From Research to Teaching Formal Methods: The B Method - TFM-B'2009

Christian Attiogbe, Henri Habrias

▶ **To cite this version:**

Christian Attiogbe, Henri Habrias. From Research to Teaching Formal Methods: The B Method - TFM-B'2009. Journées Scientifiques de l'Université de Nantes, Université de Nantes, pp.134, 2009, 2-9512461-0-2. ⟨hal-04914102⟩

HAL Id: hal-04914102

https://hal.science/hal-04914102v1

Submitted on 3 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Colloque

From Research to Teaching Formal Methods: The B Method (TFM-B'2009)

Nantes, Cité Internationale des Congrès, 8 juin 2009
Nantes, International Convention Center, 8th June 2009

ACTES / PROCEEDINGS

C. Attiogbé, D. Mery (Eds.)

Juin 2009

lina LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE

imprimé par IUT de Nantes, (Université de Nantes), juin 2009
Publié par APCB
ISBN 2-9512461-0-2
EAN 9782951246102

Journées scientifiques - Université de Nantes (FR) - 2009

Actes/Proceedings

Christian Attiogbé, Dominique Mery (Eds.)

From Research to Teaching Formal Methods: The B Method

Organisation : COLOSS Team - LINA UMR 6241

Site : www.lina.sciences.univ-nantes.fr/apcb/, Pascal André

imprimé par IUT de Nantes (Université de Nantes), juin 2009
Publié par APCB
ISBN 2-9512461-0-2
EAN 9782951246102

Présentation

"Or certains n'admettent qu'un langage mathématique ; d'autres ne veulent que des exemples ; d'autres entendent qu'on recoure à l'autorité de quelque poète ; d'autres, enfin, exigent pour toutes choses une démonstration rigoureuse, tandis que d'autres jugent cette rigueur excessive, soit par impuissance à suivre la chaîne du raisonnement, soit par crainte de se perdre dans les futilités."
de Aristote, La métaphysique, tome 1, Vrin

La première édition des journées scientifiques, multidisciplinaires, de l'Université de Nantes a eu lieu en juin 2008. Fort du succès, l'Université recommence cette année 2009, nous donnant aussi l'occasion de rééditer avec l'association de pilotage des conférences B (APCB) notre colloque sur le lien entre la Recherche et l'Enseignement des méthodes formelles, en particulier la méthode B. Ce colloque vient compléter la série de conférences internationales sur B qui a débuté à Nantes en 1996.

La première conférence B a eu lieu à Nantes les 25-26-27 novembre 1996, après la conférence ZB de Nantes les 10-15 octobre 1995. Elle a été suivie des conférences de Montpellier, York, Grenoble, Turku, Guilford, Besançon et Londres en 2008 ; la prochaine aura lieu en 2010 au Canada (ABZ'2010). Ces conférences ont donné lieu des fois à des sessions éducation où les questions et expériences sur l'enseignement sont abordées. Le fait de lier la recherche en tant que telle et l'introduction des concepts, méthodes et techniques dans l'enseignement, n'est pas encore systématique, pourtant il est admis que les travaux de recherche en génie logiciel doivent conduire à la pratique. Pour cela, il faut passer par l'enseignement, enseignement qui doit allier les fondamentaux et la pratique. Voici trois arguments en faveur de l'enseignement de la méthode B entre autres méthodes formelles :

- La méthode B et ses concepts ont un intérêt pédagogique essentiel même si la méthode n'est pas appliquée jusqu'à l'obtention de code.
- La méthode B est une méthode industrielle qui passe l'échelle de l'industrialisation en allant de la spécification au programme y compris pour des applications de très grande taille. Il existe plusieurs exemples d'applications concrètes.
- Les environnements de travail avec B sont maintenant très élaborés, disponibles dans le domaine public et font le lien avec diverses autres technologies (simulation, test, évaluation de modèle (*model checking*)).

La méthode B a montré comment mettre en œuvre les enseignements fondamentaux en informatique : la construction correcte d'algorithmes et de logiciels. Nous pensons qu'il faut continuer dans cette voie, comme cela a été fait dans plusieurs autres domaines d'ingénierie où le "formel" est naturel.

Dans cette deuxième édition du colloque, le comité de programme a sélectionné une série d'articles qui abordent, la construction de modèles formel avec Event B, la comparaison de la méthode B avec d'autres techniques formelles du point de vue de la pratique par les étudiants, des expériences d'enseignement dans différentes Universités, etc. Nous espérons que les lecteurs trouveront pleine satisfaction et matière à réflexion ou discussion autour de l'enseignement de la méthode B et des autres méthodes formelles en général.

Nous remercions le Professeur M. Leuschel, qui a bien voulu accepter notre invitation pour parler de ses travaux de recherche et d'enseignement de la méthode B avec l'outil ProB. Nous remercions la société ClearSy qui, par sa politique de mise à disposition gratuite de l'Atelier B, favorise et facilite son utilisation par les étudiants.

Nous remercions aussi ceux qui ont participé à l'organisation et à la tenue de la conférence :

- Les auteurs qui ont soumis leurs travaux à ce colloque ;
- Les membres du comité de programme qui ont fait le travail de relecture, de sélection et de correction des articles proposés ;
- Les membres de l'équipe Coloss du LINA-CNRS ; Pascal André (qui a préparé le site web), Gilles Ardouel, Henri Habrias, Isabelle Condette ;

- L'Université de Nantes qui a organisé les Journées Scientifiques de l'Université de Nantes dans la prestigieuse Cité Internationale des Congrès de Nantes. Notre colloque est un des 21 colloques de ces Journées.
- L'IUT de Nantes qui a effectué l'édition des actes.

Enfin, nous remercions très chaleureusement nos collègues Henri Habrias qui a initié la série des conférences B et la série des journées sur l'enseignement de B et qui malgré sa retraite nous a beaucoup aidé et n'a pas ménagé son temps ; Dominique Mery président de APCB, pour son animation de la communauté B et plus généralement des méthodes formelles.

Christian Attiogbé, mai 2009

Presentation

"Or certains n'admettent qu'un langage mathématique ; d'autres ne veulent que des exemples ; d'autres entendent qu'on recoure à l'autorité de quelque poète ; d'autres, enfin, exigent pour toutes choses une démonstration rigoureuse, tandis que d'autres jugent cette rigueur excessive, soit par impuissance à suivre la chaîne du raisonnement, soit par crainte de se perdre dans les futilités."
Aristote, La métaphysique, tome 1, Vrin

The first edition of the multisciplinary scientific days of the University of Nantes took place in June 2008. Due to its success, the University reconducts the manifestation, giving to us the opportunity to reconduct, with the cooperation of the APCB (the B conferences Steering Comittee), our colloquim on the link between research and teaching formal methds and more particularly the B method. This colloquim follows a series of conferences started in 1996 in Nantes. The first B conference took place in Nantes on the 25th, 26th and 27th of November 1996, after the Nantes *Z2B conference*, from 10th to 15th October 1995. It was followed by the Montpellier, York, Grenoble, Turku, Guilford and Besancon conferences and London (September 2008) ; the next edition (ABZ'2010) will take place at Orford, Canada. During some of these conferences, education sessions were held and questions on teaching and experiments was discussed.

Linking research results and the teching of formal methods is not yet systematically practicised but it is widely admitted that research in software engineering and computer science should lead to good practices. For that purpose, teaching is one vehicle ; it should embrace fundaments and practices.

Here are three principal arguments in favour of teaching the B method :

- The B method and his concepts can be considered as having an essential pedagogical interest even if this method is not used until code generation.
- The B method is an industrial method which reaches the industrialisation level going from specification to programming. There are several examples of concrete applications.
- Practical develpment frameworks with the B method are now publicly available ; they are also integrated with other technologies (simulation, test, model checking).

The method have enabled one to put into practice the teaching of fundamental topics: correct program construction, correct software development, etc. We think that it is the right way to go, as it is the case in other engineering area, where using formal methods is rather natural and the word "formal" is not used anymore.

For this second edition of the colloquim, the program committee has selected works that address: the construction of models with Event B, the comparison of the B method with other formal methods on the setting of student experiments and of teaching experiments in Universities; We hope that the reader will find in this proceeding satisfaction and matters for thinking and discussion about the teaching of the B Method and other formal methods.

We thank Professor M. Leuschel for accepting our invitation to present his work on research and teaching with the ProB tool. We thank the ClearSy company, for putting some B tools in public domain and making it easy their use by students and others.

We also thank those who was involved in the organization and the outcome of the conference:

- The authors who submitted their work;
- The programme committee members who reviezed the submitted papers;
- The members of the Coloss team at LINA-CNRS; Pascal André has prepared the Web site, Gilles Ardourel, Henri Habrias, Isabelle Condette;

- The University of Nantes who organized the Journées Scientifiques de l'Université de Nantes in the prestigious International Center of Congress in Nantes. Our conference is one of 21 symposia of these "Journées scientifiques".

- The IUT Nantes who has made the printing of the proceedings.

Finally, we warmly thank our colleague Henri Habrias who initiated the series of the B conferences and also this series of colloquia; he helped us without worrying about his time; we warmly thank Dominique Mery chairman of APCB and an animator of the formal methods community, and of B in particular.

Christian Attiogbé

Comité de programme / Program Committee

Rueda Camilo,	Universidad Javeriana-Cali, Cali, Colombia
Lars-Henrik Eriksson,	Uppsala Universitet, Sweden
Marc Frappier,	Université de Sherbrooke, Canada
Marc Guyomard,	ENSSAT, France
Jacques Julliand,	Université de Besançon, France
Michael Leuschel,	Heinrich-Heine-Universität , Germany
Dominique Mery,	Loria, Nancy, France
Mike Poppleton,	University of Southampton, UK
Ken Robinson,	UNSW, Australia
Emil Sekerinski,	McMaster University, Canada
Elena Troubitsyna,	Abo Akademi University, Finland
Istenes Zoltàn,	Elte University, Hungaria

Autres reviewers / Additional reviewers

Pascal André,	Université de Nantes, FR
Gilles Ardourel	Université de Nantes, FR
Jean-Paul Bodeveix	Université de Toulouse, FR
Mamoun Filali	Université de Toulouse, FR
Arnaud Lanoix	Université de Nantes, FR

Thèmes / Topics

- Tool support for software engineering with the B method,
- Teaching environments for the B method,
- The B method in the software engineering curriculum,
- Combining the B method with other approaches
- Case studies and exercises featuring the B method,
- Use of the B method in disciplines other than software engineering
- New advances in the B method and their incorporation into the teaching curriculum.

Table des matières / Contents

ProB for Research and Teaching:

Lessons and Outlook

Michael Leuschel

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de *

In this invited talk we present our experiences in using the ProB validation tool for research and teaching. ProB [6, 8] is an animator and model checker for the B-method based on the constraint solving paradigm. Constraint-solving is used to find solutions for B's predicates. As far as model checking technology is concerned, ProB is an explicit state model checker with symmetry reduction [7, 12, 10]. While the constraint solving part of ProB is developed in Prolog, the LTL model checking engine [2] is encoded in C.
Some of the distinguishing features of ProB are

- the support for almost full syntax of B, integration into Atelier B and Rodin,
- the support for Z [11] and Event-B, building on the same kernel and interpreter as for "classical" B,
- the support for other formalisms such as CSP [9] via custom Prolog interpreters which can be linked with the B interpreter [3].

In this talk, we also present recent developments around ProB and show their importance for research and teaching B:

- a new datastructure for dealing with large sets and relations,
- an improved constraint solving algorithm,
- a new parser [4], being able to deal with almost the complete syntax of Atelier B,
- a new unification-based type checker, which is more flexible than other existing type checkers, and provides support for type checking definitions,
- a completely redeveloped plugin for Rodin [1], providing full support for Event-B (see Figure 1),
- multi-level animation of Event-B models,
- proof directed model checking, whereby proof information is used to improve the performance of the model checker, and finally
- an editor and framework for generating visual animations for Event-B models called BMotion Studio [5] (see Figure 1).

We also present an outlook of future developments around ProB, such as directed model checking, linking with SAT and SMT solvers, parallelisation as well as validation of ProB.

* This research is being carried out as part of the DFG funded research project GEPAVAS and the EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

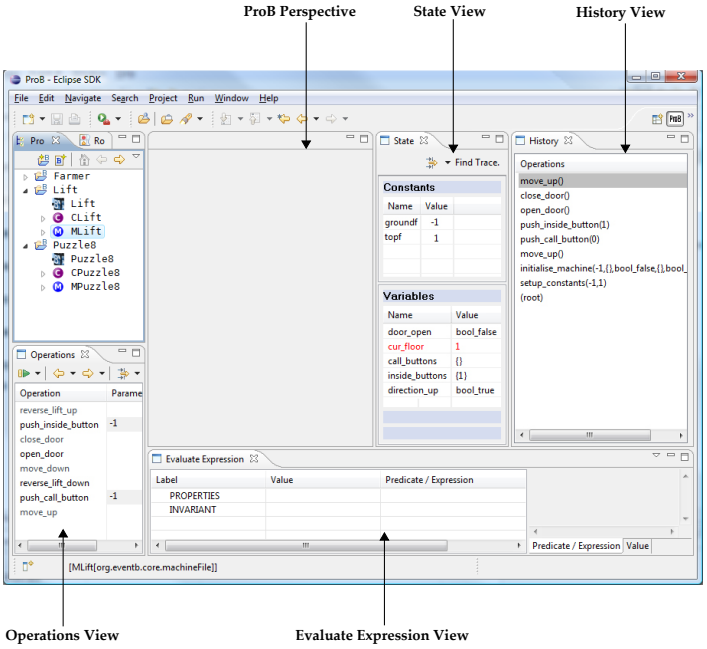


Fig. 1. The interface of the PROB plugin for Rodin

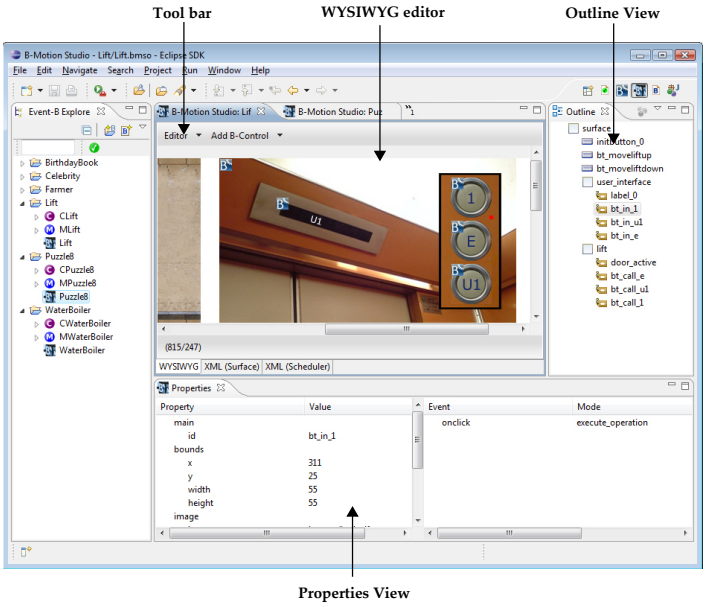


Fig. 2. The interface of the BMotion Studio tool

Acknowledgements Some of the recent developments of ProB mentioned in this talk have been developed and implemented by Jens Bendisposto, Fabian Fritz, Lukas Ladenberger, and Daniel Plagge. My thanks also go to the many people of have helped develop the ProB toolset, some of which are Michael Butler, Michael Jastram, Marc Fontaine, Corinna Spermann, and Edward Turner.

References

1. J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In *ICFEM06*, LNCS 4260, pages 588–605. Springer, 2006.
2. Y. A. Ameur, F. Boniol, and V. Wiels, editors. *ISoLA 2007, Workshop On Leveraging Applications of Formal Methods, Verification and Validation, Poitiers-Futuroscope, France, December 12-14, 2007*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*. Cépaduès-Éditions, 2007.
3. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
4. F. Fritz. An object oriented parser for B specifications. Master's thesis, Institut für Informatik, Universität Düsseldorf, 2008. Bachelor's Thesis.
5. L. Ladenberger. A visual editor for B-animations. Master's thesis, Institut für Informatik, Universität Düsseldorf, 2009. Bachelor's Thesis.
6. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
7. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
8. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
9. M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings ICFEM 2008*, LNCS, pages 278–297. Springer-Verlag, 2008.
10. M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, January 2007.
11. D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.
12. E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.

New features of Atelier B 4.0

Antoine Requet

ClearSy, Aix en Provence, France
antoine.requet@clearsy.com

Atelier B is a tool supporting the use of the B method for specification and software development. The last version has been released in January 2009, and introduces some new major features. This talk presents the most important new features: the new GUI, that provides an improved interactive prover and an editor, the support of the event-B language, and the BART refinement tool, that allows to automatically generate implementations using refinement rules. Apart from those technical improvement, this new version also introduce a new licensing policy, that provides a free version of the tool, and releases some of the components as "Open Source".

Sculpturing Event-B Models with Rodin: *Holes and Lumps* in Teaching Refinement through Problem-Based Learning

J. Paul Gibson, Eric Lallet, Jean-Luc Raffy

Le département Logiciels-Réseaux (LOR), Telecom & Management SudParis,
9 rue Charles Fourier, 91011 Évry cedex, France
(L'Unité Mixte de Recherche -SAMOVAR- UMR 5157)
pgibson@it-sudparis.eu

Abstract. We present a Problem-Based Learning (PBL) approach to teaching formal methods, using Event-B and the Rodin development environment. This approach has arisen out of a gradual adoption, over a period of 3 years, of Rodin as the main teaching tool. Just as the concept of refinement is fundamental to what we are trying to teach, we demonstrate that it is also fundamental to the teaching process. Through analysis of a small number of PBL case-studies we argue that the changes to our teaching, supported by Rodin, have started to have a positive impact on our students meeting the specified learning objectives (course requirements). However, we also argue that much more work needs to be done in order to improve our teaching of formal methods. Inspired by the analogy between software design and sculpture, we conclude by proposing that formality holds the key to mastering the harmony between the "holes" and the "lumps" in our models.

Sculpture is the art of the hole and the lump.
[Auguste Rodin, 1840 – 1917]

1 Background: teaching Event-B with Rodin

The Event-B Rodin development environment[1] is central to our teaching of formal methods. The openness of the platform, combined with our research experience, motivated us to adopt it as early as possible in our teaching. Before Rodin, we had experience of teaching a variety of formal methods; with more recent teaching using B[2] and the Atelier-B tools. Another motivating factor is that our students are all familiar with the Eclipse[3] platform, on which Rodin is built, and this helps them overcome initial feelings of unfamiliarity which often arise from using formal methods tools for the first time.

In this paper we focus our discussion on the impact of our adoption of Rodin within a single optional module called *Langages formels et applications*. (We note that the problems have also been used with other student groups and in other institutions.) Within this module, 21 hours of teaching (direct contact between lecturer and students) is programmed specifically for teaching Event-B, and the

students are also required to carry out a similar amount of self-study in their own time. The class is small — typically between 8 and 16 students. All students have already studied at least 1 year of computer science/software engineering, including foundational mathematics and programming.

In figure 1, we show how we have gradually adopted Rodin over a period of three years. In the first year, we prepared lecture slides based on Event-B case studies and tutorials that were available at the Rodin website. We also incorporated case studies inspired by formal methods research in different problem domains, for example: E-voting systems[4, 5], Distributed reference counting algorithms[6], Tree-structured File Systems[7] and the IEEE 1394 leader election protocol[8].

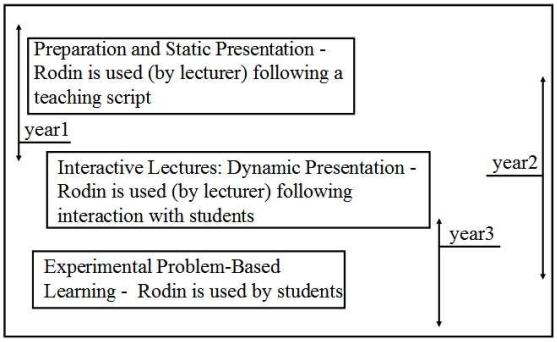


Fig. 1. The Gradual Introduction of Rodin for teaching Event-B

We re-wrote previous practical work (moving from B to Event-B in the process) so that our models could better demonstrate the facilities provided by Rodin. We did not expect students to be able to use the Rodin tool: we were just becoming familiar with it ourselves. However, we did present Event-B models, refinement sequences, and proofs that had been prepared using Rodin. We followed a traditional teaching model where fundamental theory was presented before practical case studies. As a final step towards preparing for our second year of teaching with Rodin, a single lecture was presented in a more interactive style. The students were much more responsive to the lecturer "struggling to bend the tool to their way of thinking about the problems" (as they described it), rather than the lecturer presenting a solution that had been "baked earlier". Consequently, we decided that in the following year we would try to work on all the case study problems in a more interactive way.

In our second year, we continued to start the module with traditional lectures on foundational theory. However, the second half of the module was used to let the students interact with the Rodin tools (indirectly through the lecturer). Rather than the lecturer presenting models and proofs that had already been developed, the lecturer presented the problems to the students and then attempted to show the students how they could "test their solutions" by modelling them in

Event-B and analysing them using Rodin. At the end of the second year we ran a single PBL session with the students. They were given a problem where they had to design a solution in Event-B. They did not use the Rodin tool during the design problem; but after they had submitted their designs the lecturer demonstrated how the Rodin tool could have helped them to produce better (correct) designs. The students enjoyed the PBL approach — rather than learning some piece of theory from traditional lectures the students wrestled with a problem and discovered that they lacked some fundamental understanding that would help them to solve the problem. This approach is excellent for motivating the students, when the problems are well suited to the students meeting the learning objective. However, there is a great risk that the students do not learn simply from being exposed to the problem. Based on our previous work with PBL, we decided that the potential rewards of more fully adopting PBL outweighed the risk.

In the third year we did not start with any traditional lectures. On the first day the students installed Rodin and started to experiment with the tool. Initially, they built very simple models following the directions of the lecturer. However, quite quickly they stopped asking questions of the form "what would happen if we did this instead" to trying to find out, using Rodin, the answers themselves. At the end of each class (of duration 3 hours) the students would be advised to read material that would explain how/why the tool was reacting to their experimentation. They would also be given (optional) practical work that forced them to reflect on what they had learned during the session.

As we write this paper, we are half-way through the 3rd year of teaching Rodin. It is too early to analyse the global impact of our pedagogic changes: B to Event-B, using Rodin and PBL. In fact, as the three changes are very much interdependent — and there are many other more noisy parameters to take into account — it will be difficult to draw specific conclusions as to what causes any improvement (or deterioration) in our students' learning.

2 The PBL Teaching Method

Although there are many different definitions of PBL, the common factor in every one of them is that the problem acts as the catalyst that initiates the learning process. It is said that this way of learning encourages a deeper understanding of the material, rather than surface learning. As the problem is such a critical component of the learning process it is imperative that one uses *good* problems. In 2001, Duch identified five characteristics of what makes a PBL problem *good*[9]: (1) Effective problems should engage the students' interest and motivate them to probe for deeper understanding. (2) PBL problems should have multiple stages. (3) Problems should be complex enough that group co-operation will be necessary in order for them to effectively work towards a solution. (4) Problem should be open-ended. (5) Learning objectives of the course should be incorporated into the problems.

One of the major stumbling blocks to the implementation of PBL within any discipline is the lack of a good set of problems[10]. However, the discipline of soft-

ware engineering and formal methods has many well-understood problems that have arisen out of industrial and research projects. Lecturers must be encouraged to use these problems (or parts of them) in their teaching. A recent proposal for weaving formal methods, through a software engineering programme, using problem-based learning (PBL)[11] provides good background and motivation for such a teaching approach in a software engineering programme, based on observations on how students solve problems using foundational software engineering techniques[12].

We note that the 5th of Duch's characteristics for a good PBL problem is defined in terms of learning objectives. Without explicit statement of learning objectives it is difficult, if not impossible, to evaluate and analyse the effectiveness of PBL, in general, and specific problems, in particular.

3 Learning Objectives

A main weakness when teaching design is that students fail to understand that design is a dynamic process and not just a sequence of models. This is particularly important when teaching formal methods through design problems.

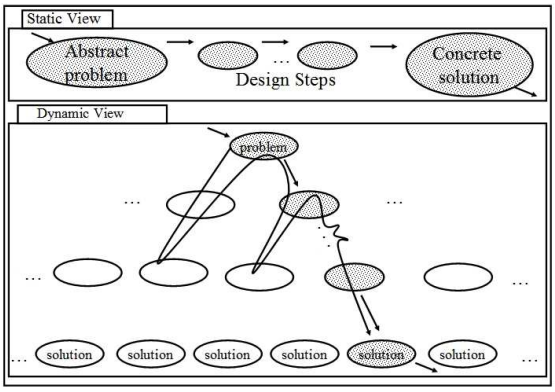


Fig. 2. Design as A Dynamic Process: The Learning Objectives

Figure 2 provides a graphical view of our main learning objectives when treating design as a process: (1) to be able to build models at different levels of abstraction, (2) to be able to prove that models at all stages of development are well defined, (3) to be able to validate that models capture precisely the needs of the client, (4) to be able to verify that each design step (from abstract to concrete) is correct, and (5) to be able to manage the process of rolling back a design decision.

We note that these objectives are generic in the sense that the modelling language and modelling tools are unspecified. Our secondary learning objectives are that the students are able to meet the main objectives when modelling with Event-B, and that they are able to use the Rodin tool for (automated) support.

3.1 Model at different levels of abstraction

Students are expected to already know about nondeterminism. We expect them to be able to build requirements models that use nondeterminism to facilitate implementation freedom. Our main objective is for them to be able to remove such nondeterminism through refinement. Further, they should learn how to reverse engineer a concrete model to something more abstract.

3.2 Well-defined Models

Students are expected to already know, in general, how theorem provers work. They are also expected to be able to carry out simple mathematical proofs by hand. Our learning objective is for them to be able to apply this knowledge when using the automated theorem proving support provided by Rodin. Our goal is to provide a problem through which the students learn about well-definedness and also learn, through experimentation, how Rodin can be used to prove that Event-B models (and parts of the models) are well-defined.

3.3 Validation — testing understanding of requirements (formally)

Students are expected to already understand that many problems occur in software engineering because of poorly understood requirements[13]. They are also expected to know that most common validation techniques involve testing an executable system (often a prototype) with the client; and that there are weaknesses to this approach. Our goal is to show that formal models offer an approach to validation that is complementary to executing code. We aim to provide them with a problem where the students quite naturally test their understanding of requirements through the formulation of theorems to be proved.

3.4 Correct Design

Students are expected to already have studied design (usually with a modelling language like the UML[1]). They must understand the role of design in bridging the gap between the problem (requirements) and the solution (implementation). Our goal is that students learn what it means for a design to be correct[14], and that they can use RODIN to prove three fundamental properties of a machine: (1) Invariants are respected. (2) Termination (where required). (3) Deadlock freeness (for interactive systems).

3.5 Design As A Process

A key objective is that students understand that design is a dynamic process, represented by a tree of decisions and compromises. (We also hope that the

[1] It is pleasing to note, for potential future exploitation in our teaching, the research and development of a Rodin plug-in for integrating Event-B and the UML[1]

students see that this tree representation is an abstraction of what happens in real software development.) The design documentation should not just be a record of the sequential final path in this tree that links the problem to the particular chosen solution: it should be a record of every design decision that was taken and why (including the decisions that were changed, i.e. rolled back). Further, the students must learn that there is as much value in the links in the design tree (which represent the correctness of the design steps) as there is in the nodes (the models).

4 Problems Presented

In this section we review a subset of the problems that have been presented to the students in order to meet specific learning objectives. We comment on students' behaviour whilst interacting with the problems, with particular emphasis on their use of the Rodin tools.

4.1 Well-defined Models: The Purse Problem

In our search for interesting problems, it was noted that the notion of a wallet (of money) had proven to be a good pedagogic case study[15]. This inspired us to consider a purse as the basis for our PBL case study. The following requirements were presented to the students:

1. A purse contains coins.
2. Coins are positive integers, but not all integers have a corresponding coin.
3. We wish to start with an empty purse, containing no coins.
4. We allow 3 operations: (a) initialise a purse to being empty (containing no coins), (b) add a coin, and (c) pay a certain (integer) sum by removing an appropriate number of coins from the purse.

Figure 3 shows a graphical representation of the problem that was presented to the students to complement the textual requirements.

It is interesting to note how the students tried to model the `Purse` using Event-B. Firstly, we witnessed the problem of confusing sets with bags as discussed by Habrias[15]. Once students realised that the problem required more than a set of coins (represented as integers) most of them they quickly defined a `Purse` to be a total function from coins to integers. (Some students also chose to specify `Purse` as a partial function from coins to integers, arguing that if a coin was not in the domain then there were no coins of that value in the `Purse`.) The students struggled to specify a generic `Purse`, parameterised by any set of coins. They knew that this type of specification should be possible but had to be shown how to specify this using an abstract `COIN` set. It was pleasing to see that many of the students then specified the notion of an empty purse in a similar, generic fashion, as shown in figure 4.

Most students then thought about the operation for paying a certain sum and decided that it was too difficult to specify directly. They were encouraged to

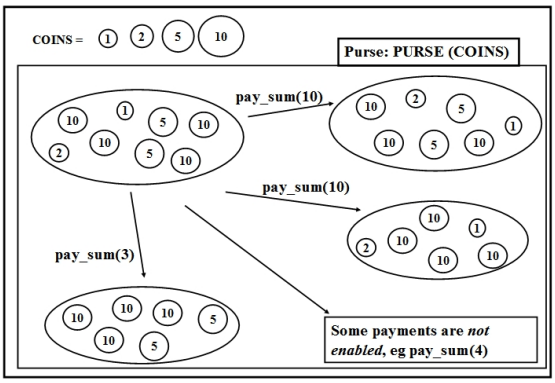


Fig. 3. The Purse **pay_sum** Behaviour

```
COINS ⊆ N1
PURSES  = COINS ⟶ N
emptyPURSES ⊆ PURSES
noCOINS ∈ emptyPURSES
emptyPURSES = { z | z∈ PURSES ∧ dom(z) = COINS ∧ ran(z) = {0} }
```

Fig. 4. A generic specification of an empty purse

think about it in an abstract, nondeterministic, fashion. However, most of them thought that this meant decomposing the payment into component parts. One of the most common ways of doing this was for students to specify the notions of "total" and "remove" (two key terms found in the textual requirements). An example of how a student specified `total` is shown in figure 5.

```
total ∈ PURSES ⟶ N
∀ep· ep∈ emptyPURSES ⇒ total(ep) = 0
∀ p, c,s· p∈ PURSES ∧ c ∈ COINS ∧ s∈ N ∧ c↦s∈ p ⇒ total(p) = c∗s + (total (p ⩤ {c↦ 0}))
```

Fig. 5. The introduction of a function to calculate the total

At this stage, the lecturer pointed out that the Rodin tool was generating proof obligations with regard to the well-definedness of their `total` specifications, as shown in figure 6.

