From Research to Teaching Formal Methods: The B Method (TFM-B'2008)

Christian Attiogbe, Henri Habrias, Frédéric Dadeau, Jacques Julliand, Regis Tissot, Michaël Leuschel, Mireille Samia, Jens Bendisposto, Li Luo, Dominique Méry, et al.

Colloque

The B Method: from Research to Teaching

Nantes, Cité Internationale des Congrès, 16 juin 2008
Nantes, International Convention Center, 16th June 2008

ACTES / PROCEEDINGS

C. Attiogbé, H. Habrias (Eds.)

Juin 2008

lina LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE

publié par APCB, juin 2008
ISBN 2-9512461-2-9
EAN 9782951246126

Journées scientifiques - Université de Nantes (FR) - 2008

Actes/Proceedings

Christian Attiogbé, Henri Habrias (Eds.)

The B Method: from Research to Teaching

Organisation : COLOSS Team - LINA UMR 6241

Site : www.lina.sciences.univ-nantes.fr/apcb/BDayNantes2008, Pascal André

publié par APCB, juin 2008
ISBN 2-9512461-2-9
EAN 9782951246126

Présentation

"L'école a pour vocation la dissidence, ce qui ne se conçoit qu'à étendre l'espace du lisible et à élargir celui du discutable."
Alain de Libera, Penser au Moyen Age, Seuil, p. 108

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. La prochaine aura lieu à London en septembre 2008. A l'occasion des conférences de Montpellier et de Grenoble, des sessions éducation ont eu lieu. Les actes en ont été publiés. L'association APCB a organisé des journées à Paris et à Nantes. Celle de Nantes a été consacrée à l'enseignement. Mais depuis quelques années, il n'y a pas eu de conférence consacrée à l'enseignement. Les travaux de recherche en génie logiciel doivent conduire à la pratique. Et pour cela, il faut passer par l'enseignement, enseignement qui doit allier les fondamentaux et la pratique. Voici deux arguments en faveur de l'enseignement de la méthode B :

- La méthode B et ses concepts ont un intérêt pédagogique essentiel même si la méthode n'est pas appliquée.
- 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.

Après le *B classique*, bien documenté, avec le *B-Book* de J.R. Abrial et les ouvrages qui ont suivi, nous disposons du B événementiel présenté lors de la première conférence B de Nantes (J.R. Abrial : *"Extending B without Changing It (for Developing Distributed Systems*). Peut-être le nouveau livre de J.R. Abrial sur le B événementiel sera disponbible pour la présente conférence. Avec le *B événementiel* et le *B classique* qui va jusqu'à la génération de code prouvé, nous disposons d'une méthode, d'une notation couvrant les différentes étapes d'un projet logiciel. Nous avons considéré qu'il était opportun de créer cette conférence pour traiter du passage de la recherche à l'enseignement. Aujourd'hui, les enseignements fondamentaux de l'informatique sont très souvent délaissés. La méthode B a montré comment mettre en oeuvre ces enseignements fondamentaux. Nous pensons qu'il faut continuer dans cette voie, comme cela a été fait dans tous les autres domaines de l'ingénierie où on fait du "formel" sans le dire. L'enseignement est aussi une occasion de prendre conscience des possibilités d'une méthode, et de nouveaux sujets connexes qui demandent de la recherche. L'enseignement ne peut être déconnecté de la recherche. Mais la recherche s'enrichit aussi de l'enseignement. Nous remercions ceux qui ont répondu à notre appel à communication. Nous espérons que cette conférence permettra aux participants de réutiliser l'expérience d'autres participants, de poser des questions, d'obtenir des réponses et ainsi d'améliorer leur enseignement ou leur pratique et de dégager des pistes de recherche. Nous remercions aussi ceux qui ont participé à l'organisation de la conférence :

- Les membres de l'équipe Coloss du LINA-CNRS : Pascal André a préparé le site web, le processus de dépôt et de revue et celui d'inscription des participants, Christian Attiogbé, l'édition des actes
- 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 chaleureusement notre collègue Didier Bert qui a été le premier président de APCB et un excellent animateur de la communauté du génie logiciel et de B en particulier.

