m\}$, let us call an assignment of values $l_k = v_k$ $(1 \leq k \leq m)$ the *state* s of π. What the Hoare triple then says is that if we start π in any state satisfying the *precondition* P, if π terminates, then we end up in a final state that satisfies *postcondition* Q.

The standard Hoare rules are displayed in Fig. 1. We employ the following conventions for schematic variables occurring in the rules: `e` is an expression, `b` is a boolean expression, `x` is a program variable, `s,s1,s2` are statements. P, Q, R, I are closed first-order formulas.

$$\text{assignment}\ \frac{}{\{P\{\mathtt{x}/\mathtt{e}\}\}\ \mathtt{x=e;}\ \{P\}} \qquad \text{composition}\ \frac{\{P\}\,\mathtt{s1}\,\{R\} \qquad \{R\}\,\mathtt{s2}\,\{Q\}}{\{P\}\,\mathtt{s1\ \ s2}\,\{Q\}}$$

$$\text{conditional}\ \frac{\{P\ \&\ \mathtt{b} \doteq \mathtt{true}\}\,\mathtt{s1}\,\{Q\} \qquad \{P\ \&\ \mathtt{b} \doteq \mathtt{false}\}\,\mathtt{s2}\,\{Q\}}{\{P\}\,\mathtt{if(b)\{s1\}else\{s2\}}\,\{Q\}}$$

$$\text{loop}\ \frac{\{I\ \&\ \mathtt{b} \doteq \mathtt{true}\}\,\mathtt{s}\,\{I\}}{\{I\}\,\mathtt{while(b)\{s\}}\,\{I\ \&\ \mathtt{b} \doteq \mathtt{false}\}}$$

$$\text{weakeningLeft}\ \frac{P \mathrel{->} Q \qquad \{Q\}\,\mathtt{s}\,\{R\}}{\{P\}\,\mathtt{s}\,\{R\}} \qquad \text{weakeningRight}\ \frac{\{P\}\,\mathtt{s}\,\{Q\} \qquad Q \mathrel{->} R}{\{P\}\,\mathtt{s}\,\{R\}}$$

$$\text{oracle}\ \frac{}{P}\quad (P \text{ any valid first-order formula})$$

Fig. 1. Rules of standard Hoare calculus.

3 Hoare Logic with Updates

The standard formulation of Hoare logic in Fig. 1 has a number of *drawbacks* in usability that are particularly problematic when used for teaching purposes:

- Because of the assignment rule, one needs to compute explicit weakest preconditions and, therefore, reasons backward through the target program.
- The compositional rule splits the proof and requires to have the intermediate state available.
- Weakening must be used before applying the rules for conditionals/loops. It would be better to delay weakening until first-order verification conditions are reached and let it be dealt with by an automated theorem prover.
- It is not easy to associate a node in a Hoare proof tree with a computation state of the target program.

We overcome these problems by introducing an explicit notation that describes finite parts of symbolic program states. This allows us to recast Hoare logic as forward symbolic execution.

3.1 State Updates

A (state) *update* is an expression of the form *Location* := *FOLTerm*. Actually, this is only the most simple form of an update, called *atomic update*. Complex updates are defined inductively: if $\mathcal{U}$ and $\mathcal{V}$ are updates, then so are $\mathcal{U}$, $\mathcal{V}$ (*sequential update*), and $\mathcal{U} \,||\, \mathcal{V}$ (*parallel update*). [4]

The more important of these is the parallel update. Consider a parallel update of the form $\mathcal{U} = l_1 := t_1 || \cdots || l_m := t_m$. Assume that we are in a computation state s. Then the update takes us into a state $s_{\mathcal{U}}$ such that:

$$s_{\mathcal{U}}(l) = \begin{cases} s(l) & \text{if } l \notin \{l_1, \ldots, l_m\} \\ t_k & \text{if } l = l_k \text{ and } l \notin \{l_{k+1}, \ldots, l_m\} \end{cases} \tag{2}$$

In words: the value of the locations occurring in $\mathcal{U}$ are overwritten with the right-hand side of the respective update. The second condition in the second clause ensures that the right-most update in $\mathcal{U}$ "wins" if the same location occurs more than once on the left-hand side in $\mathcal{U}$. Apart from that, all updates are executed in parallel. Updates are similar to a preamble or fixture as used in unit testing [18]: a piece of code that gets you into a certain state. There is, however, a difference between updates and code: the right-hand side of an update may contain any first-order term, not merely program expressions. This feature is often used to initialise a program with "arbitrary, but fixed" values.

The significance of parallel updates lies in the following property, formally stated in Lemma 3.1 below. Let us call two updates $\mathcal{U}$ and $\mathcal{V}$ *equivalent* if $s_{\mathcal{U}} = s_{\mathcal{V}}$ for any state s. Then for each update $\mathcal{U}$ exists an *equivalent* parallel update $\mathcal{V}$ of the form $l_1 := t_1 \;||\; \cdots \;||\; l_m := t_m$.

3.2 Hoare Triples with Update

We allow to write an update $\mathcal{U}$ in front of any program like this: $[\mathcal{U}]\pi$. If we are in state s the meaning is that the program is started in state $s_{\mathcal{U}}$. Within Hoare logic

[4] There are further kinds of updates [20,3], but we do not need these here.

we use updates as follows:

$$\{P\}\,[\mathcal{U}]\,\pi\,\{Q\} \tag{3}$$

where, P, Q, and π are as above, and $\mathcal{U}$ is an update over the signature of P and π. We enclose updates in square brackets to increase readability. Either one of $\mathcal{U}$ and π can be empty. The meaning of this *Hoare triple with update* is as follows: if s is any state satisfying the *precondition* P and we start π in $s_{\mathcal{U}}$, then, if π terminates, we end up in a final state that satisfies *postcondition* Q.

3.3 Hoare-Style Calculus with Updates

In Fig. 2 we state the rules of a Hoare calculus with updates that has some new features compared to standard Hoare calculus of Fig. 1:

- Composition is turned into left-to-right symbolic execution. Thereby a precise formula R is computed which is sufficient to achieve a complete calculus, but it does not subsume the composition rule as a whole as it lacks its implicit weakening.
- Weakening is pushed below application of program rules and becomes part of first-order verification condition checking.
- We employ updates for handling assignments.

One advantage of weakest precondition calculation [7] as well as backward-execution style Hoare calculus is that an assignment can be computed by simple substitution and no renaming of old variables is necessary. The price to be paid for that is the not very intuitive backward-execution of programs. The KeY program logic uses updates to achieve weakest precondition computation with *forward* symbolic execution. In our eyes, this is a major pedagogical advantage: not only follows program rule application the natural execution flow in imperative programs, but the whole prove process is also compatible with established paradigms such as symbolic debugging.

In the KeY logic as well as in the present version of Hoare logic the rules have a "local" flavour in the sense that each judgement (node) in the proof tree relates to a symbolic state during program execution.

We use the same conventions for schematic variables as above, but in addition, let $\mathcal{U}$ be an update and `s` is either a statement or the empty string. The rules are depicted in Fig. 2. Let us briefly discuss each of them.

The *assignment* rule becomes easy: assignments are directly turned into updates. In our simple language, expressions have no side effects, so we do not need to introduce temporary variables to capture expression evaluation: we can directly turn `e` into the right-hand side of an update and later evaluate the semantic denotation. The same holds for guards. Because we moved composition of substitutions into updates, we can now evaluate programs left-to-right. The weakest precondition calculation is hidden in the update rules (see Fig. 3 below).

There is one new rule called `exit` that is applied when a program is fully symbolically executed. At this point, the update is applied which computes the weakest precondition of the symbolic program state $\mathcal{U}$ with respect to the postcondition Q.

$$\text{assignment}\ \frac{\{P\}\,[\mathcal{U},\ \mathtt{x} := \mathtt{e}]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{x=e;s}\,\{Q\}} \qquad \text{exit}\ \frac{\vdash\ P \mathrel{->} \mathcal{U}(Q)}{\{P\}\,[\mathcal{U}]\ \{Q\}}$$

$$\text{conditional}\ \frac{\{P\ \&\ \mathcal{U}(\mathtt{b} \doteq \mathtt{true})\}\,[\mathcal{U}]\,\mathtt{s1;s}\,\{Q\} \qquad \{P\ \&\ \mathcal{U}(\mathtt{b} \doteq \mathtt{false})\}\,[\mathcal{U}]\,\mathtt{s2;s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{if(b)\{s1\}else\{s2\}s}\,\{Q\}}$$

$$\text{loop}\ \frac{\vdash\ P \mathrel{->} \mathcal{U}(I) \qquad \{I\ \&\ \mathtt{b} \doteq \mathtt{true}\}\,[]\,\mathtt{s1}\,\{I\} \qquad \{I\ \&\ \mathtt{b} \doteq \mathtt{false}\}\,[]\,\mathtt{s}\,\{Q\}}{\{P\}\,[\mathcal{U}]\,\mathtt{while(b)\{s1\}s}\,\{Q\}}$$

Fig. 2. Rules of Hoare calculus with updates.

Then it is checked whether the given precondition implies the weakest precondition. The premise of the exit rule (as well as the left-most premise of the loop rule) are first-order verification conditions. This is indicated by a turnstile in order to make clear that we left the language of Hoare triples.

The conditional rule simply adds the guard expression as branch condition to the precondition. Of course, we must evaluate the guard in the current state $\mathcal{U}$. As said above, this formulation requires expressions to have no side effects. It has the advantage that path conditions can easily be read off each proof node.

The loop rule is a standard invariant rule. We exploit again that expressions have no side effects, but also that we have no reference types. The chosen formulation stresses the analogies to the conditional rule. The first premise says that the precondition must be strong enough to ensure that the invariant holds after reaching the state at the beginning of the loop. In the second premise we are not allowed to use P, because P might have been affected by executing $\mathcal{U}$. In addition, we must reset the update to the empty one. In other words, started in *any* state where the loop invariant and condition hold the invariant must hold again after execution of the loop body. In practise, one uses as a starting point for the invariant those parts of P that are unaffected by $\mathcal{U}$. In those parts that *are* modified, one typically generalises a suitable term and adds that to the invariant.

3.4 Rules for Updates

We still need rules that handle our explicit state updates. Specifically, we need to (i) turn sequential into parallel updates (Sect 3.1) and (ii) apply updates to terms, formulas, and other updates. For the first task we use a Lemma from [19] (in specialised form):

Lemma 3.1 *For any updates $\mathcal{U}$ and $\boldsymbol{x} := t$ the updates $\mathcal{U}$, $\boldsymbol{x} := t$ and $\mathcal{U} \,||\, \boldsymbol{x} := \mathcal{U}(t)$ are equivalent.*

The resulting rule is depicted with the various update application rules in Fig. 3. These are rewrite rules that can be applied whenever they match. We use the same schematic variables as before and, in addition, t is a first-order term, $\mathcal{P}$ is a parallel update of the form $l_1 := t_1 || \cdots || l_m := t_m$, y is a logical variable, F is an n-ary function or predicate symbol, $\Box$ is a propositional connective, and λ is a quantifier.

On top left is the rule that turns sequential into parallel updates. The second

$$\mathcal{U}, \mathtt{x} := t \implies \mathcal{U} \,||\, \mathtt{x} := \mathcal{U}(t)$$

$$\mathcal{P}(\mathtt{x}) \implies \begin{cases} \mathtt{x} & \text{if } l \notin \{l_1, \ldots, l_m\} \\ t_k & \text{if } \mathtt{x} = l_k \text{ and } l \notin \{l_{k+1}, \ldots, l_m\} \end{cases} \qquad \mathcal{U}(y) \implies y$$

$$\mathcal{U}(F(t_1, \ldots, t_n)) \implies F(\mathcal{U}(t_1), \ldots, \mathcal{U}(t_n)) \qquad \mathcal{U}(P \Box Q) \implies \mathcal{U}(P) \Box \mathcal{U}(Q)$$

$$\mathcal{U}(\lambda y.\ P) \implies \lambda y.\ \mathcal{U}(P),\ y \notin \mathrm{free}(\mathcal{U})$$

Fig. 3. Rewrite rules for update computation.

row contains rules for applying updates to program and logical variables. Note the similarity between the rule for program variables and (2) on p. 53. Logical variables are rigid and never changed by the updates. The third and fourth row contain rules for complex terms and for formulas. These are merely homomorphism rules. In quantified formulas, again, logical variables cannot be affected, but as they may occur in updates one has to ensure that no name clashes occur ($free(\mathcal{U})$ returns the set of logical variables not bound in $\mathcal{U}$). On the whole it becomes clear that update application is basically substitution of program variables with their new values.

In fact, if we define standard substitution formally as rewrite rules, we need only two rules less! One of the additional rules is closely related to composition of substitutions. In the end we only have a very slight overhead due to the distinction between logical and program variables. Note that there is no rule to apply updates to programs. They accumulate until symbolic execution of the underlying program terminates.

4 Using KeY-Hoare

We illustrate how the system is used by proving correctness of a program computing the greatest common divisor.

```
while (!(small == 0)) {
  tmp = big % small;
  big = small;
  small = tmp;
}
result = big;
```

All variables are integers. Provided that `big` is greater or equal than `small` and both are non-negative, `result` contains the greatest common divisor of `big` and `small`. Let `_big` and `_small` be rigid constants that capture arbitrary initial values of `big` and `small`. A suitable precondition P is: `_small >= 0 & _big >= _small`. Let R be the following formula which expresses that any common divisor `x` of the integers `_big` and `_small` is a divisor of the integer `result`.

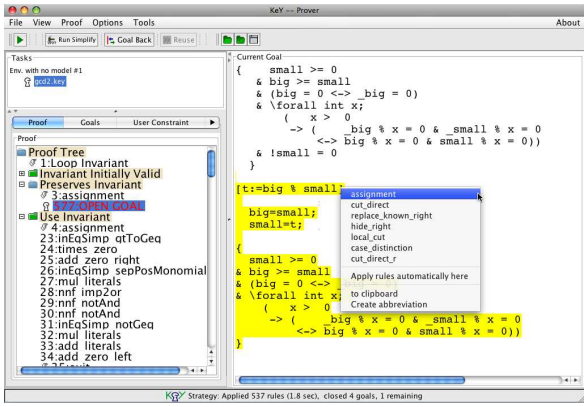
```
\forall int x;
   ((x > 0 & _big % x = 0 & _small % x = 0) -> result % x = 0))
```

Then the postcondition Q can be stated as:

```
(_big != 0 -> R) & (_big = 0 -> result = 0)
```

(concrete formula syntax see [11]). With the abbreviations introduced above, the initial Hoare triple with updates reads as follows:

$$\{P\}\,[\texttt{big} := \texttt{\_big} \parallel \texttt{small} := \texttt{\_small}]\,\texttt{gcd}\,\{Q\}$$



A file with an initial Hoare triple as proof obligation (in a simple format described in [11]) is loaded to the KeY-Hoare system. Then the user can select a rule from Fig. 2 offered in a popup-menu after moving the mouse pointer over a Hoare triple and clicking (see screenshot on the left). There is exactly one applicable rule for each program construct and the system offers exactly this rule: the user experiences statement-wise symbolic execution of the target program. The only non-trivial interaction is to supply an invariant in a dialogue box that opens when the loop rule is applied. Here the idea for the invariant is that the loop leaves all common divisors of `big` and `small` invariant, hence, `big` and `small` have exactly the same common divisors than `_big` and `_small`. One needs also to state that `big` never gets smaller than `small` and what happens when `_big==0`:

```
small >= 0 & big >= small & (big = 0 <-> _big = 0) &
\forall int x; (x > 0 ->
 ((_big % x = 0 & _small % x = 0) <-> (big % x = 0 & small % x = 0)))
```

Whenever first-order verification conditions are reached, the system offers a rule Update Simplification that applies the update rules from Fig. 3 automatically. At this point, the user can opt to push the green Go button ▶. Then the built-in first-order theorem prover tries to establish validity automatically. For simple problems discussed in the introductory courses, such as `gcd`, this works quite well. If no proof is found, typically, the invariant or the specification (or the code!) is too weak or simply wrong. Inspecting the open goals usually gives a good hint. The system allows the student to follow symbolic execution of the program and to concentrate on getting invariants and specification right. First-order reasoning is left to the system. It is possible to inspect and undo previous proof steps as well as to save and load proofs.

5 Related Work

The tutoring tool for Hoare Calculus ITS, described and evaluated in [9], does not realise a reasoning system or proof checker. Students can fill out missing Hoare triples in two different notations. ITS checks whether related triples in the different notations have the same denotation and it determines the order in which triples were filled in to see whether students used forward or backward reasoning. Another ed-

ucational tool for Hoare Calculus is J-Algo [15], a general modular framework that allows to visualise algorithms and comes with a module for the Hoare Calculus. While there is support for stepwise construction of a syntactically valid Hoare proof tableau, the lack of a reasoning system does not allow to obtain machine checked proofs. The RISC Navigator [21] provides an interactive proof assistant with an interface to external decision procedures. A main goal is to provide an easy-to-use tool suitable for educational purposes. It is used in teaching program verification, but requires to generate verification conditions of a Hoare triple by hand. Afterwards these conditions can be loaded and proven within the system. Our state updates are closely related to generalised substitutions used by the B method [1] and to Abstract State Machines [5]. A full discussion is contained in [20]. There are versions of Hoare logic that use the assignment rule from dynamic logic [12] in which case forward symbolic execution can be realised, however, at the price of introducing existentially quantified variables that hold the result of intermediate states. This is complicated to explain and difficult to use.

6 Conclusion, Future Work

We presented a verification system for a variant of Hoare-logic that supports proving by forward symbolic execution. No explicit weakening rule is needed and first-order reasoning is automated. The system is suitable for teaching program verification, because the student can concentrate on reasoning about programs following their natural control flow and proofs are machine-checked. The KeY-Hoare tool is freely available and can be easily installed. It is based on a state-of-art verification system for Java [3]. The KeY-Hoare tool is currently used in the course *Program Verification* intended for Bachelors in their final year at Chalmers University. [5] Course materials including slides, examples, exercises, and exam questions are available from the authors.

At the moment, the GUI of the KeY-Hoare tool contains several elements that are inherited from the full Java version and are not useful in the more specialised context. It should be cleaned up and simplified. The current version of KeY-Hoare does not support arrays as Java arrays are too complicated for an introductory course. It would be easy, however, to implement value-type arrays and we plan to do this soon. In a similar vein, we will also add static method calls. All this is very easy, because it can be derived from simplifying corresponding Java constructs.

Acknowledgements
We thank Wolfgang Ahrendt and Philipp Rümmer for numerous helpful comments. We thank also the anonymous referee for valuable comments and suggestions.

[5] See http://www.cs.chalmers.se/Cs/Grundutb/Kurser/prove.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[3] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2006.

[4] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2005.

[5] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[6] R. T. Boute. Calculational semantics: Deriving programming theories from equations by functional predicate calculus. *ACM Trans. Program. Lang. Syst.*, 28(4):747–793, 2006.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] M. C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

[9] K. Goshi, P. Wray, and Y. Sun. An intelligent tutoring system for teaching and learning Hoare logic. In *ICCIMA '01: Proceedings of the Fourth International Conference on Computational Intelligence and Multimedia Applications*, page 293. IEEE Computer Society, 2001.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification–Third Edition*. The Java Series. Addison-Wesley,, 2004.

[11] R. Hähnle and R. Bubel. *A Brief Reference Manual for KeY-Hoare*. Chalmers University of Technology, Department of Computer Science and Engineering, Jan. 2008. Version 0.1.6, available at http://www.key-project.org/download/hoare/students.pdf.

[12] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, Oct. 2000.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, Oct. 1969.

[14] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, second edition, 2004.

[15] J-Algo—The Algorithm Visualization Tool, 2007. http://j-algo.binaervarianz.de/.

[16] C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *Proc. 18th Intl. Conference on Theorem Proving in Higher Order Logics, Oxford, UK*, Lecture Notes in Computer Science. Springer-Verlag, Aug. 2005.

[17] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[18] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, second edition, 2004.

[19] P. Rümmer. Proving and disproving in dynamic logic for Java. Licentiate Thesis 2006–26L, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2006.

[20] P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In M. Hermann and A. Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006.

[21] W. Schreiner. Program Verification with the RISC ProofNavigator. In D. Duce and P. Boca, editors, *Teaching Formal Methods: Practice and Experience*, Electronic Workshops in Computing (eWiC), pages 1–6, London, UK, December 15, 2006. British Computer Society.

[22] K. Stenzel. *Verification of Java Card Programs*. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005.

[23] R. D. Tennent. *Specifying Software: a Hands-On Introduction*. Cambridge University Press, 2002.

[24] D. von Oheimb. Hoare logic for java in isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

How Do I Know If My Design Is Correct?

J Paul Gibson, Eric Lallet, Jean-Luc Raffy[1,2,3]

Le département Logiciels-Réseaux (LOR),
Institut National des Télécommunications,
Evry, France

Abstract

Teaching software engineering students about design is very challenging. In general, students will learn about design through a module teaching a graphical modelling language. Our experience shows that this can result in students learning how to represent and comprehend designs but having very little understanding of design as a process. When reviewing design artefacts, students often ask whether the designs are *good*. This leads to the realisation that there is lack of understanding of the fundamental question of whether a design can be said to be *correct*. Of course, the notion of *correctness* will generally be covered by another module, typically called "formal methods". Unfortunately, our experience also shows that formal methods courses can lead to students learning how to build formal models — much like they would build programs — without achieving a good understanding of nondeterminism and abstraction; and without seeing how formal methods can help in the process of design. In this paper, we argue that the teaching of software design needs to be better integrated with the teaching of formal methods. We give some concrete examples of how this can be done.

Keywords: Design, Correctness, UML, Formal Methods.

1 Introduction

One of the least well understood aspects of software development is the role of design in bridging the gap between *what* (requirements) and *how* (implementation). Inexperienced software designers fail to treat design as a process, and as a consequence become experts in representing the (static) artefacts using models/languages but fail to master the evolution of design.

During the transition from procedural to object-oriented programming languages, there was a realisation that the boundary between design and implementation was becoming even more blurred. In a controversial article in the C++ Journal, Reeves[18], stated: "...about ten years ago I came to the conclusion that, as an industry, we do not understand what a software design really is. I am even more convinced of this today." Reeves goes on to argue that considering the source code

[1] Email: paul.gibson@int-evry.fr
[2] Email: eric.lallet@int-evry.fr
[3] Email: jean-luc.raffy@int-edu.eu

as being the design overcomes one of the fundamental issues associated with software design: how can we be sure that it will work correctly?

> "...when real engineers get through with a design, no matter how complex, they are pretty sure it will work. They are also pretty sure it can be built using accepted construction techniques. In order for this to happen, hardware engineers spend a considerable amount of time validating and refining their designs."

These ideas have much more resonance when we consider recent growth in agile development[14].

In fact, the notion that the design is not finished until it has been coded and tested is not, as it would seem at first sight, at odds to a formal approach to software design. In a formal approach, designs are coded (using formal specification languages) and they are tested and refined. Unfortunately, teaching formal methods to software engineers is no guarantee that they will use them during design! Ken Robinson[20] identifies a clear problem with the teaching of formal methods: "It is frequently the case that the other courses make no reference to, or use of, the formal techniques studied in the Formal Methods course."

In this paper we argue that it is the responsibility of the teachers of formal methods to incorporate aspects of *all* other software engineering courses in their teaching (not just design). However, the focus of work in this paper is in the integration of formal methods and design, with specific examples given using UML[7] [4].

2 UML approach: the strengths and weaknesses

The main strength of UML is that it is the standard OO modelling language; with comprehensive tool support and plentiful educational resources. However, it has been openly criticised by a number of high-profile software engineers. For example, we need look no further than Bertrand Meyer for a satirical article that identifies the main weaknesses of UML[16]. In the same spirit one should read the entertaining yet insightful article by Alex Bell[6] which shows the dangers in expecting the adoption of UML to automatically improve your software development process. The UML has been extended with a more formal notation in the guise of the OCL. Expressions written in OCL offer a number of benefits: most importantly, they should make the graphical model more precise and more detailed[19]. However, the shortcomings of OCL have been well documented for a number of years[22].

We are not the first to identify the risks of replacing a general software design course with a course on UML. Engels et al. make the point quite simply[11]: "The incorporation of UML in a curriculum can and should not happen by adding a separate UML course. UML is nothing but a means to reach an end (the end in this case being the expression of software models)." We note that, in the above quotation, one can replace all instances of the string "UML" with the string "formal methods" and the resulting sentence is another which we believe to be true!

[4] In our teaching, we use B[1] for formalising aspects of design. In this paper we do not give B models, but make reference to the B-method, refinement and correctness by construction.

3 B approach: the strengths and weaknesses

B is a method [1] for specifying, designing and coding software systems. The concept of refinement [5] is the key notion for developing B models of (software) systems in an incremental way. B models are accompanied by mathematical proofs that justify them. We start from an abstract model and each subsequent model is a refinement of the previous one. Proofs of properties of B models help to convince the user (designer or specifier) that the (software) system is correct, since they demonstrate that the behaviour of the last, and most concrete, system (software) respects the behaviour of the first, most abstract model (which we assume has already been validated).

The main advantage of B is that it focuses attention on the core role of design: moving from abstract to concrete, through a process of refinement that guarantees the correctness of design decisions. The B method has evolved into Event-B, with an associated methodology and industrial strength tools[2], all of which have already been used for teaching formal software engineering[3].

A weakness of teaching B is that students find it difficult to model the system and its components abstractly. The B approach requires students to think in terms of what is required and to start by building abstractions of these high-level requirements. It also needs them to refine their abstract specifications into concrete implementations. Students can manage to do this for simple case studies but fail to appreciate how this can scale up to larger design decisions. They also fail to see how formal proof could be useful to them in their day-to-day design work. There is a mental block between the type of designs they see when using UML and the type of designs they see when working with B. They like the way UML facilitates their visualisation of structural properties. In short, they feel more confortable and confident working with pictures than working with mathematical formalism.

4 Formal Design: integrated teaching

There are many potential benefits to integrating UML with formal methods; the idea is not new[12] but continues to be a challenging topic of research, for example: [8,23,17]. From an educational viewpoint, this research is mature enough to transfer back to our teaching. However, the question of how this integration should be done is one that requires further research. We have experimented with four types of integration:

- *Case-study-driven* - continue to teach UML and formal methods as separate modules but glue them together through common case studies;
- *Formalising UML* - Extend the UML module with material focussing on the OCL and the integration of formal languages;
- *UMLing your formal method* - Extend your formal methods module to show how the models can be specified in an object oriented fashion;
- *Teach Formal (OO) Design* - focus on design as a process and use a range of notations to illustrate design activities.

It is beyond the scope of this paper to analyse these options. However, in the next sections, we give an example of the type of design problem that can be used in any of these teaching approaches.

5 An Educational Example: Queues From Stacks

A typical software engineering problem is to transform a high-level design so that it can be directly implemented on a particular architecture. One aspect of doing this is that one aims to re-use components that already exist in the chosen target architecture.

During teaching of a data structures and algorithms course, students are introduced to the abstract concepts of a queue and a stack. These two examples provide a good opportunity to introduce formal methods. We have used the following problem with 2nd year students (as a Java programming exercise), MSc students (as an OO design exercise), and with fourth-year students (as a formal verification exercise). In this paper, the emphasis is on the design process, whilst the actual modelling languages used by the students (UML, B and Java) illustrate the need to reason about correctness as formally as possible.

5.1 How can we implement a Queue using Stacks?

We specify the requirements as a Queue of integers[5] and state that the students must implement the FIFO behaviour using only two integer Stacks (LIFO behaviour) to store the queue contents. As a design exercise, students typically adopt 1 of 2 options:

- Design1: The queue is specified as having two stack components — which we will name as a `pushstack` and a `popstack`. When a push request is made of the queue then this element is pushed directly onto the `pushstack`.When a pop request is made of the queue then move all elements from the `pushstack` on to the `popstack` then pop off the last element of the `popstack` and then move all the elements back on to the `pushstack`.
- Design2: The queue is has two stack components — which we will name as a `mainstack` and a `tempstack` — and a boolean representing whether or not the `mainstack` is `ready to push`. (If it is not `ready to push` then we say that it is `ready to pop`). When `ready to push`[6]: if a push request is made of the queue then this element is pushed directly onto the `mainstack`, if a pop is requested then all the elements are moved from the `mainstack` to the `tempstack`, the `mainstack` and `tempstack` are swapped, the state is changed to `ready to pop` and the element popped off the `mainstack`.

At this stage we ask the students to evaluate the quality of their designs. Most students identify the following inter-related design quality criteria: simplicity, understandability, implementability, extensibility, modularity, maintainability,

[5] Note that this problem takes on a different nature if we allow the modelling of parametric classes of behaviour.

[6] The `ready to pop` case can be treated similarly.

re-usability, efficiency (time and memory), robustness and reliability. In our experience students will ask about the *correctness* of their design only if they have already studied formal methods. When asked if the design will work, most students reply that they will test their implementation to make sure that it does.

Analysis of Design1 and Design2 usually leads to students identifying that Design1 is easier to understand and implement, but that Design2 may be more efficient. Representing the two designs in UML often leads to the students realising that the two designs appear to be structurally the same, but quite different in terms of their dynamic behaviour. The class diagram, in figure 1, illustrates that using UML leads to further investigation of design alternatives that are not so obvious from working only with a formal modelling language.

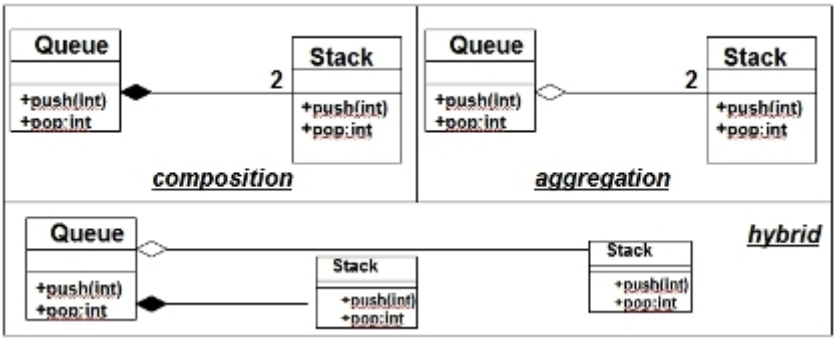


Fig. 1. Aggregation or Composition?

Most students choose to model the assocation between the Queue and its class components as composition. The remaining students usually model this using aggregation. We ask the question as to why a hybrid model (using both composition and aggregation) is, in general, never considered.

5.2 Rigorous Analysis

We ask the students to argue (demonstrate) whether the designs are correct before they implement them. Typically, they are not able to convince themselves that the designs are correct but they do identify unsafe states of the designs that should not arise. We then show them how these can be modelled using invariants. For example, in Design1 the Queue system is unsafe if the `popstack` is not empty when an element is pushed on to the `pushstack`.

We have followed two different routes from this point. Firstly, students can implement their designs (typically in Java). Secondly, students formalise their designs in B and attempt to prove that the designs are correct. In the first instance, student implementations often do not meet the queue requirements (which can be found through testing): students then need to discuss whether this signifies that their designs are incorrect. In the second instance, students usually manage to model the abstract queue requirements, but fail to see how they can refine their queue into two communicating stacks. The best students manage to model the designs in B but fail to prove the refinement relation (and hence the correctness). However, when asked to implement their B designs (again, in Java) they usually do not make the same programming errors.

5.3 How can we implement a queue of integer pairs from stacks of integers?

This extension to the problem is stated as: we wish to store co-ordinates in a queue with standard FIFO behaviour, and co-ordinates are specified as pairs of integers `(x,y)` where the class methods allow reading and writing of `x` and `y`. Our underlying implementation architecture allows us to store integers on Stacks. Most students quickly realize that they can re-use their designs to the queue of integers problem. Then, they typically propose 1 of these 3 designs:

- DesignA: propose some isomorphic function between integers and co-ordinates. To push a co-ordinate onto a queue we need only transform it into an integer (using our function); and then push this integer onto an integer stack. To pop off a co-ordinate, we just pop off an integer and transform it into a co-ordinate (using the inverse of the transform function)
- DesignB: To push on a co-ordinate `(x,y)` just push `x` onto the integer queue and then push `y` onto the integer queue. To pop off a co-ordinate then pop off an element `a`, pop off an element `b` and return the value `(a,b)`.
- DesignC: Use 2 integer queues - one for the x co-ordinate, the second for the y co-ordinate. To push on a co-ordinate `(x,y)` just push `x` onto the xqueue and then push `y` onto the yqueue; and similarly for popping.

Following their experience from the first simpler design exercise, the students immediately identify unsafe aspects in each of the designs. DesignA will certainly cause problems if we cannot prove the transform function to be isomorphic. DesignB could give rise to unsafe system states if the number of elements on the queue is odd. DesignC could give rise to unsafe system states if the number of elements on each of the queues is not the same. They are then asked to use B to model the designs and the appropriate invariant properties.

5.4 Interesting Design Aspects from the UML

In figure 2, we see an interesting design question: is it possible to combine Design2 and DesignC in order to provide a single temporary stack (shared by both Queue components) that is used for reversing the elements when moving elements from one stack to another inside the queue components?

The advantages and disadvantages of such a design should lead the students to identify that this may give rise to performance and synchronisation issues. However, they still need to ask if such a design is correct. In fact, this question is more subtle than it first seems as the question is really whether the high level structure can provide a framework for the desired behaviour.

5.5 Formal Methods and High-level structure

In figure 2, we have very clear high-level structure; but no clear underlying model of how the components will co-ordinate in order to provide correct Queue (of Int-Pair) behaviour. In the UML we could model this using sequence diagrams or collaboration diagrams. However, without providing these diagrams with formal semantics, it is not possible to establish the correctness of the design to a high degree

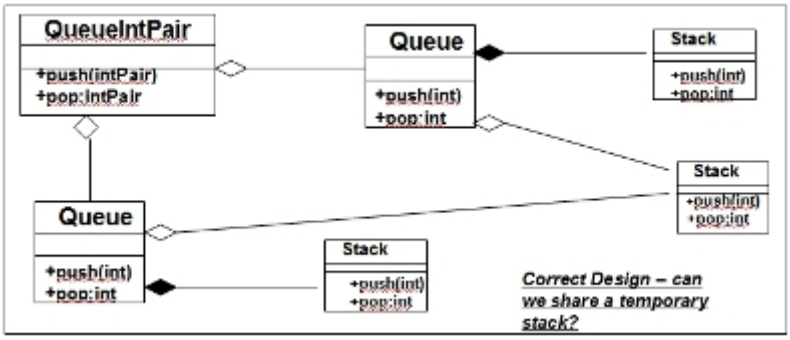


Fig. 2. Can We Build A Correct Design From This High Level Structure?

of confidence. Note that we do not reject the use of UML: the high level structural properties are much easier to represent (and validate) using UML than with a more formal notation.

5.6 Formal Methods Observation: Class invariants are fundamental

This use of invariants is key to integrating the formal with the informal. Every class in the system needs to have an associated invariant which glues together the class associations/components/attributes. For example, in our design in figure 2, we need to specify that the number of elements in each of the component queues must be the same. This is part of the invariant of the system that specifies that if this is not true then the system is in an unsafe state. A formal notation (like B) can then be used to verify that every event that changes the system state (corresponding to a class method) respects the invariant. If this proof cannot be established then the designers must make special note of the fact that this property needs to be tested in the final implementation.