The students experimented with the Rodin tool in order to see which proofs were discharged automatically and which required interaction. Although they did not know how the prover worked (and had received no lectures on the subject)

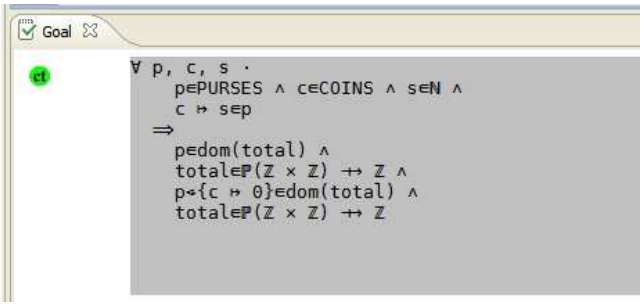


Fig. 6. The proof obligation generated for the specification of total

they were able to carry out some proofs simply by "randomly" clicking and instantiating.

By encouraging students to examine different specifications of `total` it often arises that students ask how they can test that their specifications "really work". In essence, they are asking how they can validate that the meaning of `total` corresponds to the requirements. At this stage the lecturer suggests that they formulate simple use cases. The students are able to express the fact that they want to test, for example: (1) the total of any empty purse must be 0, and (2) the total of a purse containing two 1c coins should be 2. It was surprising (to us) that, in general, students manage to express these as theorems only after receiving help from the lecturer, as in figure 7. The main problem was due to us not having yet covered the foundational material explaining fundamental concepts such as axiom, theorem, proof, completeness, consistency, etc ...

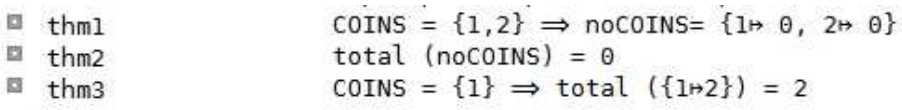


Fig. 7. Using theorems to validate understanding and specification

In the next example, of the family tree, we expect the students to follow a similar validation process.

4.2 Validation: The Family Tree Problem

The problem of specifying relations between people in order to identify families has been used by a large number of lecturers. Our goal is to use the same example but to force the students to work within a particular view that needs to be validated: we consider only relations between humans that are alive, and

we wish to enforce that each person can have 0,1 or 2 parents that are still alive. We present the problem by the tree shown in figure 8.

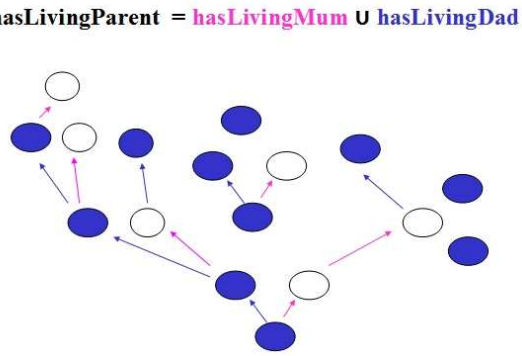


Fig. 8. Graphical representation of the required Parent behaviour

The students, having learned from their experience of the `Purse` problem validate their parent specifications — an example is given in figure 9 — with simple theorems.

axm_hommes	hommes ⊆ P
axm_femmes	femmes = P \ hommes
axm_hasLM	hasLivingMum ∈ P ⇸ femmes
axm_hasLD	hasLivingDad ∈ P ⇸ hommes
axm_hasLP1	hasLivingParent ∈ P ↔ P
axm_hasLP2	hasLivingParent = hasLivingMum ∪ hasLivingDad

Fig. 9. Formal Specification of Parent Requirements

It was interesting to note that the students went on to specify relations like `brother`, `cousin`, `aunt`, etc .... However, none of them validated (or attempted to validate) that their family tree did not contain any cycles!

4.3 Correct Design: The Purse Revisited

In the process of specifying the `Purse` behaviour we noted that the first design step — of pairing a machine with a context — led to some interesting design decisions. For example, we saw two different specification styles — see figure 10 — for events that update the state of the purse, by adding and removing coins. Some students used a style where the state updates of the machine were specified

axiomatically in the context, for example: the `add_coin` event uses an `add` function that has been specified in the context `Purse_ctx0` (see figure 11). Whilst others used a more operational style (as for event `remove`).

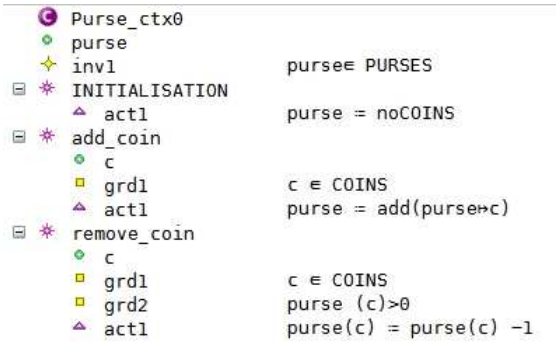


Fig. 10. Adding and Removing a coin from a Purse

```
add∈ PURSES × COINS ⟶ PURSES
∀ p1, p2, c· p1∈ PURSES ∧ p2∈ PURSES ∧ c ∈ COINS ∧ ((p1↦ c)↦ p2) ∈ add ⇒  p2(c) = p1(c)+1
```

Fig. 11. The add function defined in the Purse context

Without going into details, this approach requires additional work when proving the correctness of the context, but leads to a simple proof that the invariant is respected by the `add_coin` event (in the machine). Contrastingly, the `remove_coin` event's action is specified directly (without an additional "worker" function from the context). This approach means that the proof that `remove_coin` respects the invariant cannot re-use any properties of `remove` that could have been specified in the context.

4.4 Design As A Process: The OddEven Problem

We present the students with the problem of specifying `Odd` and `Even` numbers. We tell them that these specifications will be required in a later machine, but do not tell them precisely how. Even for such a simple problem, students typically produce different specifications. For example, see figures 12, 13 and 14.

We observed that the students, whilst in the process of proving properties of the subsequent machine, chose to change the way in which they specified `Odd` and `Even`. When asked about this, one of the more interesting replies was: "We

```
CONTEXT
  OddEven_ctx0
SETS
  set1
CONSTANTS
  EVEN
AXIOMS
  axm1   :   EVEN ⊆ ℕ
  axm2   :   ∀x · x ∈ EVEN ⇔ (∃ y · y∈ ℕ ∧ x = y+y)
THEOREMS
  thm1   :   ∀a, b · (a ∈ EVEN ∧ b ∈ EVEN ⇒ (a+b) ∈ EVEN)
```

Fig. 12. A first specification of Odd and Even

```
CONTEXT
  OddEven_ctx1
CONSTANTS
  double
  even
  odd
AXIOMS
  axm1   :   double ∈ ℕ ⤖ ℕ
  axm2   :   ∀ x· x∈ ℕ ⇒ x↦ (x+x) ∈ double
  axm3   :   even = ran (double)
  axm4   :   odd = ℕ \ even
```

Fig. 13. A second specification of Odd and Even

have learned the importance of structuring our code to make it easier to test; so why not restructure our specifications to make them easier to verify?".

4.5 Refinement and Design: The "Centralised Leader Election" Problem

The very last problem that we presented to the students had the objective of testing whether they were able to reason about the correctness of different designs through refinement of an abstract machine. The problem presented to them was a simplification of leader election:

> Given a team of players, there must be a single unique captain. There is a single event which corresponds to changing the captain. Propose alternative designs to implementing this simple system. Use the Rodin toolset to reason about the correctness of the designs.

We have observed four different types of "solution" to the problem: (i) Initial Machine too concrete and no refinement, (ii) Initial Machine too concrete and a correct refinement, (iii) Initial Machine at appropriate level of abstraction but unable to specify design as a refinement, and (iv) Initial Machine at appropriate level of abstraction and alternative correct designs specified as refinements.

```
CONTEXT
  OddEven_ctx2
CONSTANTS
  Even
  Odd
AXIOMS
  axm1   :   Even ⊆ ℕ
  axm2   :   Odd = ℕ ∖ Even
  axm3   :   0∈ Even
  axm4   :   ∀x· x∈ Even ⇒ x+1 ∈ Odd
  axm5   :   ∀x· x∈ Odd ⇒ x+1 ∈ Even
```

Fig. 14. A third specification of Odd and Even

Approximately half the class started with machines that were too concrete in the sense that they precluded some reasonable implementations. In all but one of these cases, the students who started with a concrete design were unable to refine it. Consequently, they reasoned about the design correctness informally. None attempted to reverse engineer a more abstract machine.

In a single case, the students started with a team of players where each player had an integer counting the number of times they had been selected as captain. They specified the (invariant) requirement that no player could have a `count` more than 1 larger than any other player's `count`. Then, in the abstract machine they specified that the change captain event never picked a captain whose `count` was already bigger than any other player's `count`. They then refined this event by introducing a captain `queue` where selecting a new captain popped the current captain from the front of the `queue` and pushed this player onto the back of the `queue` — so that the sequence in which captains were chosen would follow a repetitive cycle. This machine is a refinement of the first as it removes the nondeterminism in the event which chooses the captain (after the first cycle).

The remainder of the class specified the initial abstract machine at an appropriate level of abstraction. Two groups chose to specify simple solutions where the position of captain always alternated between the same 2 team members. Although this is, perhaps, the simplest design, neither of these groups were able to demonstrate that this design is correct. They attempted to prove the more concrete machine to be a refinement of the abstract machine, but failed.

The best solutions offered alternative designs and demonstrated their correctness. None of these offered a sequence of refinement steps (but this was not explicitly asked of the students). It was disappointing that only about one third of the students were able to address the problem in this way.

5 Refining our Teaching: what needs to be changed?

In order to improve our teaching formal methods[2] it is important that we learn from the students[16]. With PBL, in particular, one must take great care when

[2] More generally, student feedback is crucial in improving the teaching in any discipline.

using quantitative and qualitative analysis to evaluate the effectiveness of the problems[17].

Much like software engineers who refine their models for implementation, formal methods lecturers need to refine their teaching models. When there is a mismatch between what is required and a proposed solution then there are three possibilities: (1) the solution is correct with respect to the requirements specification yet the specification misrepresents the requirements, or (2) the specification correctly represents the requirements but the solution is not correct with respect to the specification, or (3) a combination of the two previous possibilities. In general, when a solution is acceptable it is because the initial requirements were correctly specified and the implementation was correct with respect to these requirements. (In theory, it may be possible that the solution meets the requirements despite the fact that the specification is incorrect. However, this situation is not desirable even though the client may be happy in the short term!)

There are several options when a problem is not meeting a specific learning objective: (1) Replace the problem with something completely different. (2) Fix the problem by making minor changes. (3) Change the learning objective.

This feedback into our teaching is critical in PBL but it is problematic because: (1) The frequency of change is usually tied to the academic calendar. (2) The mapping relation between learning objectives and problems is not (usually) a bijection, though it should be a total surjection. (3) Analysis of the effectiveness of problems should be done using more than 1 class of students. (4) Developing new problems is time-consuming. The simplest way to overcome these issues is to share and re-use problems between different lecturers and programmes.

Once a problem has been developed that is deemed to be effective, it is very important that one does not break its effectiveness through making change. Lecturers would greatly appreciate a formal notion of refinement with respect to their teaching material. Thus, they could make (verifiable) changes to existing problems knowing that such changes do not compromise their effectiveness (at meeting the learning objectives). Of course, this is currently beyond the state-of-the-art in educational research! However, as teachers we must aspire to achieving such refinements of our problem designs: it will improve our teaching and reduce our workload.

6 Holes and Lumps in our Event-B Models

Event-B models can be judged on a number of different criteria: (1) Well-definedness, which can be checked without knowing the intended purpose of the model. (2) Fitness for purpose, which can be checked against required behaviour. (3) Level of abstraction, which reflects whether design decisions are being taken too early/late in the development process. (4) Maintainability/Reusability, which represents how much of the modelling work (including the proofs) can be re-used if requirements change.

Design is not a prescriptive process. Students need to learn that building good designs (in Event-B) requires experience, good judgement and good fortune. Many of the students produce poor designs because their models are too

rich in detail. They miss the importance of keeping things simple. A key insight is that the best students are quite comfortable with leaving details to later stages in the development process. These "holes" correspond to abstraction. Subsequent refinements may (partially) fill in the holes. The most valuable Event-B designs are those that bridge the gap between the requirements specifications that are relatively easy to model, using lots of nondeterminism, and the determinisitic models that are directly implementable, using traditional programming languages. Our experience shows that students can quite easily produce Event-B models at these extremes of the abstraction continuum but find it very difficult to produce the intermediate design steps (that are necessary in establishing correctness).

A second issue that needs to be addressed is one of composition. Event-B, due to its refinement mechanism, has proven to be successful in teaching correctness-by-construction. However, we have fears that it is not so well-suited to reasoning about composition of systems. In fact, our students often commented on having difficulties in adding functionality (features) to already developed (and verified) machines. Further, they observed that they would like to be able to synchronise machines[3] through some form of shared events. They also claimed to miss the high level composition mechanisms that they were used to having in their favourite OO programming languages.

Design is also about knowing what needs to be added and where. Modelling and managing these "lumps" is a learning objective that we have yet to try and meet (when teaching Event-B with Rodin). There has been research in extending Event-B with richer composition mechanisms, but we are not yet ready to use them in the classroom.

7 Conclusions

We believe that our PBL case-studies, using Rodin, are improving the way in which we teach formal methods: (1) Students are happy to experiment with their models and proofs. (2) Students are more motivated by working on problems — and often spend much more time than required on self-study. (3) The students were able to better understand the foundational material presented to them (in traditional lecture format) as they could relate the theoretical concepts to the operation of the Rodin tools — with particular interest in how the provers work when discharging obligations automatically, and how to best carry out proofs interactively.

However, we need to build a more extensive problem set, and improve our feedback mechanisms for evaluating and improving problems. A major issue is the specification of our learning objectives: if we want to share problems then we need to be able to find common agreement on learning objectives. Such agreement would be complementary to the development of a formal methods body of knowledge[19]. Inspired by the analogy between software design and sculpture, we conclude by proposing that formality holds the key to mastering the harmony between the "holes" and the "lumps" in our models.

[3] This is similar to the integration of CSP and B[18].

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: A roadmap for the Rodin toolset. In: Abstract State Machines, B and Z, First International Conference ABZ 2008. Volume LNCS 5238. (September 2008) 347–351
2. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge Univ. Press (1996)
3. des Rivières, J., Wiegand, J.: Eclipse: A platform for integrating development tools. IBM Systems Journal **43**(2) (2004) 371–383
4. Cansell, D., Gibson, J.P., Méry, D.: Formal verification of tamper-evident storage for e-voting. In: Software Engineering and Formal Methods (SEFM 2007), IEEE Computer Society (2007) 329–338
5. Cansell, D., Gibson, J.P., Méry, D.: Refinement: A constructive approach to formal software design for a secure e-voting interface. Electr. Notes Theor. Comput. Sci. **183** (2007) 39–55
6. Cansell, D., Méry, D.: Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm. Theor. Comput. Sci. **364**(3) (2006) 318–337
7. Damchoom, K., Butler, M., Abrial, J.R.: Modelling and proof of a tree-structured file system. In: ICFEM 2008. Volume LNCS 5256., Springer (October 2008) 25–44 Springer LNCS 5256.
8. Rehm, J., Cansell, D.: Proved development of the real-time properties of the ieee 1394 root contention protocol with the event b method. In Ameur, Y.A., Boniol, F., Wiels, V., eds.: ISoLA. Volume RNTI-SM-1 of Revue des Nouvelles Technologies de l'Information., Cépaduès-Éditions (2007) 179–190
9. Duch, B. In: Writing Problems for Deeper Understanding. Stylus Publishing, Sterling, Virginia (2001) 47–53
10. Tien, C., Chu, S., Lin, Y.: Four phases to construct problem-based learning instruction materials. In: PBL In Context: Bridging work and Education, Tampere University Press (2005) 117–133
11. Gibson, J.P.: Weaving a formal methods education with problem-based learning. In: 3rd Int. Symposium on Leveraging Applications of Formal Methods, Verification & Validation. Volume 17., Springer-Verlag, Berlin Heidelberg (2008) 460–472
12. Gibson, J.P., O'Kelly, J.: 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, ACM (2005) 87–97
13. Gibson, J.P.: Formal requirements engineering: Learning from the students. In: Australian Software Engineering Conference, IEEE Comp. Soc. (2000) 171–180
14. Gibson, J.P., Lallet, E., Raffy, J.L.: How do I know if my design is correct? In: Formal Methods in Computer Science Education (FORMED). (March 2008) 59–69
15. Habrias, H.: Teaching specifications, hands on. In: Formal Methods in Computer Science Education (FORMED). (March 2008) 5–15
16. Gibson, J.P., Méry, D.: Teaching formal methods: Lessons to learn. In Flynn, S., Butterfield, A., eds.: IWFM. Workshops in Computing, BCS (1998)
17. O'Kelly, J., Gibson, J.P.: PBL: Year one analysis — interpretation and validation. In: PBL In Context — Bridging Work and Education. (2005)
18. Butler, M.J., Leuschel, M.: Combining csp and b for specification and property verification. In Fitzgerald, J., Hayes, I.J., Tarlecki, A., eds.: FM. Volume 3582 of Lecture Notes in Computer Science., Springer (2005) 221–236
19. Oliveira, J.N.: A survey of formal methods courses in European higher education. In Dean, C.N., Boute, R.T., eds.: TFM'04. Volume 3294 of Lecture Notes in Computer Science., Springer-Verlag (2004) 235–248

Teaching the B Method at Oxford Brookes

David Lightfoot and Clare Martin

Oxford Brookes University, UK

Abstract. The B Method has been taught to MSc students at Oxford Brookes University for over a decade. This year it has been introduced into the undergraduate curriculum for the first time, partly because of the recent advances in tool support for the language. In this paper we reflect on our teaching experiences and briefly describe how we would like to use the B Method in future to mechanise proofs in one of our research areas.

1 Introduction

Formal methods have formed an integral part of both the undergraduate and postgraduate curriculum at Oxford Brookes University for many years. Whilst much of the theoretical material taught has remained static during this time, changes in tool support, and in computer applications have continued to keep the relevant modules fresh and challenging to teach. Student feedback has shown that students enjoy learning about formal methods, and this has been reflected by their results, and in some cases, their chosen careers.

2 The B Method in the Software Engineering Curriculum

Until recently, the B Method has only been included in the postgraduate curriculum, as one of the optional modules on a number of MSc courses, including Software Engineering. This year, however, it will also appear in the undergraduate curriculum for the first time, as part of the Formal Specification module, which has previously been restricted to teaching Z. This module is optional for a number of the undergraduate degree courses, including Software Engineering and it is compulsory for the Computer Science (Mathematical) degree.

3 Textbooks

The main textbook is the one by Schneider, [S01], which is used in conjunction with some of the teaching resources from the associated website. Students are also referred to the B-Book [A06], which is in the library along with various other textbooks on B.

4 Prerequisites

Much of the mathematical content has been removed from the MSc course, and the only knowledge that can be assumed is that given during a brief maths primer at the start of the course. Many students do have a strong mathematics background, but it is still necessary to include a revision of basic set theory and logic at the start of the B module for the benefit of the weaker students. It is therefore useful to include a variety of simple and motivating exercises in B, using very restricted notation, in order to keep the stronger students interested and to help them understand the difficulty of capturing requirements precisely using mathematics.

5 Syllabus

In common with other modules in the department, the B module consists of eleven two-hour lectures, each followed by a one-hour practical. A typical schedule of topics is as follows:

Week	**Topic**
1	Introduction to module; Introduction to B
2	Specification using B
3	Fragile and robust operations
4	Worked example with audience participation
5	Relations
6	Functions and Sequences
7	Proof Obligations
8	Case Study
9	Refinement
10	Implementation
11	Revision

6 Tool support for software engineering with the B method

For the past thirteen years, the B module has been taught at Brookes using the B Toolkit [B], but this has always been slightly problematic, partly because most of the computer rooms do not run Unix. So this year we decided to experiment both with Rodin [R] and Atelier B [A]. Of these two tools, Atelier B seemed much easier to get to grips with, so this was the tool we chose for this year's run of the module. Both tools have the advantage of running under Windows as

well as Unix, so Atelier B has now been installed in all of the university pooled computer rooms, thus facilitating its use for undergraduates.

The transition from the B Toolkit to Atelier B has been almost seamless. Only minor changes have been made to the standard practical exercises and solutions because of some of the differences listed below. These differences were found, among others, on a website at Uppsala University [U], which was extremely useful:

- In the B-Toolkit, implementations assign values to deferred sets and (concrete) constants using the PROPERTIES clause. In Atelier B, the VALUES clause is used instead.
- Atelier B requires that the parameter list of an abstract machine is repeated in any refinements of that machine, including an implementation. However, the CONSTRAINTS clause should not be repeated.
- Atelier B does not allow parameters to be used in the PROPERTIES clause.

Two features of the B Toolkit that are missing from Atelier B are animation and Latex output of machines, proof obligations and proofs. These features are useful for teaching, but their loss is outweighed by the other benefits of Atelier B.

7 Experiences with Formal Methods teaching using B

The group of students who have chosen to take the B module has usually been quite small, partly because of timetabling restrictions. This has contributed to the pleasure of teaching the module, but inevitably raises questions about its viability in the future.

Teaching Methods

The primary teaching method is the traditional lecture, and all notes and practical exercises are supplied prior to the lecture, with model answers being distributed subsequently, after the students have attempted the exercises. The practical class is usually scheduled immediately after the lecture, which is convenient, but has the disadvantage that the students do not have time to go away and work through the exercises before discussing them with a tutor. All of the course materials are stored on the module website, which includes

- lecture notes
- practical exercises
- model answers to exercises
- past exam papers

– resources, such as the B reference card and the interactive prover manual
– links to associated websites
– articles on formal methods

The lectures tend to be quite interactive, in order to sustain interest from the students. For example, the basic exercise in Section 9 is sometimes used during a lecture at the start of the course, as an interactive modelling exercise. If time permits, the practical exercises are attempted in groups during the lecture, using only pencil and paper, and then discussed, before moving on to the computing laboratory to test them out on the tool. For example, the exercise on refinement in Section 9 might be discussed in class before the practical because the refinements of the operations might not initially be obvious to the students.

Although the main emphasis of the course is on mathematical modelling, the topic of proof is also covered in some depth. The mixed mathematical ability of the students means that it is not feasible to assess complicated handwritten proofs, but instead the students are taught to understand the two fundamental concepts:

– the initialisation must establish the invariant
– every operation must preserve the invariant

These principles are explained in the context of weakest preconditions, following [S01]. Students are shown some simple proofs in the lecture, and required to complete some short in-class exercises on proof, but the main objective is to make them understand what they are supposed to prove, rather than how to prove it. This is where the use of a tool becomes essential, since it is relied upon to generate the detailed proof obligations and discharge them. If the tool fails to prove anything, then the students should be able to recognise whether this is because of an error in the specification, or because it is necessary to use the interactive prover. They are given practical exercises to test their ability with this, and it is also assessed through their coursework. In general, students seem to begin to fully appreciate the value of a tool once they have experienced its ability to locate the kind of errors that result in the correctness of a system not being provable.

Guest Lectures

One novel feature of the course over the years has been the inclusion of guest lectures given by experts in the field. Experience has shown that the students find these highly motivating. Some examples from recent years are listed below:

– Yann Bruere: Real Life B - how specification is used to develop safety critical systems
– Michael Butler: Combining UML and B
– Ib Sorenson: The Specification of an Automatic Train Control System

Difficulties

In general the course runs quite smoothly, but it is not without its difficulties. Some of the problems listed below are common both to the B and Z modules.

- The main problem has been recruitment of students to the module. It is not seen to be as essential as the more practical modules, and but the students who do take it usually appreciate its value and give positive feedback.
- The mixed mathematical ability of the students can make some aspects of the module difficult to teach. Many students are overwhelmed by the volume of notation associated with relations and functions, and find it difficult to express constraints mathematically. The solution to this is to provide them with a large variety of exercises on which to practise. Supplementary exercises are often added to the module website to allow them to broaden their knowledge.
- It can be difficult to make the module seem relevant to the modern world, but this problem is usually addressed by explaining the importance of safety critical systems. The guest lectures also help. Some of the exercises are chosen to feature modern applications with which the students are familiar, such as the facebook example in Section 9. Other examples have included popular websites like Amazon and ebay.
- There are also problems with implementation machines because it is beyond the scope of the course to cover them in any depth, and students can be frustrated by the simplicity of the examples that they are taught to implement.

Assessment

Students are assessed by 30% coursework and 70% exam. For the coursework, the students are asked to produce a specification and refinement of a system of their own choice. This freedom both increases their motivation and eliminates plagiarism. Students are expected to run their specification through the tool and try to discharge all the proof obligations. The exam tests their knowledge of the B language as well as their understanding of proof obligations and their awareness of formal methods in general. It is a written exam, and therefore does not test their practical ability with the tool.

Results

Students find the module interesting and well structured, and the average mark is typically high, as is the pass rate:

Year	Number of students	Average Mark %	Pass rate %
2009	5	-	-
2008	5	71	100
2007	4	76	100
2007	13	68	85

Careers

Three students in particular have chosen to continue to work with B after completing their degrees:

- Christine Poerschke (1998) went on to do a Phd at Brookes, where she used the B notation to specify formally an existing medical decision support system. The system helped patients with insulin-dependent diabetes decide on a dose of insulin to inject and the results were presented at the ZB conference in 2003 [PLN03].
- Beeta Vagar (2005) went on to do a Phd at Surrey University, under the supervision of Steve Schneider.
- Yann Bruere (2008) went on to work for Siemens Transportation Systems in Paris where he now uses the B method to verify railway systems.

8 Combining the B method with other approaches

This year we have decided to incorporate B into the undergraduate Formal Specification module, which used to include only Z. There are two reasons for this. First, the module has been upgraded to a third year module, and therefore needs more content, and second, the module has previously relied on Z/EVES for tool support, but this tool is no longer supported by its manufacturers. Proof obligations will be covered in greater depth this year than in the past, and they will be analysed using Atelier B. Students will be examined on their knowledge of B; a sample exam question which requires students to translate a B specification into Z and supply missing preconditions is included in Section 9.

9 Exercises featuring the B method

Basic Exercise

This exercise is used right at the start of the module to encourage students to think about how to capture constraints using only basic set theory. Students are given the basic machine below, which describes some of the requirements on a degree programme at a university. Here `compulsory` denotes the set of

modules that are compulsory for this degree, and `acceptable` denotes the set of modules which are acceptable for it. `staff` and `students` represent the sets of staff and students at the university respectively, and `min` is the minimum number of modules necessary to obtain an Honours degree. The students are asked to supply the following constraints:

- No person can be both a staff member and a student
- Every compulsory module must be acceptable
- The number of compulsory modules must not exceed `min`
- There must be at least `min` acceptable modules

Whilst these constraints are quite simple to formulate, examples like these help students to understand the importance of precise mathematical descriptions of systems that they are familiar with.

```
MACHINE          degree

SETS             MODULE; PERSON

CONSTANTS        min

PROPERTIES       min : NAT1

VARIABLES        compulsory, acceptable, staff, students

INVARIANT        compulsory <: MODULE & acceptable <: MODULE &
                 staff <: PERSON & students <: PERSON & ...

INITIALISATION   compulsory,acceptable,staff,students:={},{},{},{}

OPERATIONS ...
END
```

Refinement Exercise

The following machine describes a collection of aircraft at an airport. Some of the aircraft are waiting to be assigned gates for take off, and the rest are already in their gates. There are three operations: *arrive* adds a new aircraft to the set of those waiting, *assign* moves an aircraft from the waiting set to one of the gates, and *leave* removes an airport from a gate when it takes off.

```
MACHINE           Airport(limit)

CONSTRAINTS      limit : NATURAL

SETS              PLANE; GATE

ABSTRACT_VARIABLES  waiting, assignment

INVARIANT           waiting <: PLANE
                  & assignment : GATE >+> PLANE
                  & card(waiting) <= limit
                  & waiting /\ ran(assignment) = {}

INITIALISATION    waiting := {} || assignment := {}

OPERATIONS

  arrive(pp) =
  PRE
    pp : PLANE & pp /: waiting & pp /: ran(assignment) & card (waiting) < limit
  THEN
    waiting := waiting \/ {pp}
  END;

  assign(pp,gg) =
  PRE
    pp : PLANE & gg : GATE & pp : waiting & gg /: dom(assignment)
  THEN
    waiting := waiting - {pp} || assignment(gg) := pp
  END;

  leave(pp) =
  PRE
    pp : PLANE & pp : ran(assignment)
  THEN
    assignment := assignment |>> {pp}
  END

END
```

The refinement below uses injective sequences to represent both the set of planes waiting to be assigned gates and the allocation of gates to planes. Some of the refined operations are quite lengthy, and could alternatively be written using a LET statement if preferred.

```
REFINEMENT
   Airport_r(limit)
REFINES
   Airport

ABSTRACT_VARIABLES
    waitList, gatesUsed, planesAt

INVARIANT
    waitList : iseq(PLANE) & size(waitList) <= limit &
    gatesUsed : iseq(GATE) & planesAt : iseq(PLANE) &
    size(gatesUsed) = size(planesAt) &
    ran(waitList) = waiting &
    (gatesUsed~;planesAt) = assignment &
    ran(waitList) /\ ran(planesAt) = {}

INITIALISATION
   waitList := [] || planesAt := [] || gatesUsed := []

OPERATIONS
   arrive ( pp ) =
   PRE
       pp : PLANE & pp /: ran(waitList) & pp/: ran(planesAt) & size (waitList) < limit
   THEN
       waitList := waitList ^ [pp]
   END;

   assign ( pp , gg ) =
   PRE
       pp : PLANE & gg : GATE & pp : ran(waitList) & gg /: ran ( gatesUsed )
   THEN
       waitList := waitList /|\ (waitList~(pp)-1) ^ waitList \|/ waitList~(pp) ||
       gatesUsed := gatesUsed ^ [gg] ||
       planesAt := planesAt ^ [pp]
   END;

   leave ( pp ) =
   PRE
       pp : PLANE & pp : ran ( planesAt )
   THEN
       planesAt := planesAt /|\ (planesAt~(pp)-1) ^ planesAt \|/ planesAt~(pp) ||
       gatesUsed := gatesUsed /|\ (planesAt~(pp)-1) ^ gatesUsed \|/ planesAt~(pp)
   END
END
```

Sample Exam Question

The following specification in B partially represents a specification of a web based social networking system. The set `members` is the set of all members of the system, and `nets` is the set of all social networks in the system. The relation `friends` relates each person to all of his/her friends and `belongsto` maps each person to a social network. `id` denotes the identity relation which maps each person to themselves.

```
MACHINE         facebook

SETS            NETWORK; PERSON

VARIABLES       members, nets, friends, belongsto

INVARIANT       members <: PERSON & nets <: NETWORK &
                friends : PERSON <-> PERSON &
                belongsto : PERSON +-> NETWORK &
                dom(belongsto) = members & ran(belongsto) <: nets &
                dom(friends) <: members & ran(friends) <: members &
                id(PERSON) /\ friends = {} & friends = friends~

INITIALISATION  members, nets, friends, belongsto := {},{},{},{}

OPERATIONS

  addFriend(pp,qq) =
  PRE
    pp : PERSON & qq : PERSON & pp : members & qq : members &
    {pp} /\ {qq} = {} & (pp,qq) /: friends
  THEN
    friends := friends \/ {(pp,qq), (qq,pp)}
  END;

  nn <-- numFriends(pp,qq) =
  PRE
    pp : PERSON & qq : PERSON & pp : members &
    qq : members & (pp,qq) : friends
  THEN
    nn := card (friends[{pp}] /\ friends[{qq}])
  END
END
```

1. Explain the purpose of each part of the invariant of this machine
2. Write down the state schema in Z corresponding to this machine

3. The operation `addFriend` creates a friendship between two members who were not previously friends. Explain the purpose of its precondition.
4. Translate the `addFriend` operation to Z
5. What is the proof obligation associated with the `addFriend` operation? Explain informally how it is discharged.
6. Write an operation `removeMember` in B, to remove a member `mm` from the system, together with all of that person's friendships, and their network membership.
7. The operation `numFriends` outputs the number of mutual friends of two distinct people. It is missing a precondition. What is it?
8. How would you modify the invariant to record the fact that some friends are also relatives?