Henri Habrias

Presentation

"L'école a pour vocation la dissidence, ce qui ne se conçoit qu'à étendre l'espace du lisible et à élargir celui du discutable."
Alain de Libera, Penser au Moyen Age, Seuil, p. 108

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. The next one will sit in London on September 2008. During the conferences of Montpellier and Grenoble, education sessions were held with published proceedings. The APBC Association (*International B Conferences Steering Committee*) organised meetings in Paris and in Nantes. The Nantes meeting was devoted to teaching. But since a few years, there was no meeting dedicated to teaching. Software Engineering research must lead to put it into practice. For that, it is necessary to use teaching, teaching that must conciliate the basics and the practice. Here are two 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.
- The B method is an industrial method which reaches the industrialisation level going from specification to programming. There are several examples of concrete applications.

After the *Classical B*, well documented with the B-Book by J. R. Abrial and the books which followed it, we have Event B introduced during the first Nantes B conference (J. R. Abrial : *"Extending B without Changing It (for Developing Distributed Systems)"*). Maybe the new book of J.R. Abrial on the Event B event will be available for this conference. With the *"Event B"*, and the *"Classical B"* which goes to the generation of proved code, we have a method, a language covering the different stages of a software project.
We have considered it appropriate to create this conference dealing with the transition from research to education. Today,teaching the fundaments of computer science are very often neglected. The B method showed how to implement these fundamentals. We believe that we must continue along this path, as was done in all other areas of engineering where the formal techniques are used without saying. The teaching is also an opportunity to become aware of difficulties, of points to clarify, to develop new related topics which require research. It must lead the teacher to ask questions as it must induce the student to ask questions. The teaching cannot be disconnected from the research. But research enriches also teaching.
We thank those who responded to our call for papers. We hope that this conference will enable participants to reuse the experience of other participants, ask questions, get some answers and thereby improving their understanding, their education or their practice and identify some new ways of research. We thank Jean-Raymond Abrial which is once again in Nantes to help us in our project. We also thank those who participated in organizing the conference.

- Members of the team Coloss LINA-CNRS: Pascal André has prepared the Web site, the process of review of communications and of registrations, Christian Attiogbé the edition of the proceedings
- 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 Didier Bert who was chairman of APCB and an excellent animator of the software engineering community, and of B in particular.

Henri Habrias

Comité de programme / Program Committee

Rueda Camilo,	Universidad Javeriana-Cali, Cali, Colombia
Steve Dunne,	University of Teesside, UK
Henri Habrias, [chair]	Université de Nantes, France
Lars-Henrik Eriksson,	Uppsala Universitet, Sweden
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
Guy Vidal-Naquet,	Supelec, Gif-sur Yvette, France
Istenes Zoltàn,	Elte University, Hungaria

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

Leirios Test Generator: from Research to Teaching, through Industry

Frédéric Dadeau, Jacques Julliand and Régis Tissot [1]

Laboratoire d'Informatique
Université de Franche-Comté
16 route de Gray
F-25030 Besançon, FRANCE

Abstract

Once upon a time, at the far far away University of Franche-Comté, existed a research prototype named BZ-Testing-Tools. This tool provided symbolic animation features and test generation from models written as B abstract machines. Based on the scientific and industrial success of the tool, a company had been created, named LEIRIOS Technologies. The BZ-Testing-Tools environment was transferred in the company and redesigned, giving birth to the LEIRIOS Test Generator (LTG). This tool is now being used in the industry by embedded software developers in order to assist them in the validation phase of their product. LTG is still used in the computer science department of the University of Franche-Comté for which it provides a tool support for the practical sessions of the software engineering courses. We illustrate the features of the tool and its use as a teaching environment for the B method on a concrete application of formal modelling. The main idea is that modelling for test generation is a motivating way to introduce formal methods. Teaching the complete application of a model-based test generation process is thus made possible by using the LTG tool. Our students are drawing a benefit from this experience and live happily everafter, using formal methods.

Keywords: BZ-Testing-Tools, LEIRIOS Test Generator, B abstract machine, model-based testing.