The best lesson that students can take from these types of design exercises is that abstract models must specify (sub)system invariants. Concrete models must guarantee that these invariants are respected. We have evidence that students have (partially) learnt the lesson: in their subsequent software development projects, we have seen invariant checking methods in their implementation models (code) and a structured approach to testing system component invariants at runtime. However, we have yet to see any students formally state and verify their invariants using B.

5.7 Extending and refining the example

The design problem in this paper has been re-used in a number of other different modules —

- Fault tolerance, robustness and reliability: if we know that the fundamental components can fail (following different failure patterns) then can the designs be analyzed in order to reason about the reliability of the system?
- Performance: if we have very strong requirements concerning the speed at which the system must operate then can we analyse the design options in order to reason about system-wide performance in a compositional manner?
- Maintainability and extensibility: if we extend our co-ordinate system so that we

have more than 2 dimensions then do we have to change the design?

- Automated software engineering: how likely is it that such a design could be automatically compiled into code?

We do not claim that this design example is without fault. For example, it is slightly contrived and rather simplistic. However, we have found it to be a good case study for teaching formal design concepts, and for showing that the best approach is to try and integrate different modelling languages.

6 Related Work and Conclusions

The (pedagogic) integration of design with testing is discussed in [10] but does not address the role of formal methods. Formal methods and requirements modelling is treated in [13], where the step from requirements to design is identified as being difficult to teach. In [4] there is a discussion on teaching design by contract using the OCL of UML. Recent work on teaching design by contract[15] advocates a mixed semantic approach to teaching formal design. The EU FrameWork6 project RODIN [3] acknowledges the need for research and development into the teaching of formal design, and the Eclipse-based platform provides specific plug-ins for integrating B and UML[21].

This paper reports on our view on the teaching of formal object oriented design. In a new MSc programme (starting in the next academic year) for *software engineering (of smart devices)* we have attempted to distribute formal methods throughout the programme modules and not to fall into the trap of teaching a stand-alone formal methods module that students fail to relate to all other software engineering material. The future of software design is for students to realize that formal methods are just another tool in their toolbox. This message needs to be transmitted from lecturers to students and from students to industry.

Formal methods lecturers should not just encourage their colleagues to talk about formal methods; they should also make more of an effort to incorporate other aspects of software engineering in their own formal methods modules.

References

[1] Abrial, J.-R., "The B Book - Assigning Programs to Meanings," Cambridge University Press, 1996, ISBN 0-521-49619-5.

[2] Abrial, J.-R., *A system development process with Event-B and the Rodin platform*, in: M. Butler, M. G. Hinchey and M. M. Larrondo-Petrie, editors, *ICFEM*, Lecture Notes in Computer Science **4789** (2007), pp. 1–3.

[3] Abrial, J.-R., M. Butler, S. Hallerstede and L. Voisin, *An open extensible tool environment for Event-B*, in: Z. Liu and J. He, editors, *ICFEM*, Lecture Notes in Computer Science **4260** (2006), pp. 588–605.

[4] Baar, T., D. Chiorean, A. L. Correa, M. Gogolla, H. Hußmann, O. Patrascoiu, P. H. Schmitt and J. Warmer, *Tool support for OCL and related formalisms - needs and trends*, in: Bruel [9], pp. 1–9.

[5] Back, R. J. R., *On correct refinement of programs*, Journal of Computer and Systems Sciences **23** (1981), pp. 49–68.

[6] Bell, A. E., *Death by UML fever*, Queue **2** (2004), pp. 72–80.

[7] Booch, G., "The UML User Guide," Addison Wesley, 1999, ISBN 0201571684.

[8] Borges, R. M. and A. C. Mota, *Integrating UML and formal methods*, Electronic Notes in Theoretical Computer Science **184** (2007), pp. 97–112.

[9] Bruel, J.-M., editor, "Satellite Events at the MoDELS 2005 Conference, Doctoral-Educators Symposium, Jamaica, October 2-7, 2005, Revised Selected Papers," Lecture Notes in Computer Science **3844**, Springer, 2006.

[10] Carrington, D., *Teaching software design and testing*, in: *Frontiers in Computer Science Education*, 1998, pp. 547–550.

[11] Engels, G., J. H. Hausmann, M. Lohmann and S. Sauer, *Teaching UML is teaching software engineering is teaching abstraction*, in: Bruel [9], pp. 306–319.

[12] Evans, A., R. B. France, K. Lano and B. Rumpe, *The UML as a formal modeling notation*, in: J. Bézivin and P.-A. Muller, editors, *UML*, Lecture Notes in Computer Science **1618** (1998), pp. 336–348.

[13] Gibson, J. P., *Formal requirements engineering: Learning from the students*, in: *Australian Software Engineering Conference* (2000), pp. 171–180.

[14] Martin, R. C., "Agile Software Development, Principles, Patterns, and Practices," Prentice Hall, 2002, ISBN 0135974445.

[15] McKim, J. C. and H. J. C. Ellis, *Course module: Design by contract*, in: *CSEET '05: Proceedings of the 18th Conference on Software Engineering Education & Training* (2005), pp. 239–241.

[16] Meyer, B., *UML the positive spin*, American Programmer **10** (1997).

[17] Oliver, I., *Experiences in using B and UML in industrial development*, in: J. Julliand and O. Kouchnarenko, editors, *B*, Lecture Notes in Computer Science **4355** (2007), pp. 248–251.

[18] Reeves, J. W., *What is software design*, C++ Journal **2** (1992).

[19] Richters, M. and M. Gogolla, *Validating UML models and OCL constraints*, in: A. Evans, S. Kent and B. Selic, editors, *UML*, Lecture Notes in Computer Science **1939** (2000), pp. 265–277.

[20] Robinson, K., *Embedding formal development in software engineering*, in: C. N. Dean and R. T. Boute, editors, *Teaching Formal Methods*, Lecture Notes in Computer Science **3294** (2004), pp. 203–213.

[21] Snook, C. and M. Butler, *UML-B: Formal modelling and design aided by UML*, ACM Trans. Software Engineering Methodology **15** (2006), pp. 92–122.

[22] Vaziri, M. and D. Jackson, *Some Shortcomings of OCL, the Object Constraint Language of UML*, in: Q. Li, D. Firesmith, R. Riehle and B. Meyer, editors, *TOOLS (34)* (2000), pp. 555–562.

[23] Younes, A. B. and L. J. B. Ayed, *Using UML activity diagrams and Event-B for distributed and parallel applications*, in: *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)* (2007), pp. 163–170.

Experiences of Teaching a Lightweight Formal Method

R.C. Boyatt[1] and J.E. Sinclair[2]

Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK

Abstract

This paper reports our experience of using a "lightweight" formal approach, Alloy, and its associated tool support for teaching a core undergraduate module introducing formal methods. It considers the benefits and drawbacks in terms of both the student experience and our own aims and objectives for the module. In addition, we link the practical, experimental approach supported by the Alloy Analyzer to educational theory and consider the implications of such an approach to teaching and learning.

Keywords: Formal methods, Alloy, lightweight

1 Introduction

Despite an increasing focus on formal methods education in recent years, many institutions struggle to find a place for formal methods in the Computing (and even Computer Science) curriculum. In many cases, formal methods appear to be a decreasing component rather than progressing towards pervading the curriculum as advocated by many to be the ideal situation [Win00]. Reasons cited include the difficulty of convincing students and staff of the relevance and usefulness of formal methods; the "mathematics phobia" issue (together with falling entrance requirements and declining application numbers); pressure on the curriculum due to the range of required components and the desire to provide popular modules for our students (or customers). On the positive side, there are also reports of tools and teaching approaches finding acceptance in the core curriculum by their own merits (for example, Backhouse's "Algorithmic Problem Solving" approach [Bac06]). However, prevailing conditions remain unhelpful (for instance, in terms of student numbers [UCA07,Kes05]) and formal methods are likely to be under continued pressure.

[1] Email: rboyatt@dcs.warwick.ac.uk
[2] Email: jane@dcs.warwick.ac.uk

At the University of Warwick, recent changes to the Computer Science degree have necessitated a rethink of our approach to teaching formal methods. In this paper we describe the changes made and consider the implications in terms of the student experience and what we can hope to achieve educationally. In particular, we examine our decision to investigate the use of a "lightweight" formal method and consider whether such an approach can provide an appropriate introduction. We identify the distinctive features of lightweight formal methods that are well-suited to a learning environment, and we identify areas in which caution may be needed. Our motivation was to reflect on and assess the impact of specific changes at one institution. This has led us to consider more general aspects of formal methods education, such as the way students interact with tools and the degree to which ideas in formal methods education align with certain strands of education theory.

2 The context for change

Until recently, Computer Science undergraduates at the University of Warwick studied a core Formal Methods module in their second year. Using the Z notation [Spi89], this introduced a range of skills and concepts associated with formal development and verification. Additional core modules in the first two years covered aspects of discrete mathematics and logic which provided a good background. A further, optional module is offered to fourth year students on an MEng degree.

Changes to the curriculum were motivated by a number of worthy considerations, such as allowing students greater choice, but had the unfortunate (from our point of view) result of combining the second year logic and formal methods offerings into a single module. This posed the challenge of providing some meaningful and coherent coverage of formal methods in just a few weeks. Trying to pick out parts of the old module seemed unattractive. Any notation which involved too great a start-up time or which required familiarity with too much syntax would not work. Teaching a subset of syntax may result in insufficient knowledge of the language to be able to specify things correctly. Or worse, students are forced to invent convoluted ways to describe things that only reinforce a view of formal methods as being inaccessible.

This leads to the question of what can be achieved. What are the benefits of teaching formal methods and which, if any, can be derived from just half a term's study? One answer to this might simply be to provide an existence proof to show what formal methods are, how they are used, and why they are useful. However, we want students to be involved with "doing" and to experience for themselves the important skills and benefits of abstract modelling and reasoning, rather than racing to implementation. Many students arrive with an established procedural programming mindset. The growing use of rapid application development techniques are appealing to students who like the immediacy of such approaches. Encountering formal methods challenges preconceived ideas, particularly if the method allows students to spot errors which would otherwise have resulted in a flawed implementation. To support this, it is useful to have a tool which can help reveal such errors and help users explore implications of the specifications they write.

Our approach has been to adopt Jackson's Alloy [Jac06] language, supported by the Alloy Analyser, sometimes referred to as a "lightweight" formal method. We

have now used this approach for two successive cohorts of students.

3 Lightweight formal methods

The concept of a "lightweight" formal method is characterised by Jackson and Wing [JW96] as a method which has a formal basis but is limited in its scope by one or more of: partiality in language; partiality in modelling, partiality in analysis and partiality in composition. The term is used both for the "light touch" application of a traditional method and for methods which are designed specifically to cover only certain aspects of concern. A number of studies demonstrate the former approach, for example, in requirements analysis [ELC+98,AL98]. In formal methods education, Simpson [Sim06] refers to a "light touch" application to demonstrate the relevance of formal methods to professional software engineers.

The relationship between the different uses of the terms "lightweight" and "light touch" raises an interesting question. In this paper however we are concerned in particular with one notable approach specifically designed (and designated) as a lightweight method. Alloy [Jac06], inspired by the state-based Z notation [Spi89], allows the user to express complex structures and constraints using a relational language. The language is intended to be accessible and familiar (at least in part) to programmers used to object modelling with notations such as UML's Object Constraint Language [Gro03]. Properties of the model can be checked for a size of state space suggested by the user. A further lightweight method is Escher Technologies's Perfect Developer [Cro03].

One aim of using a lightweight approach is that a lighter touch can provide quicker (and hopefully automated) feedback on some properties of the system. Such feedback can inform the ongoing construction of a specification and the development of the system itself. In addition it may be hoped that a less ambitious method may well consist of a smaller language and limited user options so that the learning time required to start using the method could be less. Obviously, the partial nature of the method (or of the way the method is applied) means that the results have to be viewed in the light of the method's limitations. For example, a property which has been checked for a state space of limited size obviously does not have the same status as a property which has been verified for all possible values.

The limited nature of lightweight methods has led to some scepticism over their value. Boute characterises such a method as one which "attempts minimizing the user's exposure to mathematics, and therefore supposedly is more suitable to software engineers" [Bou03]. He warns that marketing these methods as accessible to users with little mathematical preparation may end in disappointing results which may discredit formal methods in general. However, Boute is more optimistic about the prospects for use in education: "For many, lightweight formal methods may be very valuable as a means to illustrate more abstract concepts and thereby lowering the threshold for the mathematics relevant to software engineering" [Bou03].

Our requirements were somewhat different from a desire to introduce "mathematics by stealth". Indeed, Alloy does not claim to do this. Our experience is that, to make progress, the user must understand the logical and relational framework of the notation and be able to use it to model at an abstract level. The attractions for

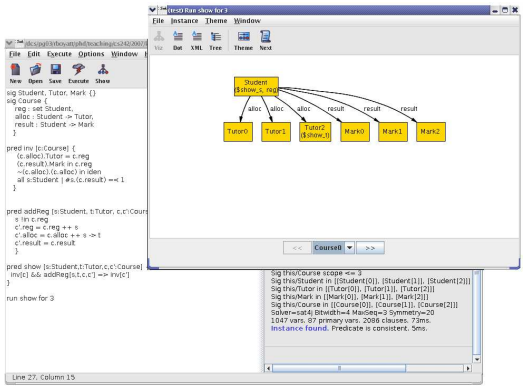


Fig. 1. Alloy Analyzer

us were the compact language, the relative speed with which small models can be written and explored and the guidance and feedback provided by small scope model checking. We consider below how far these are achieved in practice.

The Alloy notation supports modelling by *signature* and *constraint* definition, allowing quite complex structures. No particular methodology is enforced: Alloy can be used in a Z-like state+operations approach or in a variety of other ways to construct traces or describe (and solve) logical problems. Supporting the language is the Alloy Analyzer[3], a tool designed to explore models and check properties of interest. The tool uses SAT solving techniques to find, where possible, an instance of a given property which satisfies the definitions of the model. Limits must be placed on the size of the model to check. Existence of an instance demonstrates satisfiability, but failure to find an instance indicates only that one cannot be found in the current scope. The Analyzer therefore does not provide general results in the way that theorem proving can, but it does provide a quick, automatic way of exploring specifications and generating counterexamples. Jackson's "small scope hypothesis" [Jac06] postulates that most flaws can be detected in a small model.

There are obvious limitations to this approach but, with time constraints in mind, it seemed a possible way forward for students with some mathematical knowledge embarking on their first foray into formal methods. The language is relatively simple and tool support allows students to explore both the approach and the language itself but without being weighed down in large amounts of complex syntax.

We have now been teaching Alloy for two years. In addition to lectures, a good deal of teaching time is devoted to laboratory sessions in which the students can work through practical exercises, receiving help as necessary. Below, we reflect on the students' experiences. Information was obtained by talking to students; noting the questions they ask and the problems they meet; written feedback from module assessment forms and students' assessed work; observing students' interaction with the tool and the way they approach problem-solving.

4 Issues arising from students learning Alloy

This section identifies a number of issues commonly raised by students using Alloy.

[3] Available from the Alloy website – http://alloy.mit.edu

```
pred addReg [s:Student, t:Tutor, c,c':Course] {
    s !in c.reg
    c'.reg = c.reg + s
    c'.alloc = c.alloc ++ s -> t
    c'.result = c.result
    }
```

Fig. 2. Alloy predicate

One particular problem occurs in the shift from procedural programming. This is partly due to the young age at which many students start to explore computer programming, reinforced by undergraduate programming modules. This leads to entrenchment in procedural programming. Students fail to appreciate the differences both in language terms and the shift in thinking required to model declaratively rather than functionally. For example, consider the predicate shown in figure 2 in which logical conjunction is implied between lines. Students view this predicate as a series of sequential operations where, somehow, `c'.reg = c.reg + s` occurs followed by `c'.alloc = c.alloc ++ s -> t`. Another example is the way in which the predicate relates the two courses `c` and `c'`. Rather than mathematically connecting the two courses, we have observed students treating `c'` as a modified version of `c`. That is, where they have omitted part of the connecting statement, students are surprised to find Alloy altering part of the course without telling them! These issues are not confined to Alloy, but the examples generated by the tool force users to confront misconceptions much earlier than with methods we have previously used. The difficulty students encounter with moving from procedural thinking indicate that it is an important consideration for any teaching approach.

Students are frequently observed to struggle with interpreting the visualisation output of the tool. Using the Alloy Analyser can be a somewhat frustrating process. There is certainly a degree of skill required in adjusting the display settings to avoid a confusion of spaghetti lines. Careful thought, examination and sometimes further experiment is required to understand the feedback provided. Efforts are underway to provide an automatic way of arranging the visualisation output in Alloy [RCD+07]. This includes improving the aesthetic qualities of the model and also, by examining the structural properties of the model, manipulating the display to ease understanding of complex results. Even when correctly specifying the model, students can encounter difficulties when examining the feedback from the tool. A common question from students is: "is it the correct answer?". Understanding the output from any tool can be an art in itself, particularly for any reasonably complex models. Further experiment can be required to interrogate the tool in a different way or to find further examples. With practice this improves but it takes time and encouragement to cultivate the spirit of enquiry and practical skills necessary to enable students to perform further meaningful experiments on their model.

The importance to students of tool support should not be underestimated. It implies a certain level of maturity in the language, a commitment to the techniques presented beyond that of an "academic exercise" and the ability to directly experiment with the approach not afforded with a paper and pencil exercise. The Analyzer leads students to develop a model incrementally and explore it as they go along. This is popular with the students and helps reveal many errors. Laboratory sessions are needed to support this. Issues relating to tool use are discussed below.

Another challenge commonly faced in formal methods education which has been

discussed elsewhere [RJ04] is that of motivation. Any curriculum which compartmentalises formal methods risks making this worse. This year, we have started to link the formal approach in small ways to material from other modules. For example, Alloy is well-suited to exploring logical puzzles and mathematical constructs (such as partial orders and lattices) which the students meet in other contexts. This has proved useful in making connections and introducing the tool as a useful aid (particularly with visualisation) before considering it in the context of specification. This was well-received by the students and is an area we plan to develop.

5 Issues of Students and Alloy

There is obviously a limit to what can be achieved in a short space of time no matter what method is used. Bearing this in mind, we were interested to see to what extent our aims for the module could be met.

Does the approach show formal methods as useful and relevant?
The response to this is positive from two separate aspects. Firstly, only a small amount of prerequisite material needs to be covered before some quite interesting case studies can be tackled. Even students who found the material challenging expressed a respect for the subject and were interested by what could be achieved in small case studies. Secondly, in their interaction with the tool, students could see that flaws in their thinking were being picked up and demonstrated to them at a very early stage of development. This again helps to show that formal methods can extend our understanding of a specification and catch errors that otherwise head unchecked into implementation. Whilst the possibilities are not as broad as with a full formal approach, we were encouraged by the positive response of the students.

Does the approach encourage abstract thinking?
Writing an abstract model challenges students' view of procedural behaviour. Perhaps it is more accurate to say that the feedback from the Analyser provides the challenge since, very often, simply writing the statements does not bring home the point that the model is much different to a program. It is interesting to observe how effectively a small amount of interaction with a tool can bring home points that we might talk about at length in lectures! Both the explanation and the exploration seem to be important here. Why abstraction is useful and what it allows us to do are much easier to address when students can connect to the ideas in practice. As an initial step to challenge students and present ideas of modelling and abstraction, Alloy has been as effective as a full formal approach, and perhaps even better due to the immediate feedback from the tool.

Does the approach hide the mathematics?
The model checking approach means that students do not have to battle with a proof system or understand how to direct a theorem prover. However, they work with an expressive notation and build fairly complex structures and relationships. However, this is not because the system is "hiding" anything from the user: it is simply finding instances. Obviously, we would want students to go on to study verification and refinement, but as a starting point, Alloy is quite honest in what it presents. Another interpret ation of this question might be whether users can effectively write models without understanding the underlying logic and relational representation.

Whilst the use of a programmer-friendly syntax may make statements appear more approachable to users, our experience is that to make any progress students need to understand the relational and logical levels of the notation and to become competent with abstract modelling. This perhaps means that the notation is not quite so user-friendly as we might like it to be. To use it, you have to understand it.

Does the tool help student learning?

A number of authors have suggested that it is not necessary for a formal method used in an educational setting to have tool support: indeed, it is also suggested that it may be better for students to work with paper and pencil. For example, Bayley *et al* [BLM06] raise the issue of students becoming discouraged by tools which reject their early efforts with cryptic messages. They also refer to the danger that users may take false assurance from a tool which checks only for syntactic correctness. These are real concerns, but balanced against them is the very apparent fact that, unchecked, it is all too easy to write rubbish. This is not confined to students and new users! Without feedback, the user's ideas are not scrutinised and challenged, and human appraisal may leave many errors undetected. Of course, proving properties of a formal specification is one way of exploring it and demonstrating that it has "correct" behaviour. However, many errors discovered when using Alloy relate to validation, that is, to building "the right program". By seeing examples, a user realises that they have not modelled the system they had in mind. Or perhaps, a requirement is only articulated as a result of behaviour viewed in the model. Verifying a property of a specification is only helpful if it specifies the system we want.

Another class of errors we observed relates to the work of Vinter *et al* [VLK98] which suggests that errors in mental constructions of certain logical expressions are extremely common, even amongst experienced users. This relates to Boute's comments [Bou03] on the way that false conclusions are commonly drawn. Vinter *et al*'s work confirms in a formal methods context some well-known psychological results which demonstrate that logical reasoning is subject to cognitive bias and systematic misconstrual. While modelling in Alloy is just as prone to this as any other formal method, the feedback from the analyzer does serve to confront the user with instances that reveal and challenge such misconceptions.

As discussed in the previous section, another useful aspect of the tool and the exploratory nature of using it is that students relate well to this way of working. This should not outweigh the question of what is appropriate educationally, but if the two are compatible then an approach which puts students at ease and which appears to them to be relevant and usable is an advantage. We have found that using the Alloy Analyser has been very beneficial in demonstrating the ideas from lectures and notes and the practical aspect is very important for effective learning (see Section 6). However, we do have some concerns about the way the tool is used.

Using a tool to provide feedback to inform the specification is very beneficial. However, we have also observed students replacing guided thought with blind trial and error. When a student reaches the point where they are not thinking about the underlying specification but repeatedly trying out minor changes with no real understanding, the process changes from guided exploration to thoughtless hacking. Even if a "good" answer is obtained (such as no counterexamples found) the student really has little idea of what that means. This in turn can give rise to an unjustified

faith in the model with the feedback from the tool misinterpreted. This is an area we would like to explore further to see if this is a significant problem and to understand at what point students abandon logic and how best to guide them.

6 Formal Methods Education

The way of working supported by Alloy allows small experiments to be performed on partial models, promoting exploration and interactive model-building. Properties of the model can be explored without having to specify a complete system. This is similar to the idea of "extreme specification" [Cro03] [4] in which functionality of a specification is developed incrementally and guided by feedback at each iteration. Examples (and counterexamples) that satisfy the current model can be generated. Then, as these examples are understood in the context of the given model, more complex and abstract ideas arise. Experiment and visualisation help shape the mathematical understanding of a model and give insight beyond an understanding of the mathematical symbols. This is similar to Lakatos' notion of experimental mathematics [Lak76] whereby understanding is developed through exploration and (failed) proof. With a traditional approach, students typically have a view of formal methods that is steeped in rule and rote, where the procedure is fixed and inflexible. We have observed that students can lose touch with the "reality" of requirements and connections to an end result, adopting a formalist approach in which they become focused on manipulating symbols. Experimentation and visualisation, for example, of the effects of a particular operation, can help to reconnect a student to the purpose of the task. In addition, formal insight often comes from playing informally with a model. Through small experiments that inform the developer's knowledge, the understanding of the model grows. These stages typically precede any further formal representation of the model. The search for the finished model is not blind but guided by experience and meanings extracted from interaction.

Students' use of Alloy can be seen as a particular kind of exploratory learning. Papert's constructionism [Pap93], founded on Piaget's constructivist ideas, argues that students learn best when actively engaging in building knowledge structures which can be discussed and further examined. Papert and Harel are careful to establish that constructionism is not simply "learning-by-making" [PH91]. Constructionism places special emphasis on the correspondence between "making in the world" supporting the mental processes of knowledge construction. Students are constructing a public "product that can be shown, discussed, examined, probed and admired" [Pap93, p. 142]. In practical terms, this means that a good deal of time is devoted to laboratory sessions in which students can carry out their explorations in an environment where help and direction is on hand. Model development in Alloy is similar to the "bricolage" style of construction [LS68] where, as the model is built and evolves, so does the student's understanding of the situation. Testing of the model is integrated into the task of construction. The ideas embodied in a model are continually open for examination and, if necessary, revision.

Ideally, students develop a toolkit for tackling formal problems; providing them

[4] Variously credited to Susan Stepney and Helen Treharne!

with the capacity to extrapolate beyond the information given and to ask questions of their model beyond the scope of the original question. The constructionist approach to learning is connected to motivation. Students must be willing and able to learn about formal methods. This only succeeds if students can see its worth and can engage with the material appropriately. If the students find some personal interest and stake in the material, they are more likely to gain a deeper and fuller understanding of the subject. This connects with some deeper issues of the role and place of formal methods in the Computer Science curriculum.

7 Conclusion and future work

This paper is certainly not an endorsement for taking curriculum time away from formal methods. Relegating them to a small corner of the timetable leaves little room to explore ideas fundamental to any Computer Science degree. Also, it makes these ideas appear unusual and less relevant to "normal" practice. The use of Alloy described here is as an introduction to students who have already had reasonable exposure to discrete maths and logic. There are many topics, such as verification and refinement, which we do not have time to explore and which would require moving beyond the bounds of the basic Alloy provision. Concerns also arise over the way that some students, particularly weaker ones, interact with the tool. Without engagement with the model and interpretation of feedback, a trial-and-error hacking approach can result. Occasionally, students appear to lose all sense of context and reality in an attempt to gain a formal tick of approval from the tool. It is also possible to gain a false sense of security by misinterpreting the tool's feedback. Despite these concerns, our overall experience has been very positive.

We may still strive for a situation in which formal methods pervade and inform the curriculum, but in the meantime a practical approach is needed. We have investigated one possibility - and therefore our findings apply to Alloy in particular rather than to other methods and approaches referred to as "lightweight". This has worked well for our circumstances. In fact, it is interesting to consider whether this approach would provide a good introduction even if time allowed a much fuller exploration of formal methods. Although partial in some aspects, it provides an approachable first encounter which is honest in its limitations and which can demonstrate a real usefulness in identifying flaws. Whether this is enough to attract more students to return for the later, optional formal methods module remains to be seen.

In this paper we have noted some of the benefits and limitations of the use of an interactive tool. This is an area we are keen to explore further. By making some minor alterations, we are planning to record information about how the tool is used and what steps a student takes in creating and improving a model. This will allow both qualitative and quantitative analysis of aspects such as progress made towards a correct solution, iterations required, timing between checks.

References

[AL98] S. Agerholm and P. G. Larsen. A lightweight approach to formal methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, volume 41. Springer LNCS, 1998.

[Bac06] Roland Backhouse. Algorithmic problem solving - three years on. In *Teaching Formal Methods: Practice and Experience*, 2006.

[BLM06] Ian Bayley, David Lightfoot, and Clare Martin. Teaching the oxford brookes formal specification module. In J.P.Bowen P. Boca and D.A.Duce, editors, *Teaching Formal Methods 2006*, Oxford Brookes University, 2006. Available at: http://cms.brookes.ac.uk/tfm2006/.

[Bou03] Raymond Boute. Can lightweight formal methods carry the weight? In David Duce et al., editors, *Teaching Formal Methods 2003*, Oxford Brookes University, 2003. Available at: http://cms.brookes.ac.uk/tfm2003/.

[Cro03] David Crocker. Teaching Formal Methods with Perfect Developer. In David Duce et al., editors, *Teaching Formal Methods: Practice and Experience*, Oxford Brookes University, 2003.

[ELC+98] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modelling. *IEEE Transactions on Software Engineering*, 24, 1998.

[Gro03] Object Management Group. Object constraint language specification v. 2, 2003. Available from: http://www.omg.org/technology/documents/formal/ocl.htm.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[JW96] D. Jackson and J. M. Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[Kes05] Michelle Kessler. USA Today, Fewer students major in computer . http://www.usatoday.com/tech/news/2005-05-22-computer-science-usat_x.htm, 2005.

[Lak76] Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.

[LS68] C. Levi-Strauss. *The Savage Mind*. University of Chicago Press, 1968.

[Pap93] Seymour Papert. *The Children's Machine*. Basic Books, 1993.

[PH91] Seymour Papert and Idit Harel, editors. *Constructionism*, chapter 1. Ablex Publishing, 1991.

[RCD+07] Derek Rayside, Felix Chang, Greg Dennis, Robert Seater, and Daniel Jackson. Automatic visualization of relational logic models. In *First Workshop on the Layout of (Software) Engineering Diagrams (LED'07)*, 2007.

[RJ04] J.N. Reed and J.E.Sinclair. Motivating study of formal methods in the classroom. In *Teaching Formal Methods*, volume 3294. Springer, LNCS, 2004.

[Sim06] Andrew Simpson. Logic, damned logic and statistics. In J.P.Bowen P. Boca and D.A.Duce, editors, *Teaching Formal Methods 2006*, Oxford Brookes University, 2006. Available at: http://cms.brookes.ac.uk/tfm2006/.

[Spi89] J. M. Spivey. *The Z notation : a reference manual*. Prentice-Hall, 2nd edition, 1989.

[UCA07] UCAS. UK Universities and Colleges Admissions Service. http://www.ucas.ac.uk/, 2007.

[VLK98] R. Vinter, M. Loomes, and D. Kornbrot. Applying Software Metrics to Formal Specifications: A Cognitive Approach. *IEEE Metrics*, pages 216–223, 1998.

[Win00] Jeanette Wing. Weaving formal methods into the undergraduate curriculum. In *Proceedings of the 8th Int. Conf. on Algebraic Methodology and Software Technology*, volume 1816. Springer LNCS, 2000. Available at: www.cs.utexas.edu/users/csed/FM/docs/Wing-abstract.pdf.

Introducing Alloy in a Software Modelling Course

James Noble, David J. Pearce, Lindsay Groves [1]

Software Engineering
Victoria University of Wellington
Wellington, New Zealand

Abstract

Engineering degree programmes require courses in "foundational science" but many traditional foundations — calculus, chemistry, or quantum physics — are not generally relevant to Software Engineering. To address this requirement, we developed a new first-year course — Introduction to Software Modelling — that introduces the principles and practices of Software Engineering, beginning with domain analysis, finding classes and use cases, and finishing with building models in Alloy. In this short paper, we cover the rationale for the course; outline the course structure; and describe our experience with teaching Alloy to fifty first-year engineering students with minimal backgrounds in programming or logic.

1 Introduction

According to the *ACM/IEEE Curriculum Guidelines for Software Engineering* [7], Software Engineering programmes require "foundational science". Many foundations appropriate to other engineering disciplines — calculus, chemistry, or quantum physics — are not, in general, directly relevant to Software Engineering: however, most software programmes include such courses, either because existing programmes incorporate them within a common first year; because a "general" engineering background is useful so that graduates can work on interdisciplinary projects; or, for structural reasons within engineering schools.

In 2007, Victoria University of Wellington began the process of moving its existing engineering offerings — in electronics and instrumentation, taught as part of a B.Sci.Tech., and in software and network engineering taught as part of a BIT or B.Sc.(Hons) degrees — into a common engineering programme labelled with a BE degree and meeting all the criteria for engineering accreditation. Our degree is required to follow VUW's strategic plan, appealing to non-traditional engineering students (women, Tangata Pasifika) as well as those students who would otherwise have considered VUW's BIT or B.Sc. The programme also must provide space

[1] Email: {kjx,djp,lindsay}@mcs.vuw.ac.nz

Week	Topic	Lab	Assignment
1.	**Requirements** — introduction to the course, to software engineering, software modelling	1.	
2. 3.	**Use Cases I** — finding use cases, essential use cases **Use Cases II** — actors, system requirements, use case quality	2. 4.	Use Cases
4. 5.	**Domain Analysis I** — finding classes, class diagrams, inheritance **Domain Analysis II** — class diagrams, recursion, class diagram quality	5. 6.	Classes
6.	**Evaluation** — model-world connection, quality, test revision	7.	
	Midterm break		
7. 8.	**Invariants I** – terms test, introduction to writing invariants **Invariants II** — checking invariants, dealing with ambiguity	7. 8.	Invariants
9. 10.	**Alloy I** — introduction to alloy, syntax, extreme alloy! **Alloy II** — ordering, packaging, case study	9. 10.	Alloy
11. 12.	**Alloy III** — execution traces, traces in Alloy, FSMs **Alloy IV** — revision	11. 12.	Alloy II

Fig. 1. SWEN102 course plan

for students to take a minor in leadership, innovation, entrepreneurship or other subjects of current focus.

VUW has taught a software engineering major as part of the BIT since 2002, including several courses from Information Systems, and 800 hours of work experience [5]. The transition to a BE offered us the opportunity to revise the Software Engineering programme after it had been offered for around five years. We were able to remove the requirement for students to complete business-focused courses from the information systems stream, and to find something to address the requirements for engineering foundations directly in a way that would suit our students and the actual requirements of software engineering.

Our initial proposal was to teach a formal methods course, introducing students to propositional and predicate logic, and their application in describing and reasoning about software systems. After some debate, it was decided that this was not appropriate for several reasons. In particular, our students already take a course in Discrete Maths and Linear Algebra, and we did not want students to regard this as "another maths course"; and anecdotal evidence from similar universities suggests that such a course would not be popular. We eventually came to the conclusions that the main rôle of formal methods in a software engineering programme, at least in the early stages, is in modelling, and that we would be better to give a gentle introduction to formal methods within the context of a more general course on software modelling. This way, we hoped to present formal and informal approaches as different points in a spectrum of approaches to describing software systems, rather than being totally different subjects. We also wanted to ensure that students see software modelling as a useful way of understanding systems, rather than just an exercise in learning new notations, so we felt it was important that any formal notation we used be supported by tools which allowed students to explore the consequences of the models they created. After a brief consideration of our requirements and the alternatives available, we decided that Alloy [3] seemed to be the most appropriate option, though we had some reservations that the non-standard way of describing sets might cause some confusion.

The result was a new first-year course — SWEN102: Introduction to Software Modelling — that introduces the principles and practices of Software Engineering, beginning with domain analysis, finding classes and use cases, writing invariants about those models, and finishing with building those models in Alloy. SWEN102 is taught in the second semester of the first year, along with a second course in programming, a course in discrete mathematics, and various other courses (statistics, physics, or information systems) depending on students' choice.

The course objectives are that, by the end of the course, students should be able to:

1. Explain the relationship between models, software, and the real world;
2. Describe software systems in terms of models other than code;
3. Translate informal descriptions of software systems into structured textual and graphical models;
4. Create well-formed engineering models in informal notations and formal languages;
5. Manipulate, analyse, and verify properties of these models, both by hand and with tool support;
6. Evaluate the qualities of models and software systems.

As with all VUW courses, this is offered in one 12 week semester, generally split into two 6-week half-semesters. Objectives 1, 2, and 6 are threaded throughout the course; objectives 3 and 4 are primarily in the first half, and objective 5 in the second half.