10 Use of the B method in disciplines other than software engineering

One area of research that we are currently pursuing is the development of a calculus of multirelations [MCR04]. This formalism has applications to game theory and voting protocols among other things. The research has involved a large volume of proof, all of which has so far been carried out by hand. We are currently considering mechanising the proofs, both the save time and to increase confidence in their correctness. The B method has been used in the past for proving results in a very closely related relational calculus [CM07], and so we would like to explore the possiblity of using it for verifying the calculus of multirelations.

11 Conclusion

It has been a huge privilege for both authors to teach formal methods at Brookes for over ten years now, on both the Z and B modules, and also on various other ones which have now been discontinued. We are delighted with the recent advances in tool support for B that continue to make the course feel modern and worthwhile. Moreover, the constantly changing world of computer applications never fails to bring fresh inspiration for examples that motivate students. The art of capturing requirements precisely using mathematics never fails to please, and we hope to be able to continue teaching it long into the future.

12 Acknowledgements

The authors are very grateful for the comments of the anonymous referees.

13 References

References

[A06] Jean-Raymond Abrial *The B-Book: Assigning programs to meanings* Cambridge University Press, 1996.

[CM07] Dominique Cansell, Dominique Mery: Incremental Parametric Development of Greedy Algorithms *Electronic Notes in Theoretical Computer Science (ENTCS) archive* Vol 185, p47-62, 2007.

[MCR04] Martin, C. E. , Curtis, S. A. and Rewitzky, I. (2007) Modelling angelic and demonic nondeterminism with multirelations *Science of Computer Programming* 65(2) : 140-158

[PLN03] Christine Poerschke, David E. Lightfoot, John L. Nealon: A Formal Specification in B of a Medical Decision Support System. ZB 2003: 497-512.

[S01] Steve Schneider *The B Method: An Introduction* Palgrave, 2001.

[B] `http://www.b-core.com/btoolkit.html`

[R] `http://www.event-b.org/`

[A] `http://www.atelierb.eu/index-en.php`

[U] `http://www.it.uu.se/edu/course/homepage/bkp/vt07/differences`

Twelve years of B Teaching in an engineer school: from a correct by design approach to analysis techniques and tools.

Marie-Laure Potet

Vérimag, centre équation, 2 avenue de Vignate – F-38610 Gières,
Marie-Laure.Potet@imag.fr

Abstract. Twelve years ago I introduced a B course at Ensimag, a well-known french Mathematics and Computer Science engineer school. In this paper I describe this experiment and the evolution of the contents of this course. In particular I present a set of fundamental concepts that can be easily illustrated with the help of the B framework. I also present some unpublished examples and theoretical results.

1 Introduction

Twelve years ago I introduced a B course at Ensimag, a well-known french Mathematics and Computer Science engineer school. In company of Didier Bert, I also gave lectures to the " Ecole des jeunes chercheurs en programmation ", dedicated to PhD students in the domain of programming[1].

During this long period, these courses had been subject to many evolutions, due first to my better understanding and knowledge of the domain and, more important, to my perception of notions that students are able to acquire and master and finally of notions that are intrinsically very challenging to understand. Furthermore, during the last two decades, the context had changed : formal techniques have acquired a new status in industry and in the every day life of programmers. Due to safety and security constraints, a lot of tools and approaches have been developed, allowing to verify some dedicated classes of program properties. Then teaching formal methods, and more importantly the underlying concepts, are no more considered as an esoteric idea. Due to this evolutive context, the Ensimag B lectures had been subject to some evolutions, that could be described in the form of three periods:

- First period: B studied as a whole method from specification to correct code (correctness by construct).
- Second period: B used as a well-adapted support to introduce a set of fundamental aspects of programming science.
- Next period: B will be presented as a part of existing techniques for program analysis for a larger public.

[1] This school is supported by the GPL french working group.

Despite of these changes, I am convinced that the B method offers a very nice and solid framework to present a set of interesting concepts, from methodological aspects to theoretical foundations of programming. Furthermore, the existence of robust tools taking into account the global process from abstract specifications to executable code is a very attractive argument with respect to students. During these twelve years I have compiled some experiences, practical exercises and theoretical results which will be presented here.

In section 2, I present the first period of the Ensimag B teaching, with an evaluation of how some notions have been introduced with which efficiency in regard of students. I also present small and well-targeted modelling examples. In section 3 I introduce some semantical aspects that illustrate the underlying concepts relative to programming reasoning. I conclude in describing a new cursus relative to static analysis techniques, including the weakest precondition approach.

2 The B method studied as a whole

The Ensimag course dedicated to the formal aspects of programming takes place in the second year of the curriculum, corresponding to the first year of a Master degree. This course was proposed as an optional cursus, attracting students interested by theoretical aspects of computer science.

The audience is composed of future engineers having the taste for technical and fundamental aspects and with some abilities with abstraction, due to a solid background in Mathematics (issued from the very French cursus "classes préparatoires aux grandes écoles"). That means that our students have no particular difficulty with mathematical notions such as set theory. More important, they are also able to manipulate several levels of formalization such as syntax, semantics and reasoning about semantics, without too difficulty. Finally Ensimag students have some background in logic, compiling (realization of a small object oriented language compiler) and rigorous algorithmics.

2.1 The objectives

The first version of the B method course has been proposed as the successor of lectures relative to formal specifications, in which many approaches was presented (algebraic specifications, model based specifications, refinement notions as in VDM and Z, ...). At this stage the choice was to focus on the B method which integrates the main steps of formal development into a single framework. Furthermore, due to the fact that this method was supported by robust tools, the challenge was to conciliate practice with a fine understanding of theoretical concepts.

During this period, despite the fact that the aim was not to produce B specialists, I followed the top-down approach preconized by the B-method in [1]. The aim was to initiate students into formalization and modelling activities and into the top-down "correct by design" paradigm. Then the ambitious aim of this course recovers three objectives, according to students skills:

– ability to formalize behaviours and properties with the help of abstract languages;
– ability to understand the semantics of programming languages and ability to reason about programs in a formal way;
– ability to understand the theoretical concepts underlying correctness by design principle, for real programming language.

I list below the different concepts that are tackled and their understanding by students.

2.2 Modelling aspects

This part concerns the ability of students to:
– state properties with the set theory notations
– describe behaviours with the generalized substitution language (GS)
– apply weakest precondition calculus (WP)
– understand proof obligations relative to invariant properties (PO)

Notions	Students ability
use of set theory	not a problem
use of GS notation	difficulty with non-determinism
WP calculus	not really a problem
PO understanding and proof argumentation	not easy to be exact with logical reasoning

Let here some examples illustrating data structure modelling, non determinism and properties.

Examples	Used notions
a memory allocator	non-determinism
a dynamic class loader	formalisation of trees to specify a class hierarchy
a RBAC security policy	specification based on relations and search of inductive invariants
a simple elevator	how expected properties can be formalized with the help of type properties, invariant and dynamic behaviours

The allocator and the RBAC examples are given below. The dynamic class loader is described in [8] and the lift example is extracted from the course of EJCP07[2].

The memory allocator example. Let MEM be a given set, $null$ a distinguished constant of this set and $used \subseteq MEM$, the set of addresses that have been allocated. This subset is initialized with the set $\{null\}$. Here is the specification of the allocator operation:

[2] http://www-lsr.imag.fr/users/Didier.Bert/enseignement.html

$r \longleftarrow allocate =$
CHOICE
ANY v WHERE $v \in MEM - used$
THEN $used := used \cup \{v\} \parallel r := v$
END
OR $r := null$
END

In this example two forms of non-determinism are used: the CHOICE substitution corresponds to the fact that the allocator can, or not, supply a memory cell. The second form of no-determinism (substitution ANY) corresponds to the fact that the address that is returned is chosen by the allocator. Then the user of such a component has *a priori* no information about the behaviour of the allocator. In particular he systematically must test if the returned value is *null* or not. This example illustrates non-determinism.

The RBAC security policy example. We consider below a general model for role-based security policies in which roles are attached to subjects and permissions are attached to roles. As in RBAC[5], some roles can be declared in conflict. In this case a safety invariant states that a subject can not own two roles in conflict. This example illustrates how invariant properties could be enforced during the proof process in order to obtain an inductive invariant. Here is the declaration part of this model.

```
MACHINE RBAC
SETS
   SUBJECT, ROLE, PERMISSION
VARIABLES
 subject2role, role2permission, conflict
INVARIANT
   subject2role  ∈ SUBJECT  ↔ ROLE  ∧
   role2permission  ∈ ROLE  ↔ PERMISSION  ∧
   conflict  ∈ ROLE  ↔ ROLE  ∧
   subject2role  ∩  (subject2role ; conflict)= ∅
END
```

The last part of this invariant states that nobody can own two roles which are in conflict. Now we consider the operation *AddRole* that adds a new role r for a given subject s.

$AddRole(r, s) =$
PRE $r \in ROLE \wedge s \in SUBJECT \wedge r \notin conflict[subject2role[\{s\}]]$
THEN $subject2role := subject2role \cup \{s \mapsto r\}$
END

In order to establish the invariant preservation we have to prove:

$$(subject2role \cup \{s \mapsto r\}) \cap ((subject2role \cup \{s \mapsto r\}) ; conflict) = \emptyset$$

under the hypothesis $subject2role \cap (subject2role ; conflict) = \emptyset$ (the invariant initially holds) and $r \notin conflict[subject2role[\{s\}]]$ (the precondition). This proof obligation can be split into for cases:

$\{s \mapsto r\} \cap (subject2role ; conflict) = \emptyset$	$comes\ from\ the\ precondition$
$\{s \mapsto r\} \cap (\{s \mapsto r\} ; conflict) = \emptyset$	$missing\ hypothesis\ (irreflexivity)$
$subject2role \cap (subject2role ; conflict) = \emptyset$	$comes\ from\ the\ invariant\ hypothesis$
$subject2role \cap (\{s \mapsto r\} ; conflict) = \emptyset$	$missing\ hypothesis\ (symmetry)$

For the second case the invariant must be enforced by the irreflexivity property of the relation *conflict* ($conflict \cap id(ROLE) = \emptyset$). For the last case, the symmetry of conflict have to be added ($conflict^{-1} = conflict$). With these two further properties the proof obligation attached to the operation *AddRole* can now be proved.

Finally an interesting exercise, in term of modelling and properties statement, is to add hierarchical roles (a tree) and the property that this hierarchy is compatible with the relation *conflict* (it is not possible to inherit of two roles which are in conflict).

2.3 Formal development process

In this part we discuss how the notions relative to formal development process are understood by our students. We focus on the following notions:

- notion of refinement (its intuitive definition and its proof obligations)
- refinement properties like transitivity and monotonicity
- implementation constraints and proof obligations (bounded integers and overflow detection for instance)
- practical aspects (practice of the AtelierB : firstly with demos and after through exercises.)

Notions	Students understanding
refinement principle	familiar with data representation
Refinement PO	intrinsically hard to appropriate
Refinement properties	very very abstract
practical work	difficulties due to syntactic restrictions

Refinement proof obligations are intrinsically hard to integrate. Let S be an abstract substitution relative to variables x, T be a concrete substitution relative to variables y and let L be the refinement relationship between x and y. Refinement proof obligations can be presented using two forms:

$First\ form$	$L \wedge \mathsf{trm}(S) \Rightarrow [T]\neg[S]\neg L$
$Second\ form$	$L \wedge \mathsf{trm}(S) \wedge \mathsf{prd}(T) \Rightarrow \mathsf{trm}(T) \wedge \exists x'(\mathsf{prd}(S) \wedge [x, y := x', y']L)$

The first form is a bit disturbing for students due to the double negation. The second form is a little bit intuitive but requires to introduce trm and prd predicates which are again new abstract notions to integrate. I now systematically use the first form because it does not require new concepts. Furthermore the following intuitive formulation can be used: *for any concrete behaviour (T) it is not true that any abstract behaviour (S) does not establish L. Then, for each concrete behaviour, there exists at least one abstract behaviour that establishes L.*

Finally the development of examples, with the help of a robust tool such as the AtelierB, is very interesting because students can precisely understand the correctness by design approach. Furthermore they formalize informal reasonings used in algorithmics (such as invariant and variant of a while statement for instance). Furthermore practical exercises are generally motivating, even so students are generally disturbed by the semi-decidability of proofs. Here are examples we used:

Examples	Notions that are used
gcd program	proofs of iteration and absence of overflows
element research in an array	a sophisticated iteration invariant
modelling and controlling a lock	simple data refinement
a booking service example	a global development

The three first examples are available at www-verimag.imag.fr/~potet/PageB. They are tailored to be used in practical labs: components are partially specified and, if students state the right formulae, proofs are automatically established[3]. The last example is developed in several documents (in french) (web pages of Didier Bert or Marie-Laure Potet). It is also used to illustrate proof obligations relative to iteration and how these proof obligations depend on the initial specification. We develop this part of this example below.

Let $SEAT$ be the set $1..maxseat$ and let $taken$ be the subset of $SEAT$ that has been already assigned. The operation *booking* can be specified in the following way:

$place \longleftarrow booking =$
  PRE $card(taken) \neq maxseat$
  THEN
    ANY p WHERE $p \in SEAT - taken$
    THEN $taken := taken \cup \{p\} \parallel place := p$
    END
  END

At the implementation level we represent the set $taken$ by an array tab ($tab \in 1..maxseat \rightarrow BOOL \wedge tab^{-1}[\{TRUE\}] = taken$) and the booking operation is implemented in the following way:

[3] depending on the way the formula is written.

$place \longleftarrow booking =$
VAR ind IN
$ind := 1$;
WHILE $tab(ind) = TRUE$ DO
$ind := ind + 1$;
INVARIANT ...
VARIANT $maxseat - ind$
END;
$tab(ind) := TRUE$;
$place := ind$;
END

The invariant part has to be completed by the formula:

$$ind \in 1..maxseat \wedge FALSE \in tab[ind..maxseat]$$

that means that a free seat appears at the current index (ind) or in the rest of tab. Now if we modify the specification in order to choose the smaller free number the WHERE part of the specification becomes: $p = min(SEAT - taken)$. Then the invariant of the implementation has to be enforced in order to prove that we always choose the seat with the minimal number:

$$ind \in 1..maxseat \wedge tab[1..ind - 1] \subseteq \{TRUE\}$$

2.4 Conclusion of the first period

In a top-down design process (from abstract models until implementations) students become comfortable when implementations are tackled. They rediscover known notions and a formalization of their usual practice.

Furthermore, it seems difficult for them to integrate in the same time many theoretical notions such as a specification language, the weakest precondition calculus and the refinement theory, in parallel of the methodological aspects of formal development process. Then I choose to focus first on the proof of program approach, using the substitution language in its whole (without distinction between generalized substitutions allowed at the different level of components). In this way students practice proof of programs, learn how to state invariant properties and discover new notions as non-determinism. Practical exercises start with classical programs with iteration (as gcd) to go towards specifications. Finally the refinement theory is introduced. This approach has proved to be more attractive for students, because it starts from their background and knowledge.

3 B as a support to illustrate underlying concepts of formal reasoning about programs

The B method is a well-adapted framework to visit some important concepts, like semantics definition and properties underlying proofs of programs. Refinement is then introduced as an extension of this approach. In this part the aim is that

students understand some fine aspects of semantics definition and are able to reason about it. We present here three aspects: an extension of the B weakest precondition calculus taking into account a new feature (abnormally exit through exceptions), the proof of the invariant preservation by operation call, that is the base on incremental development and, finally, the theoretical framework underlying B refinement proof obligations.

3.1 Weakest precondition for exception features

Here the aim is to extend the B language by exceptions, which is an already sophisticated feature. Some results presented here are borrowed from Lilian Burdy work [2]. The original part is the proof of the correctness of this extension, based on a mapping between programs with exception and without exception. In this way students can discover that it is possible to formally reason at the semantics level. We describe below this extension.

We extend the B language with the possibility to declare exceptions. We also extend the generalized substitution language in the following way:

RAISE e	raising exception e
BEGIN S	a block with exception handling
CATCH WHEN e_1 THEN S_1	
...	the handling part
WHEN e_n THEN S_n	with $i \neq j \Rightarrow e_i \neq e_j$
END	

The new weakest precondition calculus $wpe(S, F)$. Let EXC be the set of exception names with a distinguished constant no ($no \in EXC$) that corresponds to normal behaviours. The weakest precondition calculus is now denoted by $wpe(S, F)$ with F a function relating an exceptional exit with a postcondition, i.e.

$$F \in EXC \mapsto Predicate$$

This calculus can be defined in the following way:

$wpe(\mathsf{skip}, F)$	$F(no)$
$wpe(x := v, F)$	$[x := v]\, F(no)$
$wpe(raise\ e, F)$	$F(e)$
$wpe(P \Longrightarrow S, F)$	$P \Rightarrow\ wpe(S, F)$
$wpe(S_1 \parallel S_2, F)$	$wpe(S_1, F)\ \wedge\ wpe(S_2, F)$
$wpe(S_1\ ;\ S_2, F)$	$wpe(S_1, F \lhd\!\!\!\!- \{no \mapsto wpe(S_2, F)\})$

where $\lhd\!\!\!\!-$ denotes the overriding B operator. In the case of sequencing, if S_1 stops abnormally with an exception e the expected postcondition is $F(e)$ because S_2 is not executed. Otherwise S_1 must establish the precondition issued from the execution of S_2, as in classical B. The weakest precondition of blocks with handling is defined in the following way:

$wpe($ BEGIN S CATCH WHEN e_1 THEN S_1 $\ldots$ WHEN e_n THEN S_n END, F)	$wpe(S,$ $F \Leftarrow \{e_1 \mapsto wpe(S_1, F),$ $\ldots,$ $e_n \mapsto wpe(S_n, F)\}$ $)$

An example. Let here the following program for which we want to establish the postcondition $F_1 = \{no \mapsto x = 2, stop \mapsto x = 1\}$, meaning that the postcondition $x = 2$ is expected when the program terminates normally and the postcondition $x = 1$ is expected when the exception *stop* is raised:

BEGIN $x := 1$; IF $y > 0$ THEN RAISE *stop* END ; $x := 2$ END

and here is now the same program enriched by the weakest precondition calculus (must be read from the bottom to the top):

BEGIN
(F_4) $\{(y > 0 \Rightarrow true) \ \wedge \ (\neg(y > 0) \Rightarrow true)\}$
$x := 1;$
(F_3) $\{no \mapsto (y > 0 \Rightarrow x = 1) \ \wedge \ (\neg(y > 0) \Rightarrow true), \ stop \mapsto x = 1\}$
IF $y > 0$ THEN RAISE *stop* END ;
(F_2) $\{no \mapsto true, stop \mapsto x = 1\}$
$x := 2$
(F_1) $\{no \mapsto x = 2, stop \mapsto x = 1\}$
END

with:

Formula	Its definition
F_1	$F_1(no) = (x = 2)$, $F_1(stop) = (x = 1)$
F_2	$F_2(no) = [x := 2]F_1(no) = true$, $F_2(stop) = F_1(stop)$
F_3	$F_3(no) = F_2(stop)$ if $y > 0$, $F_3(no) = F_2(no)$ if $\neg(y > 0)$, $F_3(stop) = F_2(stop)$
F_4	$F_4 = [x := 1]F_3(no) = true$

Semantics Correctness. In this part we show how to establish the correctness of the *wpe* definition with respect to the classical B weakest precondition calculus, using a systematic transformation between programs with exceptions and without exception. To do that we add a new variable exc ($exc \in EXC$). Let $\mathcal{C}(S)$ be this transformation defined in the following way:

$\mathcal{C}(x := v)$	$\hat{=}$	$x := v \ ; \ exc := no$
$\mathcal{C}(\mathsf{skip})$	$\hat{=}$	$exc := no$
$\mathcal{C}(\text{RAISE } e)$	$\hat{=}$	$exc := e$
$\mathcal{C}(S1 \ ; \ S2)$	$\hat{=}$	$\mathcal{C}(S1)$; IF $exc = no$ THEN $\mathcal{C}(S2)$ END

and:

$\mathcal{C}$(BEGIN S CATCH WHEN e_1 THEN S_1 ... WHEN e_n THEN S_n END) $\hat{=}$
$\mathcal{C}(S)$;
IF $exc \neq no$ THEN CHOICE $exc = e_1 \Longrightarrow exc := no$; $\mathcal{C}(S_1)$
OR ... OR $exc = e_n \Longrightarrow exc := no$; $\mathcal{C}(S_n)$ END
END

Now the correctness of the *wpe* calculus can be established using the following equivalence:

$$wpe(S, F) \;\Leftrightarrow\; [\mathcal{C}(S)] \bigwedge_{e_i \in dom(F)} (exc = e_i \Rightarrow F(e_i))$$

An interesting exercise (not developed here) is to prove the correctness of the raise and the catch block substitutions. Extending the weakest precondition calculus for new features is an attractive exercise for students because they have to build a formal definition. Furthermore, in this case, their solution can be proved to be correct, thanks to the natural definition of the function $\mathcal{C}$.

3.2 Semantics of operation calling

In this part we formally define operation calls and several modes of parameter passing. In particular we show that by-reference parameters are not adapted for verification, because invariant properties are not preserved. Results presented here are extracted from [7, 3].

Operation call definition. Let $r \leftarrow op(p) \;\hat{=}$ PRE P THEN S END be the definition of the operation *op* and let $v \leftarrow op(e)$ be a call of *op*. In B, incremental development is based on the encapsulation principle. That means that variables v are disjointed from variables of the component in which *op* is defined. We define here two parameter-passing modes: by copy (by value) mode or by reference (by address) mode.

By copy semantics:

$$\text{PRE } [p := e]P \text{ THEN VAR } p, r \text{ THEN } p := e \;;\; S;\; v := r \text{ END END}$$

By reference semantics:

$$\text{PRE } [p := e]P \text{ THEN } [p, r := e, v]S \text{ END}$$

Semantics by substitution corresponds to by-reference parameters when the effective parameters reduce to simple variables (as v in B). If any expressions are admitted, substitution corresponds to the by name mode passing. For the definition of substitution into substitution see [1].

In the B method, the semantics of operation calls uses the definition associated to the by-reference mode although it is a by-copy mode. This is due to the fact that operation call semantics is only defined at the level of abstract machine components in which sequentiality and while substitutions are prohibited. In the general case these two modes of parameters differ.

An example. For instance let consider the definition $op(y) \hat{=}$ PRE even(y) THEN x:=x+1 ; x:=x+y+1 END and the piece of code x:=0 ; op(x) ; print(x). The by-reference semantics produces the code x:=0 ; x:=x+1 ; x:=x+x+1 ; print(x) that prints the value 3. On the contrary the by-copy semantics produces the code x:=0 ; VAR y IN y:=x ; x:=x+1 ; x:=x+y+1 END ; print(x) that prints the value 2.

Invariant preservation by parameter passing. The by-reference parameter does not preserve invariant. For instance we can establish that the operation *op* above preserves the invariant $even(x)$. But, as shown above, the call $op(x)$ does not preserves this property (3 is not an even value). On the contrary the by-copy semantics has good properties for verification: invariants are preserved by calls.

Theorem 1 *Invariant preservation by call*

Let $r \leftarrow op(p) \hat{=}$ PRE P THEN S END*be the definition of an operation op and let* $v \leftarrow op(e)$*be an operation call, as defined before. Let* I *be a property on* x*, the set of variables of the component in which op is defined* ($x \cap v = \emptyset$)*. Then:*

$$\forall r, p \ \ (I \wedge P \Rightarrow [S]I)$$
$$\Rightarrow$$
$$(I \wedge [p := e]P \Rightarrow [\text{VAR } p, r \text{ IN } p := e \ ; \ S \ ; \ v := r \text{ END}]I)$$

On one hand, by monotonicity of $\Rightarrow$ with respect to substitution, we can derive from $I \wedge P \Rightarrow [S]I$ the formula: $[p := e]I \wedge [p := e]P \Rightarrow [p := e][S]I$ (**a**). On the other hand the conclusion reduces to $I \wedge [p := e]P \Rightarrow \forall\ p, r\ ([p := e][S][v := r]I)$. Because v does not appear in I (the encapsulation principle) then the right part reduces to $[p := e][S]I$. Then the conclusion of the rule is obtained from **(a)** because p is not free in I. Similar results can be established for refinement, i.e. refinement proofs established at the level of operation definition are preserved by by-copy operation call semantics. This property is harder to be established (see [7] for a proof).

As before, formalization of parameter-passing modes is an attractive exercise for students. On the contrary, property as invariant preservation by operation calls is a less natural question for students. They generally do not really understand why such properties are important, even so we show some incremental constructions, as the clause INCLUDES for instance.

3.3 Refinement theory

The B method gives a syntactic notion of refinement in term of proof obligations. It is interesting to give a more semantic definition. Then as in the classical theory of refinement developed by Willem-Paul de Roever and Kai Engelhardt [4], refinement notion gives this semantic point of view and proofs are conducted by the help of simulations.

Although the notions that will be introduced here are a little bit complex they are interesting according to several reasons. First they give a semantic definition of refinement in term of component substitution principle. Second they introduce how this definition can be implemented in several operational ways. And finally they allow to illustrate the classical notions of correctness and completeness of an operational procedure with respect to a semantic definition.

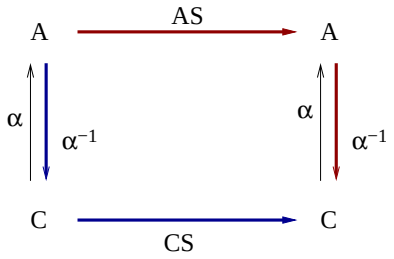
Semantic definition of refinement. Let M be a component refined by another component R. As defined in the B-Book (chapter 11, p 511) a substitution U is an external substitution for M and R if it contains no reference to the variables of components M and R (v_M or v_R). Internal variables can only be consulted or modified through operation calls. We denote by U_M and U_R the generalized substitutions obtained by U in replacing operation calls respectively by their definition in M and R. The definition below is issued from the B-Book [1].

Definition 1 *Semantic characterization of data refinement.*

M can be substituted by R if components M and R propose the same set of operations with the same interface and, for each external substitution U for M and R, we have ANY v_M IN $init_M$; U_M END $\sqsubseteq$ ANY v_R IN $init_R$; U_R END.

In this definition internal variables are encapsulated by the ANY substitution and initialized by the *init* substitution of M and R component.

Simulations and proof obligations. Definition 1 characterizes refinement as a substitution principle but gives no manner to establish refinements because all external substitutions have to be considered. Then, as in B, simulations are established with the help of a refinement relation α (the gluing invariant) and describe commutations of the following diagram:



In this diagram, AS and CS correspond to the before-after relation of the abstract and concrete substitutions[4]. For simulations can be defined (named L, L^{-1}, U and U^{-1}), depending how the diagram commutes:

[4] Termination condition are not considered here.

Simulations (notation X)	Proof obligations (notation $\sqsubseteq_\alpha^X$)
L-simulation (forward/downward simulation)	$\alpha^{-1}\ ;\ CS\ \subseteq\ AS\ ;\ \alpha^{-1}$
L^{-1}-simulation (backward/upward simulation)	$CS\ ;\ \alpha\ \subseteq\ \alpha\ ;\ AS$
U-simulation	$\alpha^{-1}\ ;\ CS\ ;\ \alpha\ \subseteq\ AS$
U^{-1}-simulation	$CS\ \subseteq\ \alpha\ ;\ AS\ ;\ \alpha^{-1}$

Two important properties are attached to simulations: **correctness** and **completeness** of their definition with respect to refinement definition (def. 1). Let M and R be two components. A simulation X is correct if whereas there exists α such that the proof obligations $\sqsubseteq_\alpha^X$ hold then M is effectively refined by R in the sense of definition 1. On the contrary, a simulation X is complete if, for every components M and R respecting definition 1, it is possible to find a relation α such that proof obligations $\sqsubseteq_\alpha^X$ hold. Results are the following ones [4]:

- correctness of L and L^{-1} simulations
- correctness of U simulation if α is total ($\alpha \in\ C \leftrightarrow\ A \wedge\ id(C) \subseteq \alpha\ ;\ \alpha^{-1}$)
- correctness of U^{-1} simulation if α is a function ($\alpha\ ;\ \alpha^{-1} \subseteq id(A)$)

If α is a total function then U-simulation and U^{-1}-simulation are two equivalent notions. All stand-alone simulations are incomplete. On the contrary a combination of L et L^{-1} is complete. We illustrate these results by an example.

Example. This example is borrowed from Steve Dunne [6]. Let's consider the two following B machines, that can be considered as equivalent ones: they both admit the same set of external substitutions.

MACHINE $CASINO1$
VARIABLES i
INVARIANT $i \in 0..36$
INITIALISATION $i :\in 0..36$
OPERATIONS
$r1 \longleftarrow spin \hat{=} r1 := i \ ||\ i :\in 0..36$
END

MACHINE $CASINO2$
OPERATIONS
$r2 \longleftarrow spin \hat{=} r2 :\in 0..36$
END

There exists α ($\emptyset$) such that CASINO2 $\sqsubseteq_\alpha^L$ CASINO1 and CASINO1 $\sqsubseteq_\alpha^{L^{-1}}$ CASINO2. On the contrary there exists no α such that CASINO1 $\sqsubseteq_\alpha^L$ CASINO2. Let's show that CASINO2 $\sqsubseteq^L$ CASINO1:

$$i \in 0..36 \Rightarrow\ [r1 := i \ ||\ i :\in 0..36]\neg[r2 :\in 0..36]\neg(r1 = r2)$$
$$i \in 0..36 \Rightarrow\ [r1 := i]\exists r2\ (r2 \in 0..36 \wedge r1 = r2)$$
$$i \in 0..36 \Rightarrow\ [r1 := i]r1 \in 0..36$$
$$true$$

CASINO1 $\not\sqsubseteq^L$ CASINO2:

$i \in 0..36 \Rightarrow\ [r2 :\in 0..36]\neg[r1 := i \mid\mid ii :\in 0..36]\neg(r1 = r2)$
$i \in 0..36 \Rightarrow\ \forall r2\ (r2 \in 0..36 \Rightarrow\ i = r2)$
$i \in 0..36 \wedge\ r2 \in 0..36 \Rightarrow\ i = r2$
$false$

Using a logical form, proof obligations relative to L^{-1}-simulation can be stated as: $\forall c, a'\ (\exists c'\ (C \wedge [c, a := c', a']\alpha) \Rightarrow \exists a\ (\alpha \wedge A))$. Let's show that CASINO1 $\sqsubseteq^{L^{-1}}$ CASINO2:

$\forall r1'(\exists r2'(r2' \in 0..36 \wedge r1' = r2') \Rightarrow \exists i(i \in 0..36 \wedge r1' = i))$
$\forall r1'(r1' \in 0..36 \Rightarrow \exists i\ (i \in 0..36 \wedge r1' = i))$
$true$

In general our students are interested by a formal definition of refinement in term of component substitution because this characterization corresponds to the classical notions of encapsulation and component contract. Without surprise, they have some difficulties with the different forms of simulation and examples.