1 Introduction

When designing a system by modelling it first, it is important to ensure two things. First, that the model behaves as expected and satisfies some properties, and, second, that the system conforms to the validated model. The first issue can be addressed by employing proof techniques, even if they can only guarantee the preservation of a given set of properties, but not that a system behaves as intended. The second issue can either be solved by employing a dedicated methodology for designing the software or by testing the implementation against the model.

The B method [Abr96] is a quitely used formal modelling technique that makes it possible to describe a complete system, from an abstract model to its implementation, through refinement steps. Each step consists of adding new information in the model w.r.t. the previous level. Proof obligations are associated to each step,

[1] Email: {dadeau,julliand,tissot}@lifc.univ-fcomte.fr

in order to ensure the correctness of the whole workflow. At the abstract level, establishing the proof obligations ensures the correctness of the model w.r.t. the invariant properties, meaning that the initialization of the model establishes the invariant, and the operations preserve it. At each refinement step, the proof obligations ensure that the newly added details preserve the coherence of the system w.r.t. the previous abstract level. This approach is complete, and produces a fully-verified system. Nevertheless, we have noticed that putting it into practice is not so simple, and the targeted audience of this method, industrials and students, are reluctant to use it.

Contrary to this "classical" use of B –where modelling objectives are to derive a correct implementation from the model– we choose to apply an approach where modelling does not aim at producing an implementation but aims at *testing* it. By eluding the successive refinement steps and considering only one B machine with a more or less large abstraction level, this approach is easier to use. Nevertheless, the absence of refinement has to be balanced by other mechanisms in order to related the abstract model with the concrete application. These mechanisms are captured by the adaptation layer, mapping tables of operations of the model and the commands available in the test bench. In this context, B is not restricted to modelling program behaviors, but it can be used to model various systems, such as an automotive windshild wiper controllers, or the security in an airport.

In 2001, the design of a complete framework dedicated to the B notation, that takes these observations into account, started at the University of Franche-Comté, in the LIFC computer science research laboratory. This environment, named BZ-Testing-Tools [ABC+02], takes B abstract machines as input and uses them for generating tests, through a black-box approach [Bei95]. First, BZ-Testing-Tools (BZ-TT) provides a *symbolic animator* in order to validate the model. This semi-automated process consists, for a user, in selecting the operations of the model he/she wants to activate. The system computes the resulting state according to the generalized substitution of the operation in the B machine. The user has then to make sure that the behavior of the models corresponds to what he/she intended. This first feature of the BZ-TT framework is complementary to the verification of properties. It thus makes it possible to validate the behaviors of the model, that will then be used for test generation purposes. Second, the *test generator* feature is dedicated to the automated computation of test sequences from the previously validated model. It extracts test targets as system states that make it possible to activate each operation. Then, the engine automatically computes a test sequence that reaches each of these targets. Coupled with the call of the tested operation, a minimal test case is build, enriched with inputs of the validation engineer.

During its design at the laboratory, the BZ-TT environment was regularily used in the software engineering courses, since it provides an efficient and playful means for validating a B model. It received good feedback from the students, who provided free beta-testing manpower. The BZ-Testing-Tools technology has been transferred in 2003 into the new born company called LEIRIOS Technologies, to be industrialized under the name LEIRIOS Test Generator [JL07]. The new version of the tool is still in use in the computer science teaching department. Whereas the use of the BZ-TT environment was limited to manipulating the animator, we have decided to

provide a full overview of the possibilities of LTG to our students. Therefore, we have designed a class project to be realized during the practical sessions [DT08]. This project consists in a concrete application of a model-based testing approach, from the design of the B model to the run of the tests on set of faulty implementations, through the test generation. This paper also reports on this experience.

The content of the paper is organized as follows. Section 2 introduces the BZ-Testing-Tools framework and its principles. The use of LTG as a tool-support for teaching software engineering is described in Sect. 3. Finally, Section 4 concludes.