2 Course Plan

Figure 1 shows the overall course plan of SWEN102. There are four main topics covered in the course: Use Cases; Domain Analysis (class diagrams); Invariants; and Alloy. Each topic is covered in a two-week integrated block of lectures, labs, and assignments, except for Alloy which is covered in two blocks (i.e. four weeks). The first half contains two blocks; the second half three. The remainder of the time in the first half is taken up with an introductory first week while the week before the break (week 6) concerns evaluation and test preparation.

2.1 Use Cases + Domain Analysis

The first two topics — use cases and domain analysis — are relatively traditional introductions to object-oriented modelling. The main difference in our approach is that we use very basic, minimalist modelling notations, teaching students how to model the "real world" in that notation, and then how to evaluate models in that notation, rather than providing (and teaching the fine details of) a rich modelling notation. So, for use cases we teach *essential use cases* [1], but without any control-flow notation, or exceptional flows. For domain analysis, we teach *class diagrams*, but classes only have attributes (not methods), undirected relationships with arity, single inheritance, and abstract classes.

In both cases, we encourage students to begin via textual analysis: reading a text and identifying *verb phrases* (for candidate use cases) and *noun phrases* (for candidate classes). Then, following the principle of separating creation from evaluation [1], we encourage the students to winnow down the candidates, removing duplicates, clarifying ambiguity, identifying and resolving inconsistencies. We teach a basic overview of the diagrammatic notation first (i.e. for class diagrams or use-case diagrams), and only then follow up with the more specific details of the notation (i.e. the steps of a use case, the attributes or a class, the arity of a relationship, etc). At this stage (well into the second week of the block) we may introduce some refinements to the notation, such as use-case preconditions, or abstract classes. Finally, we finish each block by discussing quality attributes of the models.

2.2 Invariant Analysis + Alloy

SWEN102 then proceeds to two formal topics: *invariant analysis* and the *Alloy modelling language*. These are rather less traditional in a first-year course.

The block on invariant analysis is structured the same way as before. We begin by identifying candidate invariants (e.g. no property on a Monopoly board may contain more than five houses) via textual analysis of informal requirements documentation. We also explain how invariants can be identified via analysis of class diagrams for the system in question: inspecting first classes, then relationships, then the whole diagram to find class invariants, relationship invariants, and global invariants respectively.

At this stage the candidate invariants are written informally in English, and we perform the same reviews as for other model elements: establishing vocabulary, removing ambiguity and duplicates. Our aim here is that rather than treating invariants as special "formal" elements (and thus putting off students) we teach invariants, and the practices for discovering and refining them using the same techniques we have introduced for more "informal" model elements. Once students have identified and clarified elements in English, we introduce the Alloy constraint syntax to express invariants.

The Alloy tool itself — and the Alloy language — is not introduced until the final two blocks of the course — at this point students have already completed an assignment on identifying and writing invariants *but without any tool support.* This is (these days at least) not the standard approach to teaching a programming language — even though Dijkstra may approve of writing out programs before compiling them! Our main reason for delaying the introduction of Alloy was to avoid the focus of the course being consumed by the Alloy notation/tool; that is, we wished the main emphasis of the course (as outlined in the objectives) to be on epistemology of software modelling, rather than on the technical tools we choose to employ. Of course, to teach the principles of modelling requires the use of actual notation and techniques.

3 Student Work

The detailed design of course blocks is integrated with lectures, labs, and assignments forming a complete whole — see Figure 2. Each block consists of six lectures,

starting with introductory, informal material and moving on to slightly less introductory material. Laboratory sessions are tightly tied to lecture content, providing students with practical experience (and feedback) on the techniques presented in lectures and, most importantly, plenary sessions in which the tutors give guidance not only on the techniques, but also the engineering philosophy and the epistemology of modelling.

Week	1			2		
Lectures	1	2	3	4	5	6
Prework	1			2		
Laboratories		Lab 1			Lab 2	
			Assignment			

Fig. 2. Block Structure

Each lab is preceded by an individual prework exercise (which is graded, along with participation in the lab). These prework exercises typically require students to attempt a small exercise, unsupported, that will then be repeated and expanded upon in the lab, and will finally be tested in the block assignment. The first three assignments essentially take a problem from a textual description to use cases, class diagrams, and invariants respectively: additional problems include critiquing or debugging existing models — for example, for use cases, evaluating concrete interactions against use case descriptions, or, for class diagrams, generating object diagrams or determining whether a given object diagram is consistent with a class diagram.

4 Alloy

The two final blocks present Alloy at a very basic level. The key to our presentation is that it draws strongly on the informal background the course has provided: invariants are first introduced simply as a counterpoint to classes and use cases. Our presentation of domain analysis has already introduced class diagrams, object diagrams, and the relationships between the two — both the notion of an object diagram representing some part of the real world, and also how an object diagram may be generated by (or consistent with) a particular class diagram.

The nice thing here is that Alloy's *signatures* correspond closely to class declarations with which students are already familiar. Likewise, students have been prepared for Alloy's analyser (which searches for instances of a model, or a counterexample of an assertion) by manually generating object diagrams from class diagrams. Alloy's *visualiser* is crucial here — displaying instances and counterexamples of a model in a very similar format to UML object diagrams. Thus, students are gently introduced to the concepts needed for Alloy via informal techniques, *whilst learning skills which are ubiquitous in industry.*

5 Monopoly Example

The Monopoly game provides an interesting example which illustrates the main techniques developed in the source. Students are put in the context of developing

"A Monopoly board is divided into two main areas: the border region; and, the inner region. There are 40 locations on the board, each of which is either: a property or a special area. Properties have a name, a price, a mortgage value, a rental value and can be bought or sold; the special areas are: "Go", "Free Parking", "Jail", "Goto Jail", "Chance" and "Community Chest".

A rental property is either: a street, a railway or a utility. Streets have a colour and are organised into colour groups, consisting of exactly three streets. Each street can have up to five houses or one hotel built on them; houses may only be built on a street when all the streets in its colour group are owned by the same player; a hotel may be built on a street which has four houses and, when built, all houses on that street are removed. The cost of building a house or hotel on a particular street depends upon the street in question.

...

Each player maintains a list of properties they own, and the amount of cash they have left. When it's their turn, the player roles the dice and moves the number of locations equal to their sum in the clockwise direction. If the person lands on a property owned by another, he/she must pay its rent to the owner (note, this only applies if the owner is not in jail). If the player lands on an unowned property, they have the option to purchase it for the stated price."

Fig. 3. An informal, cut-down specification of the Monopoly game.

a system for playing Monopoly, based on the (cut-down) specification in Figure 3. From this, students would be asked (depending on the assignment) to produce either *essential use-case diagrams, class diagrams, object diagrams* and/or *Alloy invariants*. The general approach we teach is to begin with a textual analysis of the specification to identify *candidate classes*, *candidate use-cases* and *candidate invariants*; these form the basis of the class diagram(s), essential use case(s) and (eventually) the alloy model itself. Figures 4, 5 and 6 illustrate the kind of answers we expect. The essential use-case cards in Figure 4 capture the interactions between user and system, highlighting the *user intentions* (i.e. what the user wants to achieve, rather than the actions they will take to achieve it) and the *system responsibilities* (i.e. what the system, at a high-level, must do, not just how it must respond). Figure 5 illustrates a typical UML class diagram for this kind of problem. The key challenge for the students here is to recognise the relationships between classes, and identify simple multiplicity constraints. Finally, Figure 6 illustrates some extracts from an Alloy model of the Monopoly specification; clearly, this is much more detailed and precise than the other models; but, it is also much harder for the students to comprehend and appreciate.

We believe that developing these models helps the students, in the first instance, simply to *comprehend* the specification document. Once passed this hurdle, they can begin (to a limited extend) to *reason* about the specification, and (eventually), to *formalise* it. Perhaps unsurprisingly, many students do not initially understand the value of doing this. Therefore, aside from the usual rhetoric on documentation and design (neither of which are really meaningful to them, given their limited knowledge), we also deliberately plant problems of *ambiguity*, *inconsistency* and *incompleteness* into the specification, and ask students to identify these at the end of the assignment.

6 Evaluation

As part of VUW's quality assurance process, we conducted a standard evaluation of SWEN102, and augmented this with an additional questionnaire giving students the opportunity to provide both qualitative and quantitative feedback on the course design. Unfortunately, few students provided qualitative feedback, so we restrict ourselves to reporting quantitative results here. The quantitative questions em-

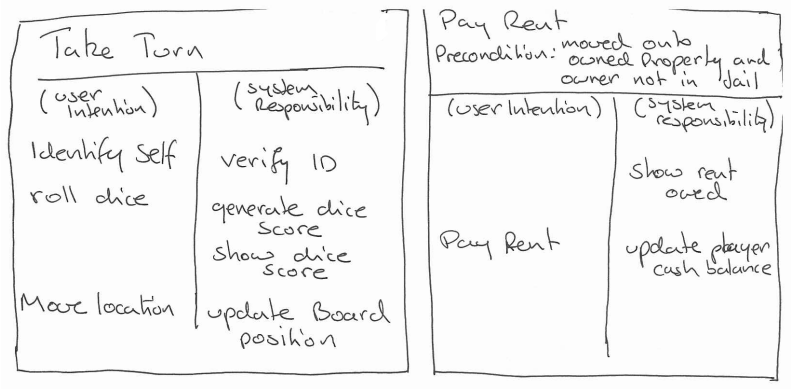


Fig. 4. Example essential use-case cards for the monopoly specification.

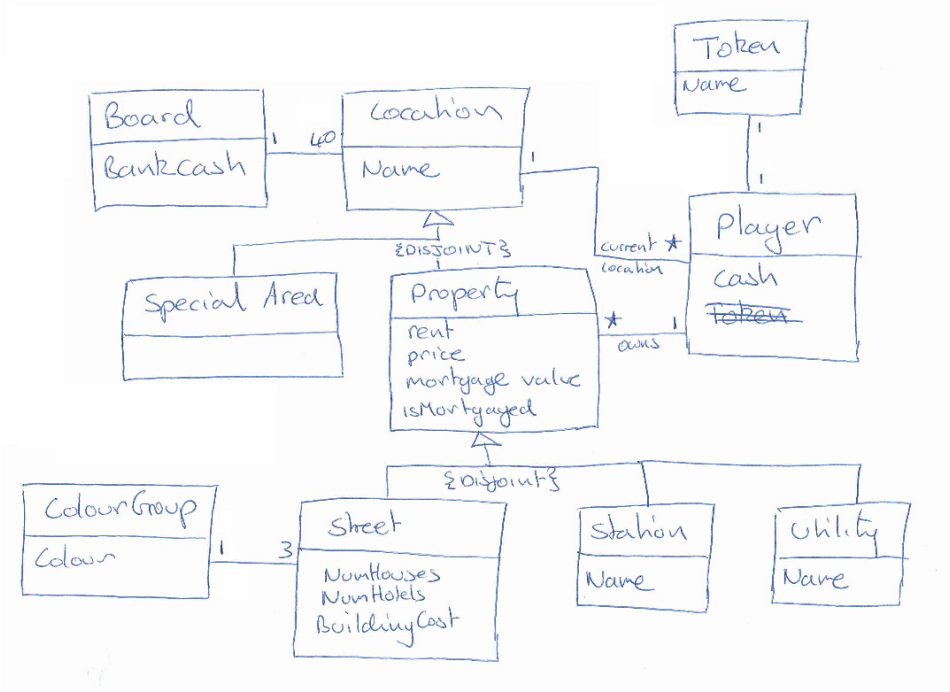


Fig. 5. An example class diagram for the monopoly specification.

```
abstract sig Building {}
sig House extends Building {}
sig Hotel extends Building {}

sig Property {
  owner: lone Player,
  buildings: set Building,
  ...
}{
 // Up to four houses or one hotel on property
 #buildings <= 4
 // at most one hotel on property
 lone h : Hotel | h in buildings
 // either a hotel or houses (not both) on property
 all h : Hotel | h in buildings implies no h' : House | h' in buildings
 ...
}
```

Fig. 6. Extracts from an Alloy model of the Monopoly specification.

ployed a 5-point Likert scale ("Strongly Agree/ Agree/ Neither Agree nor Disagree/ Strongly Disagree" unless otherwise noted) and employ both objective and affective questions. We received 39 questionnaires from 54 students nominally enrolled in the course when the evaluations where conducted.

Based on the quantative feedback, the students appeared to have enjoyed the course (only 1 student out of 39 questionnaires "disagreed", none "strongly disagreed") and it generally met its objectives: 2 students disagreed or strongly disagreed that "Informal modelling techniques (use cases, domain analysis) are clearly relevant to the work of a software engineer", and only 7 disagreed (or strongly disagreed) that "Formal modelling techniques (invariants, Alloy) are clearly relevant to the work of a software engineer". The course organisation and assessments "helped me learn" (one disagreement); similar proportions of students consider that SWEN 102 encouraged creative and critical thinking. Around 80% of students considered the workload "about right" (equally balanced on either side),

Considering quality measures, most students considered the course at least "good" (17 students), "very good" (another 17), or "excellent" (3 students). The tutorial streams were also highly valued by students: 15 strongly agreeing, and another 13 agreeing with the statement that "Tutorials provided me with good support for learning lecture content", and slightly more respondents agreeing (17) or strongly agreeing (17) that "The laboratory sessions supported my learning overall in SWEN102".

Oddly, 60% of students agreed or strongly agreed that the "online components of the course contributed to my learning" — this is surprising, given that the only online components were a course outline, copies of lectures and assignments, and a web forum used mostly to debate the choice of music to be played before lectures. Although we received only a few qualitative comments, the most common suggestion for improvement was that the course notes on the website should be updated more quickly after each lecture.

Finally, given our aim to provide an appropriate and engaging replacement for calculus in a software engineering first year, only two students considered that such a course would be more appropriate than SWEN102.

7 Discussion

We tracked student longitudinal engagement with SWEN102 by recording the number of students submitting work for each assessment item in the course, and the grades we awarded for those pieces of work. These results are shown in Figure 7, including the grade distributions for the six assignments, the terms test (given following the midterm break, after assignment 3), and the final exam. The stacked bar chart shows the grades returned for each piece of work: note that both the bottom two categories are fail grades: a D grade (roughly 40-50%) is awarded generally for work that is below a passing standard, with an E grade (less than 40%) for the remainder (which generally indicates the student submitted few, if any assignments and did not sit the exam).

The most important trend in this figure is the steadily increasing number of "E" grades across the assignments: this shows that a number of students choose not to

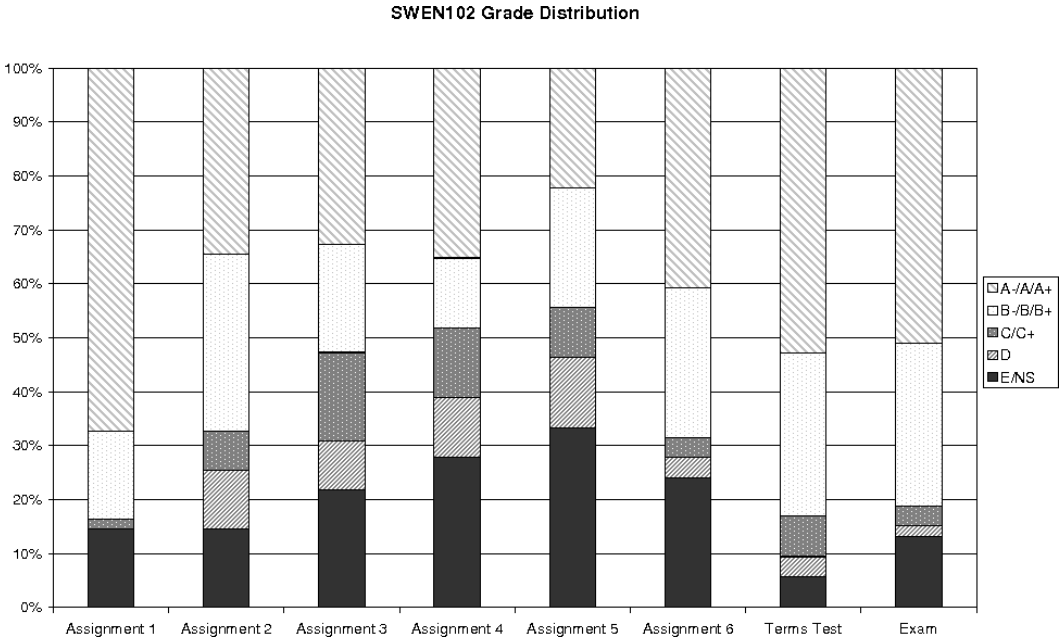


Fig. 7. SWEN102 Assessment Results

participate in assignments as the course went on — particularly in assignment 5 — although many students returned for assignment 6 and the final exam. Assignment 5 was the first Alloy assignment, so this may account for the relative lack of participation in that assignment, but that does not account for the significantly better result in the final assignment. Given that SWEN102 runs in parallel with a second course on programming (basically the ACM CS2) we suspect that the workload required in the second half of that course may lead some students to miss work in SWEN102, however we have not yet tested this systematically.

The other important implication of Figure 7 is that around 40% of students managed some form of "A" grade (roughly 75% or above) in assignment 6, and 50% in the terms test and exam. Given that this is a first year course, and that students have self-selected into the engineering program, this grade profile should not be surprising. Also, students in the bottom third of the "B" range, or in the "C" range, cannot continue with engineering in future years, even if they nominally pass all their courses. Given the design of our assessment, this means that the majority of students have certainly achieved the following basic facilities with Alloy: finding invariants from textual descriptions of problems, or from class diagrams; translating invariants between English and Alloy; writing basic invariants in Alloy; explaining the Alloy constraint language; and editing existing Alloy models. Designing Alloy models from scratch is significantly harder, and we doubt most students would be able to follow the process we teach, from free-text to a complete Alloy model, with just SWEN102. But, on reflection, this is not surprising for what is still a first-year course. Indeed, few of our students are able to build non-trivial Java programs after two first-year programming courses and, hence, modelling, integration, and design are among the objectives of our second-year.

There are relatively few courses on modelling — especially with a formal component — early in software engineering degrees [2,4]. Even in the "Software Engi-

neering First" curriculum design [7], the introductory Software Engineering courses (SE101 and SE102) remain focused on programming, albeit with a "software engineering" focus. Formal techniques and modelling are only introduced at second year. In contrast, our SWEN102 design introduces both in the second half of first year: our experience to date indicates that students are able to grasp both informal and formal modelling at a basic level in that stage of their education. How this flows through the rest of their programme, of course, only time will tell.

8 Conclusion

102. One can't proceed from the informal to the formal by formal means.
Alan Perlis, Epigrams on Programming [6]

In this short paper, we have described the design and rationale behind SWEN102, a first year course on software modelling offered for the first time at Victoria University of Wellington in 2007. Based on our experience so far, we consider that explicitly teaching software modelling — both informally and formally — is very beneficial to students at the start of their software engineering education.

Acknowledgements

Thanks to Peter Andreae, Tom Angelo, Irina Elgort, Thomas Kühne, Stuart Marshall, Alex Potanin, Ian Welch, and all those who worked on the design of SWEN102, and to our dedicated tutors — Neil Ramsay (head tutor), Hugh Davenport, Carlton Downey, Claire Lenihan, and David Stirling, without whom SWEN102 could never have happened.

References

[1] Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.* Addison-Wesley, 1998.

[2] A.J. Cowling. The role of modelling in the software engineering curriculum. *Journal of Systems and Software*, 75(1-2), Jan-Feb 2005.

[3] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, April 2006.

[4] Thomas Kühne. Fighting the "formal is futile" fallacy. In Miroslaw Staron, editor, *Proceedings of the 3rd Educators Symposium of the 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, volume 2007:01 of *Research reports in Software Engineering and Management*, page 3, Göteborg, Sweden, 2007. Department of Applied Information Technology, IT University of Göteborg.

[5] Joel W. Pauling and Peter Komisarczuk. Review of work experience in a Bachelor of Information Technology (BIT). In *Proc. ACE*, pages 125–132. Australian Computer Society, Inc., 2007.

[6] Alan Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[7] The ACM/IEEE Joint Task Force on Computing Curricula. Software engineering 2004: Curriculum guidelines for undergraduate degree programs in software engineering, August 2004.

Teaching How to Write a Proof

Adam Naumowicz[1,2]

Institute of Computer Science
University of Białystok
Poland

Abstract

In this paper we present methodological foundations of courses employing the MIZAR proof-checking system, which are currently part of the obligatory curriculum for computer science students at the University of Białystok (see also [2]).

Keywords: MIZAR in education, the art of proving, natural deduction, automated proof checking.

1 Introduction

The title of this paper is intended to cause the reader's recollection of the widely-discussed and seminal Leslie Lamport's paper "How to Write a Proof" [4]. In that paper Lamport proposed a method of writing proofs which "makes it much harder to prove things that are not true". Nowadays there are numerous systems that make it "almost completely impossible" to prove things that are not true, with the "almost" covering the always-non-zero probability of hardware and software faults. A recent comparison of leading proof systems compiled by Wiedijk [17] shows that today there is no monopoly for the best proof style. Writing proofs considered as formal depends very much on the applied system's foundations and philosophy as well as the intended goal of creating the proof.

MIZAR is a proof-checking system aimed at developing formal mathematics in a rigorous way under the control of a computer, but without unnecessary departure from the standard mathematical practice (see e.g. [8], [15]). To this end, the system has been equipped with an underlying proof language quite close to the so-called "mathematical vernacular" based on a declarative style of natural deduction. Its distinctive features, which can be inferred from the aforementioned Wiedijk's

[1] This work has been partially supported by the FP6 IST project no. 510996 *Types for Proofs and Programs* (TYPES).

[2] Email: adamn@mizar.org

comparison [17], make it one of the most "mathematical in spirit" of all the systems and include inter alia:

- readable proof input files,
- ZFC set theory and classical logic forming the base,
- the use of (dependent) types,
- low but quite efficient automation,
- large mathematical standard library.

Considering these features, it is not surprising that the bibliography of this project [10] shows a long history of using its various versions in mathematics instruction on different levels: from secondary school courses to writing PhD theses (e.g. in [5],[13] one may find historic expositions, while [7] and [2] present recent works).

Owing especially to the readability and writability of proofs, the Mizar system is now renowned for the biggest library of formalized mathematical data, Mizar Mathematical Library (MML). Recent statistics of the library are presented at the MMLQuery website [11]. The system has been involved in several big formalization projects, attracted almost two hundred authors of serious contributions who represent more than a dozen countries, despite being initially developed in a much disadvantageous environment behind the 'Iron Curtain' in Poland through 1980s and 1990s.

2 Whom to teach writing proofs?

For over three decades now it has been tried to get working mathematicians more involved in the use and development of proof-assistants. Nowadays mathematicians utilize lots of scientific software to name just computer algebra systems, geometric presentation tools, etc. but for some reasons a widespread adoption and dissemination of proof-assistants in mathematical practice is still far ahead on the horizon. This dissemination process seems to be mainly impeded by the fact that the state-of-the-art proof-assistants are still not considered useful enough in the research aimed at obtaining new results. As Lamport puts it in [4], "Mathematicians tend to be conservative, and many are unwilling to consider that there might be a better way of writing proofs."

Worse still, the mathematical community presents a very sceptical attitude towards various initiatives, like the utopian but very inspiring QED project directed at advancing research in this field (see the QED archives [12]). Despite the evidence collected recently that it is certainly possible to write fully formal mathematical expositions/formalizations for quite advanced mathematics, be it classical theories with well-established theorems [3], specialized monographs [1] or recent mathematical journal papers [6], the de Bruijn factor seems still too high to persuade working mathematicians to make the extra effort required by proving at least parts of their work rigorously under the control of a proof-assistant system (see [16] for the explanation of the de Bruijn factor).

Therefore, as mathematicians find the length of formal proofs and the unfamiliar

format rather intimidating, it has been observed that instead of trying to convince this community, a new approach worth trying is to appeal to the new generation. Arguably, today's students who do not yet carry any such bias of traditional mathematical practice may be even better users of proof-assistants if we manage to instill the idea into their minds early enough. When Lamport proposed his structured proof method presented in [4], he envisaged a growing interest in proving not only pure mathematics, but also various computer science applications. In particular he intended to use his system for proving the correctness of algorithms. Today this is done in practice, and it shows that apart from mathematicians an important and prospective group of users of various proof tools are also computer scientists. Although there are now well-established and intensively used formal specification methods, like the Z notation and related systems [9], particularly devised for describing and modeling computing systems, introducing a general-purpose proof-assistant like Mizar into the CS curriculum may have an even wider range of applications.

3 How difficult is it to learn writing proofs with Mizar?

As Wiedijk aptly pointed out [3], *proof assistants tend to resemble their implementation language*, and so Mizar is about as complex as the Pascal programming language. If this is really the case, then the answer to the above question should definitely be "Not at all".

Let us remember that Pascal is an imperative computer programming language, developed around 1970 by Niklaus Wirth as a language particularly suitable for structured programming. A derivative known as Object Pascal was later designed for object oriented programming – this is the language that the current Mizar system is implemented in.

Initially, Pascal was a language intended to teach students structured programming, and generations of students have "cut their teeth" on Pascal as an introductory language in undergraduate courses. Owing to that, Pascal is far too often considered by many as suitable ***only*** for educational purposes. Once common criticism in the spirit of Brian Kernighan's famous paper in defense of the C language "Why Pascal is Not My Favorite Programming Language" (although almost completely irrelevant to modern Pascal variants) is still a source of unjust prejudice in many computer science communities.

Nevertheless, variants of Pascal are still widely used today and all types of Pascal programs can be used for both education and "serious" software development. To name just a few notable examples, the original operating system of Apple Lisa computers was once coded in Pascal, and the primary high-level language used for development in the first couple of years of the Macintosh was also Pascal. Additionally, the popular typesetting system TEXwas written by Donald E. Knuth in WEB – the original literate programming system using Pascal.

Indeed, in many respects Mizar is similar to its implementation language. Even on the syntactic level there are certain similarities which support the claim that

[3] See Wiedijk's slides to the "Formalization of Mathematics" course at the TYPES Summer School in Göteborg – August 2005 (http://www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/wiedijk_sl.pdf)

learning Mizar should not in general be more complicated than learning programming in Pascal.

The lexis of Object Pascal consists of 29 special symbols, e.g. `#`, `[`, `@`, `<=`, etc. and 65 reserved words, e.g. `if`, `then`, `procedure`, etc. together with 39 so-called "directives" (words with a reserved meaning only in certain contexts, e.g. `abstract`). In Mizar we have 27 special symbols and 110 reserved words (6 of them are not actually implemented yet), so the numbers are almost identical! What is more, there are 10 symbols shared by both languages and also 15 identical reserved words. It is quite surprising for two so much different projects: a proof language and a programming language. The most significant difference on the syntactic level, however, is that current Mizar, unlike Pascal, is case-sensitive.

But even more importantly, Mizar has semantic features which make it very suitable for teaching purposes, just like Pascal. An important similarity is surely due to the high level of both languages, as they tend to use as many as possible words from the natural English. On the one hand this is why there is this number of shared reserved words. On the other hand, this allows the creation of constructs and expressions close to natural language, relatively easy for new users to assimilate. The same concerns several features we can describe as "syntactic sugar" (their sole purpose is to make the source text more readable for humans). However, these features are devised with the intention to preserve the source text's conciseness rather than make it much more verbose. This way the problems we may find learning the syntax of other languages which use elements of natural language excessively (like COBOL or SQL) are avoided.

Moreover, both the Pascal and Mizar languages are notably highly structured and also strongly typed. The latter is especially responsible for supporting the production of rigorous and semantically unambiguous expositions.

Pascal being an imperative programming language allows to manipulate the state (memory) of the machine by means of variables and other more complex expressions. Mizar's imperative part consists of steps of natural deduction, e.g. `assume`, `take`, `consider`, etc. They allow to manipulate the current proof-goal, the so-called *thesis*. These imperative constructs, however, are used in a declarative way, i.e. stating what is to be proved rather than how to prove it.

Another feature of Mizar inherited directly from its implementation language is the way one works with the system. This is often called "lazy interaction" – not a full interaction, but rather running the system on a source text in order to reveal errors and then correct them using the step-wise refinement method. Mizar reports its errors in much the same way a Pascal compiler reports warnings and errors. As Mizar does not stop processing on encountering the first error, its error recovery mechanism allows to check correctness of incomplete texts similarly to compiling modularized programs. This is particularly useful when users want to develop various parts of a proof e.g. postponing a proof or its more complicated parts. Especially in the educational context this can be useful to first sketch a proof's structure, revise it when needed, and also develop parts independently by several people.

It should be noted that although Mizar possesses certain didactic qualities, its developers hope this will never make it subjected to so much undeserved prejudice

as Pascal.

4 Basic framework of Mizar-aided courses at the University of Białystok

The Mizar system has been developed at the University of Białystok (formerly the Białystok branch of Warsaw University) since 1970s. The research involved the members of the mathematics department, so naturally all teaching experiments concerned local mathematics students. The teaching has never had a permanent position in the mathematics curriculum. Mizar-aided classes were organized mainly as voluntary monographic courses, e.g. "Lattice theory", "Category Theory", "Topology", etc. The situation changed when there emerged a new university unit, the department of Computer Science and then the core Mizar developers formally became its stuff and were assigned teaching duties concerning CS students. This gave the opportunity to instill more Mizar-based instruction into the curriculum.

At the same time, basic issues related to the use of computer tools (that some mathematics students used to have) with CS students became far less problematic. The distribution of the system and teaching materials, as well as assessment can now be done easily via the Internet. And although the teaching is currently being done with the students gathered in class at a certain time, it would be possible to use the Internet medium to even more extent reducing the involvement on the side of the instructors.

Additionally, as the Mizar-oriented teaching is done by instructors who are also involved in teaching programming, some elements of the methodology of teaching programming languages can also be used. Currently the CS curriculum contains three one-semester Mizar-based courses for undergraduate (bachelor level) and graduate (master level) students:

- Undergraduate level:
  (i) "Introduction to formal methods"
  (ii) "Constructive methods in CS"
  (iii) "Abstract methods in CS"
- Graduate level:
  (i) "Software verification"
  (ii) "Proof verification"

The paper [2] presents in great detail one particular course - "Introduction to logic and set theory" (it corresponds to "Introduction to formal methods" in the current curriculum). However, the methodology and procedures are common for all the above courses.

4.1 Undergraduate level

The "Introduction to formal methods" course is devised to introduce the fundamental formal mathematical apparatus and form a basis for other strictly mathematical subjects like "Discrete mathematics" or "Mathematical analysis". The course is taken during the first semester of study, so there is virtually no knowledge that

can be assumed, since the mathematics education in secondary schools often varies. Therefore the accompanying lecture must run in a way in parallel: introducing the standard mathematical symbolism and its Mizar counterparts together with hints to "the art of proving". In brief, the syllabus comprises:

- logical formulae and basic structures of conditional proofs,
- Boolean properties of sets (also Venn diagrams),
- families of sets and their properties,
- binary relations (composition, the converse relation, selected properties - e.g. reflexivity, transitivity, etc.),
- functions (domain and codomain, image, etc.).
- equivalence relations, partitions and ordering relations.

The second undergraduate course, "Constructive methods in CS", heavily depends on the knowledge covered by the previous one. The students are supposed to know how to use all basic proof techniques. The Mizar symbolism is dominating as there is not much purely mathematical contents in the syllabus:

- Peano arithmetic,
- various forms of the induction principle,
- higher-order reasoning with Mizar schemes (statements with second order free variables),
- the axiomatics of set theory.

The aim of the last undergraduate course, "Abstract methods in CS", is to get students acquainted with abstract methods of describing objects used in high-level mathematics, but even more importantly, also in computer science for formal description and specification. The course comprises Mizar formalizations in the following fields:

- lattice theory,
- universal and many-sorted algebra,
- elements of category theory.

Below there are typical examples of Mizar proofs created by students for their first two undergraduate courses. As a rule, students were expected to complete up to five such short tasks during each class (90 minutes).

Example 1: For any relation R, if R is transitive, then $R \circ R \subseteq R$.

```
for R being Relation holds R is transitive implies R*R c= R
proof
  let R be Relation;
  assume a: R is transitive;
  let a,b be set;
  assume [a,b] in R*R;
  then consider c being set such that
  c: [a,c] in R & [c,b] in R by RELATION:def 7;
```

```
  thus [a,b] in R by c,a,RELATION:def 12;
end;
```

Example 2: There exist relations R, S and T such that $R \circ (S \setminus T) \not\subseteq (R \circ S) \setminus (R \circ T)$.

```
ex R,S,T being Relation st not R*(S \ T) c= (R*S) \ (R*T)
proof
  reconsider R={[1,2],[1,3]} as Relation by RELATION:2;
  reconsider S={[2,1]}, T={[3,1]} as Relation by RELATION:1;
  take R,S,T;
  b: [1,2] in R by ENUMSET:def 4;
  d: [2,1] in S by ENUMSET:def 3;
  [2,1] <> [3,1] by ENUMSET:2;
  then not [2,1] in T by ENUMSET:def 3;
  then [2,1] in S \ T by d,RELATION:def 6;
  then a: [1,1] in R*(S \ T) by b,RELATION:def 7;
  e: [1,3] in R by ENUMSET:def 4;
  [3,1] in T by ENUMSET:def 3;
  then [1,1] in R*T by e,RELATION:def 7;
  then not [1,1] in (R*S) \ (R*T) by RELATION:def 6;
  hence not R*(S \ T) c= (R*S) \ (R*T) by RELATION:def 9,a;
end;
```

Example 3: By induction, for natural numbers i, j, k if $i + k = j + k$, then $i = j$.

```
reserve i,j,k,l for natural number;
i+k = j+k implies i=j;
proof
  defpred P[natural number] means i+$1 = j+$1 implies i=j;
  A1: P[0]
  proof
   assume B0: i+0 = j+0;
   B1: i+0 = i by INDUCT:3;
   B2: j+0 = j by INDUCT:3;
   hence thesis by B0,B1,B2;
  end;
  A2: for k st P[k] holds P[succ k]
  proof
    let l such that C1: P[l];
    assume C2: i+succ l=j+succ l;
    then C3: succ(i+l) = j+succ l by C2,INDUCT:4
    .= succ(j+l) by INDUCT:4;
    hence thesis by C1,INDUCT:2;
  end;
  for k holds P[k] from INDUCT:sch 1(A1,A2);
  hence thesis;
end;
```

4.2 Graduate level

On the graduate level the main focus is on one of the most important applications of automated theorem proving – software verification. At the same time, students are supposed to become competent Mizar users and be able to individually produce formalizations in various domains.

The "Software verification" course covers the following theoretical and practical aspects:

- various semantics of software description (operational, denotational, axiomatic),
- program correctness criteria,
- mathematical models of computers,
- practical verification of exemplary algorithms (sequential instructions, loops, jumps, recurrence),
- using the "describer" technique to generate proof conditions.

The objective of the second Mizar-based course on the graduate level is to enable students to choose a domain in which they could carry out formalization. The formalization may form a basis of one's MSc thesis. On this level the students are supposed to be proficient Mizar users and be trained enough to develop themselves new contributions to the Mizar Mathematical Library. The syllabus topics of the classes comprise:

- the formal theory of mathematical proofs in connection with computer proof-checking systems,
- structuring and managing databases of formalized proofs,
- practical usage of discussed mechanisms based on selected fields of mathematics.

5 Getting to know how to write Mizar proofs

The methodology adopted for the realization of the above mentioned courses must reflect the gradual way of getting to know the system from its very basics till becoming an independent competent user. Therefore the methodology changes slightly with the advance of formalized material. There are, however, certain assumptions we consider crucial from the didactic point of view.