4 Conclusion

I present below some general conclusions about this twelve years of B teaching at Ensimag and some propositions for a new course (under development) dedicated to a larger audience and a larger spectrum.

4.1 Conclusion of my experiments

During this long period the B method has proved to be a solid framework for teaching formal methods. First the underlying concepts (set theory, generalized substitution notation) are simple and expressive enough. That means that students can easily and quickly write specifications and codes. Second, when examples are well-made, the AtelierB tool can be used successfully by students. Its main advantage, among other tools dedicated to formal methods, is the support of the global development process, from abstract specification until executable programs. Even so the interactive prover is not very intuitive, with a few introduction students are able to develop some simple proofs. Tractable examples and their educational aims have been given section 2.

Nevertheless, the B method has been designed to be efficient and tractable in an industrial context, encapsulating some complex notions and introducing a set of restrictions that are not always easy to justify. Then teaching B, and using it to study theory of programming, requires to break this framework in order to enter into details of the internal machinery, as described section 3.

Finally, the main difficulties for students are relative to refinement theory whereas the refinement process is intuitive enough. Nevertheless, this complexity seems inherent to all formal methods.

4.2 The future

The curriculum of Ensimag is under modification in order to be closer of the LMD reform. In the Information System Engineering curriculum, it has been decided to impose a course with some formal contents. The audience should now be around 50-60 people.

Furthermore, as pointed in the introduction part, formal techniques have acquired a new status in industry and in the every day life of programmers. Due to safety and security constraints, a lot of tools and approaches have been developed, allowing to verify some dedicated classes of program properties. Because our future engineers have to know the state-of-the-art technologies, we have decided to study a larger set of formal approaches, from rapid and very approximative tools to precise but interactive approaches, depending on the target of the verification process (bug finder, verification of some particular form of properties, proof of assertions ...). Depending on the chosen approach, some questions as false-positives (a program is declared as erroneous whereas it is correct) and false-negatives (an incorrect program is not detected) can be studied, with respect to a given notion of correctness.

Moreover some advanced features will be examined as object oriented programs, pointer and component oriented verification. On the contrary, some other aspects will not be presented at all, such as refinement and modelling. It seems more realistic to plan a new course (at the M2 level and as an optional one) dedicated to these aspects. In this case a broader spectrum has to be targeted, including temporal logic, communication models (automata, process algebra) and refinement theory (data refinement as well as behaviour refinement).

References

1. J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. Lilian Burdy and Antoine Requet. Extending B with Control Flow Breaks. In *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, pages 513–527, 2003.
3. D.Bert, S. Boulmé, M-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C programs. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*. Springer, 2003.
4. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
5. Ferraiolo D.F. and Kuhn Richard. Role-Based Access Control. In *Proceedings of the 15th NIST-NSA National Computer Security Confe rence*, pages 554–563, Baltimore, MD, USA, October 1992. Nat'l Inst. Standards and Technology.
6. Steve Dunne. Introducing Backward Refinement into B. In *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, pages 178–196, 2003.
7. M-L. Potet. *Spécifications et développements formels: Etude des aspects compositionnels dans la méthode B*. Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble, 5 décembre 2002.
8. Nicolas Stouls. *Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés*. Thèse de doctorat, Institut Polytechnique de Grenoble, 2007.

High-Level versus Low-Level Specifications: Comparing B with Promela and ProB with Spin

Mireille Samia, Harald Wiegard, Jens Bendisposto, Michael Leuschel

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{samia|bendisposto|leuschel}@cs.uni-duesseldorf.de
harald.wiegard@uni-duesseldorf.de

Abstract. During previous teaching and research experience, we have accumulated anecdotal evidence that using a high-level formalism such as B can be much more productive than using a low-level formalism such as Promela. Furthermore, quite surprisingly, it turned out that the use of a high-level model checker such as ProB was much more effective in practice than using a very efficient model checker such as Spin on the corresponding low-level model.
In this paper, we try to put this anecdotal evidence on a more firm empirical footing, by systematically comparing the development and validation of B models with the development and validation corresponding Promela models. These experiments have confirmed our previous experience, and show the merits of using a high-level specification language such as B, both in a teaching and in a research environment.

Key words: Model Checking, ProB, Spin, B-Method, Promela

1 Introduction

In the past, we have accumulated a variety of anecdotal evidence about using high-level and low-level formalisms for formal modelling, both in the context of teaching and in the context of academic or industrial applications.

For example, while teaching a formal methods course at the University of Southampton[1], on multiple occasions, the students were able to develop a B model of a particular task and validate it using ProB, whereas attempts using Promela and the model checker Spin turned out to be fruitless. A similar situation arose while teaching a course at the University of Düsseldorf[2]. On the one hand, it is not surprising that the use of a high-level formalism makes the model development easier. However, one would expect that using an extremely well tuned model checker such as Spin would result in better validation.

In this paper, we present an empirical case study, which tries to provide a more empirical foundation for these anecdotal evidences:

[1] CM401: Formal Design of Systems, taught in 2003 and 2004 at master's level
[2] Softwaretechnik III, taught once, also at Master's level

– How much time does it take to develop a B model compared to an equivalent low-level Promela model?
– How big is the B model compared to the Promela model?
– What is the performance of the model checkers ProB and Spin on these models?

The methodology we adopted was the following:

– a student developed B and Promela models for a variety of tasks, ranging from protocols to puzzles. The goal was to cover a wide variety of scenarios,
– the student carefully measures his time usage. In order to arrive at a more just result, the student alternated between first developing the B model and first developing the Promela model.
– The student also carefully measured the model checking performance, and whether the tools were successful.

The student had a background in both B and Promela, and has had experience both with ProB and with Spin.

2 Background

The B-Method, originally developed by J.R. Abrial [1], is based on the notion of abstract machines and refinement. In B, a system is seen as a set of states and operations, which modify the state. In order to prove consistency of the system, one has to prove, that each operation preserves the model's invariant. B uses the same notation for all the stages of the development of a system, which goes on for successive refinement steps. There are several tools to support developing models with B, one of them is the model checker ProB [5]. Promela[3] is a formalism for the verification of models [3], in particular protocols. It is based on concurrent communicating processes, that communicate via channels. The message transmission can be immediate or with delay, i.e. a channel can have a capacity of elements stored inside. The syntax and datatypes of the Promela language are influenced by the high-level language C. Promela is supported by the model checker Spin that is used and verifies the correctness of a model. It offers the possibility to verify the validity of LTL formulas, which are automatically converted into *never claims*.

3 Case Studies of Different Models

Figure 1 contains a summary of all the case studies that were conducted. The column entitled "development time" describes how much more time consuming the development of the Promela model was compared to the B model. Similarly, the column entitled "source code length" describes how much longer the Promela models were compared to the B models (counting the number of tokens). Finally,

[3] Process or Protocol Meta Language

the verification time using ProB for B and Spin for Promela are given. One can see that in one instance (Needham-Schröder protocol) Spin was much more efficient than ProB. Overall, however, the performance is comparable and despite various attempts, the student was unable to verify the Promela versions of the vehicle administration and the reservation system using Spin. The Promela version of the "railway interlocking model" could not be completed in time.

	Development Time (Promela / B)	Source Code Length (Promela / B)	Verification Time (in sec.) ProB	Spin
Prime Number Test using Random Numbers	1.3	1.0-3.5	1.0	1.0
The Bridge Puzzle	5.0	1.9	1.0	1.0
The Bier Glass Puzzle	2.0	1.7	0.1	0.1
Administration of a Vehicle	4.5	4.5	0.7	-
Parking Garage	6.0	1.0	0.5	0.5
Water Boiler	18.0	1.8	0.05	1.0
Reservation System	10.0	3.3	130.0	-
Needham-Schroeder Public Key Protocol	3.0	0.8	150.0	2.0
Echo Algorithm	1.4	1.4	1.0	1.0
Railway Interlocking Model	-	-	5400.0	-

Fig. 1. The results of the comparison of ProB with Spin.

Below we provide more details about the individual experiments, except for the "prime number test" and the "vehicle administration" examples, which are two very simple specifications and which we do not elaborate on further.[4] [5]

3.1 The Bridge Puzzle

Description of the Model In this puzzle, several persons are supposed to cross a narrow bridge within a specified time. Each person moves at a different speed, and at most two persons are allowed to cross simultaneously. To cross safely they have to carry a torch, and the group has only one.

Source Code Size In the developed Promela model, a choice must be done for every person in a nondeterministic if-statement, whether or not he/she crosses the bridge. It is also necessary to query individually the time required to cross the bridge. The result is, that the size of the Promela model depends on the

[4] The vehicle administration problem did uncover one curious behaviour of Spin though, where the use of a variable labeled *auto* lead to inexplicable errors when model checking (but not simulation).

[5] The models can be found in the student's bachelor thesis in [8].

number of persons. That means, that the model becomes inflexible, which is based on the predefined number of people. [6]

On the other hand, B allows the definition of sets. In the model, we define a set that consists of an arbitrary number of persons. Adding another person is as simple as adding it to the set and putting a tuple into a function that links persons and the time they need to cross the bridge. The rest of the model remains the same no matter how many elements the set contains. This results in a very compact and flexible code.

The code of the Promela model, which is an example with four persons, is about twice the size of the B model's code. The more persons a model has, the greater is the gap between the size of the two models. The reason is that for each additional person, the Promela model requires more extra code than the B model.

Verification Time To solve the puzzle, we need to find a trace of operations that leads into a state, where the maximum time has not yet elapsed and all persons crossed the bridge safely. ProB can automatically search for such a condition. For a successful search for the defined goal in Figure 2, ProB needs significantly less than one second. In Figure 3, History lists the steps of the successful search for the bridge puzzle.

```
GOAL == time =< maximum_time & ran(person_pos)={dest}
```

Fig. 2. The GOAL-Definition in ProB.

Finding the solution with Spin also takes about one second. In order to find it, the user has to encode the goal into a neverclaim that is less intuitive than plain predicate logic.

Development Time The development time of the Promela model was five times longer compared to the development time of the B model.

3.2 The Beer Glass Puzzle

Description of the Model The model treats — similar to the brigde puzzle — the search for a solution for a given task. In this model, an amount of beer, such as 400 ml, should be measured using a 500 ml large glass and a 300 ml small

[6] An experience Promela user may be able to reduce the size of the model by modelling each person as an individual process. This, however, also makes the model development more tricky.

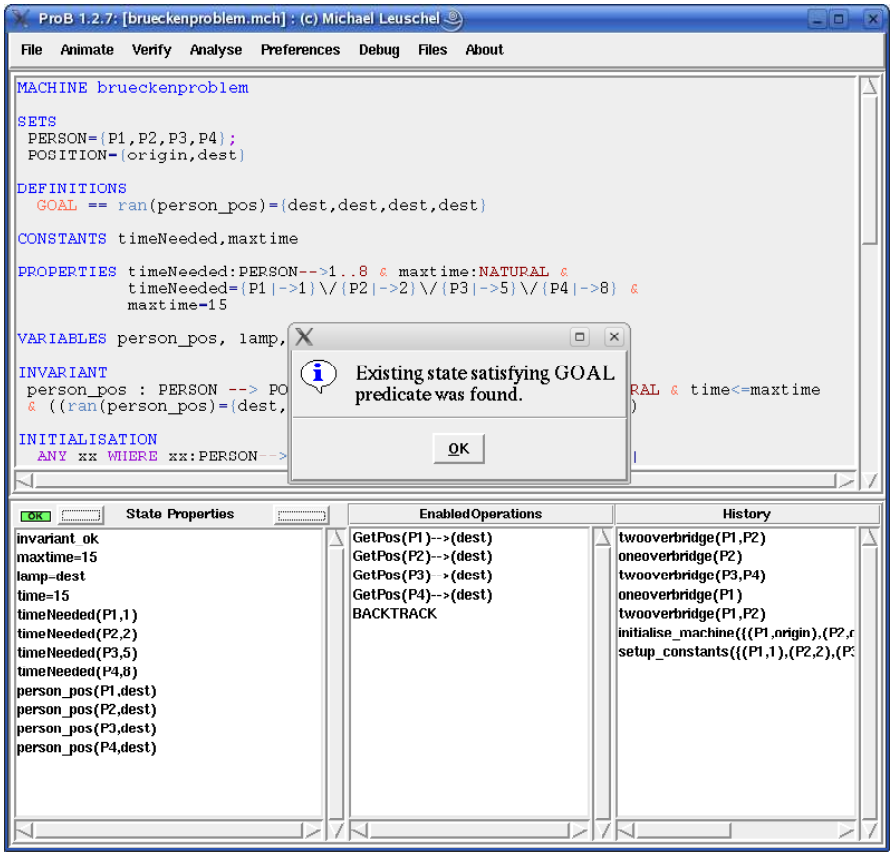


Fig. 3. ProB found a solution for the bridge puzzle.

glass. Both glasses can be only entirely filled or fully emptied. Moreover, the content of a glass can be decanted in the other glass till this glass is filled. Only a maximum total amount of liquid can be used, which is defined by the constant *limit*.

Source Code Length Similar to the bridge puzzle, the advantage of the high-level mathematical B notation is also shown here. In the Promela model, many `if`-statements are needed to select all possible cases. For instance, filling the big glass with the content of the small one comes with a case distinction. It is necessary to handle the two possible cases "content of the small glass fits completely in the large glass" and "content of the small glass does not fit completely in the large glass", e.g., if the big glass already contains beer (cf. Figure 4).

However, in B, the minimum function can be used resulting in a much more compact code (cf. Figure 5). The Promela code of the beer glass puzzle is 1.7 times longer than the B Code.

```
if
:: atomic{((big+small)<=maxBig)
           -> big=big+small;   small=0
         }
:: atomic{else
           -> small=small-(maxBig-big);
              big=maxBig
         }
fi
```

Fig. 4. In Promela, the Else case should be treated separately.

```
big  :=min({big+small,maxBig}) ||
small:=small-(min({big+small,maxBig})-big)
```

Fig. 5. The B Code is compact due to the use of the minimum function.

Verification Time The search for a solution with ProB is also easier than with Spin. Using predicate logic, it is easy to specify a goal, which ProB should find. We used the predicate $(big = solution \vee small = solution) \wedge used \leq limit$ as the goal. The goal is achieved, when a state is found, where the desired quantity of liquid is either in the small or in the large glass, and the selected limit is exceeded. ProB is able to find a solution in less than one second. In Spin, no goal state can be defined. Instead, we need to use a NeverClaim. It can be automatically generated from the LTL Formula $G\neg foundSolution$. The NeverClaim is a representation of the negation of the LTL formula. If a solution exists, the LTL-formula is violated. Spin recognizes this case. Using a "guide simulation", it can display the executed steps which lead to the violation of the formula. Spin can also find the solution in less than a second.

Development Time Although the model was first created in B and then in Promela, the creation of the Promela model has taken twice the time compared to the B model.

3.3 An Example of a Parking Garage

Description of the Model The example of a parking garage is a simple model: as long as the parking garage is not full, a car can enter, and then can go out.

Source Code Length In Promela and also in B, the model's code is quite compact. In both cases, the variable *usedSpace* is incremented or is decremented by the actual number of the cars inside the parking garage.

Whenever the Promela model uses several independent processes (or a process with multiple instances), the query of the actual value of the variables and their changes must be included in an *atomic*. Otherwise, the variable can accept an illicit value, whenever, for instance, between querying whether there is a car park in the parking garage and the execution of `usedSpace++`, another process increases the variable to its maximum allowed value. With a proper NeverClaims, Spin finds the source of errors - with one exception.

If the type of *usedSpace* is byte and the value of the variables was previously zero, then `usedSpace--` sets the variable's value to 255. Initially, the student's model used byte, meaning that the Promela model did not correspond to the desired specification and that certain errors were not detected by Spin (e.g., overflowing as increasing 255 by 1 gives 0 or underflowing as decreasing 0 gives 255). This problem does not occur in B and ProB. If a variable, defined as $NATURAL$ and having the value zero, is decreased by one, then the variable value is -1. Then, an invariant's violation occurs.

Verification Time Taking into account these restrictions, the models with Spin and ProB can be verified in a second.

Development Time The time spent on the B model is about fifteen minutes, and on the Promela models 90 minutes.

3.4 The Water Boiler

Description of the Model We specified a simple water boiler. It may only be turned on, if the lid is closed and water is filled. The lid may only be opened, whenever the water boiler is turned off. When the lid is open, water can be filled or distributed.

Source Code Length For each action of the water boiler model, the B model offers an operation, which can be executable depending on the preconditions. To ensure that no illicit condition occurs, the model has several invariants. The Promela source code is around the factor 1.8 longer than the B code.

Verification Time In the modelling phase, due to the possibility of interactive simulations, the student found it slightly easier to be convinced of the desired functionality of the B model. For the verification, ProB requires 0.05 seconds.

The Promela-model consists of an active process "User", which can execute non-deterministically one of the possible actions. To ensure that the corresponding preconditions remain unchanged before the end of their respective action, every action was wrapped in an `atomic` statement. This, however, meant that the Promela model was no longer verifiable using Spin. The reason is that, the

actions "fill water" and "distribute water" both have a `do`-loop, whose termination is not guaranteed. It is possible to replace the `do`-loop by an `if`-construct with an entry for every potential value of *ii*. This requires more work in programming, but allows both a proper verification as well as a realistic simulation. The verification of the model with Spin is then possible within about a second.

Development Time Because of this difficulty, the modelling in Promela took about 18 hours. The creation of the model with ProB was possible within one hour.

3.5 The Echo Algorithm

Description of the Model The Echo algorithm [2] is designed to find the shortest paths in a network topology. A start node sends an explore-message to all neighbors. Each node is marked with red, when it receives an explore-message for the first time. Moreover, it memorizes the nodes, from which it received the message, as a shortest path to the initialization node. It also sends, in turn, explore-messages to its other neighbors. Whenever the node receives either an explore-message or an echo-message from all its neighbors, to which it sent one of such messages, the node will be marked green and sends an echo-message to the nodes, from which it had first received an explore-message. When all nodes are marked green, the cycle is finished.

In the B model, every type of message type has one corresponding operation. The operations are active, as soon as a node sends the appropriate message. The execution of the operation reflects the receipt of the message by the node's recipient. By the non-deterministic order, the selection of the active operations is assured that any various long message runtime in the channel of the model is taken into account. It is possible that many messages are simultaneously on the channel. The edges are depicted as functions between the nodes. With the proper invariants, it is easy to verify if a protocol ensures that all nodes are marked green, when no message is in the channel. Furthermore, all shortest paths must be known as soon as all nodes are marked green.

In addition, with ProB, it is also possible to verify the accuracy of an LTL-formula. This example presents the LTL-formula `"GF{ran(state)={GREEN}}"`, i.e. the verification, to check whether there always exists a final state, in which all nodes are marked green. The verification of the model with ProB is possible within one second.

The Promela model offers a simple modelling of the message channels. As in the B model, many messages can pass simultaneously in the channel. The modelling of the message communications between the nodes is possible. For the Echo Algorithm, the simulation of Spin provides a graphics of the protocol's execution (cf. Figure 6).

A clear disadvantage of the Promela model is the cumbersome modelling of the edges between the nodes. Each node has an array, in which its neighbors' nodes' are defined. This is much more cumbersome and less compact than the presentation of the edges as a function between the nodes as in the B model.

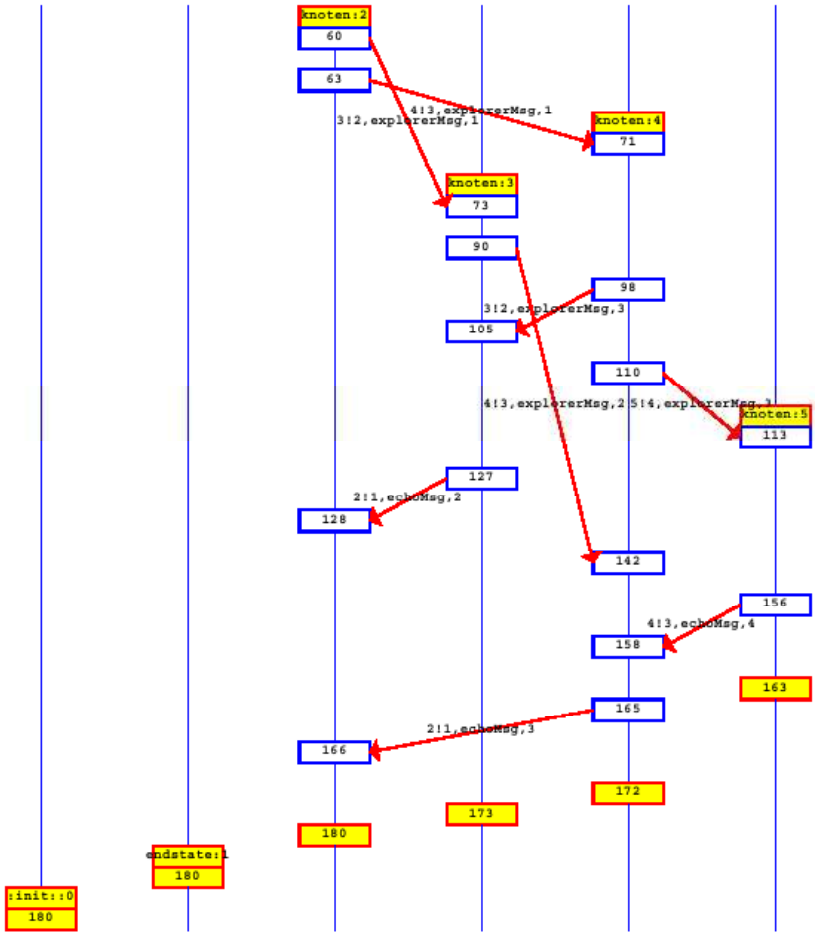


Fig. 6. The Graphical Output of the Spin-Simulation.

Despite the simple simulation of the message channels, the Promela-model is by a factor 1.4 longer than that of the B model. The same is true for the time required to create the model. With Spin, the verification is possible within about a second.

3.6 A Reservation System

Description of the Model This model specifies a reservation system for trains, which makes it possible to book, in different wagons, different kinds of seats, such as first class, second class, and so on.

The Promela model consists of a self-defined type of an array called *train*, which contains different wagons. During the initialization, Spin lacks the op-

portunity to assign the places a category. Hence, it requires much more time to create the model.

During the creation of the Promela model, it is often useful to execute a simulation with Spin in order to verify if the result is as expected. Due to the use of arrays, in particular nested arrays, the simulation with Spin is extremely slow. In this model, the Spin simulation requires, for every step of the test, about 0.9 seconds. This is a very considerable testing time, since just one seat reservation thus requires about five minutes.

For each of the reservation server, customers and vendors, there exists a process in the Promela model. The communication between these processes is done through the message channels. This is especially important for the communication between sellers and the reservation system. Another process exists, called Watchdog process, which checks if every place is marked as reserved (cf. Figure 7). The statement `assert(false)` does not guarantee the detection of an error, whenever a condition is not satisfied.

Due to high memory assignment and the considerable search needed, no available verification method of Spin could verify the reservation system model.

```
active[1] proctype watchdog()
{
  byte ii=0;
  do
  :: (ii<(MAXRESERVATIONS-1)) -> ii++
  :: (ii>=(MAXRESERVATIONS-1)) -> ii=0
  :: (reservations[ii].dest>0)
     -> if
        :: (train[reservations[ii].train]
             .waggon[reservations[ii].waggon]
             .compartment[reservations[ii].comp]
             .place[reservations[ii].place].status==belegt)
        :: else -> assert(false)
        fi
  od;
}
```

Fig. 7. A Watchdog process for the reservation system.

On the other hand, in the B model, each place is assigned a category. No nested arrays and loop traversals are needed. ProB verifies the model within 130 seconds. The verification time depends on the size of the set *seat*. The more seats exist, the longer the verification takes.

The Promela model is longer by a factor of 3.3. While the B model was developed and verified with ProB within a day, the Promela model took ten days to develop.

3.7 Needham-Schroeder Public Key Protocol

Description of the Model The Needham-Schroeder public key protocol is an authentication protocol for creating a secure connection over a public network [7]. The model consists of a network with the two normal users called Alice and Bob, an attacker named Eve and the keyserver. The first version of this protocol, developed in 1978, contains an error which was found in 1995 [6]. The error is that Eve is allowed to communicate with Bob, where Bob thinks to communicate with Alice. This error is found by ProB as well as Spin. In the corrected version of the protocol [6], ProB and Spin find no error.

In the Promela model, a separate process is created for every network user and the keyserver. The communication between the processes is easily described, since Promela supports the modelling of message channels.

The following LTL formula was used to check whether the protocol can successfully run to completion:
`<>((okAlice && okBob && aliceTalksToBob && bobTalksToAlice))`. In order to validate the protocol, the LTL formula in Figure 8 has been checked.

```
[]((okAlice && okBob) -> (aliceTalksToBob <-> bobTalksToAlice))
```

Fig. 8. LTL-Formula violated by Needham-Schroeder-Protocol

A verification of the model with Spin was impossible, because high memory was needed. A simpler version of the model, which reduces the messages sent by Eve, was created. Spin verifies if the previous mentioned bug exists, within about two seconds.

In the B model, the messages sent over the network are stored in global variables. The actions executed by Alice, Bob and Eve are created as single operations. The B model differs significantly from the Promela process-oriented model. Instead of LTL formulas, the B model used invariants (ASSERT_LTL would have been also possible). Because the verification of the first model takes several hours, similar to the Promela model, the new version of the B model was simplified. ProB finds the well-known bugs of the Needham-Schroeder public key protocol within about 150 seconds.

The Needham-Schroeder protocol is the only model, where the B source code is longer than the Promela source code (factor 1.3). However, the creation of the Promela model is about three times longer than the B model.

3.8 A Railway Interlocking System

Description of the Model In this model, signals (such as slow or green) and switches for the block sections are possible. No two trains can be on the same block section. No two crossed block sections can be entered at the same time.

The B model, the signals and the switches are assigned to functions. Variables include, for instance, the current status of the signals and the position of the train.

The model fulfills the requirements of a railway system, since it is possible to enter and move trains in the system. Moreover, invariants are formulated as conditions to meet the correctness of the system. Within approximately 1.5 hours, ProB can verify a system with an initial example of two trains, seven block sections, two switches, a crossing rail and eight signals. This complex model was created within three weeks. The model is flexible, since within a few minutes further block sections or trains can be added by changing the respective initialization of the function variable.

In the Promela model, functions and sets cannot be used. The model consists of individual processes for signals, switches, trains and the "Supervisor" in the interlocking system. The communication is via message channels.

The dependencies between signals, switches and block sections are stored in arrays, which makes the Spin simulation extremely slow. For the execution of just one single command, Spin needs about a second. Hence, the simulation of a small part of the model requires several minutes.

Despite the longer work, it was impossible to create a fully functional Promela interlocking system. The Promela model contains more code than the B model.

For the railway interlocking system, the B method is most appropriate than the Promela model.

4 Conclusion and Future Work

In summary, we have studied the elaboration of B-models for ProB and Promela models for Spin on ten different problems. With one exception (the Needham-Schroeder protocol), all B-models are more compact than the corresponding Promela models. On average, the Promela models were longer by a factor of 1.85 (counting the number of symbols) and took roughly a factor of 2-3 more time to develop than the B models. No model took less time in Promela, while some models took up to 18 times more time in Promela. One Promela model could even not be fully completed within the timeframe of this study.

More surprisingly, the study also found that ProB and Spin were comparable in practical model checking performance as measured by the user. This is despite the fact that ProB works on a much higher-level input language analysed via a Prolog interpreter, and thus being much slower (when looking purely at the number of individual states that can be stored and processed) than Spin which is compiling to C.

This study confirms our own anecdotal evidence while teaching B and Promela, as well as developing our own B and Promela models within research projects.

We believe that this study shows the merits of using a high-level specification language such as B, both in a teaching and in a research setting. Note that, based on the findings in this study, we have tried to provide some explanations for the model checking performance in this study in [4].

As far as teaching formal modelling, we believe that it is both easier for a student to write models in a high-level specification language and easier to make use of tool support to validate those models. We also believe that it maybe appropriate to give students a first introduction to model checking using a high-level language such as B, also because the high-level tools are more forgiving and more helpful in locating errors.

References

1. Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Ernest J.H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4), 1982.
3. Gerard J. Holzmann. *The Spin model checker: primer and reference manual*. Addison-Wesley, Boston, 2004.
4. Michael Leuschel. The high road to formal validation. In *ABZ*, pages 4–23, 2008.
5. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
6. Gavin Lowe. An attack on the Needham-Schroeder public key authentication protocol. *Information Processing Letters*, 56(3):131–136, November 1995.
7. Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. 21(12):993–999, Dezember 1978.
8. Harald Wiegard. Vergleich des model checkers ProB mit Spin. 2008. Bachelor's Thesis, Institut für Informatik, Universität Düsseldorf.

BiCoax, a proof tool traceable to the BBook ⋆

Samuel Colin[1] & Georges Mariano[2]

[1] Univ Lille Nord de France, UVHC
LAMIH, CNRS UMR 8530
F-59313 Valenciennes Cedex 9, France
scolin@hivernal.org
[2] Institut National de Recherches sur les Transports et leur Sécurité
georges.mariano@inrets.fr

Abstract. We introduce BiCoax, a shallow embedding of B set-theoretic first-order logic into the Coq proof assistant. This tool aims at validating the mathematical foundations of B described in the BBook and providing the B community a proof tool matching those foundations. While this is still a work in progress, BiCoax has become usable for mundane proof work in B projects.

1 Introduction

BiCoax[3] is a Coq[25] library for proving formulas defined in the first-order set-theoretic logic of the B formal method. This work is initially based upon similar efforts done by Rocheteau [23] with another proof assistant named PhoX [22] and it is available as part of the Brillant platform (https://gna.org/projects/brillant).

The original aim of this work was to provide a B *proof tool* based on a generic proof assistant, for proving B Proof Obligations. This aim was completed with a secondary objective of *validating* the mathematical foundations of B described in the BBook[1]. This book indeed introduces very precisely these mathematical foundations and claims many of their properties, sometimes providing a proof along. Unfortunately, the number of these proofs is reduced in comparison of the number of claimed properties.

Ideally, the design of any formal method shall be supported with tools when possible, e.g. for avoiding inconsistencies introduced by mere human errors. BiCoax subscribes to this approach: the use of a tool helped with the discovery a few simple typographic errors as well as minor semantic mistakes in the BBook.