2 BZ-Testing-Tools: Features and Principles

The BZ-Testing-Tools environment [ABC+02] is dedicated to the validation of systems specified by B machines. The Z extension had been studied in [Utt00]. The whole process is depicted in Fig. 1. It starts by designing a formal model that is animated in order to ensure its conformance w.r.t. the initial requirements. Then, the model is used as an input for the tests generation process, which automatically generates tests according to model coverage criteria selected by the user. Dedicated scripts are then used to concretize the tests in order to run them on the system under test (SUT). This section presents the two main features of the BZ-Testing-Tools framework: the symbolic animator, and the test generation engine. For each one, the principles are described.

2.1 Symbolic Animation of B Models

The first feature proposed by BZ-Testing-Tools is the possibility to perform *symbolic animation*. Animating a model is a semi-automated tool-supported process which consists in simulating the execution of the model, in interaction with the user which chooses the operations to activate and parameters to pass to these operations. The animation engine computes the resulting state which is displayed in the GUI of the tool. The user checks that the system behaves as expected in the informal require-

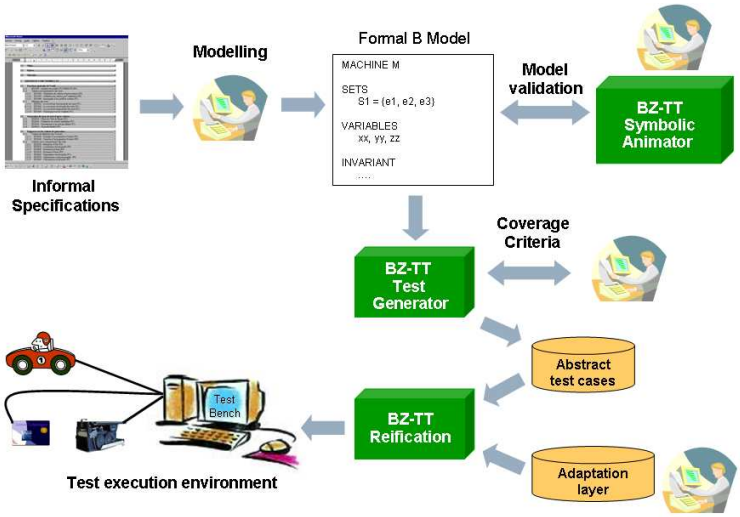


Fig. 1. Process associated to the BZ-Testing-Tools Framework

ments from which the model has been designed. This validation task completes the verification that can be done by proving the correctness of the system w.r.t. the invariants. This is the principle of another B animator, ProB [LB03].

The animation of the models is based on the decomposition of the operations into *behaviors*. Behaviors can be seen as paths in the control flow graph extracted from the generalized substitution of a B operation.

Example 2.1 (Extraction of the behaviors). Let us consider an operation to illustrate the extraction of operation behaviors. The `checkPin` operation hereafter is typical from smart card applications which are the main industrial use of the BZ-TT approach. This operation describes the (simplified) verification of a PIN code given as a parameter (`p`) against the current value of the PIN (`pin`). Notice that this verification is performed only if the parameter is correct and if the PIN differs from its initial value, considered to be -1. The effect of the operation is to update the `isHoldAuth` variable that indicates the authentication status of the user. It also instantiates the result status word `sw` that indicates whether the verification succeeded or failed.

```
sw ← checkPin(p) ≙
   IF p ∈ 0..9999 ∧ pin ≠ -1 THEN
      IF p = pin THEN isHoldAuth := true ‖ sw := ok
      ELSE isHoldAuth := false ‖ sw := wrong_pin
      END
   ELSE sw := wrong_conditions
   END
```

This operation contains three behaviors, that are represented using before-after predicates in which the after values of the state variables are primed:

(i) `p` $\in$ 0..9999 $\wedge$ `pin` $\neq$ -1 $\wedge$ `p = pin` $\wedge$ `isHoldAuth' = true` $\wedge$ `sw = ok`

(ii) `p` $\in$ 0..9999 $\wedge$ `pin` $\neq$ -1 $\wedge$ `p` $\neq$ `pin` $\wedge$ `sw = wrong_pin`

(iii) (`p` $\notin$ 0..9999 $\vee$ `pin` $=$ -1) $\wedge$ `sw = wrong_conditions`

The symbolic animation feature makes it possible for the user to leave some parameters of the operations unspecified. As a consequence these parameters are replaced by symbolic values. The same applies principle applies to the state variable whose values are related to the parameters. Thus, instead of considering concrete states, in which the state variables have a defined and concrete value, symbolic animation considers groups of concrete states, named *symbolic states*. The symbolic values of variables and parameters are managed by underlying customized constraint solvers [BLP04] that make it possible to valuate (i.e. instantiate) these variables, in order to valuate the corresponding execution sequence afterwards.