First of all, the amount of Mizar notions introduced at each stage should be reduced to minimum, i.e. the smallest possibly subset of Mizar which allows completing a certain task. For instance, the first session of the introductory course in logic is devoted solely to propositional and predicate calculus. We have to bear in mind that our students at the same time must also learn the standard mathematics notation of these notions and the way of installing, running, and interacting with the Mizar system. Therefore it is extremely convenient to prepare ready-made dedicated local environments which spare the students the intricacies of coping with much technical detail completely unnecessary at that stage. The first environment should therefore include only predicates with various arity and no particular deno-

tation, as it is all one needs to practice the use of logical connectives, quantifiers and the rules of first-order logic. Similarly, before writing complete proofs of e.g. Boolean properties of sets, students should start with the so-called formal proof sketches (cf. [14]), to have the syntax and proof structure correct first.

In general, we believe the whole instruction on the undergraduate level should be based on "incremented" local environments instead of using the full system with the standard mathematical library (which would require certain extra effort of searching the whole library). Teaching the interaction with the full system should be an integral part of the graduate-level courses.

The high-level features of the Mizar language and system should not be introduced too early. It seems much more proper to first show students how to prove things in an elementary way, and only when they master it, allow for a reflection when they are presented a more "elegant" high-level way to do the same. On the one hand, this can be done on the level of the language's "syntactic sugar" expressions like `then`, `hence`, `thesis` which should be introduced when students already know how to write proofs using "strict" Mizar. On the other hand, there are many system's capabilities normally available to users which allow to write shorter proofs, like the automatic definition expansion, the use of implicit general quantifiers, the use of semantic correlates rules for thesis elimination, the forward/backward proof distinction and so on – their introduction should be postponed too.

The acquisition of the formal Mizar language should also be split into two layers: passive and active. Only passive knowledge (the ability to read with general understanding) of certain constructs is enough at first. Taking as an example the whole part of Mizar's grammar which concerns definitions, we note that although the students should be able to read and refer to the definitions presented in Mizar abstracts (interface information only with all justifications removed), writing the student's own definitions (active knowledge required) is only the part of the second graduate-level course. In practice, this process resembles the way in which students master the use of programming languages. The major part of writing a Mizar article consists of using pre-existing theorems and definitions, just like the major part of writing a computer program consists of referencing available library procedures/functions.

6 Conclusions

Implementing the usage of the Mizar proof-assistant in the obligatory curriculum for CS students at the University of Białystok proved feasible. There were no major administrative or methodological difficulties encountered during the process. Last semester, on completion of their second Mizar course students were asked to fill in a simple anonymous questionnaire asking for their attitude towards Mizar-aided courses and whether they observed any beneficial influence of using Mizar on their study of other subjects. From the answers we collected, we learned that also from the students' perspective the courses were quite successful. The results confirmed that the simple methodology adopted with a combination of interdependent undergraduate- and graduate-level courses make the teaching process not harder than that of a popular programming language. As the questionnaire was fully

anonymous, there were several students who confessed to a complete lack of interest in Mizar as they could not see any "*practical use of it*". However, the vast majority of our respondents were more enthusiastic about it. To quote just a few typical comments, the courses "*were a good opportunity to learn and use logical thinking*", "*involved more thinking than other courses*", "*were quite similar to programming courses*" and "*helped with understanding proofs used in other mathematical subjects like algebra and analysis*".

In the perspective of few years there will be graduates who completed all the Mizar-aided courses. At that point we will be able to fully assess the results of this project by the number of MSc theses based on new contributions to the Mizar Mathematical Library. The choice to focus on the instruction of new CS adepts rather than mathematicians seems reasonable and prospective in order to facilitate the important use of formal methods in software verification and specification.

References

[1] Bancerek, G., *Development of the Theory of Continuous Lattices in* Mizar, In M. Kerber and M. Kohlhase (Eds.), "Symbolic Computation and Automated Reasoning", proceedings of the CALCULEMUS 2000 Symposium, 65–80.

[2] Borak, E. and A. Zalewska, *Mizar Course in Logic and Set Theory*, In M. Kauers et. al. (Eds.) "Towards Mechanized Mathematical Assistants", proceedings of 14th Symposium Calculemus 2007 and 6th International Conference MKM 2007, LNCS 4573 (2007), 191–204.

[3] Geuvers, H. et al., *The Algebraic Hierarchy of the FTA Project*, In S. Linton and R. Sebastiani (Eds.) "Proceedings of the CALCULEMUS 2001 Symposium", Siena, 2001, 13–27.

[4] Lamport, L., *How to Write a Proof*, American Mathematical Monthly **102(7)** (1993), 600–608, URL: http://research.microsoft.com/users/lamport/pubs/lamport-how-to-write.pdf.

[5] Matuszewski, R. and P. Rudnicki, *MIZAR: the First 30 Years*, Mechanized Mathematics and Its Applications, **4(1)** (2005), 3–24.

[6] Naumowicz, A., *An Example of Formalizing Recent Mathematical Results in MIZAR*, In C. Benzmüller (Ed.) "Towards Computer Aided Mathematics", Journal of Applied Logic **4(4)** (2006), 396–413.

[7] Retel, K. and A. Zalewska, *MIZAR as a Tool for Teaching Mathematics*, Mechanized Mathematics and Its Applications **4(1)** (2005), 35–42.

[8] Rudnicki, P. and A. Trybulec, *Mathematical Knowledge Management in MIZAR*, In B. Buchberger and O. Caprotti (Eds.) "Proceedings of the First International Workshop on Mathematical Knowledge Management: MKM 2001", URL: http://www.emis.de/proceedings/MKM2001/rudnicki.ps.

[9] Spivey, J.M., *The Z Notation: a reference manual*, URL: http://spivey.oriel.ox.ac.uk/mike/zrm/zrm.pdf.

[10] The Bibliography of the Mizar Project, URL: http://mizar.org/project/bibliography.html.

[11] The MMLQuery Project, URL: http://mmlquery.mizar.org.

[12] The QED Project Archives, URL: http://www-unix.mcs.anl.gov/qed/.

[13] Trybulec, A. and P. Rudnicki, *Using Mizar in Computer Aided Instruction of Mathematics*, Norwegian-French Conference of CAI in Mathematics, Oslo, 1993, URL: http://mizar.org/project/oslo.ps.

[14] Wiedijk, F., *Formal Proof Sketches*, In W. Fokkink and J. van de Pol (Eds.) "7th Dutch Proof Tools Day, Program + Proceedings", CWI, Amsterdam, 2003, URL: http://www.cs.ru.nl/~freek/notes/sketches.pdf.

[15] Wiedijk, F., *Mizar: An Impression*, URL: http://www.cs.ru.nl/~freek/mizar/mizarintro.pdf.

[16] Wiedijk, F., *The de Bruijn Factor*, URL: http://www.cs.ru.nl/~freek/factor/factor.ps.gz.

[17] Wiedijk, F. (ed.), "The Seventeen Provers of the World" (foreword by Dana S. Scott), LNAI 3600, 2006.

Teaching Distributed Algorithms Using SPIN

Claude Jard[1]

Université Européenne de Bretagne
IRISA/ENS de Cachan
Campus de Ker-Lann, 35170 Bruz, France

Abstract

This paper presents a teaching experiment to introduce formal methods and distributed algorithms at the undergraduate level in the department of computer science and telecoms of the ENS de Cachan in France. The course intends to teach some basic notions on formal modeling and analysis of distributed software. We used the free-software SPIN. This kind of experiment was found to be useful for our students since they gain an understanding of the importance and necessity of formal methods for designing even simple protocols.

Keywords: Distributed computing, Formal methods, Model-checking, SPIN, Promela.

1 Context

1.1 The ENS de Cachan in France

ENS de Cachan is a prestigious public institution of research and higher education, founded in 1912. It is one of the French "Grandes Ecoles" which are considered to be the pinnacle of French higher education. Students who enter the ENS have a double status: they are both students, registered in State Universities, and "normalien", i.e. members of the ENS. ENS de Cachan has two campuses, one located in Cachan near Paris, and one locate in Rennes in Brittany (West of France). Admission is decided by a highly selective national competition usually taken after a two-year post-baccalauréat preparatory program. The first year of education corresponds to the international BA/BS bachelor degree. Our students enter then in a Master program, before becoming a PhD candidate for most of them. They stay four years in the school.

1.2 Computer Science and Telecommunication department

This recently created department (2003) is devoted to Communication and Information Technology and Sciences. Its goal is to develop training and research in

[1] Email: Claude.Jard@bretagne.ens-cachan.fr

computer science, on the border of telecommunication. It is located in the Brittany extension of the ENS de Cachan on the Ker-Lann campus, in the south of the city of Rennes. The curricula are organized within the scope of the Master's degree entitled "Computer Science and Telecommunication" co-delivered by ENS de Cachan and the IFSIC Institute at the University of Rennes 1.

In addition to the usual set of degrees offered at IFSIC (Higher Training Institute for Computer Science and Communication), Department students are offered complementary courses in the fields of mathematics applied to statistics and signal processing, electronics and optics. But above all, these future researchers are offered personalized coaching, to help them maturing their taste for research and their professional objectives. For instance, this includes personal tutoring by professional academic staff, seminars, team work, internships in France and abroad, and, most important of all, a daily contact with researchers. We train 15 students each year, among which a dozen "normaliens" from ENS de Cachan and a few university students selected on the basis of their qualifications and by interview.

Telecommunications have been a leading research and development activity in Brittany for a long time. The Department is closely associated with the IRISA Research Institute (www.irisa.fr), a Joint Research Unit between CNRS, INRIA, University of Rennes 1 and INSA of Rennes. IRISA includes more than 180 permanent academic research staff, and as many PhD students. Together with the technical and administrative support staff, this sums up to more than 500 people. This makes IRISA one of the largest Computer Science laboratory in France.

2 The course on distributed algorithms. Its objectives.

The course takes place in the first semester of the first year of the school. It is a specific course offered to the department students. It is clearly their first approach of the area of distributed computing. This is quite unusual, but we found important to introduce the parallel and communication aspects of computer systems in the same time as the traditional concepts of sequential systems. After all, distributed computing is at the heart of modern systems and we are in a Computer Science and Telecoms department... Distributed computing means designing and implementing programs that run on two or more interconnected computer systems. For this level of course (undergraduate), we are assuming students will study distributed programs for a fully functioning Internet [5]. The intent is to use a network; not to study networks (this will be taught in the second year). Modern distributed programming includes multimedia systems, client-server systems, web programming and collaborative systems. What common fundamental principles and difficulties do they possess that will serve our students' needs for the next ten years?

Our students have been mainly selected on a basis on an excellent background and performance in mathematics. When there were recruited, they had the choice to follow a degree course in maths or in computer science. Maths in France are clearly the most prestigious way and those who decided to join the computer science track (by interest or realism, considering the possible opportunities of carrier) have to manage this frustration. This context benefits to the formal methods. A challenge for the teachers is to prove that computer science can be rigorous and

mathematically based. The situation is different in standard universities in France, where students often choose Computer Science against Mathematics to learn more practical things.

The last point is that we found distributed computing is an excellent playground to start a research oriented activity based on small collective projects. Distributed algorithms remain small, but presents surprisingly complex and counter-intuitive behaviors. This makes the students strongly aware of problems of software engineering.

To sum up:

- our context is favorable of the use and promotion of formal methods,
- we chose to start directly by teaching some aspects of distributed computing at the early stage of the degree course in computer science,
- it is a good place for research-oriented activities, and
- the intrinsic complexity of even small distributed algorithms is a good carrier of the promotion of formal methods in the context of a mathematically based software engineering.

From the algorithmic point of view, the objective of the course is to put emphasis on the difficulty of designing correct distributed algorithms. The study of several classical paradigms (reliable transfer, mutual exclusion, termination) serves to design a general methodology of description. The course is divided into six sessions of two hours long. It also comprises several projects, conducted by groups of 2-3 students.

3 About the use of SPIN

3.1 SPIN and Promela

SPIN [4] is a popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM. Since 1995, (approximately) annual SPIN workshops have been held for SPIN users, researchers, and those generally interested in model checking. SPIN is an automata-based model checker. Systems to be verified are described in Promela (Process Meta Language), which supports modeling of asynchronous distributed algorithms as non-deterministic automata. Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas, which are negated and then converted into Buchi automata as part of the model-checking algorithm. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user. Unlike many model-checkers, SPIN does not actually perform model-checking itself, but instead generates C sources for a problem-specific model checker. This rather antique technique saves memory and improves performance, while also al-

lowing the direct insertion of chunks of C code into the model. SPIN also offers a large number of options to further speed up the model-checking process and save memory.

Promela programs consist of processes, message channels, and variables. Processes are global objects that represent the concurrent entities of the distributed system. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

3.2 Selected usage

Assuming our students are familiar with a standard sequential imperative programming language like C, the best originality of Promela is its non-deterministic control flow constructs *if* (selection command) and *do* (repetition command), based on the concept of guarded commands introduced by Dijkstra [2]. The selection is a list of guarded commands, of which one is chosen to execute. If more than one guards are true, one statement is non-deterministically chosen to be executed. If none of the guards are true, the result is undefined. Because at least one of the guards must be true, the empty statement "skip" is often needed. The repetition executes the guarded commands repeatedly until none of the guards are true.

This type of abstract construct is a bit disconcerting for the students. On one hand, specifying behaviors in Promela is a good opportunity to explore the existing continuum between the different abstraction levels, which are concerned with modeling activities, from the code level to the possible logical level used to express properties. A crucial question in formal methods is actually to choose a relevant level of abstraction.

On the other hand, I must admit that a graphical representation like the communicating finite state machines of Figure 1 is better suited for a pedagogical presentation on the blackboard. In practice, I used automata, even extended automata to take into account variables and symbolic conditions to model more complex distributed algorithms. The translation to Promela was based on a systematic way to code communicating automata into Promela programs (see the code of Figure 2).

The other graphical aspect is to present the execution traces and the different examples and counter-examples. It is quite easy to use informal message diagrams on the blackboard in exact correspondence with the message sequence charts displayed by SPIN (see Figure 3).

Once the Promela program is written, the interactive simulator is used to validate the code and we discuss how to instrument the code to perform verification. The course is not a course on verification. It just explains the general idea of exhaustive simulation provided by the enumeration of the possible reachable states. It is clearly out of scope at this level to present the temporal logic, its translation into Buchi automata and the on-the-fly construction of the graph product, on which the satisfaction of properties is computed. We thus preferred to use the simple method of assertions, which can be easily inserted into the Promela code, at the price of inventing special global variables to compute the invariants. The difficulty is to explain that these variables are not part of the model, but are just there for

verification purpose.

SPIN possesses the interesting feature to detect violations of assertions and when this occurs, to graphically present the shortest scenario leading to the violation in term of message sequence charts. This was quite impressive for the students, since we had in a few seconds for our small examples the discovering of counter-examples for various conjectures difficult to reject by hand. It was also a good example to see how computers may help the design of other computers despite the theoretical barrier of computability, seen in another course in parallel.

To sum up:

- the presentation of the algorithms is based on a graphical representation on the blackboard as well as the execution traces, shown as message diagrams.
- the formalization is completed using a translation towards Promela and the insertion of global assertions,
- the ability of SPIN to rapidly present non-trivial counter-examples is impressive and proves the interest of automated tools in that case.

4 Introduction using the simplest protocol

We start the course by introducing the simplest protocol that we can imagine between two processes A and B communicating asynchronously through reliable FIFO queues. I present the problem of modeling the interaction between a user A and a timer B. The user has only two states: it can be awoke or sleeping. When awoke, it can ask to set the timer (message a) and decides to sleep. After a certain period of time (not modeled here and abstracted as a spontaneous transition), it can either wake up and ask for the stop of the timer (message b), or it is awoke by a timeout ringing (message c), sent by the timer process. Symmetrically, the timer process has two states: timer set or not. The timer is set upon receiving the set message. It is stopped when it decides to send the timeout message or when it receives the stop message. Messages from A to B are stored in a FIFO communication channel linking A and B, while the messages in the other direction are stored in a second FIFO communication channel linking B and A. In case of bounded channels, we consider that sending a message on a full channel will block the communication until a possible reception. The protocol is explained using the formalism of communicating automata illustrated in the left part of Figure 1.

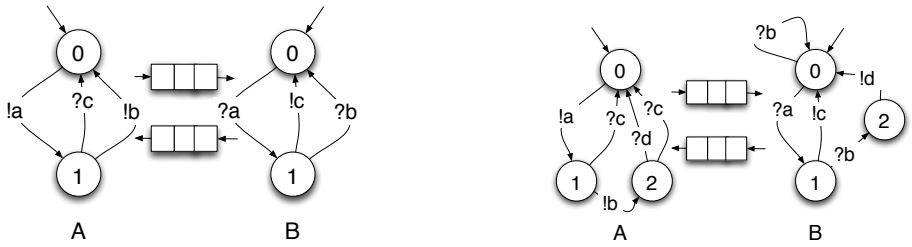


Fig. 1. Our first protocol. Left part: a buggy version. Right part: a corrected one.

The translation into Promela is shown on Figure 2. Notice the introduction of the global variable *alternate* to check that it is not possible to have two consecutive

message a in the communication channel from process A to process B. My favorite joke is to prove informally that is not possible: the only mean to put a second a in the channel is to execute again the transition from state 0 to 1 of the process A. And this implies for it to go back to state 0, which demands for the process B to consume the first message a. This is false, but the shortest counter-example needs to consider two cycles in the protocol with an asynchronous collision between messages b and c in the channels. This happens at depth 8 in the state graph, which is not easy to guess by hand. Figure 3 shows a screen shot of this invalidation. We concluded that distributed algorithms are complex objects, since even for the simplest one, it is difficult to obtain an evidence of correctness. A corrected version is proposed as shown in the right of Figure 1, where a supplementary message d is introduced to acknowledge the stop of the timer.

```
/*-------------------------------------------------------------------------*/
/* My favorite first example of complex behaviors in distributed systems.
   Buggy version including assertion */
#define N 8 /* Size of channel AB */
mtype = {a,b,c}; chan AB = [N] of {mtype}; chan BA = [1] of {mtype};
bit alternate = 1;  /* For verification purpose */

active proctype A() {do
 ::  atomic{ assert(alternate); alternate = 0; AB!a };
 if :: BA?c; :: atomic{ AB!b; alternate = 1 } fi od}

active proctype B() {do
 :: atomic{ AB?a; alternate = 1 };
 if :: AB?b; :: BA!c fi od}
 /*-------------------------------------------------------------------------*/
```

Fig. 2. The wrong Connect-Disconnect protocol expressed in Promela

Exhaustive simulation is a good method to find bugs. Of course, it suffers from the state explosion phenomenon. It is not the sole viable approach for analyzing systems, especially when tempting to obtain a formal proof of correctness. It is easy to discuss this aspect with the students on the erroneous version of the protocol, which has an infinite state space. We were lucky that the situation with two consecutive a in the AB communication channel was occurred when bounding the channel by three messages (bounded channels are required by SPIN, which is based on an enumerated model-checking). In general, one cannot assert that bounding channels by a given value is enough to check a given property. This is undecidable, even for our simple model of two communicating finite state machines. It is interesting to note that we can predict in our small example the number of states in function of the size n of the AB channel, using the formula $\lfloor \frac{n^2}{4} \rfloor + 3n + 3$. This formula has been used to test several model-checker in the world.

At this point of the course, it is possible to try to build a formal proof of correctness of the "corrected" protocol. This rises the question of how to express the desired properties. One possibility is to use the temporal logic framework provided by SPIN. As I explained above, it was not realistic to implement the idea until now, since their first course on logic was put after my presentation. But this could be changed next year, and will offer a good opportunity to show an interesting application of logic. In Linear Temporal Logic, the property of the alternation of message a can be written as $\Box[(!a \wedge \Diamond\bigcirc !a) \Rightarrow \bigcirc\neg(\neg(!b \vee !c)\mathcal{U}!a)]$.

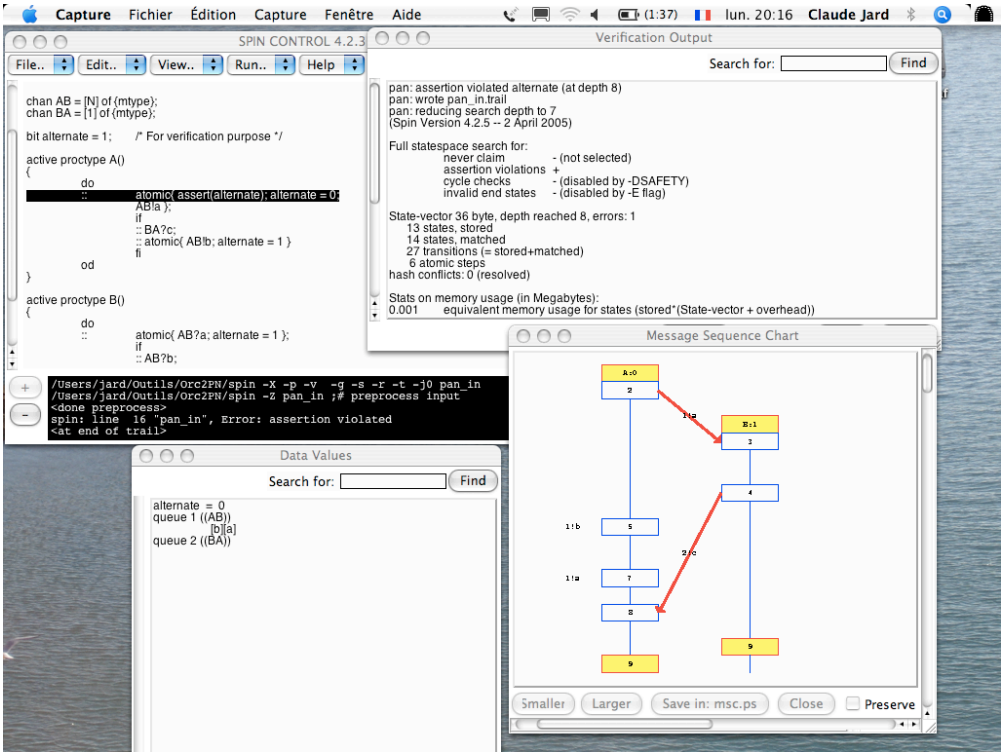


Fig. 3. Screen shot of the Spin session discovering the shortest path to the error.

Another kind of formal proof has been discussed during the following SWP project, in which some liveness property can be proved using transition invariants in a Petri net model.

5 The SWP project

The next step of the course is a homework project, based on the study of a simplified transport protocol, supposed to represent the essence of the Internet TCP. This "Stop and Wait Protocol" was described in [1], an invited paper in a conference on formal methods in 2003. Stop-and-Wait protocols have been shown to operate correctly over media that may lose packets, however, there has been little discussion regarding the operation of these protocols over media that can re-order packets. The model is analysed using a combination of hand proofs and automatic techniques based on the analysis of Coloured Petri net models. After a collective discussion to explain the content of the paper, present the Petri net model and the associated proofs, the goal of the project is to obtain similar results using SPIN. The results are the following:

- A proof of the counter intuitive property for a Stop-and- Wait protocol that the number of packets that are stored in the network can grow without bound. This is

true for any positive values of the maximum sequence number and the maximum number of retransmissions.

- It is shown that the protocol may fail: loss of packets is possible and duplicates can be accepted as new packets by the receiver, even though the sender and receiver perceive that the protocol is operating correctly.

```
/*-------------------------------------------------------------*/
/* "Stop and Wait protocol" from J. Billington, Forte'2003.
 *  A simplified transport Protocol after Internet/TCP */
#define true 1
#define false 0
#define empty 0
#define MaxData 2
#define MaxRetrans 1
#define MaxChannel 3           /* 2*MaxRetrans+1 */
#define MaxSeqNo 1
chan mess_channel = [MaxChannel] of {int,int}; /* Sender->Receiver (no_seq, data) */
chan ack_channel  = [MaxChannel] of {int};       /* Receiver->Sender (no_seq) */

active proctype Sender() /* Sender automaton */ {
 bit sender_ready = true; /* Two states automaton */
 int seq_no = 0;             /* Sequence numbering of data */
 int retrans_counter = 0; /* Retransmission counter */
 int data = 0;                 /* VERIF: the transmitted data */
 int m;                          /* Last number of the received ack */
 do :: atomic{sender_ready && data<MaxData -> data++; /* send_mess */
                printf ("MSC: Send request: %d\n", data);
                mess_channel!seq_no,data; sender_ready = false; }
     :: atomic{(retrans_counter<MaxRetrans) && !sender_ready -> /* timeout_retrans */
               mess_channel!seq_no,data; retrans_counter ++; }
     :: atomic{ack_channel?m -> /* receive_ack */
        if :: (m==(seq_no+1)%(MaxSeqNo+1)) && !sender_ready -> retrans_counter = 0;
           seq_no = m; sender_ready = true;
           :: (m!=(seq_no+1)%(MaxSeqNo+1)) -> skip; /* receive_dup_ack */ fi; } od}

active proctype Receiver() /* Receiver automaton (one single state) */ {
 int rec_no = 0; /* Sequence number of received messages */
 int rec_indication = 1; /* VERIF : Test of non detection of duplicated acks */
 int data; /* Last received data */
 int sn;    /* Its sequence number */
 do :: mess_channel?sn,data  -> /* receive_mess */
         if :: (sn==rec_no) -> rec_no = (rec_no+1)%(MaxSeqNo+1);
               printf("MSC: receive indication : %d\n",data);
               assert (data==rec_indication);/* VERIF */
               rec_indication++; /* VERIF */
            :: (sn!=rec_no) -> skip fi;
        ack_channel!rec_no; od}

active proctype devil () /* Simulation of the network level
                            (lossy channels with possible reordering) */ {
 int s,d;
 do :: atomic{mess_channel?s,d -> skip; } /* Loose message */
      :: atomic{(len(mess_channel) > 1) -> mess_channel?s,d; mess_channel!s,d; }
                                          /* Reorder message */
      :: atomic{ack_channel?s -> skip; } /* Loose ack */
      :: atomic{(len(ack_channel) > 1) -> ack_channel?s; ack_channel!s; }
                                          /* Reorder ack */ od}
/*-------------------------------------------------------------------------*/
```

Fig. 4. The Stop and Wait protocol expressed in Promela

The motivating fact is that it seems these failures (based on the problem of the incrementation of the sequence numbers using a finite modulo) could be reproduced in the real TCP, by a particular positioning of its large set of parameters.

Figure 4 shows the complete Promela program we obtain. We can notice the introduction of the "Devil" process to simulate losses and reordering of messages in the communication channels. As previously, some global variables have been introduced to be able to write assertions. To this goal, we maintain the exact numbering of data, to be able to check the correct sequence of receptions at the user level. In the screen shot of Figure 5 we detect a shortest counter-example in the case of reordering and for a small modulo of 2. Without reordering, as expected,

SPIN does not detect faults during verification, which can be performed only for small values of the different parameters because of the classical state explosion problem.

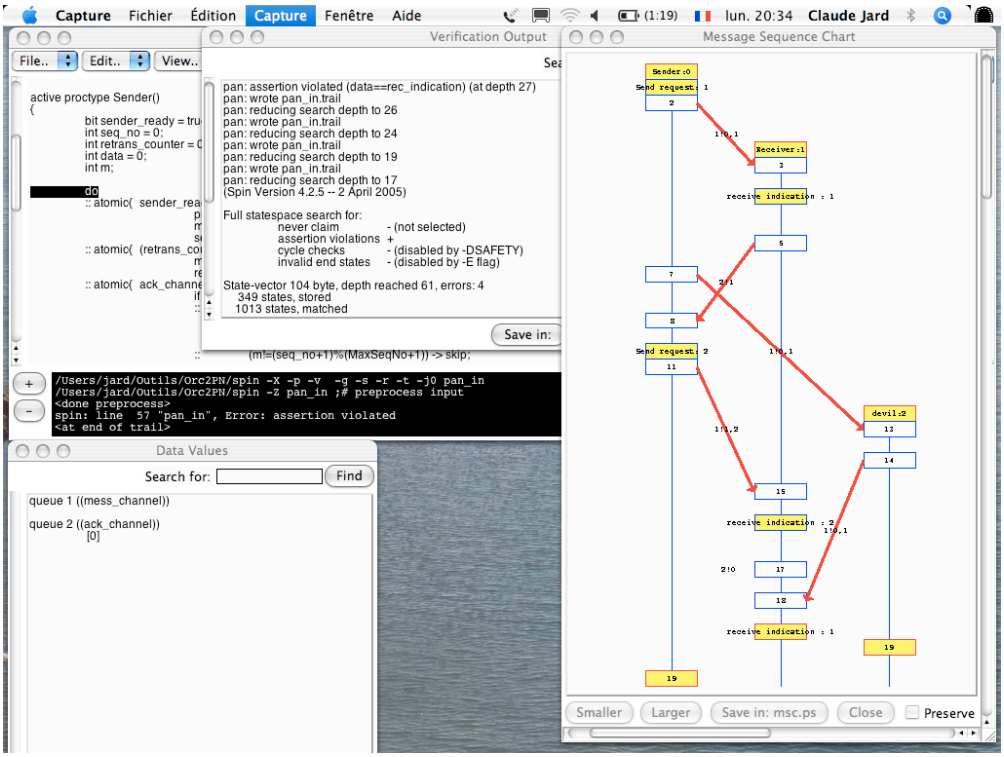


Fig. 5. Screen shot of the Spin session discovering the shortest path to the error.

The different models and results obtained by the students were presented during a short oral presentation (project defense), followed by a general discussion about what they have learnt.

The rest of the course, using a similar approach, is dedicated to the study of other basic distributed algorithms. For the problem of termination detection, I chose the Dijkstra's algorithm using a ring of processes. In this algorithm, a colored token circulates on the ring. If the token comes back to the initiator without having changed its color, the termination can be claimed. One can show it is correct in an asynchronous environment (recall that the original algorithm was designed for synchronous communication) under the assumption that no application message can be delayed more than the total time needed for the token to complete its tour on the ring. SPIN was able to produce a counter-example in the latter case.

The last part of the course presents the consensus problem and mainly the result of Fisher, Lynch and Paterson on the impossibility of achieving a consensus in presence of process failures [3].

6 Conclusion

6.1 What do the Students Learn ?

At the beginning of the course, our students are almost unaware of distributed computing and formal description techniques. Experiments using cases studies motivate them because they see it "with their own eyes". The main lesson is that it is difficult to understand the behavior of a protocol without the help of formal tools. Even for a simple protocol, they understand that automatic tools are necessary in order to detect errors. We expect that they will remember that techniques exist and hopefully that they will convince more people about their usefulness. The evaluation of the understanding of students is done throughout the case studies and the presentation of their home-works.

6.2 Future evolutions

I want to improve the practical aspect of the work in order to enforce the proof of potential impact of this kind of formal approach. To that respect, the SWP example is promising. The idea is to try to use the possible failures found on the simple model to force real TCP to make a mistake. From the application point of view, a duplication that is not detected may have spectacular effects (imagine a banking application for example). A realistic experiment is to use the NS simulator in which most of the TCP parameters can be changed. A more ambitious project is to settle a real experiment on the Internet. But before that, we have to evaluate if the parameters of our network environment make the fault possible (if it is the case, the probability of occurrence will be very low in practice of course).

References

[1] Billington, J., Gallash, G. E., "How Stop and Wait Protocols Can Fail over the Internet", Formal Techniques for Networked and Distributed Systems - FORTE 2003, Lecture Notes in Computer Science, Volume 2767/2003, pp. 209-223.

[2] Dijkstra, E., "Guarded commands, non-determinacy and formal. derivation of programs", Commun. ACM 18 (1975). http://www.ccs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF.

[3] Fisher, M. J., Lynch, N. A., Paterson, M. S., "Impossibility of Distributed Consensus with One Faulty Process", Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 374-382.

[4] Holzmann, G. J., "The SPIN Model Checker: Primer and Reference Manual", Addison-Wesley, 2004. http://spinroot.com/spin/whatispin.html.

[5] Hyde, D. C., "Importance of Teaching Cluster Computing", Proceedings of the 8th International Parallel Processing Symposium, IEEE Computer Society Press, 1994, pp. 956-961.

Teaching logic using a state-of-the-art proof assistant

Cezary Kaliszyk Freek Wiedijk

Radboud Universiteit Nijmegen, The Netherlands

Maxim Hendriks Femke van Raamsdonk

Vrije Universiteit Amsterdam, The Netherlands

Abstract

This article describes the system ProofWeb developed for teaching logic to undergraduate computer science students. The system is based on the higher order proof assistant Coq, and is made available to the students through an interactive web interface. Part of this system is a large database of logic problems. This database will also hold the solutions of the students. The students do not need to install anything to be able to use the system (not even a browser plug-in), and the teachers are able to centrally track progress of the students. The system makes the full power of Coq available to the students, but simultaneously presents the logic problems in a way that is customary in undergraduate logic courses. Both styles of presenting natural deduction proofs (Gentzen-style 'tree view' and Fitch-style 'box view') are supported. Part of the system is a parser that indicates whether the students used the automation of Coq to solve their problems or that they solved it themselves using only the inference rules of the logic. For these inference rules dedicated tactics for Coq have been developed. The system has already been used in type theory courses and logic undergraduate courses. The ProofWeb system can be tried at http://proofweb.cs.ru.nl.

Keywords: Logic Education, Proof Assistants, Coq, Web Interface, AJAX, Natural Deduction, Gentzen, Fitch

1 Introduction

At every university, part of the undergraduate computer science curriculum is an introductory course that teaches the rules of propositional and predicate logic. At the Radboud Universiteit (RU) in Nijmegen this course is taught in the first year and is called 'Beweren en Bewijzen' (Dutch for 'Stating and Proving'). At the Vrije Universiteit (VU) in Amsterdam this course is taught in the second year and is called 'Inleiding Logica' ('Introduction to Logic'). Almost all computer science curricula have similar undergraduate courses.

For learning this kind of elementary mathematical logic it is crucial to work many exercises. Those exercises can of course be done in the traditional way, using pen

[1] Email: {cek,freek}@cs.ru.nl {mhendri,femke}@few.vu.nl

[2] This research was funded by SURF project 'Web-deductie voor het onderwijs in formeel denken'.

and paper. The student is completely on his own, and in practice it often happens that proofs that are almost-but-not-completely-right are produced. Alternatively, they can be made using some computer program, which guides the student through the development of a completely correct proof. A disadvantage of the computerized way of practicing mathematical logic is that a student often will be able to finish proofs by random experimentation with the commands of the system (accidentally hitting a solution), without really having understood how the proof works. Of course, a combination of the two styles of practicing formal proofs seems to be the best option. So computer assistance for learning to construct derivations in mathematical logic is desirable. Currently the most popular program that is used for this kind of 'computer-assisted logic teaching' is a system called Jape [2], developed at the university of Oxford.

This paper describes our development, named ProofWeb. This system is much like Jape (it might be considered to be an 'improved Jape-clone'). The two main innovations that our system offers over other similar systems are:

- The system makes the students work on a centralized server that has to be accessed through a web interface. The proof assistant that the students use will not run on their computer, but instead will run on the server.

  A first advantage is flexibility. The web interface is extremely light: the student will not need to install anything to be able to use it, not even a plug-in. When designing our system we tried to make it as low-threshold and non-threatening as possible. The student can work from any internet-connection at any time.