⋆ The present work has been partially supported by the ANR-SETIN project: TACOS : Trustworthy Assembling of Components: frOm requirements to Specification, the European Community, the Délégation Régionale à la Recherche et à la Technologie, the Ministère de l'Education Nationale, de la Recherche et de la Technologie, the Région Nord-Pas de Calais, the Centre National de la Recherche Scientifique. The authors gratefully acknowledge the support of these institutions.

[3] BiCoax can be downloaded through Subversion (https://gna.org/svn/?group=brillant) or through a tarball placed in http://download.gna.org/brillant/snapshots/

As for providing a proof tool for the B method, BiCoax implements all operators and most theorems up to the middle of the third chapter of [1], namely the introduction of integers. This means that the missing constructs are sequences and trees, which are not as frequently used in B projects as function constructs, for instance. As a consequence, we think that BiCoax is already mature enough for being used as a proof tool in the development of mundane B projects.

After justifying the need for our implementation with respect to similar work in Sec. 2, we give a short introduction to the involved formalisms in Sec. 3. We describe our implementation choices in Sec. 4 and their consequences on the theoretical side in Sec. 5. We present a feedback of some manual and automated experiments with our tool in Sec. 6. We conclude with the various perspectives in Sec. 7.

2 Related work

2.1 Objectives

As stated in the introduction, the goal of BiCoax is to provide a proof tool for B based on the first reference on B theory [1]. It is possible to try re-implementing a proof tool in a given programming language, but using a generic proof assistant provides us with the ability to double-check the theorems already established on paper. This gives us our second goal of tracking the inconsistencies of the BBook. We left aside other objectives such as the automation of proofs, although we hope that our choice of a proof tool will help make our task easier when these objectives are given a higher priority.

Moreover, two additional parameters must be chosen when implementing through another formalism: the kind of embedding and what levels of B will be implemented. The embedding can be either *shallow* or *deep*. For B, an embedding usually will be shallow if it reuses the logic of the target formalism directly (such as the first-order logic of Coq) and it will be deep if the syntactic structure of B is modeled again in the target formalism. The implementable levels of B can be divided into the following categories: (i) mathematical foundations [1, chapters 1-3], (ii) generalized substitutions [1, chapters 4-5,9-10], (iii) theory of abstract machines [1, chapters 6-8] and (iv) refinement [1, chapters 11-13].

For a proof tool, the sole mathematical foundations are sufficient and the embedding is often, but not exclusively, a shallow one. With all the previous items we can now characterize BiCoax: it is a shallow embedding of the B set-theoretic logic in Coq. As such it reuses directly the first-order logic connectors of Coq with the axiom of excluded middle and the axiom of choice. At the time of writing, BiCoax covers the first three chapters of the BBook but the integers, the sequences and the trees, i.e. the last half of the third chapter is yet to be implemented. Let us now describe more precisely similar work.

2.2 Similar work

[23] presented a comprehensive view of other efforts directed at the validation of B or the creation of a proof tool for B. We shall sum up here the description of these efforts and amend this description with more recent work.

	Bowen [14]	Chartier [16]	Bodeveix [12]	Bodeveix [13]	Berkani [9]	Rocheteau [23]	Jaeger [19]	BiCoax
formalism	HOL	Isabelle/HOL	PVS	PVS	Coq	PhoX	Coq	Coq
embedding								
Deep		⋆	⋆		⋆		⋆	
Shallow	⋆			⋆		⋆		⋆
B level								
Foundations	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
Language		⋆	⋆					
Machines		⋆	⋆					
Objectives								
Validation		?	⋆		⋆		⋆	⋆
Tool	?			⋆		⋆	⋆	⋆

Table 1. B implementations at a glance

Table 1 sums up the existing implementations of B, what embedding they use, what levels of B they implement and whether the implementation is proposed to validate the semantics of B or to provide a proof tool. Note that we did not include "refinement" because it is also taken care of at the "Machines" level.

We also included in Table 1 an implementation of Z, because the mathematical foundations of B and Z are very similar. It turns out that the various implementations are done with usually mature proof assistants: HOL, Isabelle/HOL, PVS and Coq. The only exception is PhoX, which did not meet the same adoption despite an efficient equational reasoning engine.

Let us firstly give more details at implementations that are more than five years old. Bowen & Gordon [14] propose a shallow embedding of Z in HOL with HOL viewed as a proof tool. They justify the choice of a shallow embedding for avoiding too complex notations. The goal of Chartier [16] is the derivation of a predicate for defining formally PO generation and its validation. His implementation is a deep embedding realized with Isabelle/HOL and it supports not only the foundations of B but also the generalized substitutions and the representation of abstract machines. Bodeveix & al. [12] have a similar but less ambitious goal, as their validation involves only the generalized substitutions. This time the deep embedding is done with PVS and automated with the PBS tool by Muñoz. The work of Bodeveix & Filali [13] concerns the typechecking of B, this time with a shallow embedding in PVS. This work gave way to a typechecker which was later integrated into the BRILLANT platform. Berkani & al. [9] proposed a deep embedding of B into Coq for validating the logic rules of the prover of the AtelierB commercial tool. To the best of our knowledge, all the works we just cited have no usable tool available, either because it does not exist anymore or because the proof assistants they are based upon have evolved too much.

More recent implementations consist of B/Phox [23], BiCoq [19] and BiCoax. Rocheteau [23] introduces a shallow embedding of the foundations of B in PhoX: the set constructs are translated into PhoX[22], this translation being proved correct. This work is now abandoned because its direct successor is actually BiCoax: the very first working source code of BiCoax is a translation of the PhoX constructs into Coq.

We are left with comparing BiCoax with BiCoq [19]: the objective of Jaeger & Dubois can be seen as a gathering of all the objectives of the previously cited works. Their goal is to validate the theory of B and propose a proof tool for B. They realize a deep embedding of B into Coq. Choosing a deep embedding avoids the interference of the classical logic of B with the more intuitionistic logic of Coq and allows the proposition of decision procedures. Their implementation also has the peculiarity of using De Bruijn indices for quantified variables. [19] does not make it clear what parts of [1] are covered, although we can infer that they include the first two chapters.

With respect to all the works presented here, we can see that the most relevant work to compare our efforts with is BiCoq [19]. The most recurring question in comments about earlier presentations of BiCoax was about the reuse of BiCoq: why not develop on the base of BiCoq ? The first answer is purely practical: several attempts to contact the authors of BiCoq before and during the development of BiCoax were unsuccessful, thus we simply did not have BiCoq at disposal. We then were deprived of the most straightforward option of reusing BiCoq.

The second answer is about the durability and the availability of BiCoax. We wanted to avoid the fall into oblivion from which all the previous work, B/PhoX [23] excepted, seem to have suffered. We also wanted to include BiCoax in the tool suite of the BRILLANT platform, which requires the release of BiCoax under open-source license terms. As a result, BiCoax is now part of the BRILLANT platform and is available for anyone to try. It is also durable as it is proposed in the same conditions as B/PhoX, which was the only work we could reuse in the end. Moreover, using Coq as an underlying proof tool makes the durability of BiCoax linked to the effective durability of Coq.

In light of the previous comparisons, we thus make the strong but independently verifiable claim that BiCoax is the *most complete* academic tool for proving B or event-B projects, *complete* to be read in the sense of "matching the BBook".

We shall conclude this section by mentioning the provers of B commercial tools. Their main objective is very pragmatic: having an efficient automated proof tool for a possibly big number of proof obligations. They function mostly by saturation of premises until a contradiction in the hypotheses is found. They also include mechanisms for the users to specify their own decision procedures. For instance, the prover of the B-toolkit includes a tactic language comparable to the tactic language of Coq, although much simpler. Those commercial provers are also based upon an adapted B set-theoretic logic, hence comparing them with academic tools is somewhat less relevant. What makes these tools interesting is their efficiency: they are thus a good choice for evaluating the performance of B academic proof tools.

3 A short presentation of the involved formalisms

3.1 The B and event-B formal methods

B is a formal software development method used to model and reason about systems [1]. The B method has proved its strength in industry with the development of complex real-life applications such as the Roissy VAL [7]. The principle behind building a B model is the expression of system properties which are always true after each evolution step of the model. The verification of a model correctness is thus akin to verifying the preservation of these properties, no matter which step of evolution the system takes. A successor of B called event-B[21] takes the modelling further in allowing the reasoning about systems in an event-based fashion. The verification step of an event-B model is done in a similar way as B with taking into account deadlock problems due to the fact the formalism is event-based.

In both cases, the verification entails the generation of so-called *proof obligations* (POs) which are set-theoretic first-order logic formulas to be proved in the context of the B or event-B theory. PO generation for B is supported by several tools like B4Free[6], , AtelierB[4] , the B-toolkit[5] or the BGOP of our BRILLANT [18] platform. For event-B, the only tool is Rodin[24]. The proof is also supported by the tools mentioned previously except for the BGOP, whose only task is PO generation. Proof in BRILLANT shall be supported by BiCoax.

There are a few differences in the theory of B and event-B, though. Event-B gives different definitions for some of the basic constructs,supposedly compatible with B. As BiCoax is primarily about validating the BBook, we shall focus in this document on B alone, although we might occasionally mention event-B to state how our implementation fares or shall fare in an event-B context.

3.2 Coq

Coq [25] is a proof assistant based on the calculus of inductive constructions. This implies the following important points:

- The logic is an intuitionistic one, hence properties such as excluded middle or the axiom of choice, fundamental in the logic of B, are not available immediately. Coq fortunately provides modules in its libraries for such axioms
- Types are first-class citizens. It is possible to build datatypes associating basic values with types or to make the existence of a datatype rely upon the validity of a proposition, for instance.

Types are also organized along a hierarchy of *sorts*:

Set is the sort of program types. If this datatype is supposed to be implementable, this sort shall be preferred. Natural numbers and integers belong to `Set`, for instance

Prop is the sort of propositions, hence logical formulas

Type is the sort of `Set` and `Prop` and it constitutes the rest of the type universe hierarchy: a datatype built upon another datatype of $Type_n$ will inhabit $Type_{n+1}$, although these indices are hidden to the user.

The following example shows how the union property of two sets is modeled in the ENSEMBLES module of Coq's standard library:

```
Inductive Union (B C:Ensemble) : Ensemble :=
  | Union_introl : ∀ x:U, In B x → In (Union B C) x
  | Union_intror : ∀ x:U, In C x → In (Union B C) x.
```

This datatype can be interpreted two ways. In the propositional perspective, Union_introl and Union_intror are axioms describing the properties of the union from the notion of set belonging. In the type theory perspective, Union_introl is an element of type **Union B C** and it is parametrized by a (set belonging) property.

Another important functionality of Coq in our context is the ability to write *tactics* for automating repetitive proof tasks. Thanks to constructs reminiscent of a programming language and to pattern-matching, it is possible to associate specific proof steps to goals of a known shape. These proof steps can involve existing tactics and theorems.

4 Implementation choices

After a short presentation of the targetted organization of our library and of the design choices made before implementation, we exhibit its particularities.

4.1 Organization and design choices

Ideally, it should be possible to trace BiCoax theorems and definitions back to the corresponding entry of the BBook. We thus chose to split BiCoax the same way as the part of the BBook we model. The BCHAPTER1 module introduces the theorems of first-order classical logic (we reuse the logical connectives of Coq), the BCHAPTER2 introduces basic and derived set constructs and BCHAPTER3 introduces high-level constructs. In order to manage code size, we chose to have one file for each section. There are two noticeable exceptions: the first chapter and properties listings. The first chapter holds in one single file because no definitions are introduced and proofs are short. The properties listings of the BBook follow the "one table = one file" policy. Finally each definition or theorem shall refer in a comment the corresponding page, section and table row of the BBook, when applicable. Table 2 sums up what sections of the BBook are modelled, into how many and which files.

When implementing the various definitions of the BBook, we used the following guideline: if a matching definition exists in the standard library of Coq, we use it. If not, we implement it preferably as an inductive definition: this constitutes more of a stylistic choice which is consistent with the module of the standard library we reuse the most. In all cases, BiCoax is initially based on B/PhoX[23] which means that the equivalence or the equality of each translated operator shall be assessed. Hence each BiCoax module containing definitions will also contain theorems showing the equality or the equivalence of the definition with the corresponding BBook definition.

BiCoax might also be used by non-specialists in the future: POs can be big, hence readability of formulas is an important concern. Coq allows the definition of additional Unicode notations and event-B introduced Unicode notations for B constructs [21].

Chapter	BBook section	Filled-in files	files
1	All	1	BCHAPTER1
2	Basic set constructs	2	BBASIC, BINCLUSION_PROPS
	Derived constructs	2	BDERIVED_CONSTRUCTS{,_PROPS}
	Relations	1	BRELATIONS
	Functions	1	BFUNCTIONS
	Catalogue of properties	13	..._LAWS, EQUALITIES_...
3	Generalized Intersection/Union	1	BGENERALIZED_UNION_INTER
	Fixpoints	1	BFIXPOINTS
	Finite sets	1	BFINITE_SUBSETS
	Infinite sets	1	BINFINITE_SUBSETS
	Natural numbers	13	BNATURALS{_BASICS,...}
	Integers	1	BINTEGERS{_BASICS,...}
	Finite sequences	0	BSEQUENCES
	Finite trees	0	BTREES
	Labelled trees	0	BLABELLED_TREES
	Binary trees	0	BBINARY_TREES
	Well-founded relations trees	0	BWELL_FOUNDED

Table 2. Global organization

We thus decided to introduce a notation scope that can be activated when necessary. This notation scope also follows the guidelines of [21] for associativity and priority of notations. Some event-B notations are in a "private" zone of Unicode because no corresponding symbol exists officially: in that case we decided to reuse other similar-looking symbols. Here follows an example of Unicode notation for set belonging:

`Notation` "x ∈ y" := (In *y x*) (`at` *level* 11, *no associativity*): *eB_scope*.

Many definitions are parametrized by the types of the sets they manipulate. In that case we declared these type parameters implicit so that Coq infers them. This design choice makes the formulas feel less crowded.

We shall now detail what parts of Coq were reused and when not, what changes we introduced w.r.t. the definitions of the BBook.

4.2 From B to Coq

Reused constructions. BiCoax is a shallow embedding, hence we reuse some parts of Coq for our benefit.

The most basic connectors are directly reused, i.e. logical connectors (conjunction, disjunction, negation, etc) and pairing. On a related note, reusing the pairing of Coq shielded us from discovering the fact that the injectivity of pairing is never showed in B, as noticed by Jaeger in the French version of [20] at the end of Sec. 8.2.

We reused the definitions of the Coq's standard library ENSEMBLES module for basic set constructs: set belonging, inclusion, union, intersection, empty set, set difference. This reuse is also justified by the fact that set belonging in this module match very closely set belonging in B. Let U be a datatype and A a value of type (Ensemble$U$): A is actually a function of U to propositions ($U \rightarrow Prop$). Set belonging is defined as :

```
Definition In (A:Ensemble) (x:U) : Prop := A x.
```

This means that for the ENSEMBLES module, belonging to A is the same as verifying the A predicate, which matches exactly the B definition of set belonging.

We also imported the CLASSICALEPSILON module for the choice operator as well as related constructs and tactics proposed by Castéran [15]. We could not reuse the notion of relation of the ENSEMBLES module because it lacks the property being itself a set (of couples), hence we modeled relational and functional constructs in the "B way" upon the basic set constructs mentioned above.

Earlier communications about BiCoax raised the question of why we decided to reuse the ENSEMBLES module instead of FSETS. This is because this module requires an ordered datatype upon which sets of this datatype can be built. In B, the only basic datatype is that of the BIG abstract set: as we have no information about the order of its elements, it was thus not possible to use FSETS. Let us note here that it would be possible to do so with event-B as it defines the natural numbers as a basic datatype, which we know are an ordered type.

Our rule of thumb for definitions is to use inductive constructions when possible. The third chapter of the BBook introduces at its beginning the notion of fixpoint and makes almost systematic use of it for defining the rest of B constructs: this is where our choice of Coq becomes the most fruitful as inductive (and thus fixpoint-based) constructs are pervasive in Coq. As a consequence, all inductive constructs of B are translated into inductive definitions. The existing inductive definitions we reused were the notion of finite sets as the **Approximant** of the ENSEMBLES module, the notion of natural number and all arithmetic operators for natural numbers of the ARITH module.

Proposed constructions When not reusing definitions of the standard library of Coq, we had to balance ease-of-use and traceability to the BBook of the definitions we proposed. While it is not the place here to present all definitions we introduced, we think that the definition of partial function sums up what challenges choosing a definition poses. The BBook suggests two equivalent definitions of partial function:

- `{f | f ∈ A ↔ B ∧ (f⁻¹;f) ⊆ id(B)}`
- `{f | f ∈ A ↔ B ∧`
  `∀(x,y,z).(x,y,z ∈ A×B×B ∧ x↦y ∈ f ∧ x↦z ∈ f ⇒ y = z)}`

The second definition make explicit the image unicity property for functions, which is often used in proofs involving partial functions. This property can of course be deduced from the first form of the definition, but it makes it less immediately usable for the end-user. In the second definition, the type information about x, y and z seems redundant: it should be deducible from the belonging of $x \mapsto y$ and $x \mapsto z$ to f. These remarks lead us to the final shape of the corresponding BiCoax definition:

```
Inductive partial_function (U V: Type) (A: Ensemble U) (B: Ensemble V): Ensemble
(Ensemble (U*V)) :=
  pfun_intro : ∀ (f: Ensemble (U*V)),
    In (relation A B) f
  → (∀ (x: U) (y: V), In f (x,y) → (∀ (z: V), In f (x,z) → y=z))
  → partial_function U V A B f.
```

The use of a curryfication of pfun_intro arguments instead of a conjunction also removes one or two additional decomposition steps in proofs later on.

In the end, we believe it to be easy for the end-user to manipulate the various definitions. The danger is then that the definitions we propose do not match the BBook definitions anymore. This imposes on us the verification that our definitions are equivalent to the BBook definitions.

4.3 Validity of modified constructs

Let us look at the definition of shallow embedding of a B term in PhoX as defined in [23], where † is a translation function:

$$(f\ t_1 \cdots t_n)^\dagger \equiv f^\dagger\ t_1^\dagger \cdots t_n^\dagger$$

For the embedding to be correct, it must guarantee that it is the same to handle a translated complex term or to handle the same term for whom each subterm is translated. When the embedding is a deep one, this verification can be assisted by the tool used for it. Because our embedding is a shallow one, we can also do such a verification but it is meaningful only for connectors directly reused: for instance, proving with Coq that $(P \leftrightarrow Q) \leftrightarrow (P \leftrightarrow Q)$ is trivially verified.

Fortunately this meaninglessness is limited to logical connectives. For other constructs, the verification, even if done within Coq itself, is more meaningful. When the definition is predicative we use the logical equivalence of Coq and for terms we used Coq's Leibniz equality, whether they be sets or other constructs. Here we advise the reader to see e.g. the verification of the validity of set union in the BDERIVEDCONSTRUCTS module.

When dealing with datatypes the verification becomes a bit more convoluted as the operators defined upon this datatype must also be transformed. We thus introduce homomorphic functions for translating one datatype into the other and we verify that one operator in the one datatype is isomorphic with the corresponding operator in the other datatype. For instance, in the BNATURALS* modules the homomorphisms are nat_of_bbN and bbN_of_nat and for all arithmetic operators (except logarithm at the time of writing) the isomorphism is verified.

All the design choices we presented here gave us a clear guideline for implementing the first part of the BBook and we present some theoretical results and remarks in section 5.

5 Theoretical results

So far BiCoax amounts to about 27000 lines, or 768KB. It contains 1163 theorems and lemmas and was written with 3 estimated person-months. What has been implemented is split between:

Chapter 1 All non-trivial theorems about first-order logic with predicates have been proved. This part is somewhat trivial but what makes it most interesting is seeing which theorems actually require the excluded middle for being proved

Chapter 2 All this chapter was implemented and all properties were proved

Chapter 3 All sections up to and excluding integers have been implemented. The only theorems left unproved at the moment are the Dedekind-infinity theorem for infinite sets and the properties about natural logarithm, for which we also proposed a definition purely based on the nat datatype of Coq.

At this point, we found very few mistakes in the BBook, which makes it a solid reference for basic mathematical concepts. The mistakes include:

- What we think are typographical errors, e.g. a u instead of a v in a hypothesis
- False properties, the ones numbered 33 and 34 in MEMBERSHIP_LAWS. Given their location, they are most likely copy/paste errors
- Property 2.5.1, where the right-to-left implication is actually false. This mistake most probably occurred because one of the hypotheses needed for proving the definition was overlooked (which the use of a tool would not allow). As a consequence the proof of theorem 3.5.3 is wrong, while the theorem itself is true: we did the proof differently, because the theorem 3.5.3 is needed after
- Properties of addition: the codomain of the addition was deduced to be $\mathbb{N}$ from theorem 3.5.3, while it can only be deduced to be $\mathbb{P}(BIG)$. Many definitions depend on addition, hence blindly following the theorems would have induced too many chances. As a consequence we decided to define the addition upon `(ℕ ◁ succ)`, which leads to the desired properties, instead of `succ`.

We put counterexamples and more detailed explanations in an ERRATA file distributed along with BiCoax. The following sections present interesting or difficult points pertaining to our implementation.

5.1 Axioms in BiCoax and dependability

The dependability of BiCoax can be attributed to the trustworthiness of the following items: Coq, the axioms we included and the non-inconsistency brought by the introduced axioms. Coq has existed for about two decades and many academic and industrial users trust it, hence we will not discuss it further here.

The very fundamental axioms we needed and thus included are: the excluded middle (EM), the constructive indefinite description (epsilon) and set extensionality (Set_ext). The derived axioms we introduced so far are the infinity of the *BIG* set and the axioms related to integer negation for defining the negative part of the set of integers in the BBook. If BiCoax were to be inconsistent, it would thus come from the use of these axioms which are necessary for our implementation.

Without entering into details, we know that the excluded middle, while making a strong assumption about the decidability of statements, does not cause inconsistencies by itself. Used with the axiom of choice, it implies proof-irrelevance [8], which is still not inconsistent. As epsilon can be seen as a weaker form of the axiom of choice, it is then very possible that proof-irrelevance is present in our implementation. We also know that epsilon allied with the impredicativity of Coq's Set sort leads to inconsistency [15, introduction]. As of version 8.0, Set is predicative by default hence we also avoided inconsistency here.

As a consequence, the other places from where inconsistencies might originate are the derived axioms coming from B: it can thus be said that BiCoax can be considered as dependable as B. We however do not plan to scrutinize these axioms in the near future to look for potential inconsistencies.

5.2 Pitfalls of function application

As reckoned by Jaeger [19, section 3], implementing function application in a shallow embedding is a tricky exercise. According to the BBook, functional application requires the choice operator (epsilon) and a relation having the property that the image of any of its element is unique (the functional property). As we used Castéran's [15] *unique choice* iota operator, unicity becomes an intrinsic property of the image.

```
Definition app (U V:Type) (A: Ensemble U) (B: Ensemble V)(f: Ensemble (U*V))
(x:U) (applicability: In (partial_function A B) f ∧ In (domain f) x) :=
  iota V
    (codomain_unique_inhabitation U V A B f x applicability)
    (fun y:V ⇒ f (x,y)).
```

This might be the definition the most foreign to its B origin, as the problem of its soundness in Coq comes much into play: it requires an additional parameter which is a proof that the datatype of the codomain is inhabited. Fortunately such a proof can actually be deduced from the the mandatory well-definedness side-condition of functional application. The definition above is thus what we think is the best trade-off between Coq soundness and friendliness to the end user, who is likely to have already seen proofs of well-definedness in other B tools. Its use in formulas requires an additional proof of existence of well-definedness. This explains why the EQUALITIES_EVALUATION module contains so many existential quantifications.

5.3 The types of B and Coq

As expected, BiCoax exhibits some of the peculiarities of B typing. For instance, any set must be based on a given datatype: hence there is not one but several empty sets [1, section 2.3.3]. Overlooking this fact when implementing a prover "from scratch" might lead to inconsistencies. We still could make it look like there is only one empty set without breaking typechecking thanks to Coq's notation mechanism:

```
Notation "∅" := (Empty_set _) (at level 10, no associativity): eB_scope.
```

Coq is indeed able to infer type parameters when they are simple enough: this permitted us to propose a unified notation for the empty set.

We also removed in some definitions the "B typing" parts, i.e. the predicates stating to what sets the bound variables belong. Indeed, these predicates are sometimes redundant (see e.g. the partial function definition of Sec. 5.2). Systematically proving the equivalence of the new definition with the corresponding BBook definition comforted us in our action.

5.4 Natural numbers and arithmetic operations in BiCoax

Implementing the part of the BBook introducing natural numbers faced us with a dilemma: reusing the convenient Coq's natural numbers or strictly following the BBook. In order not to sacrifice usability, we decided to do both, finding easy-to-overlook peculiarities in the process. For using both datatypes, we had to propose an isomorphism in the form of two homomorphims: bbN_of_nat is a recursive function with a straightforward definition similar to the B fixpoint construct for natural numbers. The nat_of_bbN homomorphism is simply the cardinal of the B natural, which is a subset of BIG.

We proved that B arithmetic operations and Coq's functions are equivalent, sometimes *under conditions*. Coq functions are indeed total w.r.t. their arguments (e.g. $0-1=0$ for natural numbers in Coq) while B operations are partial ($x-y$ is defined under the condition that $x \geq y$). This suggests that for any other tool implementing B naturals, the partiality of these operations may have been overlooked: hence some formulas might be provable when they should not.

As a conclusion for the theoretical side of BiCoax, we can state that our work, while not ground-breaking, is useful. It helped the trimming down of previously uncovered mistakes of the BBook. It has also helped and will help in stating how B types and structures match their "programming-like" counterparts as we did for arithmetic operations on natural numbers. We shall now present how BiCoax fares on the more practical side.

6 Experimental results

Experimentations with BiCoax were twofold: using it in a purely automated way for determining what the tools in the upper part of the toolchain are expected to provide and interactively with B toy projects for getting a better idea of the completion of BiCoax.

6.1 At the end of the B toolchain

We inherited from B/Phox, the ancestor of BiCoax, a way of generating PhoX proof obligations from B POs described in an XML format. We adapted the XSL stylesheet for doing so to Coq instead of PhoX, resulting in the `bgop2bicoax` XSL stylesheet. The transformation from a PO to BiCoax is straightforward: after inserting the importing of the BLIB module of BiCoax, the formula is translated to a Coq theorem using the B constructs defined in BiCoax. A tactic call is inserted, followed by a proof-saving command. Nowadays this part of the toolchain is mostly used for correcting the PO-to-BiCoax transformation step.

Our first tests were with a publicly available B project, the boiler, in a "case-study" flavor and an "industrial" flavor. The PO generator of BRILLANT issues 2335 POs for the case-study boiler and 2563 for the industrial boiler. PO generation is as close to the atelierB as possible so as to make subsequent comparisons more relevant. In a nutshell, successful proofs (see Tab. 3) correspond to typing proofs and are realized in 45s on average with the (still experimental) `firstorder` Coq tactic. Failures come mostly from B disambiguation errors (such as cartesian product and multiplication) or XSLT transformation errors (missing constructors, etc). The various errors we encountered are as follows:

T1: `Cannot infer an instance for the implicit parameter U of app`
T2: `The reference ... was not found in the current environment`
T3: `Cannot infer a term for ...`
T4: `',' expected after [binder_list] (in [constr:binder_constr])`
T5: `Cannot infer a type for ...`
T6: `The term "..." has type "..." while it is expected to have type "..."`
T7: `Attempt to save an incomplete proof`

	Case-study	Industrial
Nbr PO	2335	2563
Nbr BPhoX	1871	43
Nbr BiCoax	237	52
Error T1	1446	24
Error T2	310	2450
Error T3	175	8
Error T4	66	15
Error T5	25	1
Error T6	20	-
Error T7	13	11

Table 3. Proof results

The discrepancy of T1 and T2 errors in the proof results seems to be a consequence of the more complex shape of expressions in the industrial version of the boiler. In the end, the use of BiCoax in an automated fashion seems possible but will require corrections in the upper part of the toolchain in the near future.

6.2 Using BiCoax interactively

It is our belief that the interface to proof tools should be easy to use interactively, because on non-trivial B projects the end-user will spend most of his/her proving-time issuing commands to the prover (the automated part is invisible, as expected). Hence interactive proof shall be made easy, which was also part of our motivation in using Coq as a proof tool.

As a quick test of the usability of BiCoax, we tried to prove two B toy projects, the "LittleExample" project appearing in [1, Sec. 11.2] or [19] and a bounded stack, with no further assumptions than the POs translated manually. As a result, the proofs were successful and took no more than half an hour for each project. They are present in the BMISC module of BiCoax along with the corresponding B projects in comment. It is this somewhat unexpected success that turned our opinion to the idea that BiCoax is almost ready for mundane proof tasks. Here follows a top-down proof tree of the demonstration of the PO assessing that the `read` operation preserves the invariant of the "LittleExample" machine:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \forall(n,y).y \in FIN(\mathbb{N}_1) \wedge n \in \mathbb{N}_1 \Rightarrow (y \cup \{n\}) \in FIN(\mathbb{N}_1)}{n : Z, y : (Ensemble\ Z), H : (y \in FIN(\mathbb{N}_1) \wedge n \in \mathbb{N}_1) \vdash (y \cup \{n\}) \in FIN(\mathbb{N}_1)}\ \text{intros}}{n : Z, y : (Ensemble\ Z), H : (\ldots) \vdash (\{n\} \cup y) \in FIN(\mathbb{N}_1)}\ \text{rewrite commutativity\_1}}{n : Z, y : (Ensemble\ Z), H : (\ldots) \vdash (n,y) \in (\mathbb{N}_1 \times FIN(\mathbb{N}_1))}\ \text{apply augmented\_set\_in\_finite\_sets}}{\ldots, H : (y \in FIN(\mathbb{N}_1) \wedge \ldots) \vdash y \in FIN(\mathbb{N}_1)}\ \text{intuition} \qquad \dfrac{}{\ldots, H : (\cdots \wedge n \in \mathbb{N}_1) \vdash n \in \mathbb{N}_1}\ \text{intuition}\ \Big]\ \text{constructor}$$

We perceived that our experience with proving the above projects could be better though: some very trivial theorems were missing, such as the non-emptiness of a singleton, and tactics could have helped us solving repetitive proof tasks. In the future, such interactive sessions shall give us directions for automating the proof of B projects, maybe to the point of proposing specialized tactics for domain-specific B projects.

Because the projects we proved are not very complicated, we could not illustrate here the full extent of using other tactics provided by Coq. For instance, when proving

equality of arithmetic equations we can use the `ring` tactic: it normalizes arithmetic terms and attempts to prove their equality. This tactic is actually defined for structures that form a ring (hence here, natural numbers) but it could be applied to any other ring (e.g. one formed with set operations). Another high-level tactic called `omega` follows the same idea for mixes of equations and inequations in Presburger arithmetic.