Example 2.2 (Symbolic animation). Consider the following execution sequence: `init`; `setPin`(X_1); `checkPin`(1234)
in which the `setPin` operation is used to set the value of the PIN code according to the following B operation:

`sw ← setPin(p) ≙ IF p ∈ 0..9999 THEN pin,sw := p,ok ELSE sw := ko END`

The call to operation `setPin` contains a symbolic parameter, which makes it possible to activate the two behaviors of this operation. The first behavior is activated if $\texttt{p} \in 0..9999$. Thus, a symbolic value is used to represent the value of the parameter X_1 with the associated constraint $X_1 \in 0..9999$. Since the `pin` state variable is related to the parameter, its after value also becomes symbolic X_2, with the associated constraint $X_2 = X_1$. The second behavior, which has no effect apart from instantiating the status word, is activated if $\texttt{p} \notin 0..9999$.

The animator GUI provides the user a way to visualize the possible behaviors to activate. He then has to choose the one from which the animation will continue [2]. Suppose that the user selects the first behavior of the `setPin` operation.

The call to the second operation, `checkPin`, detailed in Example 2.1, with a concrete parameter value will have an influence of the current (symbolic) value of `pin`. Because of the value of `p`, it is only possible to active behaviors (i) and (ii) from `checkPin`. If behavior (i) is activated (`pin=p`) then the symbolic value representing the `pin` variable X_2 is instantiated to 1234, and, because of constraint $X_2 = X_1$, we have $X_1 = 1234$. If behavior (ii) is activated ($\texttt{pin} \neq \texttt{1234}$), constraint $X_2 \neq 1234$ is added and propagated so as to obtain: $X_1 = X_2 \wedge X_1 \in 0..1233 \cup 1235..9999$.

2.2 A Model-Based Test Generator

The BZ-TT Test Generator [LPU02] is used to automatically compute tests from a B machine. It considers a model coverage criterion that consists in activating all the behaviors of each B operation. We present here the principles used to generate the tests, and the possible coverage criteria the user can input in order to drive the test generation. Finally, we briefly explain how the generated tests are concretized into executable tests.

2.2.1 Test Targets and Test Cases

Each behavior is composed of an activation condition, i.e. a predicate that has to hold for the behavior to be activated. This predicate corresponds to the before part of the behavior (that does not present any primed variables, or result values) and it is called the *test target*. The computation of the test target is inspired from [DF93] which considers a DNF decomposition of a before-after predicate describing the operations.

Example 2.3 (Test target computation). Consider the `checkPin` operation introduced in example 2.1. The test target corresponding to the different behaviors are the following.

1. $\texttt{p} \in 0..9999 \wedge \texttt{pin} \neq \text{-}1 \wedge \texttt{p} = \texttt{pin}$ behavior (i)
2. $\texttt{p} \in 0..9999 \wedge \texttt{pin} \neq \text{-}1 \wedge \texttt{p} \neq \texttt{pin}$ behavior (ii)
3. $\texttt{p} \notin 0..9999 \vee \texttt{pin} = \text{-}1$ behavior (iii)

The composition of BZ-TT test cases are shown in Fig. 2. Each test starts with a *preamble* that is a sequence of operation calls that reaches a state satisfying the test target. This preamble is automatically computed by the tool. The test *body*

[2] The animator GUI also provides backtracking features, that make it possible to rewind the animation.