  A second advantage is that the student does not need to worry about version problems with the software or the exercises. Since everything is on the same centralized server, the students have at any time the right version of the software, exercises, and possibly solutions to exercises available, and moreover the teachers know at any time the current status of the work of the students.

- The system makes use of a state-of-the-art proof assistant, namely Coq [3], and not of a 'toy' system.

  Coq has been in development since 1984 at the INRIA institute in France. It is based on a type theoretical system called *the Calculus of Inductive Constructions*. It has been implemented in Objective Caml, and has been used for the formal verification of many proofs, both from mathematics and from computer science. The most impressive verification using Coq is the verification of the proof of the Four Colour Theorem by Georges Gonthier [5]. Another important verification is the development of a verified C compiler by Xavier Leroy and others [8].

  The choice for a state-of-the-art proof assistant fell on Coq because both at the RU and at the VU it is already used in research and teaching.

  An advantage of using a state-of-the-art proof assistant is again flexibility. The same interface can be used (possibly adapted) for teaching more advanced courses in logic or concerning the use of the proof assistant.

The system ProofWeb comes equipped with two more products.

- A large collection of logic exercises. The exercises range from very easy to very difficult, and will be graded for their difficulty. The exercise set is sufficiently large (over 200 exercises) that the student will not soon run out of practice material.

More about the exercise set can be found in Section 6.

- Course notes, with a basic presentation of propositional and predicate logic, and a description of how to use the system ProofWeb. We want the presentation of the proofs in the system to be identical to the presentation of the proofs in the textbook. Therefore we develop both the 'Gentzen-style' and the 'Fitch-style' natural deduction variants.

There are already numerous systems for doing logic by computer, of which Jape is the best known. A relatively comprehensive list is maintained by Hans van Ditmarsch [9]. Of course many of these system are quite similar to our system (as well as to each other). For instance, quite a number of these systems are already web-based.

The distinctive features of our system are the use of a serious proof assistant, together with a *centralized* 'web application' architecture. The work of the students remains on the web server, can be saved and loaded back in, and the progress of the student is at all times available both to the student, the teacher and the system (i.e., the system has at all times an accurate 'user model' of the abilities of the student).

Our system has been developed for teaching logic in the natural deduction style. There exists a school of teaching logic due to Dijkstra and Gries, called 'Calculational Logic', in which reasoning is done through rewriting with equations. The Coq system is powerful enough to support this kind of reasoning as well, but we have not developed this style of logic in our system.

In the rest of the paper we present our experiences so far (Section 2), next we present the architecture of the interface (Section 3). Sections 4 and 5 are concerned with the supporting infrastructure of tactics and exercises, and Section 6 with the presentation of the collection of exercises. Finally, in Section 7 we give an outlook on future work.

2 Experience so far

In the beginning of the project, ProofWeb was developed as a web-interface for using Coq on a centralized server. In this status, the system was already used in three master courses on type theory using Coq:

(i) In fall 2006: the course 'Logical Verification' at the VU [10], taught by Femke van Raamsdonk. The course also recapitulates some undergraduate logic, using natural deduction in Gentzen style.

(ii) In spring 2007: the course 'Type Theory' at the RU, taught by Freek Wiedijk and Milad Niqui. This course corresponds to the Logical Verification course at the VU.

(iii) In spring 2007: the course 'Type Theory and Proof Assistants' in the 'Master Class Logic 2006-2007', taught by Herman Geuvers and Bas Spitters. This course is similar to the previous ones, but is aimed at master's students from all over the Netherlands.

These courses were opportunities to test the interface of ProofWeb on more mature students. The efficiency of the server turned out not to be a problem. At peak times around sixty students use about 2Gb memory and a fraction of a CPU. This might be thanks to the fact that the students are not using tactics that involve automation.

In 2007, ProofWeb was used in two different undergraduate logic courses. In both courses the students used the special tactics, the display, and the database with exercises to practice natural deduction proofs.

(i) In spring 2007: the course 'Beweren en Bewijzen' at the RU [1] taught by Hanno Wupper and Erik Barendsen. This is a computer science undergraduate course in logic using Gentzen style 'tree' proofs. See Section 4 for a more elaborate discussion about ProofWeb and Gentzen style natural deduction.

(ii) In fall 2007: the course 'Inleiding Logica' at the VU [6] taught by Roel de Vrijer. This is a computer science undergraduate course in logic using Fitch style 'flag' proofs. See Section 5 for a more elaborate discussion about ProofWeb and Fitch style natural deduction.

3 Architecture of the interface

The architecture of the interface to Coq used in ProofWeb is an implementation of an architecture for creating responsive web interfaces for proof assistants [7]. It combines the current web development technologies with the functionality of local interfaces for proof assistants to create an interface that behaves like a local one, but is available completely with just a web browser (no Java, Flash or plug-ins are required).

To provide this it uses the *asynchronous DOM modification* technology (sometimes referred to as *AJAX* or *Web Application*). This technique is a combination of three available web technologies: JavaScript – a scripting programming language interpreted by the web browsers, *Document Object Model* (*DOM*) – a way of referring to sub elements of a web page that allows modification of the page on the fly creating dynamic elements and XmlHttp – an API available to client side scripts, that allows requesting information from the web server without reloading the page.

The technique consists in creating a web page that captures events on the client side and processes them without reloading the page. Events that require information from the server send the data in asynchronous XmlHttp requests and modify the web page in place. Other events are processed only locally. The server keeps prover sessions for all users and the clients are presented with an interface that is completely available in a web-browser but resembles and is comparably responsive to a local interface.

The architecture described in [7] was designed as a publicly available web service. Using it for teaching required the creation of groups of logins for particular courses. The students are allowed to access only their own files via the web interface, and teachers of particular courses have access to students' solutions through the admin interface.

An example of the use of the interface can be seen in Fig. 1.

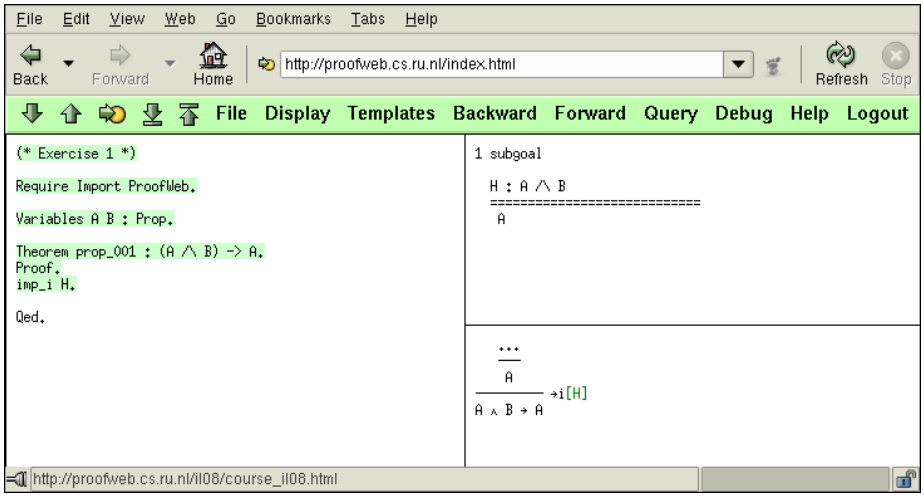


Fig. 1. A propositional logic exercise in ProofWeb.

4 Gentzen style natural deduction for first-order logic

A first aim in the development of ProofWeb was a have an exact correspondence between the derivations on paper and the derivations in the system. The student should then work with a set of dedicated tactics, because the standard Coq tactics are too powerful. (For instance, then one could solve the exercises in propositional logic using the tactic `tauto` instead of building the actual derivation.) We naturally arrived at a set of backward working tactics: every proposition (the current goal) is deduced from another proposition (the new goal) using a deduction rule. This imposes a relatively strict way of working. The proof trees have to be constructed from 'bottom to top'. On the one hand, this makes the construction of a deduction more difficult than on paper, because there is no possibility of building snippets of the proof in a forward way, using what is known from the hypotheses and their consequences. But on the other hand, the method forces the student to ponder the general structure of the proof before deciding by what step he will eventually end up with the current proposition.

As an example we present the tactic for disjunction elimination, which gives a good impression of the way additional tactics are implemented:

```
Ltac dis_e X H1 H2 :=
  match X with | ( _ \/ _ ) =>
   let x := fresh "H" in
   assert (x : X);
   [ idtac | elim x; clear x; [intro H1 | intro H2] ]
  end || fail "(the argument is not a disjunction
    or the labels already exist)".
```

If the current goal is C, the tactic `dis_el (A \/ B) G H` will create the following three new goals:

(i) $A \vee B$;

(ii) C, with the extra assumption A with name (or proof, if viewed constructively) G;

(iii) C, with the extra assumption B with name H.

An example proof with our set of tactics (in all exercises the domain is non-empty):

```
Theorem pred_076 : all x, exi y, (P(x) \/ P(y)) -> exi x, P(x).
Proof.
imp_i H.
insert G (exi y, (P(x0) \/ P(y))).
f_all_e H.
exi_e (exi y, (P(x0) \/ P(y))) y0 J.
ass G.
dis_e (P(x0) \/ P(y0)) K K2.
ass J.
f_exi_i K.
f_exi_i K2.
Qed.
```

4.1 Visualization

A second aim is a visual presentation of proofs as in Jape. This meant requesting the proof information from Coq and converting it to a graphic format. Coq internally keeps a proof state. This proof state is a recursive OCaml structure, that holds a goal, a rule which allows to obtain this goal from the subgoals, and the subgoals themselves. The Coq commands that allow inspecting the proof state (`Show`, `Show Tree` and `Show Proof`) were not sufficient to build a natural deduction tree for the proof. We added a new command `Dump Tree` to Coq that allows exporting the whole proof state in an XML format. An example of the output of the `Dump Tree` command for a very simple Coq proof:

```
<tree><goal><concl type="A -> A"/></goal>
  <cmpdrule><tactic cmd="intro x"/>
    <tree><goal><concl type="A -> A"/></goal>
      <cmpdrule><tactic cmd="intro x"/>
        <tree><goal><concl type="A -> A"/></goal>
          <rule text="intro x"/>
          <tree><goal><concl type="A"/><hyp id="x" type="A"/>
        </goal></tree></tree>
      </cmpdrule>
      <tree><goal><concl type="A"/><hyp id="x" type="A"/>
    </goal></tree></tree>
  </cmpdrule><tree><goal><concl type="A"/><hyp id="x" type="A"/>
</goal></tree></tree>
```

ProofWeb is able to parse the XML trees dumped by Coq and generate natural deduction diagrams (Fig. 2). The user's browser may request diagrams (when no text is being processed) and displays them in a separate frame in the interface along with the usual Coq proof state.

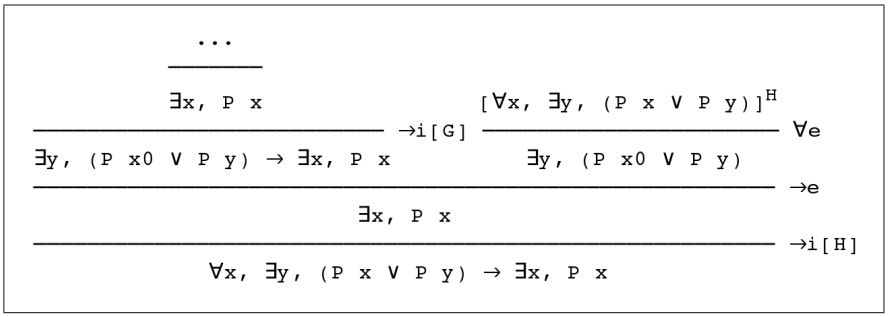


Fig. 2. A natural deduction tree as seen on the web page.

5 Fitch style natural deduction for first-order logic

The ProofWeb system also has the possibility to use the system for so-called *Fitch-style* natural deduction proofs.[3] Fitch-style proofs have the graphical advantage over Gentzen-style proofs of being linear (as opposed to having a branching tree structure), which makes them more convenient to display for large proofs, like the ones constructed by the students for final assignments. Another name for these kind of proofs is *flag-style proofs*, because the assumptions of a subproof are often written in the shape of 'flags'.

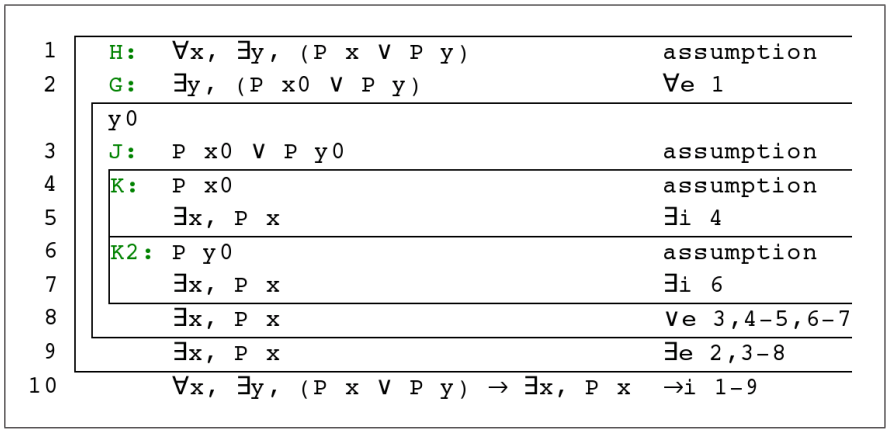


Fig. 3. A Fitch-style deduction rendered by the system.

6 The exercise set

As a part of the project a set of over 200 exercises was developed. If a student logs in via the web interface as a participant to a specific course, he sees the list of exercises for the course. It gives for each exercise the name of file that holds the exercise, an indication of the difficulty, the current status of the exercise, and a button for resetting the exercise to its initial state.
The four possibilities for the status of an exercise are:

– Not touched (in grey)
– Incomplete (in red)
– Correct (in orange)
– Solved (in green)

The colors are meant to resemble the colors of a traffic light.

- The status Not touched means that an exercise has not been opened, or has been opened but has not been saved.
- The status Incomplete means that the file is incomplete or wrong. If you want to know why ProofWeb thinks there is an error in your solution, you can click the why? link next to the status. You will get a new window that shows the error message of Coq.

[3] This style of proof was initially developed by Stanisław Jaśkowski in 1934 and perfected by Frederic Brenton Fitch in 1952.

- The status Correct means that the file is a correct Coq-file. However, the file is not accepted as a solution for the exercise in the course. This happens for instance if more automation (present in Coq) is used than intended for the course, for instance: by proving a propositional formula with the tactic `tauto` instead of using the tactics corresponding exactly to the logical rules used in the course. It also happens if the file is empty. If the status is Correct you can click on the why? link to find out what you did that ProofWeb thinks is not allowed in the course.

  This is a feature of ProofWeb is meant as a service for the teacher, but of course in addition manual verification may be required, for instance if the exercise is to give the definition of a certain object in type theory.
- The status Solved means that the file is correct Coq and moreover is accepted as a solution in the course.

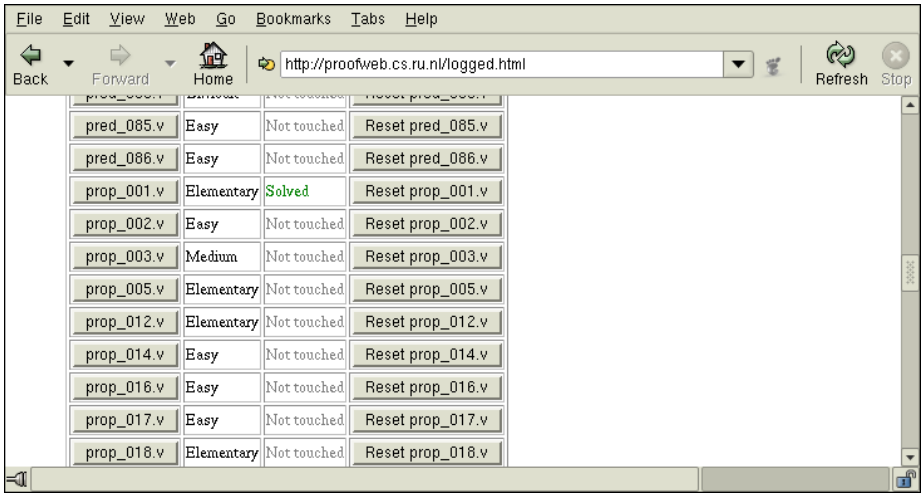


Fig. 4. Tasks assigned to students and their status.

7 Future work

Some of issues that currently are being worked on are the following.

- A first version of the course notes is available via the web-page of the system.
- The deduction trees are currently rendered as text or HTML in IFrames, and can be optionally opened in a separate browser window to allow easy printing as PostScript or PDF. However students may need to use the trees in texts, and for that a dedicated TEX or image rendering of the trees could be implemented.
- The interface uses some web technologies that are not implemented in the same way in all browsers. It includes a small layer that is supposed to abstract over incompatible functionalities. Currently this works well with Gecko based browsers (like Mozilla, Firefox, Galeon, Epiphany and Netscape), Webkit based browsers (like Safari and Konqueror), and the Opera browser. Also, some effort has been made to make the system work reasonably well with common versions of Internet Explorer however it needs further attention.

- The system a log of each interaction of each student session is already stored on the server. Using these logs, it is possible to develop software for 'replaying' student sessions. We are currently discussing whether it is useful to develop such an extension of the system.
- The system was designed in a way to be used in standard university courses. It might be useful to create a more complete online environment that would include introductory explanations and adaptive user profiles, therefore allowing students to learn logic without teacher interaction.

If the development of ProofWeb is finished, a possibility is to integrate it with a system that supports the development of more serious proofs with the Coq system. One of the other projects that currently is being pursued is the creation of a so-called 'math wiki' [4]. Here, traditional wiki technology is integrated with the same proof assistant front end that our system is based on.

References

[1] *Beweren en Bewijzen*.
URL http://www.cs.ru.nl/~wupper/B&B/index.html

[2] Bornat, R. and B. Sufrin, *Jape's quiet interface*, in: N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report (1996), pp. 25–34.

[3] Coq Development Team, "The Coq Proof Assistant Reference Manual Version 8.1," INRIA-Rocquencourt (2006).

[4] Corbineau, P. and C. Kaliszyk, *Cooperative repositories for formal proofs*, in: M. Kauers, M. Kerber, R. Miner and W. Windsteiger, editors, *Calculemus/MKM*, Lecture Notes in Computer Science **4573** (2007), pp. 221–234.

[5] Gonthier, G., *A computer-checked proof of the Four Colour Theorem* (2006).
URL http://research.microsoft.com/~gonthier/4colproof.pdf

[6] *Inleiding Logica*.
URL http://www.cs.vu.nl/~tcs/il/

[7] Kaliszyk, C., *Web interfaces for proof assistants*, Electr. Notes Theor. Comput. Sci. **174** (2007), pp. 49–61.

[8] Leroy, X., *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*, in: *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2006), pp. 42–54.

[9] *Logic courseware*.
URL http://www.cs.otago.ac.nz/staffpriv/hans/

[10] *Logical Verification*.
URL http://www.cs.vu.nl/~tcs/lv/

A Prototype Environment for Verification of Recursive Functional Programs

Nikolaj Popov [1,3] Tudor Jebelean [2,3]

RISC
Johannes Kepler University
Linz, Austria

Abstract

We present an experimental prototype environment for defining and verifying recursive functional programs, which is part of the *Theorema* system. A distinctive feature of our approach is the hint on "what is wrong" in case of a verification failure. The prototype is designed in order to improve the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

Keywords: Formal Methods, Program Verification, Software Engineering, Theorema.

1 Introduction

The use of formal methods for software design is motivated by the expectation that, performing appropriate mathematical reasoning can contribute to the reliability of a design. In our opinion, these methods should be used in practice, however, their acceptance by industry is not yet very broad. The authors are convinced that in order to increase the practical impact of formal methods, the education of future software engineers should be improved.

We present an experimental prototype environment for defining and verifying recursive functional programs, which is part of the *Theorema* system. In contrast to classical books on program verification [6],[4],[10] which expose methods for verifying correct programs, we put special emphasize on verifying incorrect programs. The user may easily interact with the system in order to correct the program definition or the specification.

[1] Email: popov@risc.uni-linz.ac.at

[2] Email: jebelean@risc.uni-linz.ac.at

[3] The program verification project is supported by INTAS Ref. Nr 05-1000008-8144. The Theorema project is supported by FWF (Austrian National Science Foundation) – SFB project F1302. Additional inspiration came from discussions made within the frame of Aktion Österreich-Ungarn – project 66öu2.

There are various tools for proving program correctness automatically or semi-automatically, (see, e.g., [12],[1],[2]), and this is where our contribution falls into. As a distinctive feature of our prototype is the hint on "what is wrong" in case of a verification failure.

This work is performed in the frame of the *Theorema* system [3], a mathematical computer assistant which aims at supporting all phases of mathematical activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides both functional as well as imperative programming constructs. Moreover, the logical verification conditions which are produced by the methods presented here can be passed to the automatic provers of the system in order to be checked for validity. The system includes a collection of general as well as specific provers for various interesting domains (e. g. integers, sets, reals, tuples, etc.). More details about *Theorema* could be found at `www.theorema.org`.

2 Programming, Specification and Verification

As a first step, we emphasize the importance of formalization of specifications, that is, each program definition is accomplished by its input and output specifications. As it turns out, providing a program specification or giving the definition of what a program is expected to do along with its source code, is commonly considered to be unusual or even redundant. A special emphasis goes to the fact that a program (its source code) cannot be correct on its own, but only correct with respect to a given specification.

The question whether the formal specification correctly describes the program is normally called verification and is discussed in this paper.

We furthermore perform a check whether the program under consideration is *coherent* with respect to its specification, that is, each function call is applied to arguments obeying the respective input specification.

The program correctness is then transformed into a set of first-order predicate logic formulae by a Verification Condition Generator (VCG) – a device, which takes the program (its source code) and the specification (precondition and postcondition) and produces several verification conditions, which themselves, do not refer to any theoretical model for program semantics or program execution, but only to the theory of the domain used in the program.

However, there is no "universal" VCG, due to the fact that proving program correctness is undecidable in general. On the other hand, in practice, proving program's correctness is possible in many particular cases and, therefore, many VCG-s have been developed for serving a big variety of situations. Our research is contributing exactly in this direction.

Our prototype is designed for recursive programs which may have multiple choice *if-then-else* with zero, one or more recursive calls on each branch (but no nested recursion allowed) – these are the most used in practice.

For coherent programs we are able to define a necessary and sufficient set of verification conditions, thus our condition generator is not only *sound*, but also *complete*. This distinctive feature of our method is very useful in practice for pro-

gram debugging – as we also demonstrate by an example.

3 Coherence and Verification Conditions

We consider the correctness problem expressed as follows: *given* the program which computes the function F in a domain D and given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$, *generate* the verification conditions VC_1, ... VC_n which are sufficient for the program to satisfy the specification. The function F satisfies the specification, if: F terminates on any input x satisfying I_F, and, for each such input, the condition $O_F[x, F[x]]$ holds. This is also called "total correctness" of the program.

$$(\forall x : I_F[x])\ O_F[x, F[x]], \tag{1}$$

Any VCG should come together with its *Soundness* statement, that is: for a given program F, defined on a domain D, with a specification I_F and O_F if the verification conditions VC_1, ... VC_n hold in the theory of D, then the program F satisfies its specification I_F and O_F.

Moreover, we are also interested in the following question: What if some of the verification conditions do not hold? May we conclude that the program is not correct? In fact, the program may still be correct. However, if the VCG is complete, then one can be sure that the program is not correct. A VCG is complete, if whenever the program satisfies its specification, the produced verification conditions hold.

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on "what is wrong". Indeed, most books about program verification present methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

3.1 Coherent Programs

In this subsection we state the principles we use for writing coherent programs with the aim of building up a non-contradictory system of verified programs. Although, these principles are not our invention (similar ideas appear in [8]), we state them here because we want to emphasize on and later formalize them.

Building up correct programs: Firstly, we want to ensure that our system of coherent programs would contain only correct (verified) programs. This we achieve, by:

– start from basic (trustful) functions e.g. addition, multiplication, etc.;

– define each new function in terms of already known (defined previously) functions by giving its source text, the specification (input and output predicates) and prove their total correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our system. The next we want to ensure is the easy exchange (mobility) of our program implementations. This principle is usually referred as:

Modularity: Once we define the new function and prove its correctness, we "forbid" using any knowledge concerning the concrete function definition. The only knowledge we may use is the specification. This gives the possibility of easy replacement of existing functions. For example we have a powering function P, with the following program definition (implementation):

$$P[x, n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } P[x, n-1] * x$$

The specification of P is:
The domain $\mathbb{D} = \mathbb{R}^2$, precondition $I_P[x, n] \iff n \in \mathbb{N}$ and a postcondition $O_P[x, n, P[x, n]] \iff P[x, n] = x^n$.

Additionally, we have proven the correctness of P. Later, after using the powering function P for defining other functions, we decide to replace its definition (implementation) by another one, however, keeping the same specification. In this situation, the only thing we should do (besides preserving the name) is to prove that the new definition (implementation) of P meets the old specification.

Furthermore, we need to ensure that when defining a new program, all the calls made to the existing (already defined) programs obey the input restrictions of that programs – we call this:

Appropriate values for the auxiliary functions. The following example will give an intuition on what we are doing. Let the program for computing F be:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x],$$

with the specification of F (I_F and O_F) and specifications of the auxiliary functions H (I_H and O_H), G (I_G and O_G). The two verification conditions, ensuring that the calls to the auxiliary functions have appropriate values are:

$$(\forall x : I_F[x])\ (Q[x] \implies I_H[x])$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]).$$

3.2 Recursive Programs and Generation of Verification Conditions

As we already mentioned, there is no universal VCG. Thus, in our research, we concentrate on constructing a VCG which is appropriate only for a certain kind of recursive programs – those which are defined by multiple choice *if-then-else* with zero, one, or more recursive calls on each branch (but without nested recursion). They are defined as those F:

$$\begin{aligned}(2)\quad F[x] = \ & \textbf{If } Q_0[x] \textbf{ then } S[x] \\ & \textbf{elseif } Q_1[x] \textbf{ then } C_1[x, F[R_1[x]]] \\ & \textbf{elseif } Q_2[x] \textbf{ then } C_2[x, F[R_2[x]]] \\ & \dots \\ & \textbf{else } Q_n[x] \textbf{ then } C_n[x, F[R_n[x]]].\end{aligned}$$

where Q_i are predicates and S, C_i, R_i are auxiliary functions ($S[x]$ is a "simple" function (the bottom of the recursion), $C_i[x, y]$ are "combinator" functions, and $R_i[x]$ are "reduction" functions). We assume that the functions S, C_i, and R_i satisfy

their specifications given by $I_S[x]$, $O_S[x,y]$, $I_{C_i}[x,y]$, $O_{C_i}[x,y,z]$, $I_{R_i}[x]$, $O_{R_i}[x,y]$. Additionally, assume that the Q_i predicates are non-contradictory, that is $Q_{i+1} \Rightarrow \neg Q_i$ and $Q_n = \neg Q_0 \wedge \ldots \wedge \neg Q_{n-1}$, which we do only in order to simplify the presentation.

Note that functions with multiple arguments also fall into this scheme, because the arguments x, y, z could be vectors (tuples).

Type (or domain) information does not appear explicitly in this formulation, however it may be included in the input conditions.

Considering Coherent Recursive programs, we give here the appropriate definition:

Let S, C_i, and R_i be functions which satisfy their specifications. Then the program (2) is coherent if the following conditions hold:

$$(\forall x : I_F[x])\ (Q_0[x] \implies I_S[x]) \tag{3}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_F[R_1[x]]) \tag{4}$$

$$\ldots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_F[R_n[x]]) \tag{5}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_{R_1}[x]) \tag{6}$$

$$\ldots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_{R_n}[x]) \tag{7}$$

$$(\forall x : I_F[x])(Q_1[x] \wedge O_F[R_1[x], F[R_1[x]]] \implies I_{C_1}[x, F[R_1[x]]]) \tag{8}$$

$$\ldots$$

$$(\forall x : I_F[x])(Q_n[x] \wedge O_F[R_n[x], F[R_n[x]]] \implies I_{C_n}[x, F[R_n[x]]]). \tag{9}$$

It is not that a program which is not coherent is necessarily not correct. However, non-coherent programs are somehow inconsistent, namely proving their correctness would involve knowledge about their auxiliary functions which is out of the official scope. Thus, if we allow them in our system of verified programs, the modularity would be lost.

After performing the coherence check, we go to the verification. The upcoming theorem gives the necessary and sufficient conditions for a program to be correct. These conditions are taken as the *Verification Conditions*.

Theorem 3.1 *Let S, C_i, and R_i be functions which satisfy their specifications. Let also the program (2) be coherent. Then, (2) satisfies the specification given by I_F and O_F if and only if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{10}$$

$$(\forall x : I_F[x])(Q_1[x] \wedge O_F[R_1[x], F[R_1[x]]] \implies O_F[x, C_1[x, F[R_1[x]]]]) \tag{11}$$

$$\ldots$$

$$(\forall x : I_F[x])(Q_n[x] \wedge O_F[R_n[x], F[R_n[x]]] \implies O_F[x, C_n[x, F[R_n[x]]]]) \tag{12}$$

$$(\forall x : I_F[x])\ (F'[x] = 0) \tag{13}$$

where F' is defined as:

(14) $F'[x] =$ **If** $Q_0[x]$ **then** 0

elseif $Q_1[x]$ **then** $F'[R_1[x]]$
elseif $Q_2[x]$ **then** $F'[R_2[x]]$
...
else $Q_n[x]$ **then** $F'[R_n[x]]$.

Based on this statement we construct a VCG, which takes as an input program (2) annotated with its specification I_F and O_F, and generates the verification conditions (10), (11), (12) and (13). Moreover, the theorem gives, in fact, two statements, namely:

– *Soundness*: If (10), (11), (12) and (13) hold, then the program (2) is correct, and

– *Completeness*: If (2) is correct, then (10), (11), (12) and (13) hold.

A precise proof of the theorem, based on the fixpoint theory of programs [11], is presented in [7], and completed in [9].

3.3 Proving the Verification Conditions

As we have already said, the coherence check is done at the beginning of the verification process—it is also realized by proving the validity of the respective conditions: (3), (4), (5), (6), (7), (8) and (9). Partial correctness is guarantied by (10), (11), (12), and termination—(13).

Proving any of the three kinds of verification conditions has its own difficulty, however, our experience shows that proving coherence is relatively easy, proving partial correctness is more difficult and proving the termination verification condition (it is only one condition) is in general the most difficult one. The latter one is expressed by using a *simplified version*(14) of the initial program (2), and the condition itself expresses a property of that *simplified version* (13). The proof typically needs an induction prover and the induction step may sometimes be difficult to find. Fortunately, due to the specific structure, the proof may be omitted, because different recursive programs may have the same *simplified version*.

Proofs of the verification conditions may be done by using a *Theorema* prover (see, e.g., [3],[5]) or by delivering the proof problem itself to another specialized tool. For serving the termination proofs, actually for omitting the proof redundancy, we are now creating libraries containing *simplified versions* together with their input conditions, whose termination is proven. The proof of the termination may now be skipped if the *simplified version* is already in the library and this membership check is much easier than an induction proof – it only involves matching against simplified versions.

4 Example and Discussion

In order to make clear our experiments, we consider again a powering function P, however we provide this time a different implementation, namely *binary powering*:

$$\begin{aligned} P[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

This program in the context of the theory of real numbers, and in the following formulae, all variables are implicitly assumed to be real. Additional type information (e. g. $n \in \mathbb{N}$) may be explicitly included in some formulae.

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x,n] = x^n. \tag{15}$$

The (automatically generated) conditions for **coherence** are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow \mathbb{T}) \tag{16}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n]\ \Rightarrow \mathrm{Even}[n]) \tag{17}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n]\ \Rightarrow \mathrm{Odd}[n]) \tag{18}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2} \Rightarrow \mathbb{T}) \tag{19}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow \mathbb{T}) \tag{20}$$

One sees that the formulae (16), (19) and (20) are trivially valid, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the 1 *constant-function-one* and the $*$ *multiplication*.

The formulae (17) and (18) are easy consequences of the elementary theory of reals and naturals. For the further check of **correctness** the generated conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 1 = x^n) \tag{21}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n]\ \Rightarrow n/2 \in \mathbb{N}) \tag{22}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2}\ \Rightarrow m = x^n) \tag{23}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n]\ \Rightarrow (n-1)/2 \in \mathbb{N}) \tag{24}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow x * m = x^n) \tag{25}$$

$$(\forall x, n : n \in \mathbb{N})\ P'[x,n] = 0, \tag{26}$$

where

$$\begin{aligned} P'[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P'[x * x, n/2] \\ & \textbf{else } P'[x * x, (n-1)/2]. \end{aligned}$$

The proofs of these verification conditions are straightforward.

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program P is now almost the same as the previous one, but in the base case (when $n = 0$) the return value is 0.

$$\begin{aligned} P[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, for this buggy version of P we may see that all the respective verification conditions remain the same—and thus the program is correct—except one, namely,

(21) is now:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 0 = x^n) \tag{27}$$

which itself reduces to:

$$0 = 1$$

(because we consider a theory where $0^0 = 1$).

Therefore, according to the *completeness* of the method, we conclude that the program P does not satisfy its specification. Moreover, the failed proof gives a hint for "debuging": we need to change the return value in the case $n = 0$ to 1.

Furthermore, in order to demonstrate how a bug might be located, we construct one more "buggy" example where in the "Even" branch of the program we have $P[x, n/2]$ instead of $P[x * x, n/2]$:

$$\begin{aligned} P[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, we may see again that all the respective verification conditions remain the same as in the original one, except one, namely, (23) is now:

$$(\forall x, n : n \in \mathbb{N})\ (\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x)^{n/2}\ \Rightarrow m = x^n) \tag{28}$$

which itself reduces to:

$$m = x^{n/2}\ \Rightarrow m = x^n$$

From here, we see that the "Even" branch of the program is problematic and one should satisfy the implication. The most natural candidate would be:

$$m = (x^2)^{n/2}\ \Rightarrow m = x^n$$

which finally leads to the correct version of P.

5 Conclusions

The approach to program verification presented here is a result of an experimental work with the aim of practical verification of recursive programs. Although the examples presented here appear to be relatively simple, they already demonstrate the usefulness of our approach in the general case. We aim at extending these experiments to industrial-scale examples, which are in fact not more complex from the mathematical point of view. Furthermore we aim at improving the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

Furthermore, we want to approach the problem of program synthesis which may replace the nowadays standard way of programming. However, before going to synthesis, one has to establish the importance of formalization of specifications and this is what we are doing.

References

[1] Y. Bertot, P. Casteran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer Verlag, 2004.

[2] F. Blanqui , S. Hinderer, S. Coupet-Grimal, W Delobel, A. Kroprowski. A Coq library on rewriting and termination. *http://coq.inria.fr/contribs/CoLoR.html*

[3] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. In *Journal of Applied Logic, vol. 4, issue 4*, pp. 470–504, 2006.

[4] B. Buchberger and F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics (in German)*. Springer, 2nd edition, 1981.

[5] B. Buchberger, D. Vasaru. Theorema: The Induction Prover over Lists. In *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 1997.

[6] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.

[7] T. Jebelean, L. Kovács, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2006. To appear.

[8] M. Kaufmann and J. S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.