7 Conclusion

We have proposed a shallow embedding of the mathematical foundations of B into Coq, in order to provide a proof tool for B based on a generic proof assistant and in order to validate the definitions and theorems of the BBook through implementation. When relevant, we proposed handier alternative definitions, either ours or from Coq's library. We systematically proved the equivalence with the corresponding B definitions in the process, to the extent allowed by a shallow embedding.

Our implementation covers the first two and a half chapters of the BBook up to and almost including the integers, which amounts to about 1100 theorems. This implementation helped us uncover minor BBook mistakes not documented elsewhere. It also brought more focus on confusing or overlooked parts of B. For instance, the difference between "B typing" as set belonging and typing in the sense of type theory is hopefully clearer. Moreover we exhibited potential pitfalls of the partiality of B arithmetic operators (subtraction, division and natural logarithm) which might have been overlooked in other implementations of B.

Experimental results showed that an automated use of BiCoax is feasible but it mostly requires changes in the upper part of the B toolchain. The ease, initially unexpected, with which we proved two B toy projects leads us to believe that BiCoax is almost ready for the proof of mundane B projects.

The short-term perspectives include what we know or perceive BiCoax lacks: the sections of the BBook left to be implemented, trivial theorems not present in the BBook and tactics for repetitive proof tasks. Long-term perspectives include the implementation of several of the numerous B extensions that appear in the literature. Such extensions consist for instance in the introduction of other datatypes in B or the extension of B with a temporal logic [17].

Acknowledgements We would like to thank Arnaud Lanoix for proof-reading earlier versions of this paper.

References

1. Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.
2. AFADL2000, LSR/IMAG. *Approches Formelles dans l'Assistance au Développement de Logiciels*, LSR/IMAG – BP 72 38402 Saint-Martin d'Heres Cedex – Grenoble – France, January 2000. LSR/IMAG.
3. AFADL2003, IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.

4. AtelierB. http://www.atelierb.eu.
5. B-Toolkit. http://www.b-core.com.
6. B4Free. http://www.b4free.com.
7. F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *Formal Specification and Development in Z and B*, volume 3455 of *LNCS*, pages 334–354. Springer-Verlag, 2005.
8. Franco Barbanera and Stefano Berardi. Proof-irrelevance out of excluded-middle and choice in the calculus of constructions. *Journal of Functional Programming*, 6:519–526, 1996.
9. Karim Berkani, Catherine Dubois, Alain Faivre, and Jérôme Falampin. Validation des règles de base de l'Atelier B. In *AFADL'2003* [3], pages 121–136.
10. Didier Bert, editor. *B'98 : The 2nd International B Conference, Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science (Springer-Verlag)*, Montpellier, April 1998. B1998, LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer-Verlag.
11. Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors. *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, Grenoble, France, January 2002. LSR-IMAG.
12. J.-P. Bodeveix, Mamoun Filali, and C. A. Munõz. Formalisation de la méthode B en COQ et PVS. In *AFADL'2000* [2], pages 96–110.
13. Jean-Paul Bodeveix and Mamoun Filali. Type synthesis in B and the translation of B to PVS. In Bert et al. [11], pages 350–369.
14. Jonathan P. Bowen and Michael J. C. Gordon. A shallow embedding of Z in HOL. In *Information and Software Technology*, pages 269–276, 1995.
15. Pierre Castéran. Utilisation en Coq de l'opérateur de description. pages 30–44, January 2007.
16. Pierre Chartier. Formalisation of B in Isabelle/HOL. In Bert [10], pages 66–82.
17. Samuel Colin. *Contribution à l'intégration de temporalité au formalisme B : utilisation du calcul des durées en tant que sémantique pour B*. PhD thesis, Université de Valenciennes et du Hainaut-Cambrésis, October 2006.
18. Samuel Colin, Dorian Petit, Jérôme Rocheteau, Rafaël Marcano-Kamenoff, Georges Mariano, and Vincent Poirriez. BRILLANT : An open source and XML-based platform for rigourous software development. In *SEFM (Software Engineering and Formal Methods)*, Koblenz, Germany, September 2005. AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press.
19. Éric Jaeger and Catherine Dubois. Why would you trust B ? In *LPAR, 14th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 288–302, 2007.
20. Éric Jaeger and Thérèse Hardin. A few remarks about formal development of secure systems. In *HASE '08: Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 165–174, Washington, DC, USA, 2008. IEEE Computer Society.
21. C. Métayer, J.-R. Abrial, and L. Voisin. RODIN deliverable 3.2: Event-B language, May 2005. http://rodin-b-sharp.sourceforge.net.
22. Christophe Raffalli and Paul Rozière. *The PhoX Proof checker Documentation*. version 0.83.
23. Jérôme Rocheteau, Samuel Colin, Georges Mariano, and Vincent Poirriez. évaluation de l'extensibilité de phoX: B/PhoX un assistant de preuves pour B. In Valérie Ménissier-Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 37–54. INRIA, 2004.
24. Rigorous Open Development Environment for Complex Systems. http://www.event-b.org/platform.html.
25. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. http://coq.inria.fr/doc-eng.html.

Constructing a Formal Event Model of Linux File System Access Permissions: A Research Based Teaching Approach

David Cumbor and Bill Stoddart
School of Computing, Univ of Teesside, UK
Email D.Cumbor@tees.ac.uk

School of Computing, University of Teesside, Middlesbrough, UK

Abstract. Based on documentation of the Unix file system, supplemented by examination of Unix configuration files and experimental observations, we develop an Event B style model of Unix file permissions and associated operations. The work is presented in the context of a "research based teaching" approach, in which practical investigations play a leading role and formal event modelling is used to capture our emerging intuitions in a precise way.

1 Introduction

In a previous application of Formal Methods to operating systems [6] Morgan and Sufrin comment: "As anyone who has ever used an operating system will confirm, the manuals cannot tell the whole story about the behaviour of the system. Indeed, almost every programmer .. sets up a number of *experiments* by which she attempts to discover how it really behaves."

Morgan and Suffrin introduce their paper with the remark "The Unix Operating System is widely known and its filing system is well understood". Nevertheless, the formal modelling of Unix file access is still a current research topic, with one recent contribution (2008) being "Some aspects of Unix file-system security by Markus Wenzel"[8] in which Isabelle is used as the modelling environment.

In this paper we describe a research based teaching approach to investigating Unix file access permissions through the creation of an Event B style model. Our model is based partly on documentation of the Unix file system, as given in Unix "Man" and "Texinfo" files, and as described in text books such as "The Unix Programming Environment"[5]. However, this is supplemented by examination of Unix configuration files and experimental observations on a Unix (Linux) system. The work is presented in the context of a "research based teaching" approach, as described in the book [7]. Practical investigations play a leading role; formal event modelling and natural language descriptions are used to capture

our emerging intuitions in a precise way. Rather than being studied in isolation and as something apart from practical computing, formal methods are seen here as a tool to be used in a practical investigation; the formal description of a fairly complex system takes place in a step by step manner at the same time as practical knowledge is gained relating to the system's operation.

Students are presented with the given sets of the model. A structured discussion leads to the emergence of the constants, variables, invariant clauses and events. They are asked to devise experiments that test some given hypotheses about the emerging model, and also to formulate such hypotheses themselves. Successful hypotheses — those that withstand rigorous testing - are used to formulate the constraints of the model. Other student exercises associated with this project include the provision of Pearl scripts which mark up the textual output of experiments for inclusion in LaTeX documents.

The investigations cover users, groups, files (including directories and paths), individual file owners, group file owners, the file access permissions of owners, groups and others, and the effect on these permissions on various actions taken on files. Enough is reported here to illustrate our approach, but we do not provide a complete model.

An essential part of the process is to describe in an abstract state which serves as a foundation for a conceptual model. Some components of the abstract state will correspond to information held within the Unix operating system, whilst others describe conceptual elements, such as "groups", and "users" and "files". The associated dynamic part of the model would describe changes to the abstract state brought about e.g. by adding users and groups, associating users with groups, and changing ownerships and permissions. They also describe the conditions under which certain events, such as reading a file, can occur. We are concerned with how the description of events emerges through a process of refinement as experiments progress, rather than with the presentation of completed model, which is reserved for a later publication.

We present the model in Event B [2, 3], though we do not adhere to machine readable notations: it is convenient for us to adopt a freer format, and for succinctness we base our notations on Abrial's Generalised Substitution Language. Natural language descriptions are woven into the formal text. Definitions of notations used may be found in [1].

The rest of the paper is organised as follows. In Section 2 we outline our method of conducting experiments. In Section 3 we give some formal background relating to our use of pre-conditions as well as guards in an event based model. In Section 4 we describe the static aspects of our model. In Section 5 we study the development of an event from the model and the use of both pre-conditions and guards during refinement, and in Section 6 we draw conclusions and mention future work.

2 The Experiments

We consider the development of our model to have been a simple adaptation of the "scientific method", with the aim of re-discovering the conceptual design of a human artifact rather than investigating the natural world.

Our experimental material is the Unix file system with its associated file permissions, and the users and groups of the Unix system.

As with other branches of science, it is important to perform experiments within a controlled environment to avoid extranious side effects. For example we avoid consideration of block devices, and the effect of mounting a block device already containing files created on another Unix system.

We admit that any model and associated documentation produced in this way is always provisional - it is based on the few machines we have been able to observe. It is always possible that our observations have exceptions, or are based on some "hidden state" such as a particular configuration value. Nevertheless, the potential refutability of our model is also a strength, as it can lead to the search of a wider model of which the present model will form a part.

As examples of hypotheses we might wish to investigate:

Hypothesis 1. A non-superuser requires appropriate permissions to access a file, but a superuser can always access a file, no matter what permissions are set.

Hypothesis 2. All users require suitable permissions to be set before accessing a file, and the superuser can always set permissions on any file.

These hypotheses are obviously incompatible, and are investigated by experiments whose results are given below. The results are cut and pasted from an xterm, and some notes have been added by the student who performed the experiments. Lines commencing with \% are comments from the supervisor suggesting additional points that should be investigated.

```
We will create files as user ben and as Superuser (root), then
remove permissions from the files and see what abilities we have.

1.1 Control User

Here, as the prompt shows, the current user is ben

ben@doomed ~ $
ben@doomed ~ $ touch myfile
ben@doomed ~ $ ls -l myfile
-rw-r--r-- 1 ben ben 0 Feb 25 11:09 myfile
ben@doomed ~ $ echo "test me" > myfile
ben@doomed ~ $ cat myfile
```

```
test me
ben@doomed ~ $ chmod u-rw myfile
ben@doomed ~ $ ls -l myfile
----r--r-- 1 ben ben 8 Feb 25 11:10 myfile
ben@doomed ~ $ cat myfile
cat: myfile: Permission denied
ben@doomed ~ $
```

The user ben created a file called 'myfile', then we removed the owner's read and write permissions from the file. The user ben was then unable to access the file.

% There is an assumption here that when current user creates a
% file, that file will be owned by the current user. An inter-
% action should be added confirming this.

1.2 Super User

Here the prompt shows the current user to be root.

```
doomed ~ # touch sufile
doomed ~ # ls -l sufile
-rw-r--r-- 1 root root 0 Feb 25 11:11 sufile
doomed ~ # echo "testt file" > sufile
doomed ~ # cat sufile
testt file
doomed ~ # chmod u-rw sufile
doomed ~ # ls -l sufile
----r--r-- 1 root root 11 Feb 25 11:11 sufile
doomed ~ # cat sufile
testt file
doomed ~ # chmod 000 sufile
doomed ~ # ls -l sufile
---------- 1 root root 11 Feb 25 11:11 sufile
doomed ~ # cat sufile
testt file
doomed ~ # echo "and more" >> sufile
doomed ~ # cat sufile
testt file
and more
doomed ~ #
```

The root user creates a file called ‘‘sufile’’. We remove the root users permission from its file but the root user can still access it. Even when we remove all permissions the root user can

```
still access the file.
```

From these experiments we can tentatively accept hypothesis 1, that a root user access a file irrespective of the file permissions. This becomes a "working" hypothesis", it is accepted for the moment, but we still keep a look out for future results that might require us to refine it by detailing more precisely when it is true: so far we have only tested this hypothesis on a file that was owned by the super-user.

3 Pre-conditions, guards, and developing a model through experiments

In the most usual use of Event B, events are able to occur spontaneously when their guards are true. We can contrast this with the operations of a classical B Machine, where the operations are designed to be *invoked* at an interface, e.g. by another program.

Events have guards, which act as enabling conditions. Operations, in classical B, may have pre-conditions. these function as "instructions on the box", telling a programmer when an operation can safely be invoked. However, in our approach to experimental exploration of a complex system, we use both guards and pre-conditions in the context of an event based model.

We note that guards can also serve to describe a user interface in which some operations are unavailable, e.g. are invoked via buttons that are greyed out if the firing condition of the event is not satisfied. In this scenario it makes sense to combine them with pre-conditions, which represent a condition outside which *we provide no information as to whether the event can fire or what its effect (possibly disastrous) may be.*

As we perform more experiments we can extend our description of the system to cover more cases. The pre-condition can then be loosened. For example we might first find out how to perform some file operation as the owner of the file, and would include, in the pre-condition of our description, the constraint that the user should be the file owner. Later, as we obtain a complete understanding of file permissions, we would loosen this pre-condition to cover all the ways in which permissions can be obtained.

A further refinement step can be made when we have a pre-condition expressing all the possible ways of having the requisite file permissions. At that point we can switch the pre-condition expressing required firing conditions into a guard, since outside these conditions the event cannot fire.

No difference can be detected between guards and pre-conditions when checking the invariant properties of a specification[2]. We see the difference only when

refining, where we are able to tighten guards and weaken pre-conditions. In the current investigation, the process of refinement yields more and more detailed descriptions of file access events.

To add some weight to these assertions we now conduct a short mathematical discussion.

If S describes an update and Q a post condition, we write $[S]\,Q$ for the condition that an occurrence of S will result in a post-condition satisfying Q.

We write an event with guard g and effect S as $g \Longrightarrow S$. Such an event can only occur if g is true, and its effect will then be given by S. The associated predicate-transformer rule is:

$$[g \Longrightarrow S]\,Q \;\equiv\; g \Rightarrow [S]\,Q$$

We write an event with pre-condition P and effect S as $P \mid S$. We don't know whether such an event can occur outside P or not, and if it can occur we know nothing about its effect. The associated predicate transformer rule is:

$$[P \mid S]\,Q \;\equiv\; P \wedge [S]\,Q$$

If S and T are events whose effects act in the same state space, we can say that S is refined by T if a post-condition guaranteed by S is also guaranteed by T, i.e. if the following holds for an arbitrary post-condition Q

$$S \sqsubseteq T \equiv [S]\,Q \Rightarrow [T]\,Q$$

The following results, which show how to mingle guards and pre-conditions, may then be proved:

$$(g_2 \Rightarrow g_1) \;\Rightarrow\; (g_1 \Longrightarrow S \sqsubseteq g_2 \Longrightarrow S)$$
$$(P_1 \Rightarrow P_2) \;\Rightarrow\; (P_1 \mid S \sqsubseteq P_2 \mid S)$$
$$P \mid g \Longrightarrow S = (P \Rightarrow g) \Longrightarrow P \mid S$$
$$P \mid S \sqsubseteq P \Longrightarrow S$$
$$P \mid g \Longrightarrow S \sqsubseteq g \Longrightarrow P \mid S$$

We leave the proofs as exercises for the diligent reader.

4 Modelling an abstract state

A serious limitation of the Unix Manual concept, where each operation has its effect described on its own "man" page, is that the description of operation effects is not complemented by a systematic description of *what is being affected,*

i.e. by a description of the system state. This limitation is equally present in the GNU Texinfo system which officially superseded the "man" page form of documentation. Following TexInfo, in a historical sense, we see a change of attitude exemplified by remarks such as "the documentation is in the listings" or by the use of WIMP interfaces, supposed to provide an obvious access to system internals that requires no conceptual explanation. I.e. we see, more or less, an abandonment of any serious documentation attempt.

In the Event B approach, on the other hand, we begin our construction of a model by providing a description of the model's state. This state, we should note, will be abstract and conceptual - that is, it does not necessarily represent the details of what the concrete elements of state in a system are, but rather gives a description of internal state that is consistent with observations of the system. Were we seeking to implement a system, these components of abstract state would be refined into components of concrete state, but that will not be our aim here; rather we want to use the notion of abstract state as a documentation tool, allowing us to think about, and describe, a system at the most convenient conceptual level, i.e. the level which contains the necessary and sufficient information for describing system behaviour.

4.1 Abstract sets

Our model is based on a number of abstract sets, such as *FILE*, the set of all conceivable files. We say "conceivable files" to indicate that this set does not refer just to the files in the system at a particular time. We call that set *file*, and generally use a consistent naming convention by which, where an upper case word denote and abstract set, and the same name in lower case denotes a variable representing those elements of the abstract set which are present in the current system state.

The other given sets of our model are:

USER the set of all users

GROUP the set of all groups

NAME the set of all names. We will assume that a Unix name e.g. `foo` is associated with the name *foo* in the model.

ID the set of all user and group identities.

CONTENT the set of all possible file contents.

We also have a set of permissions, for which we list the elements:

$PERM = \{r, w, x\}$

Our model will only consider read write and execute permissions, and execute permissions will only be of interest in as far as they affect directories.

4.2 Constants

We have a number of constants:

$r \in FILE$ is the root directory of the file system. We are not concerned to always choose names that correspond with names used in Unix, where the root directory is referred to as `/` (forward slash).

$root \in USER$ is the super user.

$root\_id$ is the root users associated identity.

4.3 Variables

The state of our model is represented by a number of variables. We introduce them with some initial type information and brief comments, and subsequently add some additional "state invariant" information.

$file \subseteq FILE$ is the set of files held in the system.

$file\_name \in file \rightarrow NAME$ is the function associating each file with its name. We mean here the local name rather than the whole path, so many files can have the same name.

$file\_owner \in file \rightarrow user$ a function associating each file with its owner.

$file\_group \in file \rightarrow group$ a function associating each file with a group of users who may access the file via its "group permissions". $group$ is declared below.

$dir \subseteq file$ is the set of files which are directories.

$wd \in dir$ the working directory.

$cu \in user$ the current user.

$user \in \mathbb{P}(USER)$ the set of users in the system. A user is identified by name at the user interface, and by a user-id internally.

$user\_id \in user \rightarrow ID$ a function associating each user with an ID.

$user\_name \in user \rightarrow NAME$ a function associating each user with a name.

$group \in \mathbb{P}(GROUP)$ the set of groups in the system. As with users, each group is associated with both a name and an identity.

$group\_name \in group \rightarrow NAME$ the function associating each group with its name.

$group\_id \in group \rightarrow ID$ the function associating each group with its ID.

$group\_members \in group \rightarrow \mathbb{P}(users)$ a function associating each group in the system with its members

$owner\_permissions \in file \rightarrow \mathbb{P}(PERM)$ a function associating each file with the access permissions of the files owner.

$group\_permissions \in file \rightarrow \mathbb{P}(PERM)$ a function associating each file with the access permissions of members of the file's group (other than the owner).

$others\_permissions \in file \rightarrow \mathbb{P}(PERM)$ a function associating each file with the access permissions of users who are not the owner and not in the file's group.

$path \in file \rightarrowtail \text{seq}(NAME)$ is a function associating each file with a unique path. A file at `/mnt/thefolder/thefile` can be thought of, using a suitable naming convention, as having a path modelled by the sequence $\langle r, mnt, thefolder, thefile \rangle$, where r is our name for the root directory. There is a one to one correspondence between paths and files.

$parent \in file \setminus \{r\} \rightarrow dir$ Each file except the root directory has a parent file which is a directory. The intention is, for example, that if
$path(f) = \langle r, mnt, thefolder, thefile \rangle$
then
$parent(f) = path^{-1}(\langle r, mnt, thefolder \rangle)$.

4.4 Invariant and initialisation

Part of our invariant has been given as each variable was introduced. It also includes the following clauses.

The parent of a file f other than the root directory is characterised by:
$path(parent(f)) = front(path(f))$

To state that the *parent* relation defines a tree, we need the property, already given, that the root directory is not in the domain of parent, together with the property that sufficient applications of the *parent* function will yield the root directory:

$$\forall f.f \in file \Rightarrow \exists n.parent^n(f) = r$$

We assume the system is initialised to some configuration satisfying the invariant. A more detailed initialisation could be used to specify an initial file structure for a Unix system, but is beyond the scope of the current investigation.

5 Modelling file access events

One approach to modelling Unix access permissions might be to give a formal specification to the Unix commands such as `touch, cat, cp, mv` etc. However, this would be a clumsy and require considerable duplication if used as a direct

approach, since there is often more than one way to perform the same task. For example both `cp` and `cat` can be used to copy files, and a file move can be done by a m̄v= command or by a copy followed by a `rm`. In addition there is a mismatch between Unix shell commands, which can take variable numbers of arguments and generally have large numbers of command line options, and the strongly typed style used in B.

We therefore look for a collection of atomic "events", happening within the system, and which can be used as a basis for a conceptual model. The events we choose are:

- a user is added to the system
- a group is added to the system
- a user is added to a group
- the current user changes her working directory
- the current user creates a file
- the current user reads a file
- the current user writes a file
- the current user removes a file
- the current user changes the ownership of a file
- the current user changes permissions on a file

Our understanding of when these events can take place and their exact effect emerges gradually through our experimental examination of the system. To capture our current knowledge at a particular stage in the investigation we make use of pre-conditions *which are not usually part of the conceptual apparatus of Event B*. The pre-condition represents *the system state we have investigated so far*. As our understanding progresses through performing additional experiments we can widen the precondition to include additional cases.

We see how the works for a possible version of the create file event. In this version we give a name as argument, and a file of that name is created in the working directory. The file will belong to the user that creates it, its initial group (we have observed) will be a group having the same name as the user. From student experiments listed in Appendix B we know the file can be created by an ordinary user if that user owns the parent directory and has write and execute permission there:

$$
\begin{array}{l}
\mathit{Create}(\mathit{name}) \mathrel{\widehat{=}} \\
\quad \mathit{owner}(\mathit{parent}(\mathit{wd}) = \mathit{cu} \wedge \\
\quad \{w, x\} \subseteq \mathit{owner\_permissions}(\mathit{parent}(\mathit{wd})) \wedge \\
\quad \mathit{user\_name}(\mathit{cu}) \in \mathrm{ran}(\mathit{group\_name}) \\
| \\
\quad \text{any } f.f \notin \mathit{file} \Longrightarrow \\
\qquad \mathit{file} := \mathit{file} \cup \{f\} \ \| \\
\qquad \mathit{file\_name}(f) := \mathit{name} \ \| \\
\qquad \mathit{parent}(f) := \mathit{wd} \ \| \\
\qquad \mathit{path}(f) := \mathit{path}(\mathit{wd}) \leftarrow \mathit{name} \ \| \\
\qquad \mathit{file\_owner}(f) := \mathit{cu} \ \| \\
\qquad \mathit{owner\_permissions}(f) := \{r, w, x\} \ \| \\
\qquad \mathit{group\_permissions}(f) := \{r\} \ \| \\
\qquad \mathit{others\_permissions}(f) := \{r\} \ \| \\
\qquad \mathit{file\_group}(f) :\in \{g \mid g \in \mathit{group} \wedge \mathit{group\_name}(g) = \mathit{user\_name}(\mathit{cu})\}
\end{array}
$$

We use the pre-condition to state the prevailing condition when we do an experiment, and the body of the event to describe the result. We also use guards, which, of course, have a different semantics and a different purpose. For a *RemoveFile* event, the event can only fire if the file selected for removal exists, and this condition is a guard rather than a pre-condition. The difference between pre-conditions and guards is seen during refinement, where we are allowed to tighten guards and weaken pre-conditions.

The pre-conditions record under what circumstances we have obtained our experimental results. We observe that creating a file associates that file with a singleton group with the same name as the current user. That file already exists. We do not yet know what group would be chosen if this group does not exist (and the documentation is not going to help us in this respect!).

Nor do we know if a user who is not the owner of the parent directory can perform a *CreateFile* operation. After some further experiments, however, we find that what is required is simply that the user has write and execute permissions on the parent directory.

Permissions work as follows:

The super-user always has all permissions.

The owner has “owner” permissions, she cannot take advantage of “group” or “others” permissions.

All group members other than the file owner have the “group” permissions associated with the file. They cannot take advantage of “owner” or “others” permissions.

Any user who is not the file owner and not in the group associated with the file will have the “others” permissions.

This is somewhat counter intuitive, in that the general user can be given more liberal permissions than the file owner or a user who has a specific relationship with the file via the files associated group.

Since the concept of permissions is ubiquitous in our model we define a condition to formally capture the above rules.

$$
\begin{array}{l}
permissions(user, file) == \\
\quad \text{if } user = root \text{ then } \{r, w, x\} \\
\quad \text{elseif } user \neq root \wedge user = file\_owner(file) \text{ then} \\
\quad\quad owner\_permissions(file) \\
\quad \text{elseif } user \neg root \wedge user \neq file\_owner(file) \wedge user \in file\_group(file) \text{ then} \\
\quad\quad group\_permissions(file) \\
\quad \text{else } others\_permissions(file) \\
\quad \text{end}
\end{array}
$$

We can then broaden the pre-conditions of the *Create* event as follows to show that any user with relevant permissions on the parent directory can create a file.

$$
\begin{array}{l}
Create(name) \mathrel{\widehat{=}} \\
\quad \{w, x\} \subseteq permissions(cu, parent(wd)) \wedge ... \\
\quad\quad \text{rest of definition as before}
\end{array}
$$

Note that it is *this relaxation of a condition during refinement that tells us we are dealing with a pre-condition here rather than a guard.*

At this point a reader may suggest that permissions should function as guards rather than pre-conditions, since if permissions do not allow an event, that event cannot take place. As noted earlier, this is a further refinement step.

6 Conclusions

We have presented a refinement based method for investigating the behaviour of a complex system through the construction of a theory based on experimental observation. Our subject matter has been the Unix file store. The material is suitable for investigation by computer science students working at final year undergraduate or at research level. Rather than isolating formal methods from practical computing experience, the approach attempts to unite theory and practice.

Writing system documentation based on experimental evidence is always subject to doubt. Results may fail to match predictions, but at least we then have a definite refutation of our theory. At such points we would search for a new theory, typically one with an additional, currently "hidden" variable. Often we can guess where the model is subject to this form of limitation - for example the initial

permissions granted to a new file are given definitively in the model, but are no doubt subject to change under different security policies. Further developement of these ideas, a more complete description of Unix file permissions, and an informal document, abstracted from the formal model and describing Unix access permissions, will form part of the first author's PhD thesis.

References

1. J R Abrial. *The B Book*. Cambridge University Press, 1996.
2. J-R Abrial. Extending B without Changing it (for Developing Distributed Systems). In H Habrias, editor, *The First B Conference, ISBN : 2-906082-25-2*, 1996.
3. J R Abrial. *Modeling in Event B*. Cambridge University Press, 2009.
4. I Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice Hall, 1987.
5. Brian W. Kernighan and Rob Pike. *Unix Programming Environment (Prentice-Hall Software Series)*. Prentice Hall, 1984.
6. C Morgan and B Sufrin. Specification of the unic filing system. In Hayes [4], pages 91–140.
7. Briony J Oates. *Researching Information Systems and Computing*. Sage Publications Ltd., 2006.
8. Markus Wenzel. Some aspects of unix file-system security, 2008. Available from www.cl.cam.ac.uk/ lp15/Isabelle/dist/library/HOL/Unix/document.pdf.

Appendices

A Unix commands used in experiments

The Unix commands we use in our experiments are:

`adduser` for adding a new user to the system

`addgroup` for adding a new group to the system

`cd` for changing the working directory

`chmod` for changing permissions of a file

`chown` for changing the owner of a file

`chgrp` for changing the group to which a file belongs

`ls` for listing files and associated ownership, group ownership and file permission details, and for seeing when we can list such details.

`touch` for creating files for use in experiments

`echo` for generating example content to place in files

`cat` for testing read and write permissions on files.

`cp` for copying files

`mv` for moving files

B Experiments on the effect of directory permissions

```
Directory Permissions

This experiment is to see what effect the permissions of a
directory have on a file within said directory. The directory
/mnt/testarea has full permissions to every one.

doomed ~ # ls -l /mnt/
drwxrwxrwx 4 root root 4096 Feb 9 10:37 testarea
doomed ~ #

The user bob creates a folder that only he can access

bob@doomed ~ $ cd /mnt/testarea/
bob@doomed /mnt/testarea $ mkdir thefolder
bob@doomed /mnt/testarea $ ls -l
drwxr-xr-x 2 bob bob 4096 Feb 25 11:17 thefolder
bob@doomed /mnt/testarea $ chmod 700 thefolder
bob@doomed /mnt/testarea $ ls -l
drwx------ 2 bob bob 4096 Feb 25 11:17 thefolder
bob@doomed /mnt/testarea $

The user bob creates a file then tries to change the ownership
of the file to the user ben

bob@doomed /mnt/testarea $ touch thefolder/thefile
bob@doomed /mnt/testarea $ ls -l thefolder/thefile
-rw-r--r-- 1 bob bob 0 Feb 25 11:18 thefolder/thefile
bob@doomed /mnt/testarea $ chown ben thefolder/thefile
chown: changing ownership of ‘thefolder/thefile’: Operation not
permitted
bob@doomed /mnt/testarea $ chown ben: thefolder/thefile
chown: changing ownership of ‘thefolder/thefile’: Operation not
permitted
bob@doomed /mnt/testarea $

Since bob is unable to change ownership root does it

doomed ~ # chown ben /mnt/testarea/thefolder/thefile
```

```
bob@doomed /mnt/testarea $ ls -l thefolder/thefile
-rw-r--r-- 1 ben bob 0 Feb 25 11:18 thefolder/thefile
bob@doomed /mnt/testarea $

The user ben tries to access the file he owns in thefolder

ben@doomed /mnt/testarea $ pwd
/mnt/testarea
ben@doomed /mnt/testarea $ cat thefolder/thefile
cat: thefolder/thefile: Permission denied
ben@doomed /mnt/testarea $

The user bob modifies the containing folder's permission so
the others permissions include read.

bob@doomed /mnt/testarea $ chmod o+r thefolder
bob@doomed /mnt/testarea $ ls -l
drwx---r-- 2 bob bob 4096 Feb 25 11:18 thefolder
bob@doomed /mnt/testarea $

The user ben tries to access his file again

ben@doomed /mnt/testarea $ cat thefolder/thefile
cat: thefolder/thefile: Permission denied
ben@doomed /mnt/testarea $

We remove read permission and add execute

bob@doomed /mnt/testarea $ chmod o-r+x thefolder
bob@doomed /mnt/testarea $ ls -l
drwx-----x 2 bob bob 4096 Feb 25 11:18 thefolder
bob@doomed /mnt/testarea $

The user ben is now able to read and write to the file

ben@doomed /mnt/testarea $ cat thefolder/thefile
ben@doomed /mnt/testarea $ echo "poop" >> thefolder/thefile
ben@doomed /mnt/testarea $ cat thefolder/thefile
poop
ben@doomed /mnt/testarea $

Conclusion: it is only once the execute permission of a
containing folder is available that a user can modify a file
he has within it.
```

Acknoledgements

We acknowledge the contribution of the anonymous referees.