[9] L. Kovacs, N. Popov, T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, Paphos, Cyprus, 2006.

[10] J. Loeckx, K. Sieber. The Foundations of Program Verification. Teubner, second edition, 1987.

[11] Manna, Z.: *Mathematical Theory of Computation*. McGraw-Hill Inc. (1974)

[12] PVS: Specification and Verification System. *http://pvs.csl.sri.com*

Teaching Model-Based Testing with the Leirios Test Generator

Frédéric Dadeau, Régis Tissot [1,2]

Laboratoire d'Informatique
Université de Franche-Comté
16 route de Gray
F-25030 Besançon, FRANCE

Abstract

This paper proposes a technique to encourage the interest of students in learning formal methods. Our course is focused on the B method, involving basic knowledge of set theory, invariance proofs, refinement techniques and so on. While lectures and tutorials cover a large range of such concepts, the practical work is focused on applying the principles of a model-based approach in the context of test generation. This paper explains the pratical outcome of the course, through the Leirios Test Generator tool, that gives an interesting and playful use of the B method, by simulating the execution of the model through animation, and by generating tests –based on the B model– that can be run on an implementation. In order to make sure that students will be interested in applying these techniques, we challenge them to play a game consisting in detecting mutants of a program with their model-based tests. The feedback from the students is very positive here, and suggests that formal methods are more likely to be understood if their interest is shown through a concrete application.

Keywords: B notation, model-based testing, proof, animation, testing, mutation

1 Introduction

The B method [Abr96] is one of the most common system development method used in France for teaching formal methods. The reason that motivates this choice is twofold. First, the B notation is well tool-supported by provers (the AtelierB [Cle04] or its free version, B4free [Cle]), animators (BZ-Testing-Tools [ABC+02] or ProB [LB03]), platforms (Rodin [Abr06], for EventB) or test generators (BZ-Testing-Tools, Leirios Test Generator (LTG) [3] [JL07]). Second, the B notation requires minimal basic notions of first-order logic, and set theory, which makes it a well-suited specification language for the teaching of formal methods.

The B method starts by the writing of a formal specification, named **abstract machine**, that gives a functional view of the system. This abstract machine is

[1] Email: {dadeau,tissot}@lifc.univ-fcomte.fr

[2] This work is partially funded by the Région Franche-Comté.

[3] LTG is the commercial version of the BZ-Testing-Tools developed at the University of Besançon.

spiced up with invariant properties that represent properties that have to hold at each state of the system execution. It means that (*i*) the initialization has to establish the invariant, (*ii*) the operations have to preserve the invariant (meaning that if the invariant is satisfied before the operation, then it also has to be satisfied after the execution of the operation). Operations are written in terms of *Generalized Substitutions* that are built on basic assignments, composed of more generalized and expressive structures, that may e.g., represent conditional substitutions (IF...THEN...ELSE...END) or non-deterministic buildings (CHOICE, ANY).

The **refinement** steps consist in increasing the detail level of the model. The additional elements of the refinement have to be linked to the abstract machine. The refinement step can be iterated over and over again, until the highest detail level is reached. Finally, when the refinements steps are completed, an **implementation** may be written, based on the last refinement, that describes a ready-to-execute system. From there concrete code (e.g. C or Ada) may be generated. The interest of specifying by refinement is that the properties that have been established w.r.t. the invariants at each level are preserved through the refinement relationship. This considerably simplifies the writing of a complete formal model "at once" which is source of numerous errors/defaults, and it is also an help for the proof of the model.

B is the starting point for studying formal methods in our University. It is the first modeling language that is introduced to our students. Even if the language seems to be easy-to-learn and easy-to-use, there exists several problems to make them practice formal methods. As researchers in the domain of formal methods, we exhort our students to practice formal methods in the software engineering process, even when they will be working in industry. The major problem is that they usually do not see the concrete application of writing a formal model.

This paper proposes an approach, based on a concrete application of the formal methods that helps the students to understand their usefulness. The idea is to force them to work with a model-based test generator that represents a concrete application of formal modelling. Section 2 presents the formal methods course to which the students attend. Then, we present the tools we use in practical work session in Section 3, namely B4free for the proof part, and LTG for the testing part. The description of what has to be done during the project is given in Section 4. We present the case study on which the students work as a practical session project in Section 5. Finally, Section 6 analyses the results obtained in this "teaching experiment", before concluding and presenting the future courses in Section 7.

2 Presentation of the Course

B is taught in our university to the master students of computer science; the main notions of first-order logic and set theory are seen two years before. The main problem with the B-method is its lack of concretization. Most of the students complain about the too high abstraction level of the models. Their greatest difficulty is to understand how models can be employed, and thus, they doubt of their usefulness. A corollary difficulty in learning the B-method is to see how do an abstract B model and a concrete code implementation relate to each other, due to the difference of abstraction level between them.

2.1 Theoretical Sessions

Lectures cover the basics of B notation, and B method. This include abstract machines, proof obligations generations, machine compositions, refinement and implementations. Tutorials concretize these concepts, through exercises.

Our objective is to use formal methods in a concrete way so as to ensure that the students understand their usefulness. Thus, practical sessions cover a different aspect of B method, by illustrating the use of formal models in the software development process.

2.2 Practical Sessions

Practical sessions start by learning the basics of B notation, using the JEdit editor improved with the B plug-in, which offers a toolbar for mathematical symbols and a typechecking feature. Then, students are asked to prove their models using the B4free prover. One of the first reaction of the students after having written and proved their model is: "ok, and so what? can it be executed?". This question is quite logical since the machine represents a system, or a program, and a natural reflex is to check whether it acts as expected.

Unfortunately, proof can hardly be used for that step. Thus, we use the Leirios Test Generator tool, which provides an user-friendly animator in order to validate the behaviour of the model.

2.3 Student's point of view

We noticed that it is quite complicated to teach students a refinement approach. It appears that they are expecting concrete lessons; doing the effort of writing one model is acceptable, but adopting a complete refinement-based approach represents for them a waste of time. Note that this vision is shared by most of our industry partners.

3 Tools

3.1 B4free/Click'n'prove

Click'n'prove[4] is a graphic interface (see Fig. 1), developed by J.-R. Abrial and D. Cansell, for the interactive prover B4free, which is a single-user version of the "Atelier B" developed by ClearSy System Engineering.

3.1.1 Interest of the tool

The main objective of using Click'n'prove is to provide to students a complete framework including formal modelling and verification. Typechecking and proof features of this framework make the students apply the notions they learned during the theorical sessions. Students are ask to produce a model which is both syntactically correct and consistent.

[4] http://www.loria.fr/~cansell/cnp.html

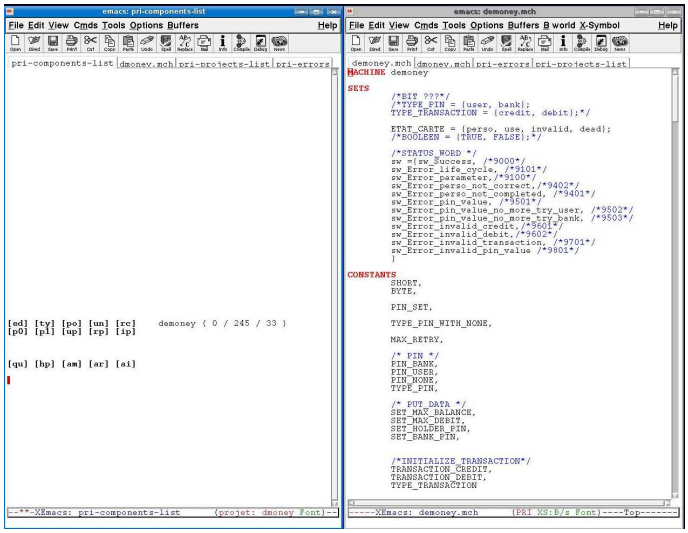


Fig. 1. A screenshot of the Click'n'Prove GUI

Working with this tool shows the students what is an interactive prover, the automatic proof features and their limitations. When automatic proof fails, the interactive proof mode forces the student to understand the proof obligations' definitions, and mechanisms. Then they have to correct their model or to solve by themselves the proof obligations with the interactive proof support.

3.1.2 Features of the tool

Click'n'prove is grounded on the B-method introduced by J.-R. Abrial [Abr96]. This methods aim to define the development of computer software process based on refinement. In order to support a B-method based development process, Click'n'prove provides two functionalities. The first one of these features is the *typecheck* functionality that makes it possible to check whether a B model is syntactically correct.

The second feature of Click'n'prove is the *proof support*. First, the software generate proof obligations have to be proved to ensure that all properties expressed in the invariant are still satisfied after each operation of the system and that all properties expressed over a more abstract level are still verified through its refinements.

The automatic prover solves most of the proof obligations. When some proof obligations are not automatically proved, the user enters the interactive proof mode. In this mode, the proof obligations are summarized as a list of hypothesis and goals to reach. The user disposes of some features to solve the remaining proof obligations, such as proof by case, proof by contradiction, selecting/removing hypothesis from the invariant, add assertions, etc.

3.1.3 Which features do we use ?

The type-checking confronts the students to their syntactic errors. Students realize that even though the specification of a system seems informal, the related model should be formal. Generally, syntactic errors relate to multiple variable assignment in one operation, incomplete initialization of variables, or mismatch between sets and elements.

Through the proof obligations solving process, students are confronted to the incoherence of their model. When a property fails to be proved automatically, they have to discover the problem, and thus, they need to understand the proof obligations. First, they have to verify if the proof obligation can be proved. If not, they have to answer to some questions, e.g., "Are the invariant properties correct ?", "Are there conflicting properties in the invariant?" or "Do all the behaviours of the operations preserve the invariant?" in order to locate their mistake(s). Otherwise, if the proof obligation can be proved, they have to answer other questions, e.g., "Does the invariant contain all the properties needed to solve the proof obligation?", "How is it possible to help the prover solving this proof obligation? by case? by contradiction? etc.".

3.1.4 Feedback from the students

The main problem is that most of the student find that the emacs-based graphical interface is not intuitive. This essentially results from the following reasons. Click'n'prove does not have a graphical interface like the majority of the software they use generally. The tool suffers from a lack of flexibility: a model can not be modified out of the Click'n'prove editing window, otherwise the project is locked. Finally, the substitution of mathematical symbols is systematic (e.g. string "pin" is displayed "πn" in the editor window).
Nevertheless in practice, they quickly accommodate to this tool. Contrary to Click'n'prove, the second tool we consider, LTG, has a user-friendly graphical user interface as described below.

3.2 Leirios Test Generator

LTG is a software developed by LEIRIOS Technologies, it is a test generator based on constraint solving techniques (see Fig. 2).

3.2.1 Interest of the tool

In practice, the students do not refine their model to generate the code of an implementation. Thus, thanks to LTG, students confront their model to a real implementation. The model animator of LTG offer a complementary approach for the students to validate their model. The tests generator feature introduces them the notions of coverage, verdict and search heuristics.

3.2.2 Features of the tool

LTG includes a model animator which is used to validate the model or to manually define tests sequences. This animator verifies the preservation of invariant properties after operation calls. It makes it possible to check whether the model behaves as expected in the informal specifications.

For each test generation campaign, the user can set some parameters which are crucial for the results, namely: the initial state of the system, the state to reach by a test sequence –crucial if one wants to concatenate tests sequences–, the selection of the operations to cover, the condition coverage criteria, some other criteria e.g. transition pair coverage, the observation of the system after a test, the heuristic em-

ployed to reach the test targets, including *preambule helper* strategies (ie. operation sequences defined by the user to help the software to find tests sequences).

3.2.3 Which features do we use?

Each student's B model is given as an input to LTG. The animator helps the student to understand which behaviours are covered by the model. This feature can be used to "replay" a test sequence and analyse the state of the system in order to find more precisely why the test sequence fails. Then, LTG is used to derive test sequences based on the student's model analysis. In order to generate relevant tests, students have to precisely set the campaign generation parameters. During the practical work sessions, they have to produce test campaigns.

For each test campaign, students have to define a initial state and the set of operations which are tested. Then, for each tested operation, they select the coverage criteria for the conditions and the sequence of observation. The use of observation operations shows them the importance of having an accurate verdict for a test sequence –the more comparisons are done, the more conformance verdicts are precise.

In addition, students have to set up strategies for test target search. They can concatenate different automatic strategies –width-first, forward/backward search with best first, preamble helpers. The preamble helpers strategy forced them to forecast the system states which are the most difficult to reach.

3.2.4 Feedback from the students

Although LTG is user-friendly, beginners may experience problems for setting the parameters of the test campaign. For instance, setting the accurate model coverage criteria requires to find the trade-off between the number of tests, their relevance

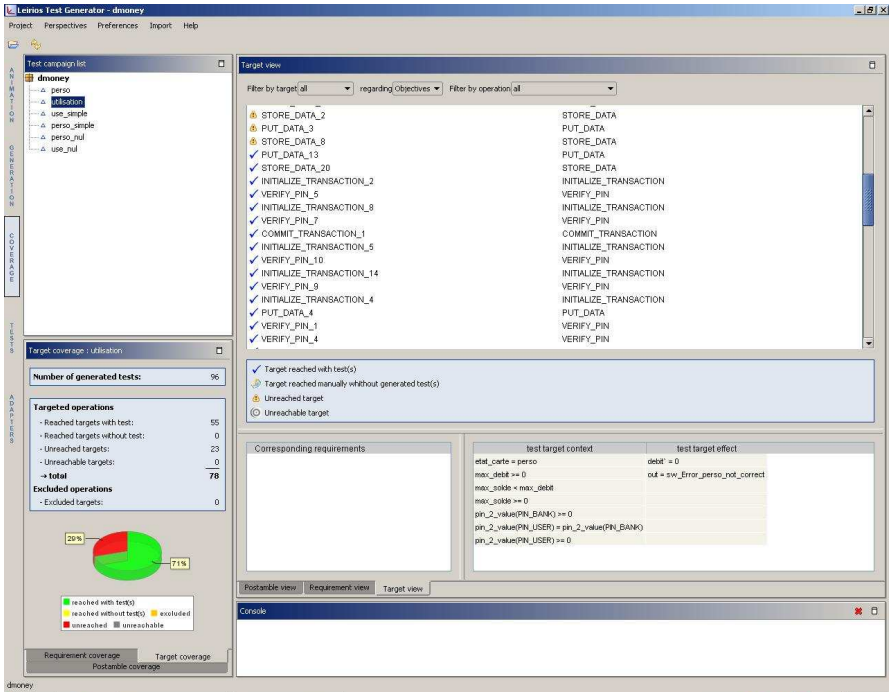


Fig. 2. A screenshot of the Leirios Test Generator GUI

and the capabilities of the test generator. The other main problems for the students are the addition of system observations after the tests and the configuration of the heuristics applied to reach the test targets.

4 An (Supposedly) Interesting Project

Before going into the detail of the project assigned to the students, let us first introduce the notions that we want to illustrate with the project.

4.1 Promoting Interest

First, we have to find a theme that will catch the students' interest. Being involved in research projects in embedded software, and especially smart cards, we have noticed that these represent a great challenge in the software industry, and one of the only domains in which significant efforts are made to use formal methods. We plan to make students model a system of a banking application, namely an electronic purse. In order to make it sound real, we use a simplified version of the Demoney applet [MM], a demonstrative electronic purse, that describes a realistic behaviour of smart cards, and contains enough details to be quite difficult.

Then, we have to define the goal of the project, that will guarantee that the students spend time on their models. Our first thought was to make them successively refine their models and generate an implementation. Nevertheless, this does not seem to be very motivating: smart card applets do not necessary require a refinement approach to be modelled, and the time available for the project would have been a bit too short.

4.2 Maintaining Interest

One of the greatest difficulty in writing a formal model is to know what degree of precision has to be put in it. This fact is a crucial point that has to be well understood by the students. In general, the accuracy and precision of the model is highly related to the degree of motivation shown by the students, including various external factors, e.g. the time they devote to the project.

Our objective is therefore to ensure that students will be putting efforts writing an efficient model that is both complete and coherent. For this purpose, we propose a new approach which consists in confronting their model to a real world application. This constitutes a validation challenge that requires them to write the most accurate possible model. To do so, an informal specification is given to them. This latter is based on the original specification of Demoney, and thus, it teaches the students to decode such documents.

Following a model-based testing approach [Bei95], our idea is to ask the students to build a model that they first have to prove, in order to ensure its coherence, and from which they have to derive test cases. This exercice is already practiced in the practical work sessions, during which the tests are run on a correct implementation that is used to help them making sure that their model is correct. Once this step is done, they are asked to run their test suites on several mutations of the original implementation, in order to check the accuracy of their model.

In our context, the task is different. The correct implementation is given to them, along with the mutants, but without any identification of the correct and faulty implementations. The challenge for the students is to find, among all the implementations, the one that is correct and explain why the others are wrong, ie. which part of the informal specification has not been respected by the implementation, based on the analysis of their tests.

In order to avoid copying out results on mutant detection, each team recieves a different archive, that contains a "personalized" set of 20 implementations. In order to increase the challenge, each team is given a different set of mutants and a diffferent file name relating to the correct implementation.

5 Case Study

The study case we propose to the student is based on the specification of *Demoney* –Demonstrative Electronic Purse– developed by *Trusted Logics* for research applications. No implementation of this specification is used in the real world, but it is closed to credit card applets and therefore pertinent. Demoney is a good example of a critical system, a banking transaction, which requires extensive validation and verification, simple enough to be used by students.

Our specification describes a card with four life cycle states: a personalization state, a use state, an blocked state and a disabled state. The card is secured with two PIN objects –user and bank– which are associated to retry counters values. Moreover, the card contains information e.g. the current balance, the maximal balance and the maximal debit.

During the *personalization* phase, the PUT_DATA operation makes it possible to set the informations and the objects of the card. Then, the STORE_DATA operation terminates this phase if and only if all the informations are setup and coherent with the security policy. Once the card has quit the personalization phase and started a use phase, it will never come back to the former state.

During the *use* phase, the card holder can invoke some commands to gain authentication on PIN –VERIFY_PIN– and to initialize a financial transaction –INITIALIZE_TRANSACTION– that has to be completed –COMMIT_TRANSACTION. If the user fails so much times to be authenticated that the retry counter of the user PIN reached zero then the card become *bloked*.

When the card is *blocked*, the authenticated bank can unblock the user PIN and change its value. But if the card is blocked and the retry counter of the bank PIN reaches zero then the card become *disabled* and will never be used. In addition, the implementation presents observation operations to retrieve the current balance, the maximal balance, the maximal debit and the state of authentication of the PINs.

The interest of this specification is to provide some behaviours which require a non-trivial command chaining to be reached. It forces the students to target these behaviours and help the test generation engine. This specification raises the notion of life cycle which is important in security testing. Moreover, the specification contains important properties of the system. Some of these properties can be expressed as invariant properties, whereas some others, e.g. operations chaining properties or more generally temporal properties, must be respected by operations

but can not be formalized as invariant properties. Thus, the students have to select what they are able to model as invariant properties or not. For instance, an invariant property over the system is: "If the card is not in the personalization phase then the balance can not be negative or higher than the maximal balance". On the contrary, the property: "All initialization of a transaction must be immediately followed by a transaction confirmation, otherwise the transaction is cancelled", can not be formalized as an invariant property.

6 Results Obtained

6.1 Evaluation of the students' work

In order to evaluate the students' work, we have defined scoring criteria based on the modelling choices, the understanding of the proof obligations and the coverage of tests. In order to evaluate the modelling choices as well as the understanding of the proof obligations, we ask the student to write a report. This latter includes explanations about the data strutures of the model, the behaviours of the operations and the invariant properties they define in order to ensure the conformance with the specification.

In order to take into account the coverage of the model and the quality of generated tests –number of tests, behaviour coverage and observations– we have based a part of the evaluation on the detection of mutants. Mutants are classified into three sets: the easily detectable ones with basic behaviours coverage criteria, those detectable with observations, and those that can only be detected with improved observations and behaviours coverage criteria selection. This simplifies the evaluation of the work: the harder the mutants are to be killed, the best is the students' score.

6.2 Feedback on the project

First of all the idea of a project appears to be a good idea. Since most of the students are only motivated by graduating, the aim is to make them learn without having noticed it. One interesting point is that the informal specification of the system they have to model is very dense, and thus, they are forced to conscienciously read the document in order to extract the pertinent informations.

The fact that the project is inspired from a real world application is also a good point, since they can really see how formal methods can be applied in the industry.

We believe this is very important that the students' first contact with formal methods links with concrete application. Thus, they understand better the usefulness of writing a model. In the years after, it will then be possible to teach them more abstract notion, that can not necessarily be concretized.

7 Conclusion and Future Courses

We have presented in this paper an approach for motivating students to learn formal methods. Our idea is based on showing the concrete application of formal methods instead of teaching it at a very abstract level, and without any link to real-life

concretization. Our solution is to use a playful and practical approach, that makes the students write a model for something interesting. We use an academic tool, named B4free, to perform the proofs of the B models we consider. We also rely on a commercial tool, named LTG, to provide a complete framework for the animation of the model. These tools are further used in a project inspired from a real world application of the formal methods in the Smart Card industry.

Since LTG is a commercial tool, universities are supposed to buy licences to run the tool. Nevertheless, a free alternative can be used. The ProB tool can be used instead of LTG in order to perform the animation. In this case, its flash-based plug-in for animation can be employed to improve the attractiveness of the interface. Concerning the test generation part, recent evolutions of the ProB authors [SLB05] let us think that the same mechanisms as LTG can be recreated at a minimal cost.

We regret in this project that the time was too short for making the students use a refinement-based approach for writing the model given to LTG. Nevertheless, such a change in the course planning is currently under study. One other improvement for the project is to adapt our course to the Event-B formalism and especially the Rodin plateform [Abr06] which appears to be the best current support for the Event-B systems. We are also looking forward to use some JML annotations [BCC+05], in order to improve the test verdict. Moreover, JML presents several possibilities for modelling programs with a few efforts. Finally, the use of real cards onto which the program is embedded, in order to increase the realism of the project, is currently under study.

References

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic, August 2002. INRIA Technical Report.

[Abr96] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[Abr06] Jean-Raymond Abrial. Tools for developing large systems (a proposal). In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, volume 4157 of *Lecture Notes in Computer Science*, pages 387–390. Springer, 2006.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[Bei95] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[Cle] ClearSy. B4free web site. http://www.b4free.com.

[Cle04] ClearSy. B reference manual v.1.8.5. 2004.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated Test Generation from B Models. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 277–281, Besançon, France, January 2007. Springer.

[LB03] M. Leuschel and M. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[MM] R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse - card specification.

[SLB05] Manoranjan Satpathy, Michael Leuschel, and Michael Butler. ProTest: An Automatic Test Environment for B Specifications. *Electronic Notes in Theroretical Computer Science*, (111):113–136, January 2005.

Generic algorithm patterns

Tibor Gregorics[1] Sándor Sike[2,3]

Department of Software Technology and Methodology, Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary

Abstract

Developing software using formal, mathematical approach has the advantage that the correctness of the solution can be formally checked, but the program construction is very time consuming and complicated process, even if software tools are used. If the programs for a relatively small set of generic problems are created such way, the solutions can be used to solve a particular problem by analogous programming. This method speeds up and simplifies the formal process and guarantees the correctness of the solution. It is also well suited for object oriented design, where the structure and overall functionality of the system can be specified, but the functionality of the methods are undefined, and with analogous programming these methods can be specified and correct programs can be created for them. The students are able to learn the technique of analogous programming relatively easily, and this process can be supported by automatic program generation or using template libraries.

Keywords: algorithm pattern, analogous programming, generic template, object oriented design.

1 Introduction

The software crises made it clear that ad hoc program development techniques are inappropriate to solve large problems, because softwares created that way did not meet the requirements and the cost and deadline became unmanageable [11].

The first attempt to overcome the software crises was the introduction of formal software development, where mathematical concepts and tools are used to specify a problem and create a correct program for it [4], [7], [8]. This approach has the advantage that correct programs can be developed and the correctness of existing programs can also be checked. One of these methodology invents the solution by composing it from available operations and proving its correctness [3]. This way a program can be created for most of the problems. However, designing a solution from scratch demands creativity, and hence time. The main disadvantage of these formal approaches that the specification and the abstract solution of a large problem requires too much effort and high level of mathematical skills from the developers.

[1] Email: gt@inf.elte.hu
[2] Email: sike@inf.elte.hu
[3] This work was supported by grant GVOP-3.2.2.-2004-07-0005/3.0

Without software support this formal development process did not provide the solution for the crises [4]. The formal approach, however, is well suited for solving smaller (safety critical) subproblems of a large system.

Later object oriented software development has been introduced and hoped to be the solution for creating large systems [9], [10]. This is a semi formal, diagram driven approach with standardized notation called UML [1]. This method has also its limitations, but the UML and the wide range supporting tools has made this approach probably the most widely used software development model. In object oriented development the most critical phase is to create a proper model or design. This process is supported by different type of patterns [5], [6], that guarantees the quality of the model created. Software tools are capable to translate an object oriented design to a programming language, however the resulting program is usually not complete, the functionality of methods in the classes are missing. These methods should be implemented later by a programmer [5].

The missing functionality of the methods shows that object oriented development still requires "traditional procedural" programming knowledge. Experience shows that if only object oriented design is learned by students, they will not be able to functionally complete the code created. The functionality of these methods usually is relatively simple, since the object oriented model determines the structure and main functionality of the system, thus formal specification and program creation can be applied at this point. However, this still slows down the development process and its mathematical background is a severe drawback force for most of the students.

Another method which still guarantees the correctness of the resulting program is based on the idea of relating the problem to a similar problem whose solution is already known. The program for the actual problem can be constructed mechanically using the analogy. We call this method *analogous programming*. Twenty years of teaching experience shows that students can solve more problems with analogous programming than create programs using traditional formal method.

The basis of this method is the concept of *algorithm pattern*. Such a pattern is obtained from solving a general problem, proving the correctness of the solution, and includes the specification of the problem and the description of the appropriate program. Whenever the actual problem is analogous to a general problem of an algorithm pattern in terms of substitutions, the program can be derived by applying the same substitutions to the program of the pattern.

When analogous programming is used to develop a program then, instead of creating a new solution from scratch and checking its correctness, only the analogy between the actual problem and the algorithm pattern has to be established in terms of substitutions called mapping, and verify that the mapping really holds. Applying the same mapping to the program is a straightforward task, that can be automated [2] or supported by generic templates.

Analogous programming can be used since most of the small programming prob-

[4] In the last decade software tools and systems has been developed making formal approaches practically available, but the use of these tools still requires strong mathematical background, that limits their widespread usage.

[5] The tools differ in this aspect. If additional knowledge can be specified by annotation or extending UML, then automatic (partial) implementation of these methods is possible.

lems can be seen as a special case of a general task, such as summation, searching for an element satisfying a condition, or finding a maximum. In addition more complex problems can be composed from such simple general tasks.

Next, the technique of analogous programming is described shortly, then the use of generic templates for implementing algorithm patterns is demonstrated.

2 Analogous programming

In formal development methods programs are composed from available elementary operation. The first step of this process is to create the formal specification of the problem to be solved by describing the input and the output and the relation between them. Variables and type value sets are used for data description, and logical functions, called precondition and post condition, determine the relation. The precondition contains the constraint (if any) the input must satisfy.

A calculus is needed to check if an elementary operation solves a problem described by a precondition post condition pair. Programming constructs are used to build program from elementary operations. The programming constructs allowed for program creation are limited to sequence, alternate and loop. Rules expressed by preconditions and post conditions are used for proving the correctness of the applied construction, assuming its components are correct. These rules are used to decompose the specification of the problems into smaller problems until an elementary operation is appropriate to solve the resulting subproblem.

This technique is illustrated for the next problem. Let us assume that $[m..n]$ is a subinterval of the integer numbers, $\mathbb{Z}$, and $f : [m..n] \to H$ function is given, where H is an arbitrary set with an adding operation, $+ : H \times H \to H$, and 0 is a neutral element of this operation. Our task is to calculate the sum of the values of function f on interval $[m..n]$, i.e., the value of $\sum_{k=m}^{n} f(k)$ expression.

The state space of the specification, A, contains the data relevant to the problem with their types. The precondition Q expresses that the endpoints of the interval are fixed, and the post condition R determines the relation between the input and the output.

$$A = (m : \mathbb{Z}, n : \mathbb{Z}, s : H)$$
$$Q \ : \ m = m' \wedge n = n' \quad \text{(where } m', n' \text{ are arbitrary fixed initial values)}$$
$$R \ : \ Q \wedge s = \sum_{k=m}^{n} f(k)$$

Sequence construct is used to create the program. The first component is an assignment the second component is a loop. The condition between the two component is Q' and the invariant of the loop is P. For the loop a new integer variable i is introduced to the state space to mark the end of the subinterval for which the sum is already calculated [6].

$$Q' : s = 0 \wedge i = m$$

[6] If $n + 1 < m$, then the interval in P is empty, but i must belong to a set, that is the reason for adding m to the interval in P.

$$P : Q \wedge i \in [m..n+1] \cup \{m\} \wedge s = \sum_{k=m}^{i-1} f(k)$$

It is obvious that if Q holds, then Q' will be also satisfied after executing the $s, i := 0, m$ assignment, that will be the first part of the sequence. $Q' \Rightarrow P$ and $P \wedge i > n \Rightarrow R$, thus the loop starts properly, and if $i \leq n$ is the loop condition then it ends correctly. $t = n - i + 1$ expression can be used as terminating function to ensure the termination of the loop. The value of this function can be decreased by $i := i + 1$ assignment. This is the second part of the sequence used as the kernel of the loop. For satisfying P invariant $s := s + f(i)$ is needed as the first part of the sequence. The result is the following program.

$s, i := 0, m;$
while $i \leq n$ **do**
    $s := s + f(i);$
    $i := i + 1;$
endwhile

Analogous programming is based on the idea that if the specification of a new problem is similar to a specification of a problem already solved, we expect that the program should be also similar. Let us assume that this similarity is expressed by mapping the variables and data types of the two problems. If the result of the mapping is that operations correspond each other, and further the preconditions and post conditions are identical, then the same decomposition will produce a correct program, if the operations are replaced according to the mapping. This approach is applicable even if the preconditions and post conditions are not identical after the mapping, but the precondition of the new problem implies the precondition of the problem solved, and the post condition of the problem solved implies the post condition of the new problem. This way the time consuming decomposition and formal checking process is skipped, only a mapping must be established between the specification of two problems, and the mapping must be applied to the existing program to create the solution for the new problem. We call this process analogous programming.

The application of analogous programming needs a set of problems already solved, that can be used in the mapping process. The size of this set should be limited to keep it manageable. This requires the use of general problems, where generosity can be achieved by using functions to define an aspect of a problem. The general problem program pairs are called algorithm patterns. A small set of algorithm patterns can be used to solve a huge number of frequent, small tasks in programming. Combining these patterns, i.e., decomposing tasks to the patterns, allows us to increase the size of problems to be solved this way.

The specification and the program of the previously solved problem in this section is called the algorithm pattern of summation, or shortly summation. Next analogous programming is demonstrated by using summation.

Example 2.1: Let us calculate the factorial of a given n natural number!

The specification of the problem is given next.

$A = (n : \mathbb{N}, s : \mathbb{N})$
$Q\ :\ n = n'$
$R\ :\ Q \wedge s = \prod_{k=1}^{n} k$

After comparing this specification to the specification of summation it can be seen that the problem is a special case of summation and the next table shows the mapping that establish the analogy between the current problem and summation.

summation	current problem
m	1
n	n
H	$\mathbb{N}$
f	id
0	1
$+$	$*$

Applying the mapping to the program of summation leads to the solution of our current problem.

$s, i := 1, 1;$
while $i \leq n$ **do**
    $s := s * i;$
    $i := i + 1;$
endwhile

Example 2.2: Decide whether all the values of an $f : \mathbb{Z} \to \mathbb{Z}$ function are even on an $[m..n]$ interval!

The specification of the problem and the mapping to summation:

$A = (m : \mathbb{Z}, n : \mathbb{Z}, s : \mathbb{L})$
$Q\ :\ m = m' \wedge n = n'$
$R\ :\ Q \wedge s = \bigwedge_{k=m}^{n} (2|f(k))$

summation	current problem
m	m
n	n
H	$\mathbb{L}$
$f(k)$	$2\|f(k)$
0	$true$
$+$	$\wedge$

The program derived from summation:

$s, i := true, m;$
while $i \leq n$ **do**
    $s := s \wedge (2|f(i));$
    $i := i + 1;$
endwhile

The algorithm pattern of summation can be further generalized if the elements of an integer interval is replaced by values provided by an iterator, since in the pattern the elements of the interval are accessed sequentially, they are iterated. The specification and the program of this new version is shown next.

$A = (h : seq(E), s : H)$
$Q : h = h'$
$R : Q \wedge s = \sum_{k \in h} f(k)$
where: $f : E \to H$

$s := 0;\ h.first();$
while $\neg h.end()$ **do**
    $s := s + f(h.current());$
    $h.next();$
endwhile

Example 2.3: Collect the even values from an integer sequence to a set. The specification, the mapping and the program is given next [7].

$A = (x : seq(\mathbb{Z}), s : 2^{\mathbb{Z}})$
$Q : x = x'$
$R : Q \wedge s = \bigcup_{k \in x} g(k)$
$g : \mathbb{Z} \to 2^{\mathbb{Z}}$
$g(i) = \begin{cases} \{i\}, \text{ if } 2|i \\ \emptyset, \quad \text{else.} \end{cases}$

summation	current problem
E	$\mathbb{Z}$
H	$2^{\mathbb{Z}}$
f	g
0	$\emptyset$
$+$	$\cup$

$s := \emptyset;\ x.first();$
while $\neg x.end()$ **do**
    if $2|x.current()$ **then** $s := s \cup \{x.current()\};$ **endif**
    $x.next();$
endwhile

As the examples presented show, an algorithm pattern can be used for the specification of a problem, if the pattern is named and the mapping is given. In fact, frequently we see a problem as a special case of a pattern, and only a mapping should be established. This allows us to omit the formal specification of the problem and begin the program creation with establishing the mapping when analogous programming is used.

If the mapping is given, the remaining task of program development can be automated [2] or supported by software tools. In the next section we show the pos-

[7] Adding $\emptyset$ to the result needs no computation, thus this branch of function g realized by $SKIP$.

sible use of generic templates in C++ to support this process. The benefit of these templates is that the (novice) programmer can focus only to the implementation of the mapping, thus becoming familiar with the use of algorithm patterns. We are not saying that this is the optimal way of solution in most of the cases. Direct implementation results in simpler code, but the emphasis here is on the education, to practice the application of algorithm patterns.

3 Using generic templates for algorithm patterns

The possible use of templates to support the implementation of algorithm patterns is illustrated for the summation pattern. This pattern uses an iterator object to access the elements for the function whose values must be summed. Two abstract template classes are used in the C++ implementation. The first defines the interface of the iterator objects used in summation (Fig.1), the second implements the algorithm of summation leaving the realization of two functions to the derived concrete class (Fig.2). These two functions realize the initial assignment of the neutral element and the adding operation.

The use of summation pattern requires two classes: a concrete iterator and the realization of the two abstract method of summation (Fig.3). Sequential file will be used in the examples of this section, therefore we create an template iterator class that can iterate elements with >> operators defined (Fig.4). The possible use is demonstrated on two examples.

Example 3.1: Count how many words begins with 'a' in a text file!

```
template <class Item>
class Iterated
{
   public:
      virtual ~Iterated() {}
      virtual void first() = 0;
      virtual void next() = 0;
      virtual bool end() const = 0;
      virtual Item current() const = 0;
   protected:
      Iterated()          {}
};
```

Fig. 1. The interface for iterator objects

```
template <class Item, class ResultType>
class Summation
{
   public:
      virtual ~Summation() {}
      ResultType  Result() const { return sum; }
      void  Do();
   protected:
      Iterated<Item>* it;
      ResultType      sum;
      Summation() {}
      virtual void Null() = 0;
      virtual void Add(const Item& e) = 0;
};

template <class Item, class ResultType>
void Summation<Item, ResultType>::Do()
{
   Null();
   for ( it->first(); !it->end();  it->next() )
      Add(it->current());
}
```

Fig. 2. The abstract implementation of the summation pattern

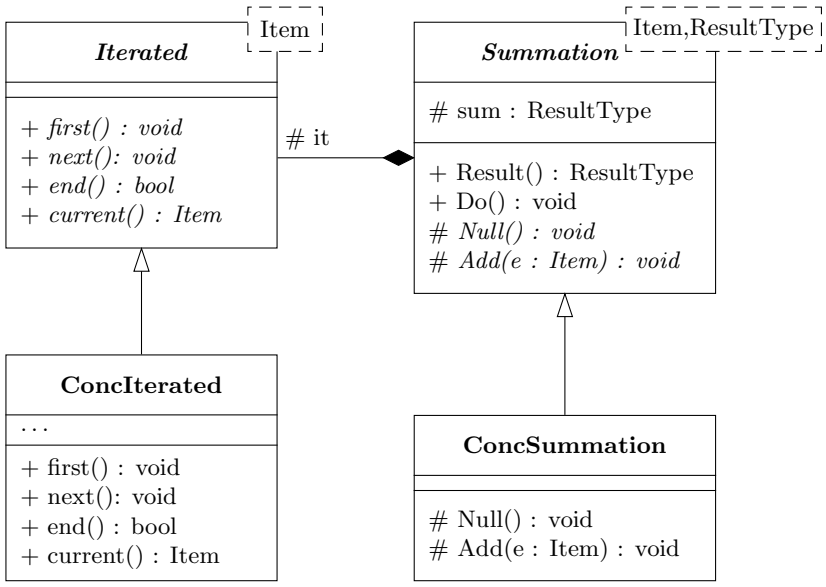


Fig. 3. The structure of the use of the summation pattern

```
template <class Item>
class SeqInFile : public Iterated<Item>
{
   protected:
      ifstream f;
      Item     df;
      string   fname;
   public:
       SeqInFile(const string& s) : fname(s)  { f.open(fname.c_str()); }
       ~SeqInFile()          { f.close(); }
       void first()          { f >> df; }
       void next()           { f >> df; }
       bool end()      const { return f.eof(); }
       Item current() const { return df; }
};
```

Fig. 4. Template class to iterate the elements of a sequential file

Words can be seen as strings in C++, and the value of function to be summed is 1, if the first character is 'a', 0 otherwise, i.e.:

$$g : string \to \mathbb{Z}$$

$$g(s) = \begin{cases} 1, \text{ if } s_0 =' a' \\ 0, \text{ else.} \end{cases}$$

Using the following mapping we get the program of Fig.5.

summation	current problem
E	$string$
H	$\mathbb{Z}$
f	g
0	0
$+$	$+$

```
class AChar : public Summation<string, int>
{
   public:
      AChar(const string& str)
      {
         it  = new SeqInFile<string>(str);
      }
   protected:
      void Null() { sum = 0; }
      void Add(const string& e) { if ( e[0] == 'a' ) sum++; }
};

...

   AChar s("input.txt");
   s.Do();
   cout << s.Result() << endl;
```

Fig. 5. Counting the number of words beginning with 'a'

Example 3.2: Collect the words longer than k character from a text file to another text file!

Strings must be collected in this example, thus the result is sequence. The neutral element is the empty sequence (<>), and the adding operation is concatenating a string to the sequence (<< operator). The correspondence between the problem and the pattern is given next.

$$g : string \to seq(string)$$

$$g(s) = \begin{cases} < s >, \text{ if } |s| > k \\ <>, \quad \text{else.} \end{cases}$$

summation	current problem
E	$string$
H	$seq(string)$
f	g
0	$<>$
$+$	concatenation

Adding the empty sequence to the sum needs no activity, therefore only the first case in g must be implemented when summation is realized. The solution is shown in Fig.6.

```
class KWord : public Summation<string, ofstream*>
{
   public:
      KWord(const string& inp, int k, const string& out)
      {
         it = new SeqInFile<string>(inp);
         this->k = k;
         sum = new ofstream(out.c_str());
      }
      ofstream*  Result() const { sum->close(); return sum; }
   protected:
      void Null() {  }
      void Add(const string& e)
      {
         if ( e.length() > k )  (*sum) << e << endl;
      }
   private:
      int   k;
};

...

   int   k = ...;
   KWord words("input.txt", k, "output.txt");
   words.Do();
```

Fig. 6. Collecting words longer than k

4 Conclusion

In this paper we have shown that analogous programming can be used in numerous practical cases to simplify and replace the time consuming, complicated formal software development process so that the correctness of the resulting program is still guaranteed. One of the simplest analogous pattern, summation, has been used to illustrate that wide range of problems can be solved by finding analogy between a generally specified problem and the concrete problem. The examples shown demonstrate that the set of problems to be solved can be extended, if not only trivial mapping are concerned, but substitutions with different type value sets and operations are allowed.

A small set of similar algorithm patterns makes possible to solve a huge set of small practical problems with analogous programming. These problems can represent the functionality of methods of classes in an object oriented design or simple services of components. Examples for general algorithm patterns are summation, linear searching, maximum selection and elementwise processing.

It has been also illustrated that algorithm patterns can be used to specify a problem by defining the mapping between the general and actual problem. In fact, this mapping is sufficient to derive the correct code for the problem either by using specification macros and generate the code automatically, or by using general templates where only the mapping must be realized when the concrete class is specialized. Both method are adequate to support the education of analogous patterns.

We can conclude that analogous programming, which has a formal mathematical background but can be used without deep mathematical knowledge, can be a useful, practical technique in software development or programming methodology, since it covers that field (realization of the functionality of methods and services of components) with simple tools, that are unsupported or weakly supported by recent technologies.

References

[1] Booch G., Rumbaugh J., Jakobson I., "The Unified Modeling Language User Guide (Second Edition)", Addison-Wesley, 2005.

[2] Csepregi Sz., Dezső A., Gregorics T., Sike S., *Automated Implementation of Service Required by Components* , PROVECS 2007, http://lina.atlanstic.net/provecs/2007/provecs2007proceedings.pdf, 2007.

[3] Dijkstra E.W., "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, 1973.

[4] Dijkstra E.W. and C.S. Scholten, "Predicate Calculus and Program Semantics", Springer-Verlag, 1989.

[5] Fowler M., "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997.

[6] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley Longman, 1995.

[7] Gries D., "The Science of Programming", Springer Verlag, Berlin, 1981.

[8] Hoare C.A., *Proof of Correctness of Data Representations*, Acta Informatica **1** (1972), 271-281.

[9] Mayer B., "Object Oriented Software Construction", Prentice Hall, 1997.

[10] Rumbaugh J. et al., "Object Oriented Modeling and Design", Prentice Hall, 1991.

[11] Sommerville I., "Software Engineering", Addison-Wesley, 1983.

Formal Methods: Never Too Young to Start

J Paul Gibson[1]

Le département Logiciels-Réseaux (LOR),
Institut National des Télécommunications,
Evry, France

Abstract

In many countries around the world, there is a crisis in the teaching of mathematics and computer science. Governments have tried to address the problem by investing in computers in schools; when they should have invested in teaching computer science in schools. Formal methods bridge the boundary between computing and mathematics in a natural way. Through our experience of teaching algorithmic thinking in schools, young children have been observed using concepts such as refinement, proof, abstraction, complexity, non-determinism, equivalence, etc... in their own reasoning about problems. We argue that this ability needs to be better leveraged in order to improve both the teaching of mathematics but also to improve childrens' understanding of computer science as a discipline in its own right. We give concrete examples of the type of formal methods teaching that succeeds.

Keywords: Schools, Computer Science, Proof, Algorithms

1 Introduction: Why teach formal methods in schools?

The current state of mathematics teaching around the world is causing problems for the teaching of computer science. Children are taught "mathematics" at all levels of school education, yet computer science lecturers continue to struggle with undergraduate students whose mathematical abilities are poor. Furthermore, many of these students have been (mis)led in their schools to believe that computer science does not require mathematics. The recent introduction of computers into schools has not, in general, improved the situation: computers are mostly used as a tool to support the teaching of other subjects (including mathematics); whilst often giving a false impression that computer science is using a computer. Some schools, in an attempt to teach computer science, have introduced children to programming. We believe that this is a step forward, but it carries the risk that children think that programming is computer science, and that computer science is programming. We believe that the best way of introducing children to computer science does not

[1] Email: paul.gibson@int-evry.fr

require a computer: it requires the teaching of rigorous (formal) reasoning about computations and algorithms.

This paper reports on the teaching of formal methods to children as young as seven years old. It supports the well-established view that there are advantages in teaching mathematics constructively. For example Clements stated in 1990[4]: "Educational research offers compelling evidence that students learn mathematics well only when they construct their own mathematical understanding." The work reported is motivated by our own experiences in teaching computer science to young children: Java programming[8], the Computer Science Unplugged Tutorials[1], and problem based learning (PBL)[13,12]. It builds on research that suggests that formal software engineering concepts form the basis for how children learn to solve problems[9].

In "How to Solve It: A New Aspect of Mathematical Method", Polya[15] states: "A good teacher should understand and impress on his students the view that no problem whatever is completely exhausted." Our approach has been to take problems that one would normally see in university and to rework them for school children. The children are then encouraged to take the problems in whatever direction they wish, and as far as they want. In this way, young children quickly identify fundamental concepts in computer science.

This paper reports on some of the case studies with which we have had most success. We do not claim to have carried out a verifiable educational experiment: we report on observations that have been made over a number of years through interaction with hundreds of children (aged 7 to 18).

2 Learning Theory and Models

There are hundreds of well-published complementary, and competing, theories of learning. The highly cited review by Hilgard and Bower[11], published over half a century ago, is a good introduction to the foundations of learning theory. In this paper, we focus on the well-accepted theories that have had most influence on our own research into formal reasoning and problem solving.

Cognitive structure is the concept central to Piaget's theory. (See the work by Brainerd[2] for a good overview and analysis.) Piaget's most interesting experiments, with respect to the work presented in this paper, focused on the development of mathematical and logical concepts. His theory has guided teaching practice and curriculum design in primary (elementary) schools in the last few decades. However, his work predates the development of computer science as a discipline and it is therefore unsurprising that it does not make reference to any formal methods concepts. Piaget's theory is similar to other *constructivist* perspectives of learning (e.g., Bruner [3]), which model learning as an active process in which learners construct new concepts upon their current knowledge and previous experience.

Similarites can be seen between the constructivist view and the *theories of intelligence* such as proposed by Guildford's *structure of intellect* (SI) theory [10] and Gardner's *multiple intelligences*[7]. Typically, these theories structure the learning space in terms of skills, for example: reasoning and problem-solving, memory operations, decision-making, and language. These skills do not explicitly mention

computation and proof, but one can see that there are strong overlaps.

Piaget's ideas also influenced the work by Seymour Papert in the specific domain of computers and education[14]. Papert argues that children can understand concepts best when they are able to explain them algorithmicaly through writing computer programs. We believe that they learn better when working at higher levels of abstraction, and that what they should really be doing is mathematical modelling.

Alan Schoenfeld argues that understanding and teaching mathematics should be treated as problem-solving [16]. He identifies four skills that are needed to be successful in mathematics: proposition and procedural knowledge, strategies and techniques for problem resolution, decisions about when and what knowledge and strategies to use, and a logical *world view* that motivates an individual's approach to solving a particular problem. In this respect, Schoenfeld has quite nicely identified fundamental aspects of formal methods in computer science and software engineering.

3 Learning Objectives: review of our sessions

In each of the sessions that we run in the schools, we are mindful that our goal is that the children learn fundamental concepts. Following the PBL philosophy, we do not explicitly teach the children about these concepts; we help the children to discover them through interaction with a specific problem. Each problem has the goal of the students discovering at least one of the following:

- *Proof* — the evidence or argument that compels one to accept an assertion as true.
- *Theorem* — a proposition that is true in all cases.
- *Conjecture* — an unproven proposition for which there is some sort of empirical evidence.
- *Constructive proof* — demonstrates the existence of a mathematical object with certain properties by giving a method (algorithm) for creating such an object.
- *Algorithm* — any procedure involving a series of steps that is used to find the solution to a specific problem.
- *A deterministic algorithm:* behaves predictably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states.
- *Correctness* — an algorithm can be proven to be correct with respect to a specification.
- *Refinement* — the verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable program. It guarantees the correctness of the program by construction.
- *Invariant* — an expression whose value does not change during algorithm execution; which can implement a required safety property in the specification
- *Computational Complexity* — the scaleability of algorithms with respect to the use of resources (typically time and space).

It is beyond the scope of this paper to analyse all of these learning objectives, and to demonstrate how our sessions help children to reach them. Rather, we give — in the next 3 sections — examples of sessions that we run with the youngest children (aged seven to nine). These illustrate how the formal methods concepts arise quite naturally out of the games that we play.

4 Parity: Algorithms, Verification and Proof

For this problem children need to be familiar with the concepts of even and odd numbers. They do not need to be able to add (although they think they do). We present the children with sums and ask if the answers are odd or even numbers. For example: *Is the answer to the sum 1 + 2 + 2 + 1 + 2 + 1 + 1 +1 + 1 + 2 +1 even or odd*? The children typically follow the algorithm:

```
add up the numbers
decide if the sum is odd or even (by looking at the last digit)
```

They then arrive at (hopefully) the answer 15 and then they say that 15 is odd (because 5 is odd). We then ask the children for more complicated sums and claim that we can perform the task faster than they can (even if they use a calculator): *1285 + 45362 + 12987 + 367235 + 12 + 887877 + ...* Of course, we have a trick, which we ask the students to try and work out.

We then bring in some (younger) students from a class that we have taught how to recognise odd and even numbers (represented as a string of digits). We give them a torch (flashlight). They then perform the following algorithm (after practising with us until we know they execute it correctly):

```
In the beginning the light (which can be switched) is off
For each number to be added:
 If light is off and the number odd  then switch
  else if light is on and the number even  then switch
   else do not switch
When no more numbers left
  If light is off then answer is even else the answer is odd.
```

The older children are amazed that the younger children — who cannot count (very well) — are quicker than them at doing the sum. Of course, they never really do the sum . . . this is the trick that we want the older children to discover. We then ask the younger children to explain the algorithm to the older children. The younger children, in general, know the algorithm but do not understand what it is doing. The older children do not know the algorithm but know what it is supposed to do. Typically, they question the correctness of the algorithm by asking if the trick will always work:

"They did it once (for a very difficult example) so it must work", or
"Ask them to do it a few more times to make sure they are not just guessing", or
"I will not ever be convinced until I know exactly what they are doing"

In order to convince the sceptical students we ask them to consider the following table:

```
Odd + Odd = Even          Even + Even = Even
Odd + Even = Odd          Even + Odd = Odd
```

The students are then asked to construct a table for the addition of three numbers, and to see if they can use the two tables to find out something that may help us how to understand whether the algorithm works (is correct). Typically, they identify the associativity and commutativity of addition (over odds and evens). At this point we introduce some physical elements (toys) in order to help them play with the problem. In this case, we used different coloured bricks that can stick together. We demonstrate the addition of the numbers 1 and 2, as shown in figure 1.

We ask the students to demonstrate/prove the elements of the addition table, for example: `Odd + Odd = Even`. In general, we see the sort of "proof" as illustrated in figure 2. Whether the physical manipulation of the sticky bricks constitutes a

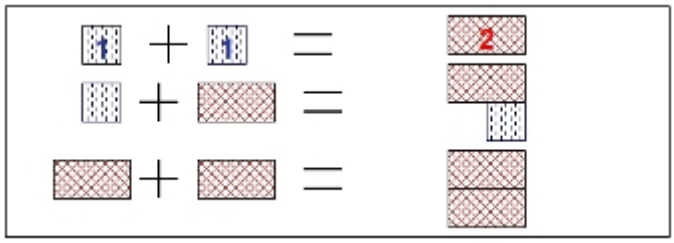


Fig. 1. Addition using Sticky Coloured Bricks

proof is open to question. What should not be questioned is that children are able to demonstrate why the parity addition trick works.

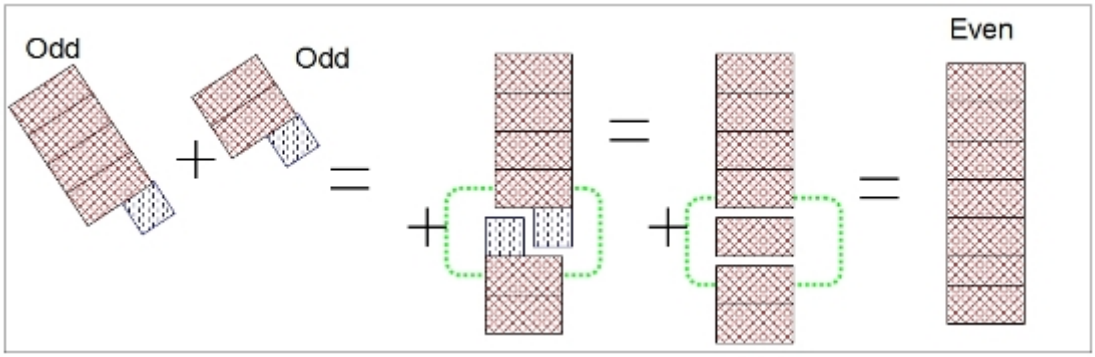


Fig. 2. Parity "Proof"

5 Primes: Algorithms, Correctness and Complexity

The goal of this exercise is to test whether the students can reason about the computational complexity of an algorithm. Furthermore, we would like them to see that checking the answer of a computation is often simpler than finding the correct answer. We use the classic problem of testing whether an integer is prime. The children do not need to know how to multiply or divide but they do need to know about rectangles. We use sweets (wrapped in paper) for the concrete representation of numbers (in unary notation). In figure 3, we see how the children check whether the numbers 7 and 6 are prime.

As a game, we provide a larger number of sweets and ask the children to race against each other (often in teams) in order to construct a rectangle. The winning team gets to eat their sweets. As a challenge, we give them 17 sweets and see that they get quite frustrated. We explain that when they cannot make a rectangle then the number of sweets is said to be prime. When playing the game, some of the children do not trust us when we say that another team has found the rectangle before they have. Consequently, we tell them the width and height (multiples) and they see immediately that they are quicker at checking that the answer is correct than at trying to find the answer themselves. After playing this game, the older children identify classic algorithms for primality and even manage to execute them in parallel using different players in their teams to check different multiples. In order

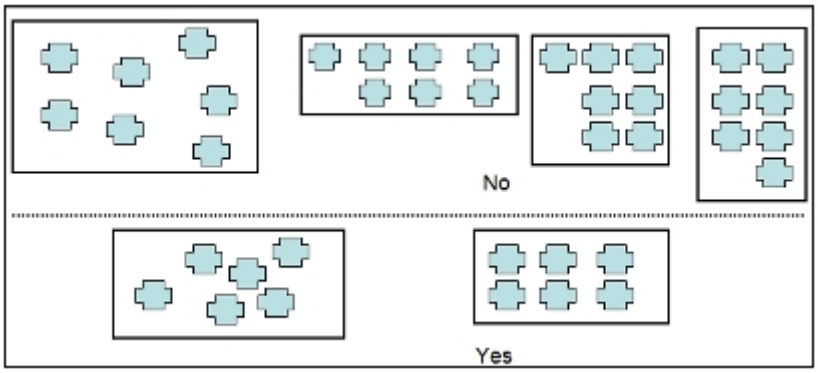


Fig. 3. Primes Algorithm: Can we make a rectangle?

to win the sweets, the children try to speed up their algorithms. In general, they are too young to recognise that they do not need to check all rectangles (and that they can stop at the "square in the middle").

During this game there are a number of surprising things that can occur. The most interesting remark that we had was from an 8-year old who — on seeing that the number was prime — asked if he could have another sweet.

6 Searching and Sorting: Refinement

6.1 Searching and data refinement

In searching, we initially require only that a child can match a single piece of string with another piece of string in a collection. We demonstrate that we can hide a piece of string in a box, and place a number of pieces of string in a number of boxes (one per box). Finally, we hand them a piece of string and ask them to find the matching string in one of the boxes. However, they are told that they can open only one box at a time. Again, we play the children against each other, making alternate moves of a game. In this game, a move is looking in a box for the matching string. The first player to match the string wins the game. All other children act as spectators of each game; and observing the spectators is as insightful as observing the players.

We first observe the children selecting boxes in a random manner. The first interesting observation is when children realise that they have a better chance of winning if they never look in a box that they have already looked in. This observation usually arises from one of the child spectators shouting out that a player has already looked in a particular box and that they should choose another. In terms of software engineering, the children have quite naturally identified and communicated a process refinement At this point, we ask children to play against each other using the new, improved approach. However, we preclude the spectators from speaking during a game. Very quickly, it is observed that some of the children have problems remembering in which boxes they have already looked.

In the searching example, we have observed 3 types of data refinement which the children adopted as a specific way of overcoming the problem of having to remember which boxes had already been examined:

- Children searched the boxes in an ordered fashion (left to right, e.g.) and so

had to remember only the last box searched.
- Children marked the boxes already searched (using a pencil, e.g.).
- Children moved the boxes examined into an *already examined pile*.

In the next phase, before we ask the children to play the game, we order the boxes based on the length of the strings within. Very quickly it is observed that not all the children realise that the strings in the boxes are ordered by length. The children who realise this are then observed playing in a more structured manner. Over a period of time, we observe that the children effectively refine their solution to a binary search where they do not always optimise the search by cutting the search space perfectly in two every time they make a guess. They know they need to look to the left or right of the current string box, based on the relative sizes of the search string and the string in the box.

The children are asked if it is possible to play *better*? Often they make quite solid arguments as to why their solution is optimal. At this stage, we play against them and we always find the string that is being looked for in the first guess. Without them knowing, we employ a perfect hashing function to map the string length directly to a particular box. They often accuse us of cheating; and only the more advanced students realise the trick.

6.2 Example: Sorting and process refinement

We propose the following algorithm for sorting a list of elements:

(i) Take the input list I and copy into list O.

(ii) If O is **ordered** then return this as output,
else swap two randomly chosen elements of O.

(iii) Goto step (ii)

We can specify this algorithm as a non-deterministic finite state automaton (NFSA). In the left hand side of figure 4, we illustrate sorting a list of three elements. The states are labelled with the list of integer elements. The transitions (all non-deterministic) are labelled by arcs and correspond to random swaps. Note: we have not represented the null transitions (where an element is swapped by itself) in the NFSA. The terminating state — where the list elements are ordered — is shaded.

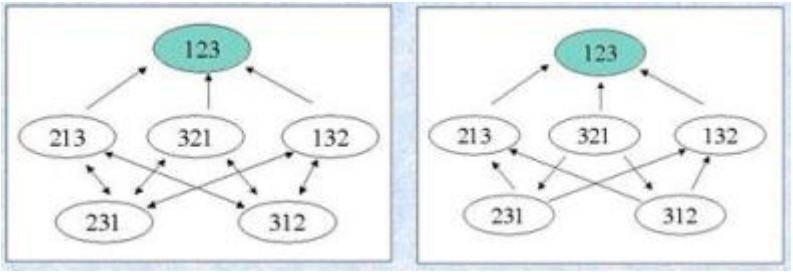


Fig. 4. *Left:* Random sorting of a 3-element list. *Right:* a sorting process refinement

We can see that from any starting state it is always possible that the terminating state will be reached, but that we may move in circles in an inefficient manner.

Refinements that remove some of the non-determinism of the system (solution) can be used to generate a more efficient solution. Consider the first refinement, in the right of figure 4, where we have removed all swap transitions that exchange elements that are already in order. The removal of non-determinism, in this case, has transformed a *correct* solution into a better, more efficient, *correct* solution.

In order to see if the children can reason about refinement and correctness, we present the sorting problem using coloured balls (to be ordered as seen in the rainbow) hidden in shoe boxes. This is illustrated in figure 5. Through playing

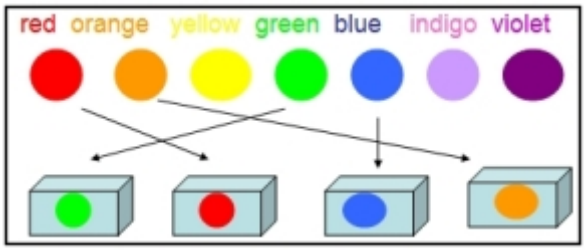


Fig. 5. Sorting with coloured balls

games, children naturally refine the inefficient solution and discover classic sorting algorithms in the process.

7 Some Interesting Observations

This paper is not reporting on a formal pedagogic experiment. We have made detailed notes concerning our sessions with the children and offer two interesting observations. With respect to algorithms, we were interested to note that, in general, a subset of students lose interest in a game as soon as the trick is explained by an algorithm. In early years (children as young as 7-years old) the percentage of children concerned is usually between 25 and 30. However, in later years (children in their late teens) this percentage usually grows to more than half the class. With respect to correctness, the yonger children are less interested in verifying their algorithms than the older children. In order to promote verification, we present the children with algorithms that are almost correct (but don't always work). Younger children appear to be disappointed in the fact that the algorithms can be broken; whilst the older (interested) children see that as being the real challenge.

8 Conclusions

We have demonstrated that it is possible to teach young children formal methods concepts through games and problem based learning. Children who are disinterested in mathematics regain an interest when they see that mathematical modelling can help them reason about algorithms and games. We do not make any claims about whether our sessions improve the childrens' mathematical ability. However, we do believe that this type of session is a good way of introducing computer science in schools. (We have also demonstrated that this problem-based learning approach can be used to teach university students[12] about computer science.)

As formal methods are foundational to computer science, it should not be any surprise that the rigorous, mathematical, analysis of algorithms and computations should be a major part of teaching computer science to children. This is not a new idea — it was discussed at the TFM workshop in Ghent in 2004[6], where daRosa presented research into the teaching of recursive algorithms as part of a high school mathematics course[5]. We already know that computer science education has need of mathematics; perhaps now the mathematics teachers can be persuaded to consider computer science (formal methods) as a good way of teaching mathematics.

References

[1] Bell, T., *A low-cost high-impact computer science show for family audiences*, 23rd Australasian Computer Science Conference **00** (2000), pp. 10–16.

[2] Brainerd., C., "Piaget's Theory of Intelligence," Prentice Hall, Englewood Cliffs, NJ, 1978.

[3] Bruner, J. S., "Toward a theory of instruction," Belknap Press of Harvard University, Cambridge, Mass,, 1966.

[4] Clements, D. H. and M. T. Battista, *Constructivist learning and teaching*, Arithmetic Teacher **38** (1982), pp. 34–35.

[5] da Rosa, S., *Designing algorithms in high school mathematics*, in: Dean and Boute [6], pp. 17–31.

[6] Dean, C. N. and R. T. Boute, editors, "Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium, November 18-19, 2004, Proceedings," Lecture Notes in Computer Science **3294**, Springer, 2004.

[7] Gardner, H., "Frames of mind: the theory of multiple intelligence," Basic Books, New York, 1983.

[8] Gibson, J. P., *A noughts and crosses java applet to teach programming to primary school children*, in: *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java* (2003), pp. 85–88.

[9] Gibson, J. P. and J. O'Kelly, *Software engineering as a model of understanding for learning and problem solving*, in: *ICER '05: Proceedings of the 2005 international workshop on Computing education research* (2005), pp. 87–97.

[10] Guilford., J. P., "The Nature of Human Intelligence," McGraw-Hill, New York, 1967.

[11] Hilgard, E. R. and G. H. Bower, "Theories of Learning," Prentice Hall, Englewood Cliffs, NJ, 1956.

[12] O'Kelly, J. and J. P. Gibson, *PBL: Year one analysis — interpretation and validation*, in: *PBL In Context — Bridging Work and Education*, 2005.

[13] O'Kelly, J. and J. P. Gibson, *Robocode and problem-based learning: a non-prescriptive approach to teaching programming*, in: R. Davoli, M. Goldweber and P. Salomoni, editors, *ITiCSE* (2006), pp. 217–221.

[14] Papert, S. and J. Sculley, "Mindstorms: children,computers, and powerful ideas," Basic Books, New York, 1980.

[15] Polya, G., "How to Solve It," Princeton University Press, 1971.

[16] Schoenfeld., A. H., "Mathematical Problem Solving," Academic Press, Orlando, Fla, 1985.

Structured Derivations: a Logic Based Approach to Teaching Mathematics

Ralph-Johan Back and Linda Mannila and Patrick Sibelius[1] ,[2] ,[3]

Department of IT, Åbo Akademi University, Turku, Finland

Mia Peltomäki[4]

Department of IT, University of Turku, Turku, Finland

Abstract

Being able to reason rigorously and comfortably in mathematics plays an essential role in computer science, particularly when working with formal methods. Unfortunately, the reasoning abilities of first year university students' are commonly rather poor due to lack of training in exact formalism and logic during prior education. In this paper we present *structured derivations*, a logic based approach to teaching mathematics, which promotes preciseness of expression and offers a systematic presentation of mathematical reasoning. The approach has been extensively evaluated at different levels of education with encouraging results, indicating that structured derivations provide many benefits both for students and teachers.

Keywords: Structured derivations, teaching mathematics, mathematics for formal methods

1 Introduction

Being able to reason rigorously and comfortably in mathematics is an essential prerequisite for studies in computer science (CS), especially when working with formal methods. Nevertheless, many CS students unfortunately show little understanding for and interest in mathematics in general and formal notation, logic and proofs in particular. For instance, Gries [10] notes that "students' reasoning abilities are poor, even after several math courses. Many students still fear math and notation, and the development of proofs remains a mystery to most." (p. 2) Almstrum [3] found that novice CS students experience more difficulty with the concepts of mathematical logic than with other CS concepts.

[1] Email: backrj@abo.fi
[2] Email: linda.mannila@abo.fi
[3] Email: patrick.sibelius@abo.fi
[4] Email: mia.peltomaki@utu.fi

One reason for students' low level of skills in formal reasoning and proofs can be traced back to their prior education: exact formalism and proof are perceived as difficult and consequently avoided at e.g. high school level (e.g. [5,12,13,18]). For instance in Finland, high school students are offered the choice of two different mathematics syllabi, including a total of 21 courses (sixteen compulsory, five elective) [15]. Despite the large number of courses, proof and formal reasoning is only mentioned in the learning objectives for one of them, the elective course "Logic and Number Theory" in the advanced syllabus. This is also the only course that introduces logical notation and truth values. How could we expect first year university students to expose high levels of proficiency in topics such as logic, exact formalisms and constructing proofs, when their only prior chance to study these topics is in one (elective) course throughout their entire general education? Proofs should be considered a way of thinking that can be applied to any mathematical topic, instead of being viewed as a distinct topic [11,18].

In addition to the lack of training in formal reasoning and proofs, studies have indicated problems in the way proofs are approached and presented in education. For instance, Dreyfus [9] claims that students often receive mixed messages. As an example he notes that many mathematical textbooks offer intuitive explanations in one solution, use examples to clarify another, and give a rigorous proof for yet another. The differences between these justifications are however not explicated, but leave students with three different views of what *could* constitute a proof. As a result, students do not know what counts as an acceptable mathematical justification.

Moreover, students are likely to engage in activities that feel worth while and relevant for their studies as a whole. However, the prevalent curriculum strategy at CS departments is to divide courses "into areas of 'theory' and 'practice'... [which] causes both faculty and students to view the theory of computing as separate and distinct from the practice of computing." [2, p. 73] In order for mathematics to be considered useful by CS students, it should thus be presented in a way that clearly links it to the computing practice.

In this paper, we present *structured derivations* [4,6,7], a logic based approach to teaching mathematics, which we argue can be used to address all the aforementioned problems. Structured derivations promote preciseness of expression and offer a systematic and straightforward presentation of mathematical reasoning, without restricting the application area. Using structured derivations, logic becomes a tool for doing mathematics, rather than a object of mathematical study.

We begin with a brief description of the structured derivations approach to constructing proofs in Section 2. The approach has been extensively evaluated since 2001 and currently the evaluation involves five institutions at high school and university level. We summarize the high school experience in Section 3 and give a more detailed account of our experience from using the approach in a first year CS course in Section 4. We conclude with a discussion section including ideas for future work.

2 Structured Derivations

Structured derivations [4,6,7] is a further development of Dijkstra's *calculational proof style*, where we have added a mechanism for doing subderivations and for

handling assumptions in proofs. With these extensions, structured derivations can be seen as an alternative notation for Gentzen like proofs in predicate calculus or higher order logic [6]. A structured derivation has the following general syntax:

$$
\begin{aligned}
\text{derivation} &::= \text{" } \bullet \text{ "[task " : "] [assumptionList] [(} \vdash \mid \Vdash \text{)] [proofSteps]} \\
\text{assumptionList} &::= (\text{postulate} \mid \text{lemma})^{+} \\
\text{postulate} &::= \text{"[" identification "]" formula} \\
\text{lemma} &::= \text{"(" identification ")" formula derivation} \\
\text{proofSteps} &::= \text{term } (\text{basicStep} \mid \text{subderivation})^{+} \\
\text{basicStep} &::= \text{relation "\{" motivation "\}" term} \\
\text{subderivation} &::= \text{relation "\{" motivation "\}" derivation}^{+} \text{ " . . . " term}
\end{aligned}
$$

Terminals are given inside quotes and nonterminals in roman font. The layout of a structured derivation is fixed. The general proof layout is as follows ("task" is a short informal explanation of what we want to do):

derivation:

```
•   task:
    assumptionList
⊢
    proofSteps
```

Below to the left, we show the layout for a derivation where all assumptions are postulates and where there are only basic proof steps. The middle box shows the layout of a lemma, where the proof of the lemma (the formula) is a derivation written directly after the formula but is indented one step to the right. On the right we show a proof step with a subderivation. A subderivation justifies the proof step and corresponds to the application of an inference rule in a Gentzen like proof system. One proof step may require one or more subderivations. The subderivations follow immediately after the motivation for the proof step and are indented one step.

simple derivation:

```
•   task:
|id] formula
⋮
[id] formula
⊢
    term
rel {motivation}
    term
⋮
rel {motivation}
    term
```

lemma with proof:

```
(id) formula
     •   task:
         assumptionList
     ⊢
         proofSteps
```

subderivation proof step:

```
    term
rel {motivation}
    •   task:
        assumptionList
    ⊢
        proofSteps
... term
```

This proof format fixes the overall structure and layout of a derivation (hence the name *structured derivations*) but it does not fix the syntax of basic entities such

as task, formula, term, relation, motivation, or identification. Thus, we can use structured derivations for proofs on different domains, and with different levels of rigor and detail (from a completely intuitive argumentation to an axiomatic proof in a logical theory).