How to make mistakes*

Stefan Hallerstede

University of Düsseldorf
Germany
halstefa@cs.uni-duesseldorf.de

Abstract. When teaching Event-B to beginners, we usually start with models that are already good enough, demonstrating occasionally some standard techniques like "invariant strengthening". We show that we got it essentially right but need to make improvements here and there. However, this is not how we really create formal models. To a beginner, getting shown only nearly perfect models is overwhelming. So we should start earlier and show how we usually get models wrong initially. This provides ample opportunity to demonstrate the strengths of formal reasoning (and the weaknesses). The principal strength of formal reasoning lies in its capacity to locate mistakes in a model and to suggest corrections. A beginner should learn how to profit from his mistakes by improving his understanding of the model. A weakness of formal reasoning is that we only find mistakes that we expect, for example, invariant violation or non-termination. Mistakes that do not fall into one of these categories may slip through.
In this article we present how a formal model is created by refinement and alteration. The approach employs mathematical methodology for problem solving and a software tool. Both aspects are important. Mathematical methodology provides ways to turn mistakes into improvements. The software tool is necessary to ease the impact of changes on a model and to obtain rapid feed back. We begin with a set of assumptions and requirements, the problem, and set out to solve it, giving a more vivid picture of how formal methods work.

1 Introduction

In Event-B [2] formal modelling serves primarily for reasoning: reasoning is an essential part of modelling because it is the key to understanding complex models. Reasoning about complex models should not happen accidentally but needs systematic support within the modelling method. This thinking lies at the heart of the Event-B method.

We use refinement to manage the many details of a complex model. Refinement is seen as a technique to introduce detail gradually at a rate that eases understanding. The model is completed by successive refinements until we are satisfied that the model captures all important requirements and assumptions. In this article we concern ourselves only with what is involved in coming up with an abstract model of some system. Note that refinement can also be used to produce implementations of abstract models, for instance, in terms of a sequential program [1,8]. But this is not discussed in this article.

* This research was carried out as part of the EU research project DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) http://www.deploy-project.eu/.

We present a worked out example that could be used in the beginning of a course on Event-B to help students to develop a realistic picture of the use of formal methods. The challenge is to state an example in such a way that it is easy to follow but provides enough opportunity to make (many) mistakes. We chose to use a sized-down variant of the access control model of [2] which we have employed for lectures at ETH Zürich (Switzerland) and at the university of Southampton (United Kingdom). We have not used the description of a computer program because it does not leave enough room for misunderstanding. We begin by stating a problem to be solved in terms of assumptions and requirements and show how the problem can be approached using formal methods. It is not possible to show everything that happens during an actual development of a formal model. But it is possible to point at the main difficulties encountered and show how to approach them. Students should be encouraged to make mistakes and experiment with formal models. On the way they will learn about strengths and weaknesses of formal methods.

We approach the model of the access control in small increments, learning at each increment something about the problem and the model. We understand this incremental approach in two ways. The first way is by formal refinement. An existing model is proved to be refined by another: all properties of the existing model are preserved in the refined model. The second way is by alteration of an existing model: properties of the existing model may be broken. When a model is shown to be not consistent, it needs to be modified in order to make it consistent. This reflects a learning process supported by various forms of reasoning about a model, for instance, proof, animation, or model-checking. This way of thinking about a model is common in mathematical methodology [6,9]. The first way is commonly used and taught in formal methods, whereas the second is at least not acknowledged. In order to apply formal methods successfully both ways need to be mastered. This is only feasible in the presence of software tools that make reasoning easy and modifications to a model painless. We have relied on the Rodin modelling tool [3] for Event-B for proof obligation generation and proof support and on the ProB tool [7] for animation and model-checking. Both tools are integrated in the Rodin platform and can be used seamlessly. In later sections we do not further specify the tools used, though, as this should be clear from the context. Also note that we present proof in an equational style [5,10] whereas the Rodin tool uses sequents as in [2].

Overview. In Section 2 we introduce Event-B. The following sections are devoted to solving a concrete problem in Event-B. In Section 3 the problem is stated. Section 4 provides a more detailed overview of Sections 5 to 9. It is intended to help the reader keeping track of how the solution of the problem advances in the ensuing sections. A first model is produced and discussed in Section 5. In Sections 6 and 8 we elaborate the model by refinement. Section 7 contains a small theory of transitive closures that is needed in the refinement. In Section 9 some further improvements of the model are made and limitations of formal modelling discussed.

2 Event-B

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts contain the static part of a model whereas machines contain the dy-

namic part. Contexts may contain *carrier sets*, *constants*, *axioms*, where carrier sets are similar to types [4]. In this article, we simply assume that there is some context and do not mention it explicitly. Machines are presented in Section 2.1, and proof obligations in Section 2.2 and Section 2.3. All proof obligations in this article are presented in the form of sequents: "premises" $\vdash$ "conclusion".

Similarly to our course and based on [2], we have reduced the Event-B notation used so that only a little notation suffices and formulas are easier to comprehend, in particular, concerning the relationship between formal model and proof obligations. We have also reduced the amount of proof obligations associated with a model. We have done this for two reasons: firstly, it is easier to keep track of what is to be proved; secondly, it permits us to make a point about a limitation of formal methods later on.

2.1 Machines

Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables $v = v_1, \ldots, v_m$ define the state of a machine. They are constrained by invariants $I(v)$. Theorems are predicates that are implied by the invariants. Possible state changes are described by means of events $E(v)$. Each event is composed of a *guard* $G(t, v)$ and an *action* $x := S(t, v)$, where $t = t_1, \ldots, t_r$ are *parameters* the event may contain and $x = x_1, \ldots, x_p$ are the variables it may change[1]. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E(v)$ by

$$
\begin{array}{lcl}
E(v) & \widehat{=} & \mathsf{any}\ \ t\ \ \mathsf{when} \\
 & & \quad G(t,v) \\
 & & \mathsf{then} \\
 & & \quad x := S(t,v) \\
 & & \mathsf{end}
\end{array}
\qquad \text{or} \qquad
\begin{array}{lcl}
E(v) & \widehat{=} & \mathsf{begin} \\
 & & \quad x := S(v) \\
 & & \mathsf{end}
\end{array}
\quad .
$$

The short form on the right hand side is used if the event does not have parameters and the guard is true. A dedicated event of the latter form is used for *initialisation*. The action of an event is composed of several *assignments* of the form

$$x_\ell := B_\ell(t, v) \quad ,$$

where x_ℓ is a variable and $B_\ell(t, v)$ is an expression. All assignments of an action $x := S(t, v)$ occur simultaneously; variables y that do not appear on the left-hand side of an assignment of an action are not changed by the action, yielding one simultaneous assignment

$$x_1, \ldots, x_p, y_1, \ldots, y_q := B_1(t, v), \ldots, B_p(t, v), y_1, \ldots, y_q \quad , \tag{1}$$

where $x_1, \ldots, x_p, y_1, \ldots, y_q$ are the variables v of the machine. The action $x := S(t, v)$ of event $E(v)$ denotes the formula (1), whereas in the proper model we only specify those variables x_ℓ that may change.

[1] Note that, as x is a list of variables, $S(t, v)$ is a corresponding list of expressions.

2.2 Machine Consistency

Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants $I(v) = I_1(v) \wedge \ldots \wedge I_i(v)$ and, thus, needs to be proved. The corresponding proof obligations are called *invariant preservation* ($\ell \in 1 \mathinner{..} i$):

$$\begin{array}{l} I(v) \\ G(t,v) \\ \vdash \\ I_\ell\left(S(t,v)\right) \quad , \end{array} \tag{2}$$

for every event $E(v)$. Similar proof obligations are associated with the initialisation event of a machine. The only difference is that neither an invariant nor a guard appears in the premises of proof obligation (2), that is, the only premises are axioms and theorems of the context. We say that a machine is consistent if all events preserve all invariants.

2.3 Machine Refinement

Machine refinement provides a means to introduce more details about the dynamic properties of a model [4]. A machine N can refine at most one other machine M. We call M the *abstract* machine and N a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v,w) = J_1(v,w) \wedge \ldots \wedge J_j(v,w)$, where $v = v_1, \ldots, v_m$ are the variables of the abstract machine and $w = w_1, \ldots, w_n$ the variables of the concrete machine.

Each event $E(v)$ of the abstract machine is *refined* by a concrete event $F(w)$. Let abstract event $E(v)$ with parameters $t = t_1, \ldots, t_r$ and concrete event $F(w)$ with parameters $u = u_1, \ldots, u_s$ be

$$\begin{array}{lcl} E(v) & \widehat{=} & \mathsf{any}\ \ t\ \ \mathsf{when} \\ & & \quad G(t,v) \\ & & \mathsf{then} \\ & & \quad v := S(t,v) \\ & & \mathsf{end} \end{array} \qquad \text{and} \qquad \begin{array}{lcl} F(w) & \widehat{=} & \mathsf{any}\ \ u\ \ \mathsf{when} \\ & & \quad H(u,w) \\ & & \mathsf{with} \\ & & \quad t = W(u) \\ & & \mathsf{then} \\ & & \quad w := T(u,w) \\ & & \mathsf{end} \end{array} .$$

Informally, concrete event $F(w)$ refines abstract event $E(v)$ if the guard of $F(w)$ is stronger than the guard of $E(v)$, and the gluing invariant $J(v,w)$ establishes a simulation of the action of $F(w)$ by the action of $E(v)$. The term $W(u)$ denotes *witnesses* for the abstract parameters t, specified by the equation $t = W(u)$ in event $F(w)$, linking abstract parameters to concrete parameters. Witnesses describe for each event separately more specific how the refinement is achieved. The corresponding proof obligations for refinement are called *guard strengthening* ($\ell \in 1 \mathinner{..} g$):

$$\begin{array}{l} I(v) \\ J(v,w) \\ H(u,w) \\ \vdash \\ G_\ell\left(W(u),v\right) \quad , \end{array} \tag{3}$$

with the abstract guard $G(t,v) = G_1(t,v) \wedge \ldots \wedge G_g(t,v)$, and (again) *invariant preservation* ($\ell \in 1 \,..\, j$):

$$\begin{array}{l} I(v) \\ J(v,w) \\ H(u,w) \\ \vdash \\ J_\ell\left(S(W(u),v), T(u,w)\right) \quad . \end{array} \tag{4}$$

3 Problem Statement

In the following sections we develop a simple model of a secure building equipped with access control. The problem statement is inspired by a similar problem used by Abrial [2]. Instead of presenting a fully developed model, we illustrate the process of how we arrive at the model. We can not follow the exact path that we took when working on the model: we made changes to the model as a whole several times. So we would soon run out of space. We comment on some of the changes without going too much into detail in the hope to convey some of the dynamic character of the modelling process.

The model to be developed is to satisfy the following properties:

P1 : The system consists of persons and one building.
P2 : The building consists of rooms and doors.
P3 : Each person can be at most in one room.
P4 : Each person is authorised to be in certain rooms (but not others).
P5 : Each person is authorised to use certain doors (but not others).
P6 : Each person can only be in a room where the person is authorised to be.
P7 : Each person must be able to leave the building from any room where the person is authorised to be.
P8 : Each person can pass from one room to another if there is a door connecting the two rooms and the person has the proper authorisation.
P9 : Authorisations can be granted and revoked.

Properties P1, P2, P8, and P9 describe environment assumptions whereas properties P3, P4, P5, P6, and P7 describe genuine requirements. It is natural to mix them in the description of the system. Once we start modelling, the distinction becomes important. We have to prove that our model satisfies P3, P4, P5, P6, and P7 assuming we have P1, P2, P8, and P9.

4 Storyboard

In this section we give a brief overview of the development as it will unfold in Sections 5 to 9. The reader is encouraged to return to this section while reading through those sections. We have tried to keep a natural flow in the presentation to make it more credible. It also requires some skill to stay on track when solving a complex problem.

It lies in the nature of the presented material that we can easily lose sight of our aim. By the way, what is our aim? This will be stated in the beginning of Section 5 before starting any work:

Our aim is to produce a faithful formal model of the system described by the properties P1 to P9 of Section 3.

Once we know what we want to achieve we can get started. In Section 5 we begin by making a rough plan of how to proceed and create a first abstract machine introducing four invariants $inv1$ to $inv4$ and abstract events $pass$, $grant$, and $revoke$. The reason for the briefness of Section 5 is simply that we needed space to make some more interesting points later on. It is not to suggest that inventing a first model would be trivial.

In Section 6 we commence with the refinement of the abstract machine, introducing invariants $inv5$ to $inv7$, and event $pass$. We uncover that the properties P1 to P9 do not say enough about the nature of the doors used in the building. Incompleteness of assumptions is a common difficulty dealing with which carries the risk of introducing undocumented assumptions into the model. At this point we have incorporated properties P1 to P6 and property P8 into our model.

In order to express P7 formally, a little bit of theory about transitive closures is needed which is introduced in Section 7. (Nothing new is "invented". We simply use an existing theory.) We use this in Section 8 to have a first go at property P7 by means of invariant $inv8$. (Property P7 is formalised as a more intricate theorem $thm1$ implied by the invariants. This is a common strategy to ease the effort of verifying invariant preservation.) However, when analysing $inv8$ we find that it is too strong and weaken it to $inv8'$.

In the remainder of Section 8 we refine events $grant$ and $revoke$. (Proving refinement of event $grant$ we find the need for invariant $inv9$, which would also permit us to prove $thm1$.) When trying to refine event $revoke$ we discover that event $revoke$ of abstract machine is wrong. We have to alter the abstract machine in order to get a model that is consistent and represents P9 adequately. By the end of Section 8 we feel that properties P1 to P9 have all been incorporated into the model. To ensure this we have made some effort to trace them into the formal model and argued that each one has been adequately captured.

In Section 9 we have another look at the model, animating it. We immediately see that the concrete machine cannot "do" anything. It deadlocks right after the initialisation of the machine. We rectify the problem and use it to illustrate limitations of a formal method. This is also a good place to advertise the usefulness of complementary techniques in analysing formal models.

5 Getting started with a fresh model

Our aim is to produce a faithful formal model of the system described by the properties P1 to P9 of Section 3. The first decision we need to make concerns the use of refinement. We have decided to introduce the properties of the system in two steps. In the first step we deal only with persons and rooms, in the second also with doors. This approach appears reasonable. At first we let persons move directly between rooms. Later we state how they do it, that is, by passing through doors. In order to specify doors we need to know about rooms they connect. It is a good idea, though, to reconsider the strategy chosen for refinement when it turns out to be difficult to tackle the elements of the model in the planned order. For now, we intend to produce a model with one refinement:

(i) the *abstract machine* (this section) models room authorisations;
(ii) the *concrete machine* (sections 6 and 8) models room and door authorisations.

In Event-B we usually begin modelling by stating invariants that a machine should preserve. When events are introduced subsequently, we think more about how they preserve invariants than about what they would do. The focus is on the properties that have to be satisfied. We declare two carrier sets for persons and rooms, $Person$ and $Room$, and a constant $\boldsymbol{O}$, where $\boldsymbol{O} \in Room$. Constant $\boldsymbol{O}$ models the *outside* of the building. We choose to describe the state by two variables for authorised rooms and locations of persons, arm and loc, with invariants

$inv1 : arm \in Person \leftrightarrow Room$	Property P4
$inv2 : Person \times \{\boldsymbol{O}\} \subseteq arm$	
$inv3 : loc \in Person \rightarrow Room$	Property P3
$inv4 : loc \subseteq arm$	Property P6

Invariant $inv2$, that *each person is authorised to be outside*, is necessary because we decided to model location by a total function making the outside a special room. For instance, person p is outside is written formally $loc(p) = \boldsymbol{O}$. In a first attempt, we made loc a partial function from $Person$ to $Room$ expressing that a person not in the domain of loc is outside. However, this turned out to complicate the gluing invariant when introducing doors into the model later on. (Because of property P7 we need an explicit representation of the outside in the model.) As a consequence of our decision we had introduce invariant $inv2$. It corresponds to a new requirement that is missing from the list in Section 3 but that we have uncovered while reasoning formally about the system. In the following we focus on how formal reasoning is used to improve the model of the system.

In order to satisfy $inv2$, $inv3$ and $inv4$ we let

initialisation
  begin
    $act1 : arm := Person \times \{\boldsymbol{O}\}$
    $act2 : loc := Person \times \{\boldsymbol{O}\}$
  end .

We model passage from one room to another by event $pass$,

$pass$
  any p, r when
    $grd1 : p \mapsto r \in arm$ — p is authorised to be in r
    $grd2 : p \mapsto r \notin loc$ — but not already in r
  then
    $act1 : loc := loc \lhd\!\!+ \{p \mapsto r\}$
  end .

Event $pass$ preserves the invariants. Granting and revoking authorisations for rooms is modelled by the two events

$grant$
any p, r when
$grd1 : p \in Person$
$grd2 : r \in Room$
then
$act1 : arm := arm \cup \{p \mapsto r\}$
end

$revoke$
any p, r when
$grd1 : p \in Person$
$grd2 : p \mapsto r \notin loc$
then
$act1 : arm := arm \setminus \{p \mapsto r\}$
end .

The two events do not yet model all of P9 which refers to authorisations in general, including authorisations for doors. Events $grant$ and $revoke$ appear easy enough to get them right. But it is as easy to make a mistake. This is why we have specified invariants: to safeguard us against mistakes. If the proof of an invariant fails, we have the opportunity to learn something about the model and improve it. The two events preserve all invariants except for $revoke$ which violates invariant $inv2$,

$$\begin{array}{ll} Person \times \{\boldsymbol{O}\} \subseteq arm & \textit{Invariant } inv2 \\ p \in Person & \textit{Guard } grd1 \\ p \mapsto r \notin loc & \textit{Guard } grd2 \\ \vdash & \\ Person \times \{\boldsymbol{O}\} \subseteq arm \setminus \{p \mapsto r\} & \textit{Modified invariant } inv2 \end{array}$$

In an instance of the model with two different rooms $\boldsymbol{I}$ and $\boldsymbol{O}$ and one person P we find a counter example:

$$arm = \{P \mapsto \boldsymbol{I}, P \mapsto \boldsymbol{O}\}, \quad loc = \{P \mapsto \boldsymbol{I}\}, \quad p = P, \quad r = \boldsymbol{O} \quad .$$

In fact, we must not remove $\boldsymbol{O}$ from the set of authorised rooms of any person. To achieve this, we add a third guard to event $revoke$:

$$grd3 \ : \ r \neq \boldsymbol{O} \quad .$$

A counter example provides valuable information, pointing to a condition that it does not satisfy. It may not always be as simple to generalise but at least one can obtain an indication where to look closer.

The model we have obtained thus far is easy to understand. Ignoring the doors in the building, it is quite simple but already incorporates properties P3, P4, and P6. Its simplicity permits us to judge more readily whether the model is reasonable. We can inspect it or animate it and can expect to get a fairly complete picture of its behaviour. We may ask: Is it possible to achieve a state where some person can move around in the building? We have only partially modelled the assumptions P1, P2, P8, and P9. We could split them into smaller statements that would be fully modelled but have decided not to do so. Instead, we are going to document how they are incorporated in the refinement that is to follow.

6 Elaboration of more details

We are satisfied with the abstract model of the secure building for now and turn to the refinement where doors are introduced into the model. In the refined model we employ two variables adr for authorised doors and loc for the locations of persons in the building (as before). The intention is to keep the information contained in the abstract variable arm implicitly in the concrete variable adr. That is, in the refined model variable arm would be redundant. We specify

$$inv5 \;:\; adr \in Person \to (Room \leftrightarrow Room) \qquad \text{Property P5}$$
$$inv6 \;:\; \forall q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}] \qquad \text{Property P4}$$

6.1 Moving between rooms

Let us first look at event $pass$. Only a few changes are necessary to model property P8,

$pass$
  any p, r when
    $grd1 \;:\; loc(p) \mapsto r \in adr(p)$
  then
    $act1 \;:\; loc := loc \lhd\!\!+ \{p \mapsto r\}$
  end .

We only have to show guard strengthening, because loc does not occur in $inv5$ and $inv6$. The abstract guard $grd1$ is strengthened by the concrete guards because $r \in \mathsf{ran}(adr(p))$. The second guard strengthening proof obligation of event $pass$ is:

$$\begin{array}{ll} & loc \in Person \to Room & \textit{Invariant } inv3 \\ & loc(p) \mapsto r \in adr(p) & \textit{Concrete guard } grd1 \\ \vdash & & \\ & p \mapsto r \notin loc & \textit{Abstract guard } grd2 \end{array}$$

Using $inv3$ we can rephrase the goal,

$$\begin{array}{lll} & p \mapsto r \notin loc & \{\, inv3 \,\} \\ \Leftrightarrow & loc(p) \neq r & \end{array}$$

Neither concrete guard $grd1$ nor the invariants $inv1$ to $inv6$ imply this. The invariant is too weak. We do not specify that doors connect *different* rooms. In fact, our model of the building is rather weak. We decide to model the building by the doors that connect the rooms in it. They are modelled by a constant $Door$. We make the following three assumptions about doors:

$$axm1 \;:\; Door \in Room \leftrightarrow Room \qquad \textit{Each door connects two rooms.}$$
$$axm2 \;:\; Door \cap \mathsf{id}_{Room} = \varnothing \qquad \textit{No door connects a room to itself.}$$

$axm3 \;:\; Door \subseteq Door^{-1}$ *Each door can be used in both directions.*[2]

These assumptions are based on our domain knowledge about properties of typical doors. They were omitted from the problem description because they seemed obvious. However, the validity of our model will depend on them. As such they ought to be included. We began to think about properties of doors because we did not succeed proving a guard strengthening proof obligation. If $adr(p)$, for $p \in Person$, would share the property of the set $Door$ given by $axm2$, we should succeed. Hence, we add a new invariant $inv7$. We realise that it captures much better property P5 than invariant $inv5$,

$inv7 \;:\; \forall q \cdot adr(q) \subseteq Door \quad .$ Property P5

Using $inv7$ and $axm2$, we can prove $\forall x, y \cdot x \mapsto y \in adr(p) \Rightarrow x \neq y$, and with "$x, y := loc(p), r$" we are able to show the guard strengthening proof obligation above.

7 Intermezzo on transitive closures

Property P7 is more involved. It may be necessary to pass though various rooms in order to leave the building. We need to specify a property about the transitive relationship of the doors. We can rely on the well-known mathematical theory of the transitive closure of a relation.

A relation x is called *transitive* if $x ; x \subseteq x$. In other words, any composition of elements of x is in x. The transitive closure of a relation x is the least relation that contains x and is transitive. We define the *transitive closure* x^+ of a relation x by

$$\forall x \cdot x \subseteq x^+ \tag{5}$$
$$\forall x \cdot x^+ ; x \subseteq x^+ \tag{6}$$
$$\forall x, z \cdot x \subseteq z \wedge z ; x \subseteq z \Rightarrow x^+ \subseteq z \quad . \tag{7}$$

That is, x^+ is the least relation z satisfying $x \cup z ; x \subseteq z$. The definition implies

$$\forall x \cdot x \cup x^+ ; x = x^+ \tag{8}$$

The transitive closure is monotonic and maps the empty relation to itself,

$$\forall x, y \cdot x \subseteq y \Rightarrow x^+ \subseteq y^+ \tag{9}$$
$$\varnothing^+ = \varnothing \quad . \tag{10}$$

8 Towards a full model of the building

Using the transitive closure of authorised rooms we can express that every person can only reach authorised rooms from the outside,

$inv8 \;:\; \forall q \cdot arm[\{q\}] \subseteq adr(q)^+[\{\boldsymbol{O}\}] \quad .$

[2] We say $Door$ is a *symmetric* relation.

This invariant does not quite correspond to property P7. However, given the discussion about properties of doors in Section 6 we should be able to prove that all invariants jointly imply property P7 which we formalise as a theorem,

$$thm1 \;:\; \forall q \cdot (arm[\{q\}] \setminus \{\boldsymbol{O}\}) \times \{\boldsymbol{O}\} \subseteq adr(q)^+ \quad . \qquad \text{Property P7}$$

We proceed like this because we expect that proving $inv8$ to be preserved would be much easier than doing the same with $thm1$. Let us continue working with $inv8$ for now and return to $thm1$ later.

8.1 Initialisation

In the abstract model all persons can only be outside initially. This corresponds to them not being authorised to use any doors,

initialisation
  begin
    $act1 \;:\; adr := Person \times \{\varnothing\}$
    $act2 \;:\; loc := Person \times \{\boldsymbol{O}\}$
  end .

The invariant preservation proof obligations for $inv5$ and $inv6$ hold, as can easily be seen letting "$arm,\ adr := Person \times \{\boldsymbol{O}\},\ Person \times \{\varnothing\}$" in $inv5$, $inv6$, and $inv7$. For invariant $inv8$ there is more work to do. We have to show:

$$\vdash \quad \forall q \cdot (Person \times \{\boldsymbol{O}\})[\{q\}] \subseteq (Person \times \{\varnothing\})(q)^+[\{\boldsymbol{O}\}]$$

Using law (10), $(Person \times \{\varnothing\})(q)^+[\{\boldsymbol{O}\}] = \varnothing \not\supseteq \{\boldsymbol{O}\} = (Person \times \{\boldsymbol{O}\})[\{q\}]$. Invariant $inv8$ is too strong! Because of invariant $inv7$ we cannot initialise adr to $Person \times \{\{\boldsymbol{O} \mapsto \boldsymbol{O}\}\}$ and because of $inv6$ we cannot use any other door. Thus, we must weaken invariant $inv8$. We replace it by:

$$inv8' \;:\; \forall q \cdot arm[\{q\}] \subseteq adr(q)^+[\{\boldsymbol{O}\}] \cup \{\boldsymbol{O}\}$$

8.2 Granting door authorisations

A new door authorisation can be granted to a person if (a) it has not been granted yet and (b) authorisation for one of the connected rooms has been granted to the person. We introduce constraint (a) to focus on the interesting case and constraint (b) to satisfy invariant $inv8'$. Thus,

grant
  any p, s, r when
    $grd1 \;:\; s \mapsto r \in Door \setminus adr(p)$
    $grd2 \;:\; s \in \mathsf{dom}(adr(p))$
  then
    $act1 \;:\; adr := adr \lhd\!\!+ \{p \mapsto adr(p) \cup \{s \mapsto r, r \mapsto s\}\}$ [3]
  end

Invariant $inv5$ is preserved by event $grant$ by definition of relational overwriting $\Leftarrow$. For invariant $inv6$ we have to prove:

$$\begin{array}{lll} & \forall\, q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}] & \textit{Invariant } inv6 \\ & s \mapsto r \notin adr(p) & \textit{Concrete guard } grd1 \\ & s \in \mathsf{dom}(adr(p)) & \textit{Concrete guard } grd2 \\ \vdash & \mathsf{ran}((adr \Leftarrow \{p \mapsto adr(p) \cup \{s \mapsto r, r \mapsto s\}\})(q)) & \\ & \quad \subseteq (arm \cup \{p \mapsto r\})[\{q\}] & \textit{Modified invariant } inv6 \end{array}$$

for all q. For $q \neq p$ the proof is easy. For the other case $q = p$ we prove,

$$\begin{array}{ll} & \mathsf{ran}(adr(p) \cup \{s \mapsto r, r \mapsto s\}) \subseteq (arm \cup \{p \mapsto r\})[\{p\}] \\ \Leftarrow & \ldots \\ \Leftarrow & s \in \mathsf{ran}(adr(p)) \end{array}$$

We would expect $s \in \mathsf{ran}(adr(p))$ to hold because doors are symmetric and because of concrete guard $grd2$, that is, $s \in \mathsf{dom}(adr(p))$. We specified symmetry in axiom $axm3$ but this property is not covered by invariant $inv7$. We have to specify it explicitly,

$$inv9 \;:\; \forall\, q \cdot adr(q) \subseteq adr(q)^{-1} \quad . \qquad \text{(see axiom } axm3\text{)}$$

We can continue the proof where we left off

$$\begin{array}{lll} & s \in \mathsf{ran}(adr(p)) & \{\, inv9 \text{ with } \text{“}q := p\text{”} \,\} \\ \Leftarrow & s \in \mathsf{dom}(adr(p)) & \end{array}$$

It is easy to show that invariants $inv7$ and $inv9$ are preserved by event $grant$. Preservation of $inv8'$ can be proved using law (8) and law (9).

Having specified invariant $inv9$ we would now succeed proving theorem $thm1$ postulated in the beginning of this section. This shows that our model satisfies property P7. We do not carry out the proof but turn to the last event not yet refined.

8.3 Revoking door authorisations

We model revoking of door authorisations symmetrically to granting door authorisations. A door authorisation can be revoked if (a) there is an authorisation for the door, (b) the corresponding person is not in the room that could be removed, and (c) the room is not the outside. Condition (a) is just chosen symmetrically to $grd1$ of refined event $revoke$ (for the same reason). The other two conditions (b) and (c) are already present

[3] Event-B has the shorter (and more legible) notation $adr(p) := adr(p) \cup \{s \mapsto r, r \mapsto s\}$ for this. We do not use it because we can use the formula above directly in proof obligations. We also try as much as possible to avoid introducing more notation than necessary.

in the abstraction. The refined events $grant$ and $revoke$ together model property P9.

$revoke$
any p, s, r when
$grd1 : s \mapsto r \in adr(p)$
$grd2 : p \mapsto r \notin loc$
$grd3 : r \neq \boldsymbol{O}$
then
$act1 : adr := adr \lhd\!\!\!- \{p \mapsto adr(p) \setminus \{s \mapsto r, r \mapsto s\}\}$
end

We expect that the guard of event $revoke$ will be to weak to preserve invariant $inv8'$. We are going to search for it in the corresponding proof. But we can get started without it, in particular, proving guard strengthening of the abstract guards $grd1$ to $grd3$ and preservation of $inv5$, $inv6$, $inv7$, and $inv9$. For instance, preservation of $inv6$:

$\forall q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}]$	*Invariant* $inv6$
$s \mapsto r \in adr(p)$	*Concrete guard* $grd1$
$p \mapsto r \notin loc$	*Concrete guard* $grd2$
$r \neq \boldsymbol{O}$	*Concrete guard* $grd3$
$\vdash$	
$\mathsf{ran}((adr \lhd\!\!\!- \{p \mapsto adr(p) \setminus \{s \mapsto r, r \mapsto s\}\})(q))$	
$\subseteq (arm \setminus \{p \mapsto r\})[\{q\}]$	*Modified invariant* $inv6$

for all q. For $q = p$ we have to prove $\mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\}) \subseteq arm[\{p\}] \setminus \{r\}$, thus, $r \notin \mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\})$. This does not look right. Indeed, we find a counter example with one person P and three different rooms $\boldsymbol{H}$, $\boldsymbol{I}$, $\boldsymbol{O}$:

$adr = \{P \mapsto \{\boldsymbol{O} \mapsto \boldsymbol{H}, \boldsymbol{H} \mapsto \boldsymbol{O}, \boldsymbol{O} \mapsto \boldsymbol{I}, \boldsymbol{I} \mapsto \boldsymbol{O}, \boldsymbol{I} \mapsto \boldsymbol{H}, \boldsymbol{H} \mapsto \boldsymbol{I}\}\}$
$arm = \{P \mapsto \boldsymbol{H}, P \mapsto \boldsymbol{I}, P \mapsto \boldsymbol{O}\}$
$loc = \{P \mapsto \boldsymbol{O}\} \quad p = P \quad s = \boldsymbol{I} \quad r = \boldsymbol{H}$

In order to resolve this problem we could remove all doors connecting to r. But this seems not acceptable: we grant door authorisations one by one and we should revoke them one by one. We could also strengthen the guard of the concrete event requiring, say, $adr(p)[\{r\}] = \{s\}$. But then we would not be able to revoke authorisations once there are two or more doors for the same room. The problem is in the abstraction! The abstract event $revoke$ should not always remove r. We weaken the guard of the abstract event using a set R of at most one room instead of r. If $R = \varnothing$, then $\{p\} \times R = \varnothing$. So, for $R = \varnothing$ event $revoke$ does not change arm and for $R = \{r\}$ it corresponds to

the first attempt at abstract event $revoke$:

$$
\begin{array}{l}
revoke \\
\quad \mathsf{any} \quad p, R \quad \mathsf{when} \\
\quad\quad grd1 : p \in Person \\
\quad\quad grd2 : loc(p) \notin R \\
\quad\quad grd3 : R \in \mathbb{S}(Room \setminus \{\boldsymbol{O}\}) \\
\quad \mathsf{then} \\
\quad\quad act1 : arm := arm \setminus (\{p\} \times R) \\
\quad \mathsf{end} \quad ,
\end{array}
$$

where for a set X by $\mathbb{S}(X)$ we denote all subsets of X with at most one element:

$$Y \in \mathbb{S}(X) \quad \widehat{=} \quad Y \subseteq X \wedge (\forall x, y \cdot x \in Y \wedge y \in Y \Rightarrow x = y) \quad .$$

With this the proof obligation for invariant preservation of $inv6$ becomes:

$$
\begin{array}{ll}
\quad \forall q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}] & \textit{Invariant } inv6 \\
\quad s \mapsto r \in adr(p) & \textit{Concrete guard } grd1 \\
\quad p \mapsto r \notin loc & \textit{Concrete guard } grd2 \\
\quad r \neq \boldsymbol{O} & \textit{Concrete guard } grd3 \\
\vdash & \\
\quad \mathsf{ran}((adr \lhd \{p \mapsto adr(p) \setminus \{s \mapsto r, r \mapsto s\}\})(q)) & \\
\quad\quad \subseteq (arm \setminus (\{p\} \times R))[\{q\}] & \textit{Modified invariant } inv6
\end{array}
$$

for all q. For $q = p$ we have to prove,

$$\mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\}) \subseteq arm[\{p\}] \setminus R \quad . \tag{11}$$

Before we can continue we need to make a connection between r and R. We need a witness for R. After some reflection we decide for

$$R = \{r\} \setminus \mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\}) \quad . \tag{12}$$

Witness (12) explains how the concrete and the abstract event are related. If there is only one door s connecting to room r, then $R = \{r\}$ and the authorisation for room r is revoked. Otherwise, $R = \varnothing$ and the authorisation for room r is kept. Now we can prove (11) using $inv6$ and (12). We note without showing the proofs that guard strengthening of the abstract guards $grd1$ to $grd3$ and preservation of $inv5$, $inv7$, and $inv9$ all hold. Only preservation of invariant $inv8'$ remains:

$$
\begin{array}{ll}
\quad \forall q \cdot arm[\{q\}] \subseteq adr(q)^+[\{\boldsymbol{O}\}] \cup \{\boldsymbol{O}\} & \textit{Invariant } inv8' \\
\quad s \mapsto r \in adr(p) & \textit{Concrete guard } grd1 \\
\quad p \mapsto r \notin loc & \textit{Concrete guard } grd2 \\
\quad r \neq \boldsymbol{O} & \textit{Concrete guard } grd3 \\
\vdash & \\
\quad (arm \setminus (\{p\} \times R))[\{q\}] & \textit{Modified invariant } inv8' \\
\quad\quad \subseteq (adr \lhd \{p \mapsto adr(p) \setminus \{s \mapsto r, r \mapsto s\}\})(q)^+[\{\boldsymbol{O}\}] \cup \{\boldsymbol{O}\} &
\end{array}
$$

for all q. Let $D = \{s \mapsto r, r \mapsto s\}$. For $q = p$ we have to show

$$(arm \setminus (\{p\} \times R))[\{p\}] \subseteq (adr(p) \setminus D)^+[\{\boldsymbol{O}\}] \cup \{\boldsymbol{O}\} \quad . \tag{13}$$

We have seen above that the term on term on the left hand side is either $arm[\{p\}]$ or $arm[\{p\}] \setminus \{r\}$. So we won't succeed proving (13) unless we add a guard to event $revoke$. We cannot use $arm[\{p\}]$ in the guard because the refined machine does not contain variable arm. If $inv6$ was an equality, we could use $\mathsf{ran}(adr(p))$ instead of $arm[\{p\}]$, obtaining the guard

$$grd4 \ : \ \mathsf{ran}(adr(p)) \setminus \{r\} \subseteq (adr(p) \setminus D)^+[\{\boldsymbol{O}\}] \cup \{\boldsymbol{O}\} \quad .$$

It says that all rooms except for r must still be reachable from the outside after revoking the authorisation for door D leading to room r. This sounds reasonable. We find that it is not possible to turn the set inclusion into an equality in invariant $inv6$. However, we can still prove the weaker theorem

$$thm2 \ : \ \forall q \cdot \mathsf{ran}(adr(q)) \cup \{\boldsymbol{O}\} = arm[\{q\}] \quad ,$$

using $inv2$, $inv6$, $inv8'$, and property (8) of the transitive closure. The authorised rooms are maintained precisely by means of the authorised doors. As a matter of fact, initially we used $thm2$ as invariant instead of $inv6$ but then weakened the invariant to $inv6$ and proved $thm2$ as a theorem. This is a useful strategy for reducing the amount of proof necessary while keeping powerful properties such as $thm2$.

9 Towards a better model

After all the serious thinking about the model we are confident that the model captures the properties P1 to P9. Assuming we have one person P and three different rooms $\boldsymbol{H}$, $\boldsymbol{I}$, and $\boldsymbol{O}$ we can inspect how the modelled system would behave.

Initially variables adr and loc have the values

$$adr = Person \times \{\varnothing\}$$
$$loc = Person \times \{\boldsymbol{O}\} \quad .$$

Event $pass$ is disabled as expected; $grd1$, that is, $loc(p) \mapsto r \in adr(p)$ cannot be satisfied for any p and r. Similarly, event $revoke$ is disabled, but also event $grant$: guard $grd2$, $s \in \mathsf{dom}(adr(p))$, cannot be satisfied for any s. Deadlock! We have not proved all properties we would expect from our model. This property seems to be implicitly contained in properties P8 and P9, but we have missed it. We have to weaken $grd2$,

$$grd2' \ : \ s \in \mathsf{dom}(adr(p)) \cup \{\boldsymbol{O}\}$$

As a consequence, we have to check again that concrete event $grant$ preserves all invariants. Fortunately, all proofs succeed.

Note, that just adding another proof obligation will not suffice to solve the problem in general. We can easily imagine a lift, say, that does not have a deadlock because some

button could always be pressed and the lift could always move; but the doors of the lift would remain always shut. If we do not specify that we expect the doors to open on some occasions, a model of the lift may not have this property. Because such properties are common sense they are often not mentioned but then they are also easily forgot. We have to analyse the model using complementary techniques such as proving, model-checking, and animation in order to find such mistakes. In the end, the best we can hope for is a model of good quality that captures the required properties well. This problem holds for formal modelling in general. However, it is very visible in the incremental approach described in this article. The proof obligations shown in Section 2 have been restricted not to take into account deadlock-freedom to emphasise the problem that we only verify properties where we expect difficulties but not more. So we can see better the benefits of using jointly the three techniques of proof, model-checking, and animation.

10 Conclusion

What we have learned: We have used proof to verify that the model is consistent and to get indications for improvements of the model. We have used model-checking before attempting a proof. If a counter example was found, the effort of proving could be saved, and the counter example could be analysed. (We could also have started a proof knowing that it would fail.) Finally we have used animation to "try out" the model, to see whether it behaves reasonably. When we animated the model, we found a problem that was not discovered by the other two techniques. Whereas trying out (or systematic testing) does not show absence of errors as proof does, proof only verifies properties where we expect problems. In this sense proof is incomplete too. We have developed a more realistic impression of what a formal method can achieve.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2008. To appear.
3. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006*, volume 4260, pages 588–605. Springer, 2006.
4. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 77(1-2), 2007.
5. D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer, 1994.
6. I. Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.
7. M. Leuschel and M. Butler. ProB : an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.
8. C. C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall, 1994.
9. G. Pólya. *How to Solve It: A New Aspect of Mathematical Method*. Princeton Science Library. Princeton University Press, second edition, 1957.
10. A. J. M. van Gasteren. *On the Shape of Mathematical Arguments*, volume 445 of *LNCS*. Springer, 1990.

Structurer Réaliser et Prouver Comment et Pourquoi

Alain COUTURIER[(1)],Michel GAZEAU[(1)],

Gérald JEAN-BAPTISTE[(1)],Gwenola KERGLONOU[(2)]

(1) CNAM des Pays de la Loire (2) ICAM Nantes

Abstract.
There are three types of difficulties encountered when teaching algorithmic. First, how do we define an algorithm and prove that it is exact? Second, for practical reasons these algorithms concerning A types are realized by B types, how do we validate this realization? And third, a program being a translation in a language belonging to a paradigm, how do we achieve this translation and how can we prove its exactness? In order to illustrate their purpose, the authors present an algorithm allowing the computation of the minimal coverage of graph.

Keywords: algorithm,minimal cover,data structures,morphism.

1 Introduction

L'enseignement de l'algorithmique rencontre d'importantes difficultés. Beaucoup d'enseignants les considèrent comme inéluctables ou les contournent à grand renfort de métaphores ou d'appels au sens commun. Ces difficultés concernent :

1. La confusion souvent faite entre un algorithme et son codage dans un langage de programmation. Généralement définis comme une suite d'opérations, les algorithmes induisent une démarche opérationnelle très proche d'une réalisation en machine et inadaptée à l'étape conceptuelle. La distinction à opérer entre une preuve et un algorithme : un algorithme n'est pas une preuve. La popularité des démarches empiriques par «essais-erreurs» donne la priorité à toute réalisation et assimile l'algorithme à son code.
2. La compréhension et l'utilisation des types : généralement abordé à partir du typage statique offert par certains langages de programmation, ce concept n'est pas compris en phase de spécification. Cela revient à se priver d'une grande partie de sa puissance d'expression.

3. Le choix, souvent inexistant, d'une structure de données permettant d'accueillir le problème posé. Cette difficulté est récurrente en recherche opérationnelle où, dès la présentation du problème à résoudre, les données sont supposées stockées dans un tableau.

Pourtant, l'utilisation de la théorie des ensembles et de la logique du premier ordre sont de nature à faciliter la résolution de ces difficultés d'apprentissage. À partir de quelques exemples, nous allons essayer de montrer :

1. qu'une définition plus rigoureuse des algorithmes lève toute ambiguïté et laisse le champ libre à plusieurs réalisations ;
2. que le *pattern* de la «machine à états» facilite la construction d'une véritable preuve ;
3. que le concept de type abstrait et de morphisme de structures permettent de formaliser le passage de l'algorithme à une réalisation prouvée, dépendante du paradigme choisi ;
4. que l'emploi de la théorie des ensembles et des langages ensemblistes étend l'espace des solutions possibles.

2 Spécifications, algorithmes, preuves, réalisations

Une spécification n'est pas un algorithme, et un algorithme n'est pas du code ou une version francisée d'un langage de programmation.

Pas de "tant que" ,ni de "répéter ... jusqu'à" , ni de "si ... fin si" dans un algorithme.

Et pourquoi pas ? Pour une raison bien simple : un algorithme doit être universel. Il doit pouvoir être utilisé pour écrire des programmes appartenant à n'importe quel paradigme. Or nous devons admettre qu'il existe des langages aussi efficaces que les langages impératifs et qui n'ont :

- ni variables,
- ni structures de répétitions.

Alors qu'est-ce qu' un algorithme ?

Thèse :
Un algorithme est un ensemble de règles de réécriture, permettant, en partant d'une situation initiale, d'aboutir à une situation finale, appelée solution du problème posé. Cette solution doit vérifier les propriétés (ou spécifications) fournies.

2.1 La machine à états

2.1.a Définition

Le concept de machine à états, adapté des travaux de JONES[1], nous permet de présenter les algorithmes dès le début de l'apprentissage et de manière indépendante du langage éventuellement utilisé :

Une machine à états (*mae*), est composée :

1°) d'un ensemble d'états E dont un final. Un état est un n-uplet de composantes caractérisant le système ;

2°) d'une suite finie d'états $s = [e_1 \ldots, e_i, e_{i+1} \ldots, e_n]$ avec $ei \in E$;

3°) d'une fonction $\Phi : E \rightarrow E$, Φ définit le passage de l'état i à l'état suivant i+1.

Pour démontrer que la *mae* est exacte :

1) on vérifie qu'un certain prédicat P (ou un ensemble fini de prédicats) est vrai à l'état initial ou état 0,

2) que si ce prédicat est vrai à l'état i, il reste vrai à l'état i+1.

Le prédicat P est appelé un *invariant*. Si la machine s'arrête au bout d'un nombre fini d'étapes, l'état final vérifie aussi ce prédicat P.

2.1.b Un exemple :

Soit $E = \{e_1, \ldots, e_i, \ldots, e_n\}$ une collection de sous-ensembles. Évaluer

$\mathbf{R} = \bigcup \mathbf{e_i}$ où $i \in I$. Si s est la suite de sous-ensembles $[e_1, \ldots, e_i, \ldots, e_n]$, l'expression que nous cherchons à obtenir est :

$R = e_1 \ldots \cup e_i, \ldots \cup e_n$ où e_i parcourant la suite s

a) La notation utilisée :

[a,b ..]	: suite d'éléments
[ ]	: suite vide
+	: opérateur de concaténation de suites
x : r	: suite non vide, composée d'une tête et d'une queue(reste) r
hd	: renvoie la tête d'une suite non vide
tl	: renvoie une copie d'une suite s, privée de sa tête

b) La machine à états :

La machine à états utilisée possède deux composantes s et e où s est la suite à traiter et e l' ensemble des résultats partiels. Trois équations gèrent cette machine :

(eq 1) état initial : ({}, s)

(eq 2) passage état i vers état (i+1) : $(e , x:r) \rightarrow (x \cup e , r)$

(eq 3) état final : $(e , [\,]) \rightarrow e$

Ces règles induisent l' algorithme suivant :

c) L'algorithme

```
(1) unions s              → unions_aux  {}  s
(2) unions_aux  e  x:r    → unions_aux  (x+e)  r
(3) unions_aux  e  [ ]    → e
```

d) Remarques :

- Cette machine à états *unions* est équivalente à un pliage gauche(foldl) s :
  unions s ≡ foldl union {} s
- Nous utilisons la notation → pour indiquer qu'il s'agit d'une règle de transformation (réécriture) et non les symboles = ou ≡ qui nous paraissent ambiguës.

Il nous faut, maintenant définir le concept de réalisation, c'est à dire comment mettre en œuvre nos algorithmes.

2.2 Morphismes de structures

La notion de type abstrait[3] permet de traiter des éléments sans chercher à connaître la manière dont ils sont représentés en machine. Les opérations sur ces éléments obéissent alors à des règles (ou axiomes).

Un type abstrait est une structure mathématique munie d'opérations de base (ou primitives). La définition d'un type (et de ses opérations de base) comporte deux parties :

- les axiomes de définition ;
- les axiomes de comportement.

Se pose alors la question : comment le réaliser ?

Soient A et B deux types abstraits munis chacun d'un ensemble d'opérateurs ϕ et ϕ'.

Réaliser A en B (réaliser les éléments de type A à l'aide d'éléments de type B) c'est :

- Fournir une fonction T : A → B
- Prouver que les «objets» B se comportent comme les «objets» A

De façon plus formelle le diagramme suivant commute : $\mathbf{T(\phi(x)) \equiv \phi'(T(x))}$

A	**T : ➔**	**B**
ϕ : ⬇		**ϕ' : ⬇**
Eqns	**T : ➔**	Eqns'

2.2.a Un exemple : réaliser le type Queue par un type Tree ?

Le type Queue est muni des opérations *queuevide, dequeue, enqueue* et il vérifie les axiomes de comportement suivants :

a) queuevide(qnil) = true où qnil est une queue vide ;
b) queuevide(enqueue(x, f) = false, f étant une queue quelconque ;
c) dequeue(enqueue(x, qnil)) = qnil
d) dequeue(enqueue(x, f)) = enqueue(x, dequeue(f)) si queuevide(f) = false ;
e) dequeue(qnil) = error

Soit Tree(A) l'ensemble des arbres de type A, solution de l'équation :

Tree(A) = {tnil} | ZTree(A)

Ztree(A) = Leaf(A) | Node({0,1}, Ztree(A), ZTree(A))

muni des opérations d'insertion (*insert*), de suppression (*delete*),de test (*istnil*) de signature :

insert : $A \times Tree(A) \rightarrow Tree(A)$

delete : $Ztree(A) \rightarrow Tree(A)$

istnil : $Tree(A) \rightarrow Bool$

Remarques :

1) ZTree est l'ensemble des arbres non vide.

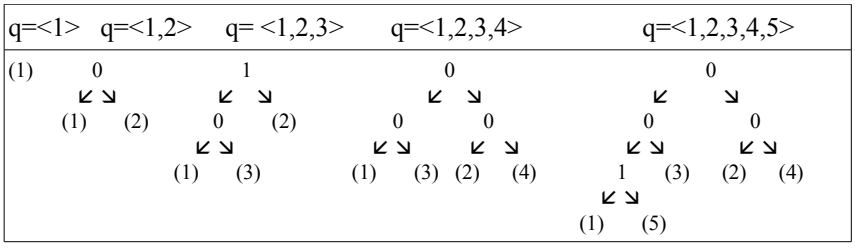
2) Un élément $t \in Tree(A)$ est :

- soit vide ;
- soit une feuille composée d'un élément x, notée leaf(x) ;
- soit un noeud ayant trois composantes (i, G, D) avec $i \in \{0, 1\}$, $G \in Ztree(A)$ et $D \in Ztree(A)$.

L'opération enqueue est réalisée par insert et dequeue par delete

Queue	Tree
qnil	tnil
enqueue(x,qnil)	insert(x, tnil) = leaf(x)
enqueue(y,enqueue(x,qnil))	insert(y, leaf(x)) = node(0, leaf(x), leaf(y))
enqueue(z,q)	si T(q)=node(0, L, R) alors = node(1, insert(z, L), R) si T(q)=node(1, L, R) alors = node(0, L, insert(z, R))

Exemple : insertion des éléments :1, 2, 3, 4, 5.



Remarque : (x) représente une feuille c'est à dire leaf(x)

2.2.b Comment vérifier la correction de la réalisation ? Quelles preuves doit-on apporter ?

Notant **T** le morphisme de transformation, ces réalisations doivent notamment vérifier les équations :

(1) **T**[dequeue(enqueue(x,qnil))] = **T**[qnil]
(2) **T**[dequeue(enqueue(x,q))] = **T**[enqueue(x,dequeue(q))] si queuevide(q) = false.

2.3 Les structures de données implicites

Appelons structures implicites les structures dites "évidentes" , et qui, par leur évidence, masquent la recherche de structures plus générales et/ou plus optimales.

Par exemple, en RO, il est courant de représenter les coûts dans un graphe valué dans un problème de recherche des chemins de coût minimal dans un graphe g = (E,G,C), par un "tableau de coûts". Or, il s'agit non pas d'un tableau mais d'une fonction (donc non obligatoirement partout définie).Si l'on continue à utiliser le terme "tableau de coûts", quelle valeur faut-il mettre pour représenter l'absence de coût ?

Le choix du langage de réalisation[4] et donc de la structure influence trop souvent l'algorithme. A contrario, aucune hypothèse de structuration des données ne doit être choisie si elle n' est pas en relation avec le problème.

Nous allons mettre en œuvre notre démarche sur une classe d'équations booléennes de la forme : $\prod\sum m_{ij}$, équations que l'on rencontre dans différents problèmes d'optimisation.

3 Résolution d' équations booléennes

3.1 Le problème

Dans certaines classes de problème (esims, couverture minimale, cliques, ...) on est amené à déterminer la solution d'équations booléennes se présentant sous forme de produits de sommes de variables :

L = (a + b + c) * (c + d) * (d + e + f + g) *... = 1 (eq 1)

où a,b,c,d ... sont des éléments de l'ensemble **B** = {0,1} muni des opérateurs classiques *non, et, ou, imp* ... Pour des raisons de simplifications d'écriture on écrit :
x' pour ***non(x)***, x + y pour ***ou(x, y),*** x*y ou xy pour ***et(x, y)*** .

La technique classique de résolution est de développer puis de simplifier l'équation (eq1) pour la mettre sous la forme de sommes de produits de monômes :

L simplifiée = a*c*d + ... + a*e*f*g ... = 1
Simplifier c'est appliquer les règles de réécriture :
(r1) a + a = a
(r2) a * a = a
(r3) a + a*b = a
(r4) a*(a+b) = a
(r5) (a+b)(a+c) = a + b*c

Exemple
Considérons l'expression L= (a + b) * (a + b + c) * (d + e) * (d + e + f) = 1.
En développant et en simplifiant on obtient :
L = a * d + a * e + b * d + b * e = 1
L'équation L possède 4 monômes donc 4 solutions

Existe-t-il une structure (au sens informatique) munie d'opérations pouvant traduire ces opérations d' algèbre booléennes ?
Comment représenter les opérations : développer, simplifier ?

3.2 Les morphismes de transformation

Nous proposons de représenter les équations booléennes par des sous-ensembles.

Terme booléen	Représentation ensembliste
monôme somme de monômes	sous-ensemble des variables ensemble de sous-ensembles

Exemples
L'opérateur ∗ est omis.

abcde abc + cde + efg	{a, b, c, d, e} {{a, b, c} , {c, d, e} , {e, f, g}}

Comment alors représenter les opérations *développer* et *simplifier* ?
Définissons deux transformations S,R et un morphisme T caractérisés par la tableau suivant :

Expression booléenne	Morphisme : T	Expression ensembliste
e = (a + b)(a + c)	$\mathcal{T}$: ➔	f = {{{a},{b}}, {{a},{c}}
simplification algébrique **S** : ⬇		simplification ensembliste **R** : ⬇
e' = a + bc	$\mathcal{T}$: ➔	f' = {{a}, {b, c}}

L'expression e = (a + b) (a + c) est

- représentée par : f = **T**(e) = {{a}, {b}}, {{a}, {c}}}
- simplifiée en : e' = **S**(e) = a + bc
- f simplifiée est représentée par : **R**(f) = {{a}, {b,c}}

Nous devons alors vérifier que : **R(T(e)) ≡ T(S(e))**, c'est à dire que le diagramme commute :

T
→
S↓ ↓**R** **T o S = R o T** où **o** est l' opérateur de composition
→
T

3.3 Le traitement

Il comporte 3 fonctions : *solve, traiter, simplifier.*

3.3.a La fonction *solve*

Cette fonction englobe le traitement général : *développer* et *simplifier.*

Signature

solve : S(S(A)) → S(℘(E))

Exemple

solve [[1, 2], [1, 3], [3 + 4][3 + 5]] → {{1, 3}, {2, 3}, {1, 4, 5} }

L'algorithme utilise une machine à états. Chaque état comprend deux composantes (e, r) où e est l'ensemble des résultats partiels et r la suite restante à traiter.

algorithme

- solve s → solve_aux (hd s) (tl s)
- solve_aux e x:r → solve_aux (traiter x e) r
- solve_aux e [ ] → e

3.3.b La fonction *traiter*

Elle consiste à developper et simplifier deux expressions e1 et e2

Signature :

traiter : ℘(℘(E)) × ℘(℘(E)) → ℘(℘(E))
e1 e2 e

Exemple :

traiter {{1}, {2}} {{1}, {3}} → { {1}, {2, 3}}

avec e1 = {{1}, {2}} correspondant, par exemple, à : (a + b)

et e2 = {{1}, {3}} correspondant, par exemple, à : (a + c)

Algorithme :

traiter e1 e2 → simplifier(developper(e1 e2))
avec :
développer e1 e2 → {u1 + u2 : u1 in e1 , u2 in e2}

3.3.c La fonction *simplifier*

Soit E une collection de sous-ensembles e1 ... en . Simplifier E, c'est supprimer les sur-ensembles de E. Si simplifier(E) → F, alors aucun élément de F n'est sous-ensemble strict d'un autre élément de F.

Le prédicat $(\forall u \in F,\ \exists v \in F \mid v \subset u \text{ et } v \neq u)$ est faux

Algorithme :

simplifier exp → exp - { u : u ∈ exp | (∃ v∈ exp : v ⊂ u and v ≠ u) }

3.3.d La fonction *transformer*

Pour faciliter la saisie, l'expression L à traiter est écrite sous la forme de suite de n-plets de variables. Par exemple, L = (a + b + c) (a + c) (b + c + d) (b + d) = 1 est transformée(codée) en :

[[a, b, c], [a, c] , [b, c, d], [b, d]]

Signature

transformer : S(S(A)) → S(℘(℘(A)))
Exemple
transformer [[1, 2], [1, 3], [2, 4]] → [{{1}, {2}}, {{1}, {3}}, {{2}, {4}}]

A nouveau l'algorithme utilise une machine à états. Un état comprend 2 composantes (e, r) où e est l'ensemble des résultats partiels et r la suite restante à traiter.

Algorithme

transformer s → transforme_aux {} s
transformer_aux e x:r → transforme_aux e□{{v} : v in x} r
transformer_aux e [ ] → e

Appliquons cette démarche à un problème classique de recherche opérationnelle [2].

3.4 Application : algorithme de couverture minimale

3.4.a Un exemple

Soit :

C = {p, q, r, s} un ensemble de caméras

D = {a, b, c, d, e} un ensemble de dépôts

r une application de C dans D qui à chaque caméra associe un sous-ensemble de dépôts qu'elle peut surveiller :

$p \rightarrow \{a, c\}$

$q \rightarrow \{b, e\}$

$r \rightarrow \{b, d\}$

$s \rightarrow \{a, d, e\}$

R = { (p, {a, c}) , (q, {b, e}), (r, {b, d}), (s, {a, d, e}) }

	a	b	c	d	e
p	x		x		
q		x			x
r		x		x	
s	x			x	x

Problème : Déterminer le nombre minimal de caméras permettant de surveiller l' ensemble des dépôts.

Solution : Elles doivent alors satisfaire l'équation L :

$$(p + s)\ (q + r)\ p\ (r + s)\ (q + s\ ) = 1\ .$$

D'après le morphisme développé en 2.2, L est transformée en :

solve {{{p}, {s}}, {{q}, {r}}, {{p}}, {{r}, {s}}, {{q},{s}}}

3.4 Prédicats

Soit f = (A, B, F) une correspondance d' un ensemble A dans un ensemble B de graphe F, où F est un sous-ensemble de $A \times B$. Si $(x, y) \in F$ alors y est une image de x (y est un successeur de y ou x est un prédécesseur de y. A toute correspondance f(A, B, F) on peut associer une fonction g=(A, $\wp$(B), G) de A dans $\wp$(B) qui associe à tout x de A le sous-ensemble de ses successeurs.

Remarque : $\wp$(E) désigne l'ensemble des sous-ensembles de E

g = (A ,B, G) : A $\rightarrow$ B

$\downarrow$ $\uparrow$

f = (A, $\wp$(B), F) : A $\rightarrow$ $\wp$(B)

$\downarrow$ $\uparrow$

h= g' = (B, A, H) : B $\rightarrow$ A avec H = G'

$\downarrow$ $\uparrow$

$k = f' = (B, \wp(A), K) : B \rightarrow \wp(A)$

Exemple :

	a	b	c	d	e	f
p	1		1		1	
q		1		1		1
r	1					1
s		1			1	
t			1	1		

F = {(a, {p, r}), (b, {q, s}), (c, {p, t}), (d, {q, t}), (e, {p, s}), (f, {q, r}) }
G = { (a, p), (a, r), (b, q), (b, s), (c, p), (c, t), (d, q), (d, t), (e, p), (e, s), (f, q), (f, r) }
H = G' = {(p, a), (r, a), (q, b), (s, b), (p, c), (t, c), (q, d), (t, d), (p, e), (s, e), (q, f),(r, f)}
K = {(p,{a, c, e}, (q, {b, d, f}), (r, {a, f}), (s, {b, e}), (t, {c, d}) }

Si l'exemple ci-dessus est associé à un problème de couverture minimale ({a, b, c, d, e, f} ensemble de dépôts, {p, q, r, s, t} ensemble de caméras de surveillance), ce problème possède une solution unique :

res = {p, q}.

Pour prouver que res est une couverture il suffit de montrer que :

$E=K(p) \cup K(q)$ où E = domain(G) est l'ensemble des dépôts. Or

domain(G) ={a, b, c, d, e, f}

K(p) ={a, c, e} et K(q) ={b, d, f}.

Remarque : l'ensemble {r, s, t} est une couverture mais non minimale car elle est de cardinal 3.

4.- Conclusion

Même si les langages de programmation évoluent moins vite que les matériels, l'élévation de leur niveau d'abstraction est continue et actuellement, une majorité de spécialistes s'accordent cependant pour dire que demain les langages seront plus déclaratifs, plus dynamiques et plus concurrents et que tous les programmeurs travailleront dans des environnements multi-paradigmes.

Aujourd'hui, un enseignement informatique qui, serait exclusivement basé sur une approche opérationnelle, correspondrait à un enseignement qui, au début des années 80, ignorerait les langages structurés.

C'est dans cette perspective que notre démarche pédagogique a été conçue.

Sans nécessité l'apprentissage d'un formalisme particulier, elle s'appuie sur un usage intensif de la théorie des ensembles, introduit une séparation nette entre algorithmes et réalisations, et implique deux familles de preuves :

- prouver les algorithmes ;
- prouver que les réalisations sont *conformes*, c'est à dire prouver que les transformations des structures utilisées préservent un ensemble de propriétés spécifiques des structures.

Son ambition est de proposer une approche progressive vers les méthodes rigoureuses de développement. Dans un première étape, elle permet d'aborder la phase de réalisation en machine en utilisant un langage ensembliste comme SETL (ou ISETL).

5.- Références

(1) JONES C.B "Software development : a rigourous approach"
(1980) Prentice-Hall New-York.
(2) GONDRAN M , MINOUX M "Graphes et algorithmes "
(1979) Eyrolles Paris
(3) Henderson P"Functional Programming : Application and Implementation "
(1980) Prentice-Hall New-Yorck.
(4) Kingston J.H "Algorithms and data structures "
(1990) Addison-Wesley Sydney

Journées scientifiques - Université de Nantes (FR) - 2009

From Research to Teaching Formal Methods: The B Method

TFM-B'2009

imprimé par IUT de Nantes (Université de Nantes), juin 2009
Publié par APCB
ISBN 2-9512461-0-2
EAN 9782951246102