We illustrate structured derivations with two mathematical problems. The first one illustrates the use of logical rules in standard mathematical reasoning, while the second illustrates subderivations and the way in which we combine formal and informal reasoning in a structured derivation (the second problem is taken from the Finnish high school matriculation exam 2006).

Our first problem is to solve the equation $(x-1)(x^2+1) = 0$. We have the following solution

---

- Solve the equation $(x-1)(x^2+1) = 0$:

  $(x-1)(x^2+1) = 0$

$\equiv$ {zero product rule: $ab = 0 \equiv a = 0 \vee b = 0$}

  $x - 1 = 0 \vee x^2 + 1 = 0$

$\equiv$ {add 1 to both sides in left disjunct and -1 to both sides in right disjunct}

  $x = 1 \vee x^2 = -1$

$\equiv$ {a square is always non-negative}

  $x = 1 \vee F$

$\equiv$ {disjunction rule}

  $x = 1$

The second problem is to determine the values of a for which the function $f(x) = -x^2 + ax + a - 3$ is always negative.

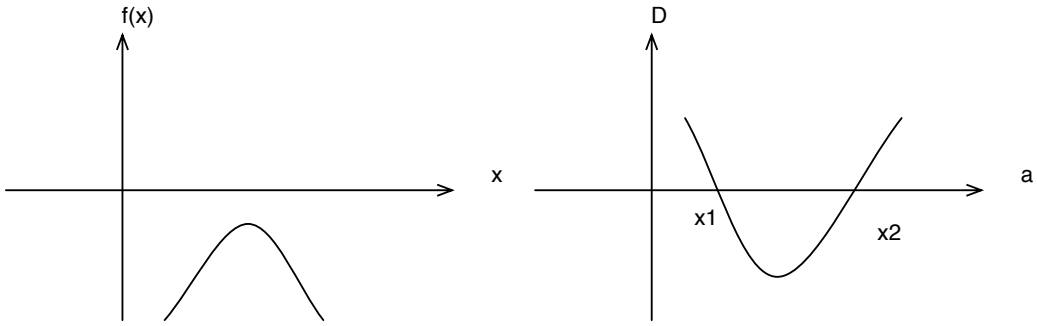


Figure 1. Downward and upward opening parabolas

The following structured derivation shows how we determine the value of a. Figure 1 illustrates the arguments used in the proof.

- Determine the values of a for which $-x^2 + ax + a - 3$ is always negative:

  $(\forall x \cdot -x^2 + ax + a - 3 < 0)$

$\equiv$ {the function is a parabola that opens downwards (the coefficient for x^2 is negative); such a function is always negative if it does not intersect the x-axis, i.e. has no roots (figure on the left)}

  $(\forall x \cdot -x^2 + ax + a - 3 \neq 0)$

$\equiv$ {this condition holds iff the discriminant D for the function is negative}
$D < 0$
$\equiv$ {substitute value of D}
- Determine the discriminant D:
  D
  $=$ {the discriminant for the equation $Ax^2 + Bx + C = 0$ is $B^2 - 4AC$}
  $a^2 - 4(-1)(a - 3)$
  $=$ {simplify}
  $a^2 + 4a - 12$

$\ldots$ $a^2 + 4a - 12 < 0$
$\equiv$ {the function $a^2 + 4a - 12$ is a parabola that opens upwards (the coefficient for a^2 is positive); such a function is negative between its roots (figure on the right)}
- Solve the equation $a^2 + 4a - 12$:
  $a^2 + 4a - 12 = 0$
  $\equiv$ {square root formula}
  $a = \frac{-4 \pm \sqrt{4^2 - 4 \cdot 1 \cdot (-12)}}{2 \cdot 1}$
  $\equiv$ {simplify the expression}
  $a = 2 \vee a = -6$

$\ldots$ $-6 < a < 2$

This proves that

$$(\forall x \cdot -x^2 + ax + a - 3 < 0) \equiv -6 < a < 2$$

In other words, the function is always negative if and only if $-6 < a < 2$.

If we are using a computer supported tool with outlining features (like the TEXmacs plug-in mentioned below), we can choose to hide the two subderivations. Omitting the more detailed steps will give us a better view of the overall structure of the proof.

Traditional approaches to teaching and presenting mathematics contain much implicit information [9,14]. Using structured derivations, all steps in the derivation are explicitly motivated and the final product thus contains a documentation of the thinking the student was engaged in while completing the derivation. This facilitates reading and debugging both for students and teachers.

Moreover, as stated in the introduction, traditional approaches to teaching proofs leave students uncertain about what rigor is required for a particular proof in a certain situation [9]. Structured derivations provide a well-defined proof format, which gives students a concrete "model" for what constitutes a proof and which can guide them in how to carry out rigorous proofs in practice. A clear and familiar format functions as a mental support that gives students belief in their own skills to construct the proof. A defined format also lets students focus on the solution rather than spending time thinking about how to put their thoughts down on paper. Furthermore, our approach provides a structure that can be used to make the presentation of mathematics more consistent in textbooks and classrooms. Due to

the well-defined syntax and simple structure, structured derivations are also well suited for presentation on the web.

3 Structured Derivations at High School Level

As stated in the introduction, Finnish students can graduate from high school without being exposed to logic or proof in their mathematics courses. This is alarming not only from a computing perspective, but from a science perspective in general. Thus, although our main concern as CS educators is to ensure that our students possess sufficient mathematical skills in order to be able to progress successfully in their studies, we feel that attention also should be put on mathematics education at lower levels. We summarize our experiences from introducing structured derivations at high school in this section, and describe our experiences from introducing the same method into a CS syllabus in the next section.

The two mathematics syllabi offered in Finnish high schools have different foci: the general syllabus focuses on developing the capabilities needed "to use mathematics in different situations in life and in further studies" [15, p. 119], whereas the advanced syllabus focuses on learning to "understand the nature of mathematical knowledge" [15, p. 122]. The advanced syllabus is practically the norm for students seeking admission to universities for further studies in, for instance, mathematics, CS, engineering, medicine and physics. Considering the need for mathematical maturity in these fields, the students would most certainly benefit from getting more training in formal reasoning and proof already at high school level (as mentioned before, these topics are currently only mentioned in one advanced elective course). This does, however, not necessarily imply that more specific courses on logic should be introduced, but rather that logic should be integrated in other courses [16].

In 2001, a longitudinal study was initiated in a high school in Turku, Finland [5,17]. The aim of the study was to investigate whether structured derivations could be used to integrate logic, proof and formal reasoning throughout high school mathematics education without the need for additional courses on logic. The research setting involved a test group and a control group, which were followed up during their entire high school period (three years). The students chose which group to belong to, but care was taken to ensure that the entry level of the students was as similar as possible in both groups. The groups had different teachers who taught the exact same material, only using different approaches; the test group teacher rewrote and taught all ten compulsory mathematics courses using structured derivations, whereas the control group teacher gave the courses in his usual presentation style. Moreover, the test group teacher spent a few hours at the beginning of the first course introducing basic notions of elementary logic and giving students formal and informal practice in working with logical connectives.

The results from the study were positive, indicating that logical notation and structured derivations can be successfully used in a high school setting. Students in the test group consistently outperformed the control group in all ten courses [5] as well as in the matriculation exam [17]. [5]

[5] Students take the matriculation exam at the end of their high school studies. The matriculations examination board approves of the use of structured derivations in the matriculation exam.

In addition to this longitudinal study, the approach has also been introduced in single courses at three other high schools. Clearly, this renders a completely different situation than when all courses are given using the same format. Despite the limited time available for the teacher to present the approach and for the students to get familiar with it and use it, the results from these courses have also been positive. Surveys and observations have shown that despite a somewhat negative initial reaction to the new strict format requiring additional writing, most students learned to use and appreciate the structured approach during one single course.

We are now in the process of developing more systematic teaching material to support the use of structured derivations in mathematics education. Back and von Wright have written "Mathematics with a Little Bit of Logic" [8], a text book that introduces the approach and that can also be used as a teachers manual. Moreover, two ordinary high school mathematics text books have been "translated" into structured derivations. In addition, all assignments in ten complete mathematics matriculation exams have been solved using structured derivations (altogether 150 solved problems). This collection of solutions is important not only as an example base, but also as a confirmation that the approach can in fact be applied on a wide variety of problems.

4 Structured Derivations for First Year CS Students

The need for practical skills in proving mathematical theorems becomes evident to our CS students already during their first year courses. A compulsory course on logic was introduced in the basic studies in the CS curriculum at our department already in the 1990s, but as it was rather theoretical, students did not see the connection to the real world and felt that the course did not give them any skills that could be useful in practice. The course was totally redesigned in fall 2006, when structured derivations were introduced to put more focus on enhancing students' logical reasoning and proof-writing abilities in practice.

The course was attended by 47 students and included 36 lectures (of 45 minutes), six exercise sessions (of 90 minutes) and a final exam. A pre- and postcourse survey as well as observations were used to evaluate the course and students' opinions about the approach. The idea of the course was to apply structured derivations to high school mathematics. We thought that applying the rigorous derivational format on familiar problems would make it easier for students to learn the methodology, as they would not have to learn any new mathematics at the same time. This, however, did not work as intended. Instead, the familiar domain hindered the students from seeing the purpose of the course, thinking that the course was just a repetition of high school mathematics, failing to understand the importance of the format used. The initial confusion was partly a result of miscommunication, as the teaching assistant did not enforce the use of the structured derivations format in the exercise sessions. This gave the students a message that was not consistent with the one they received during the lectures.

Resistance to relearn familiar material using a new format is understandable; why should one start using a new approach for doing something one has already been doing successfully in another way for 12 years. However, as the students

realized that the topics covered in the course were indeed intended to be familiar, and that structured derivations was the new thing that they were supposed to learn, the resistance faded away. The results from the final exam were good (70% of the students passed, 30% with the highest grade) and the final feedback was in general positive. In the following, we list some of the positive and negative aspects brought up by students in the open questions of the post course survey. [6]

Students identified many of the same benefits of using structured derivations as was originally hypothesized when the approach was developed:

- *"When you write out everything, careless mistakes disappear"*
- *"Writing better mathematical derivations: same motivations as earlier, but more logically constructed"*
- *"Learning a systematic way of working"*
- *"I learned to think deeper on mathematical solutions"*
- *"Useful to practice problem solving, structure and divide problems"*
- *"The different proof strategies will be useful in math courses. But the logical motivations were also important to learn considering how to prove one's programs"*

Some students found writing the derivations a bit tedious, but nevertheless found the approach interesting and useful.

- *"They feel a bit unnecessary sometimes, but on the other hand you see much more clearly what you've meant when you look at the solution again later"*
- *"Structured derivations feels like unnecessary work, but the format does make the calculations clearer"*
- *"In many cases I feel that structured derivations is a way of complicating simple things. Sure, you should be able to motivate what you're doing, but there's no need to exaggerate. On a suitable level of abstraction, this is, however, an interesting way of thinking"*
- *"The derivations were important. Even if you don't like them, you may appreciate them more during further studies"*

Students also appreciated the structured derivations simply because it was a new format that appealed to them.

- *"The approach was pretty difficult and therefore interesting"*
- *"Interesting to learn how to write them and understand why you should write them"*
- *"The approach was interesting. I've always had problems proving things"*

The negative aspects brought up by the students were mainly related to the motivations and assumptions in proofs. One student also mentioned difficulties remembering the syntax of the format.

- *"Feels somewhat unnecessary to motivate everything"*
- *"I think it's difficult to write down the motivations as many of them are obvious"*
- *"Difficult to know when a derivation is correct. What can you assume and what can't you assume?"*
- *"Difficult to remember how to write them correctly"*

Finally, some students seemed to have a negative attitude towards mathematics in general.

- *"Derivations are always uninteresting"*
- *"I'm not interested in derivations and I don't think I'll need it in future CS studies"*

We were pleased to see that only a couple of students expressed negative opinions towards mathematics after the course. Most students stated that they appreciated the approach and the benefits it provides (clarifies solutions, facilitates debugging, makes relations explicit, enforces a systematic way of working, etc). As was found in the feedback on the high school courses, students initially felt frustrated with the extra writing needed in the form of motivations, but still found the format useful. We feel that the students' feedback indicates that the majority of them did no longer

[6] The quotations have been freely translated into English by one of the authors.

fear notation after this particular course. In our opinion, this is an encouraging finding.

Based on the experience from the first version of the course, some revisions were made before giving the course again starting in fall 2007. The main changes made were that instead of applying structured derivations on high school topics, the approach is now exemplified with problems in areas that the students are not familiar with from before: propositional and predicate logic, discrete mathematics, elementary algebra, lattice theory and boolean algebra. The focus is still on using the structured derivations framework and on emphasizing the need to develop skills in constructing mathematical proofs in practice. This course is going on as we write this, and the final evaluation is thus yet to be done. However, our preliminary observations indicate that students now "got the point" of the course from the very beginning, have used the structured derivations format in their own solutions and appreciate learning new mathematical topics.

5 Discussion

The experience from using structured derivations in education has been encouraging both at secondary and tertiary level. Although the results of introducing the approach in individual high school courses have been positive, we nevertheless believe that integrating logic as a tool in all mathematics courses is to be preferred. The effects of one single course are easily canceled out by the remaining courses not mentioning logic or proof at all.

Our experiences from teaching the structured derivations course as an introductory CS course has also been very encouraging, so much that structured derivations is now the standard approach used in the logic course. We are also planning a new course for first year university students in natural sciences and engineering. The course will be specifically designed as a bridging course between high school and university, focusing on improving students' proficiency in doing mathematical proofs with structured derivations.

We are presently working on tool support for making structured derivation proofs, both on a personal computer and on the web. We use TEXmacs [1], a wysiwyg LaTeX editor, as the basic framework for writing mathematical documents. We have constructed a plug-in for TEXmacs that understands structured derivations and makes it easy to construct and browse derivations. In particular, TEXmacs now supports selective hiding and revealing of subderivations and lemma proofs. This has made it straightforward for both teachers and students to work with derivations and proofs in electronic format.

Moreover, we are also currently working on providing just-in-time on-the-spot assistance for students reading a mathematical proof. For instance, consider a step in a derivation that calls for solving an equation, like in the second example given in Section 2. In that example, the equation was solved in a subderivation. If all subderivations are initially hidden, then students who feel confident about how to solve an equation do not need to open the subderivation, while students who are uncertain can do that. Thus, one single example can be used for students at different skill levels. This feature also renders structured derivations suitable for self-study

material, as examples can be made self-explanatory on different levels, providing the reader the choice of different levels of detail.

References

[1] Texmacs homepage. Available online: http://www.texmacs.org. Retrieved on December 2, 2007.

[2] Vicki L. Almstrum, C. Neville Dean, Don Goelman, Thomas B. Hilburn, and Jan Smith. Support for teaching formal methods. *SIGCSE Bull.*, 33(2):71–88, 2001.

[3] Vicki Lynn Almstrum. *Limitations in the Understanding of Mathematical Logic by Novice Computer Science Students*. PhD thesis, Department of Computer Science , University of Texas, 1994.

[4] Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured calculational proofs. *Formal Aspects of Computing*, 9:469–483, 1998.

[5] Ralph-Johan Back, Mia Peltomäki, Tapio Salakoski, and Joakim von Wright. Structured derivations supporting high-school mathematics. In A. Laine, J. Lavonen, and V. Meisalo, editors, *Current Research on Mathematics and Science Education*. Department of Applied Sciences of Education, University of Helsinki, 2004.

[6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[7] Ralph-Johan Back and Joakim von Wright. A method for teaching rigorous mathematical reasoning. In *Proceedings of Int. Conference on Technology of Mathematics*, University of Plymouth, UK, Aug 1999.

[8] Ralph-Johan Back and Joakim von Wright. *Mathematics with a Little Bit of Logic: Structured Derivations in High-School Mathematics*. Manuscript, 2006.

[9] Tommy Dreyfus. Why Johnny Can't Prove. *Educational Studies in Mathematics*, 38:85–109, 1999.

[10] David Gries. Equational logic as a tool. In *AMAST '95: Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology*, pages 1–17, London, UK, 1995. Springer-Verlag.

[11] David Gries. *Teaching and Learning Formal Methods*, chapter Improving the curriculum through the teaching of calculation and discrimination, pages 181–196. Academic Press, London, 1996.

[12] Gila Hanna. Challenges to the importance of proof. *For the Learning of Mathematics*, 15(3):42–49, November 1995.

[13] Kirsti Hemmi. *Approaching Proof in a Community of Mathematical Practice*. PhD thesis, Department of Mathematics, Stockholm University, 2006.

[14] Uri Leron. Structuring mathematical proofs. *American Mathematical Monthly*, 90(3):174–185, 1983.

[15] Finnish National Board of Education. National core curriculum for upper secondary schools 2003, 2003.

[16] The ASL Committee on Logic and Education. Guidelines for logic education. *The Bulletin of Symbolic Logic*, 1(1):4–7, 1995.

[17] Mia Peltomäki and Tapio Salakoski. Strict logical notation is not a part of the problem but a part of the solution for teaching high-school mathematics. In *Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education - Koli Calling*, pages 116–120, 2004.

[18] A. H. Schoenfeld. What do we know about mathematics curricula? *Journal of Mathematical Behavior*, 13(1):55–80, 1994.

Teaching Software Model Checking

Cyrille Artho[1]

Research Center for Information Security (RCIS)
National Institute of Advanced Industrial Science and Technology (AIST)
Tokyo, Japan

Kenji Taguchi[2] Yasuyuki Tahara[3] Shinichi Honiden[4]

Information Systems Architecture Science Research Division
National Institute of Informatics
Tokyo, Japan

Yoshinori Tanabe[5]

Research Center for Verification and Semantics (CVS)
National Institute of Advanced Industrial Science and Technology (AIST)
Tokyo, Japan

Abstract

The use of formal methods has become commonplace in hardware design, and is becoming increasingly widespread in software engineering. While formal methods have repeatedly been applied in safety-critical projects, their technologies and tools are not widely known, due to lack of in-depth education in current curricula. In this paper, we introduce the curriculum design of software model checking, which is part of a larger education program that addresses several issues in software engineering and formal methods in general. We will also touch upon the necessity of a formal methods body of knowledge (FMBOK) for the guidance of formal methods education.

Keywords: Formal methods, education, software engineering, model checking, body of knowledge.

1 Introduction

The use of formal methods is already widespread in hardware design and is becoming increasingly common in mission-critical software projects. Despite this success, software engineering courses typically only introduce formal methods at the Ph. D. level. In our education program called Top SE [7], we address issues in software

[1] E-mail: c.artho@aist.go.jp
[2] E-mail: ktaguchi@nii.ac.jp
[3] E-mail: tahara@nii.ac.jp
[4] E-mail: honiden@nii.ac.jp
[5] E-mail: tanabe.yoshinori@aist.go.jp

engineering and formal methods from requirements to testing, and provide a comprehensive education about these subjects to graduate students and engineers from industry.

Formal methods have been recognized as a rigorous software development methodology for safety-critical systems [13]. It is now recommended to be used in safety and security areas such as functional safety (IEC 61508) [8] and security assurance (ISO/IEC 15408) [9]. On the other hand, the teaching of formal methods is still at an early stage in the sense that there is no standardized curriculum guidance based on a body of knowledge on the whole area of formal methods.

In this paper, we introduce the curriculum design of our course in software model checking. In this course named "Model Checking of Concurrent Java Programs", cutting-edge research results are integrated with foundational materials on software model checking.

A survey on the undergraduate curriculum on formal methods was carried out by Oliveira [12] 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. Unfortunately, there has not been any follow-up activity from this group since then, even though the demand for education in formal methods is growing constantly.

Formal methods could be taught within a broader topic (e.g., software verification [13]) or as a separate and independent course [10]. As we will explain later, our education program has a strong emphasis on formal methods. We provide a wide variety of courses ranging from model checking to formal specifications. When we assessed our courses on formal methods, we realized that the existing body of knowledge (BOK) on software engineering and computers was not helpful due to its lack of depth and width in its description of formal methods. This led us to pursue an idea of a body of knowledge specifically designed for formal methods.

This paper is organized as follows: The next section presents an overview of our education program. Section 3 describes the course on software model checking, which includes coverage of very recent research results. Section 4 discusses lessons learned while teaching this course for two years. The need for a body of knowledge on formal methods is explained in Section 5, while related work is covered in Section 6. Section 7 concludes this paper.

2 The Top SE program

The Top SE program is a non-accredited course at the Masters level, operated in close collaboration between industry and academia at the National Institute of Informatics in Tokyo [7]. The whole curriculum is fully funded by the Japanese government and comprises five lecture series as shown in Table 1. All courses in the five-lecture series are electives except for two basic courses, which are compulsory.

The program has the following distinguishing features:

- Cutting-edge software engineering technologies are covered comprehensively. The program encompasses software design patterns, aspect-oriented development, and model checking.

Table 1
Curriculum of Top SE.

Series	Courses
Requirements Analysis	Requirements Analysis
	Security Requirements Analysis
System Architecture	Component-based Development
	Software Patterns
	Aspect-Oriented Development
Formal Specifications	Formal Specifications (Foundations)
	Formal Specifications (Applications)
	Formal Specifications (Security)
Model Checking	Verification of Design Models (Foundations)
	Verification of Design Models (Applications)
	Model Checking of Concurrent Java Programs
	Verification of Performance Models
	Modelling and Verifying Concurrent Systems
Implementation	Testing
	Program Analysis
Basics	Basics of Software Science
	Practical Software Engineering

- Real case studies from industry are included. We regard our education program as a mission to transfer technology from academia to industry, and we tailor the program to meet the needs of industrial partners such as Hitachi, NEC, Toshiba, NTT Data, Fujitsu, and Nihon Unisys.
- The program emphasizes engineering practice and tools over theory and processes. Management and process aspects are not the focus of the program for the moment. [6]
- Collaboration with academic partners (Shinshu University, Tsukuba University, and others) ensures that our courses cover a broad spectrum. Multiple tools and methodologies are presented to teach not only the usage of tools, but also the techniques and guidelines that apply to practical software development.

As Table 1 shows, the usage of formal methods, including formal specification languages, in the early phases of the software development life cycle (requirements and design) is emphasized. Testing and program analysis are covered as well, with *Model Checking of Concurrent Java Programs* being the most advanced course in this area.

The compulsory course *Basics of Software Science* covers elementary logics (propositional and predicate logics), temporal logics (LTL, CTL), concurrency theory, abstraction, automata, model checking algorithms, and their implementation techniques. Traditionally, these topics are taught in different courses (e. g., preliminary mathematics, operating systems, and more advanced courses in theoretical computer science). Our course includes all necessary preliminary knowledge on formal methods, which makes it rather unique.

[6] A new plan is underway to include process/quality management courses.

Table 2
Module structure of the "Model Checking of Concurrent Java Programs" course.

Description		Lectures
JPF Basics	Introduction	1st
	Safety	2nd
	Liveness	3rd, 4th
	Fairness	5th, 6th
	Group Exercise	7th
Model Checking Networked Programs	Introduction, stub-based approach	8th
	Input/output caching, centralization	9th
	Centralization	10th
	Network layer for centralization	11th
	Group Exercise	12th

3 Course and curriculum design

In this section, we explain the course description shown in Table 2 in detail.

3.1 Goals of the course

The distinguishing features of model checking compared with other approaches such as testing or theorem proving are that it can be fully automated and that all possible behaviours can be checked [13]. Model checking technology can be applied to a design specification or to source code [6]. The aim of design analysis is to validate correctness of an early design. This aspect of model checking is taught in *Verification of Design Models (Foundations, Applications)*, as shown in Table 1. The direct analysis of source code, which is taught in this course, is a relatively new idea.

Verification of the final code is necessary if one wants to have full confidence in the final product. At the moment, we have to admit that heavy-weight tools such as software model checkers cannot handle large implementations yet. Still, when forsaking complete coverage of the state space (when looking for bugs rather than trying to show correctness), the Java PathFinder model checker (JPF) still manages to find complex failures in production systems that cannot be found by testing. The goal of the course is to make students proficient in this novel verification technology, and give them an understanding of its capabilities and limitations.

There are still only a few industrial practices where program code is directly model checked. Model checking the implementation directly has the potential to eliminate the process of creating an abstract model from source code. Nevertheless, due to the state space explosion problem, code usually has to be simplified (abstracted) prior to verification. In practice, verification has to focus on key aspects such as inter-thread communication and synchronization. Therefore, early prototypes (whose fine-grained design is not fixed yet) are ideal for model checking.

Our lecture does not focus on the software development process and therefore does not favor verification at a particular stage in the development cycle.

3.2 Model analysis

The first half of the course (seven lectures) is devoted to the basics of JPF. Most students have only learned the basics of model checking in classrooms and are not

familiar with software model checking in practice. Since they tackle verification of network programming in the second half of the course, they must master the basics of JPF in the first half. We also want the students to understand how the property patterns learned in the *Basics of Software Science* course, namely, safety, liveness, and fairness, are expressed and verified in the context of software model checking.

The first lecture is an introduction. A brief review of related topics learned in the *Basics of Software Science* course covers: the significance of verification techniques such as model checking, the usefulness of model checking in a concurrent or distributed environment, and advantages and drawbacks of software model checking. An overview of JPF is also presented.

Safety properties are the first target in the course. JPF is designed so that safety properties are verified in a natural way. In this part, we illustrate how they are verified with JPF by using simple examples.

The next topic regards liveness and properties in the form of Fp and $G(p \rightarrow Fq)$. Since they cannot be verified with the built-in functions of JPF, we implement a search extension for such properties. Through exercises, students learn that JPF is designed so that functions can be extended by preparing modules and how such modules are developed.

Finally, we consider fairness, which can be expressed in linear temporal logics (LTL). In the *Basics of Software Science* course, the students learned how LTL formulae are converted into Büchi automata [6]. In this course, we implement a search algorithm for JPF based on the conversion. Students study an application of automata theory through verification exercises on liveness properties under fairness.

In 2006, this part also included lectures on abstraction techniques such as data abstraction, predicate abstraction, and program slicing. However, these lectures were moved to the *Basics of Software Science* course for 2007.

3.3 Networked software

The second half of the course deals with networked software. At the time of course creation, state of the practice consisted of treating network functionality as an open call returning a non-deterministic result. This results takes all possible behaviours into account but may also generate spurious behaviours, which are not possible in practice. A less abstract stub function can be used as an alternative to a completely non-deterministic result. Such a stub returns a result that approximates the behaviour of the environment for a given test case. After briefly touching on automated, domain-specific techniques [5], the course teaches manual stub creation.

After coverage of stubs, the concept of a novel input/output caching approach is introduced. Because the implementation was in a prototype stage [4] when the course was taught, the remainder of the second half focuses on the centralization approach [15] and recent extensions that made it applicable to software communicating using TCP/IP sockets [2]. Besides the usage of JPF for analyzing the application, the theory and engineering behind the method are covered in great detail. The aim is to provide enough knowledge to allow the students to understand the tool in detail, and to cover the concepts of other communication mechanisms (such as pipes).

The exercises started with two simple server applications, which served as examples for stub usage and centralization. After that, a chat server with a main/worker thread architecture was used as a running example for the final exercises. This final exercise was rather complex, as the architecture of the chat server resembles the design of real-world servers. Hence, it constitutes a good preparation for analyzing real software using a model checker.

3.4 Model checking tools

The Java PathFinder model checker [16] (JPF) is the main tool for this lecture. We chose JPF because of its free availability, easy portability (Sun's Java Development Toolkit version 1.4 being the only prerequisite) and its facilities that allow extensions to be plugged in relatively easily. Furthermore, JPF version 3 is relatively mature and has no defects that would prevent us from carrying out exercises and case studies.

At the time when the lecture was made, no Java model checker could deal with network communication. This severely restricted the usefulness of JPF, as interesting Java programs typically require networking. During 2005 and early 2006, the necessary extensions and an additional tool, called *Centralizer*, were built to allow verification of networked software [2]. These research results were included in the lecture, because they greatly improve the applicability of software model checking.

4 Student outcome

In this section, we report on the student responses from three sources: coursework reports, students' presentations, and questionnaire results.

Coursework Reports

The students were asked to submit five coursework reports during the course. Some exercises were obligatory, and the others were non-obligatory. The former could be completed if they understood the basics of the lectures, while deeper understanding and skills were needed to complete the latter.

Almost all students submitted reports. About half of them submitted only the obligatory exercises, while a couple of students tackled almost all the exercises. Since 87 % of the obligatory exercises were answered correctly, we concluded that the students had acquired the basic knowledge. About a quarter of the students gave good answers for more than half of the non-obligatory exercises. We considered that these students had mastered sufficient skills to use JPF.

Students' Presentations

The seventh and twelfth lectures were group exercises. Students were divided into several groups, each of which consisted of five to seven students. Different tasks were assigned to each group. A group discussion of 60 minutes was followed by a presentation of 15 minutes per group.

One of the tasks in the first exercise regarded safety properties; the other was about liveness properties. The former was a classical "crossing a river" problem. Verification itself was simple. However, students were also asked to modify a VM listener so that additional information was added to the error trace. The next task concerning liveness was a variant of the dining philosopher problem. The exercise consisted of model checking, finding the reason for the breach of the liveness property, and modifying the program.

Since the task list was presented beforehand, motivated students had already started investigating the tasks, and the discussion was conducted under their leadership. Although no group completed all the assigned tasks, all groups found solutions to the exercises marked as obligatory.

During preparation of the presentation, it was observed that some of the students who had submitted only obligatory exercises in the coursework reports were not proficient in the use of the tool. However, even such students learned from other students through the discussion, since they actively took part in it. In the questionnaire described below, six out of seven students answered that the group exercise deepened their understanding.

The final exercise, discussed in the last lecture, was the most difficult one. It required both proficiency with JPF as well as an in-depth understanding of concurrency in Java. The exercise was presented one week prior to the group discussions. Some of the students who studied the material were able to complete this exercise, while others recognized some important points but did not arrive at a complete solution on their own.

Questionnaire Results

Students were asked to fill in questionnaire forms at the halfway point and at the end of the course; eight and seven students responded, respectively.

From their answers, we presume that the students were satisfied with the course – on a scale of 1 to 5, the average score for the lectures was 4.0, and the difficulty was judged to be 3.7 (1: easy to 5: difficult). The self-evaluation of proficiency score was 3.4, which was not much different from that evaluated by the lecturers.

The applicability of JPF to development work in their company was not evaluated very affirmatively — the score was 3.0. Among the free responses were: "JPF requires the user to possess a high-level knowledge," "automatic re-modeling of distributed environment is desired for easy usage," and "development of patterns is needed to apply the tool to actual problems." These comments reflect the current status of the software model checking technology. However, it is our opinion that development of the skills to utilize such an emerging technique is an advantage of "Top SE".

5 Outlook: Body of knowledge on formal methods

As shown in Table 1, formal methods are at the core of our education program. Therefore, we are very concerned with how to design the whole formal methods courses in a well-structured and organized way to ensure that all necessary knowledge areas are covered and all teaching materials are properly balanced.

To achieve this goal, we have come to realize that there is a lack of identification and specification of the underlying content of formal methods. These activities are definitely necessary for providing curriculum guidance for various formal methods courses. Indeed, formal methods appears to be a relatively mature discipline. Nonetheless, there is no established body of knowledge on formal methods. SWEBOK (software engineering body of knowledge) [1] is a comprehensive description of the software engineering knowledge standardized by IEEE CS and ACM, but it is too general for our purposes. A body of knowledge that is quite similar to our idea is PMBOK (project management body of knowledge) [14]. PMBOK was standardized by the Project Management Institute. It is widely used as the de facto standard reference for project management.

Our proposal here is to standardize a body of knowledge on formal methods (FMBOK) for general guidance of curriculum design on formal methods. We are at an early stage of developing this idea and we would like to call for an international effort to explore it. We believe this idea will be of a great benefit for all educators and practitioners working on formal methods.

6 Related work

There are a lot of courses treating model checking of programs. They can easily be found by WWW search engines such as Google, with keywords such as "software model checking", "course" and "lecture". However, most of them treat software model checking in general and are separated into small parts, each one of which is self-contained with an individual tool or technique. In addition, only a small number of the courses include exercises on practical problems. Kwiatkowska gave one of the exceptional example [11]. She assigned her students project work including case studies of verification activities. However, her work does not include any distributed system problems.

A shorter version of this course was also taught as a half-day tutorial at the Automated Software Engineering (ASE) conference in 2006 [3].

7 Conclusions

In this paper, we introduced the design and students' outcome of our software model checking course and touched upon the idea of a formal methods body of knowledge (FMBOK). Software model checking has not been widely taught yet. Several ways to teach it are possible. In order to access and compare courses, we will need a specific knowledge area for software model checking. Hence, our main future work is to develop FMBOK and its specific knowledge area for software model checking.

Acknowledgments

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

References

[1] Abran, A. and Moore, J. W. and Bourque, P. and Dupuis, R.: Guide to the Software Engineering Body of Knowledge 2004 Version SWEBOK. IEEE (2004)

[2] Artho, C. and Garoche, P.-L.: Accurate centralization for applying model checking on networked applications. In Proc. 21st Int'l Conf. on Automated Software Engineering (ASE 2006), Tokyo, Japan (2006)

[3] Artho, C.: Model Checking Networked Software. Tutorial at ASE 2006, Tokyo, Japan.

[4] Artho, C. and Zweimüller, B. and Biere, A. and Shibayama, E. and Honiden, S. Efficient model checking of applications with input/output. Post-proceedings of Eurocast 2007, LNCS 4739:515–522, 2007. Springer

[5] Ball, T. and Podelski, A. and Rajamani, S.: Boolean and Cartesian Abstractions for Model Checking C Programs. In Proc. TACAS 2001, LNCS 2031, pp. 268–285, Genova, Italy, 2001. Springer

[6] Clarke, E. and Grumberg, O. and Peled, D. Model Checking. MIT Press, 1999.

[7] Honiden, S. and Tahara, Y. and Yoshioka, N. and Taguchi, K. and Washizaki, H.: Top SE: Educating Superarchitects Who Can Apply Software Engineering Tools to Practical Development in Japan. Proceedings of International Conference on Software Engineering (ICSE '07) 708-718, IEEE (2007)

[8] IEC 61508.: Functional safety of electrical/electronic/programmable electronic safety-related systems. Bureau Central de la Commission Electrotechnique International, Geneve, (2000)

[9] ISO/IEC 15408.: Information technology - Security techniques - Evaluation criteria for IT security - Part1, Part2 and Part3. ISO/IEC 2005 (2005)

[10] The Joint Task Force on Computing Curricula, IEEE CS and ACM.: Software Engineering 2004. IEEE CS and ACM (2004)

[11] Kwiatkowska, M.: Safety Critical Systems and Software Reliability.
`http://www.cs.bham.ac.uk/ mzk/courses/SafetyCrit/index.html`.
Lectures in University of Birmingham, School of Computer Science (2006)

[12] Oliveira, J. N.: A Survey of Formal Methods Courses in European Higher Education. CoLogNet/FME TFM'04, LNCS 3295 (2004) 235–248

[13] Peled, D.: Software Reliability Methods. Springer (2001)

[14] Project Management Institute.: The Guide to the Project Management Body of Knowledge (3rd edition). Project Management Institute (2004)

[15] Stoller, S. and Liu, Y. Transformations for model checking distributed Java programs. In Proc. SPIN 2001, LNCS 2057, pp. 192–199. Springer, 2001

[16] Visser, W. and Havelund, K. and Brat, G. and Park, S. and Lerda, F.: Model checking programs. Automated Software Engineering Journal, 10(2):203–232 (2003)