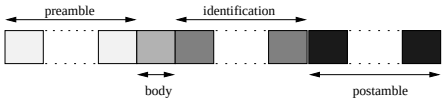


Fig. 2. Composition of an LTG Test Case

is the activation of the behavior itself by invoking the operation with appropriate parameters values. The *identification* is used to perform calls to specific operations, used to observe the system state, and thus, to deduce the verdict of the test. These operations are selected by the user which associates a different set of observers for each tested operation. The optional *postamble* feature makes it possible to reset the system in order to chain the test cases on the system under test. The postamble is optional and, unlike the preamble, it can be automatically computed by the tool.

The preamble and the postamble are computed by performing the exploration of the symbolic state space by the tool. To achieve that, a customized best-first algorithm [CLP04] is put into practice in order to find the shortest path that leads to the target state. This algorithm uses the symbolic animation engine and performs calls to the different operations of the model by systematically leaving the parameters unspecified. The reasearch is bounded on the size of the preamble, the worst case complexity of the algorithm is $\mathcal{O}(n^d)$ in which n is the number of behaviors to consider and d is the depth of the research. Once a symbolic state satisfying the test target's predicate is reached, the computation stops and the corresponding operation sequence is then instantiated in order to concretize the values of the parameters. This process is done by a labelling procedure of the constraint solver that instantiates the symbolic variables to a value that satisfies all the constraints.

2.2.2 Coverage Criteria

As shown in Fig. 1, coverage criteria are selected by the user in order to drive the test generation. BZ-TT considers two kinds of coverage criteria: a decision coverage criterion and a data coverage criterion.

The *decision coverage criterion* makes it possible to improve the test cases by decomposing the disjunctions contained in the test target predicates. This decomposition is based on four rewriting rules of the disjunctions that split the test targets, and, as a consequence, the final number of tests.

Example 2.4 (Decision coverage criteria). Consider test target #3 from example 2.3, which contains a disjunction $\texttt{p} \notin 0..9999 \vee \texttt{pin} = \text{-}1$. The following rewritings can be applied in order to improve the preciseness of the targets.

RW1: $\texttt{p} \notin 0..9999 \vee \texttt{pin} = \text{-}1$. When one of the literals is satified, no matter which one it is, the criterion is satisfied. This rewriting satisfies the Decision Coverage (DC) criterion.

RW2: $\texttt{p} \notin 0..9999 \; [] \; \texttt{pin} = \text{-}1$. The two literals are considered independently, splitting the original test target in two (separeted by []). This rewriting satisfies the Decision and Condition Coverage (D/CC) criterion.

RW3: $\texttt{p} \notin 0..9999 \wedge \texttt{pin} \neq \text{-}1 \; [] \; \texttt{p} \in 0..9999 \wedge \texttt{pin} = \text{-}1$. The two literals have to be exclusively satisfied, splitting the original test target in two. This rewriting satisfies the Full Predicate Coverage (FPC) [OXL99] criterion.

RW4: $\texttt{p} \notin 0..9999 \wedge \texttt{pin} \neq \text{-}1$ [] $\texttt{p} \in 0..9999 \wedge \texttt{pin} = \text{-}1$ [] $\texttt{p} \notin 0..9999 \wedge \texttt{pin} = \text{-}1$. This rewriting considers all the possibilities for satisfying a disjunction, splitting the original test target in three. It satisfies the Multiple Condition Coverage (MCC) criterion.

The *data coverage criterion* makes it possible to specify how the different data, state variables or parameters, have to be covered. The possible choices are "one value", meaning that a single value that satisfies the constraints will be selected, "all values", meaning that all the possible values that satisfy the constraints will be selected, "boundary value" meaning that a numerical data can instantiated to the extrema of its domain. This coverage criterion is employed to define the coverage of the state variables in the test targets, but also to instantiate the parameters after a preamble computation.

Example 2.5 (Data coverage criteria). Consider the last operation sequence issued from example 2.2 ending with the call to the `checkPin` operation using an incorrect pin code (different from 1234). Applying the boundary value coverage on the sequence would produce the two following instantiations: `setPin`(0) and `setPin`(9999), producing two different test cases.

2.2.3 Reification of the Abstract Test Cases

At the end, this test generation process produces abstract test cases[3]. The final step is to concretize the tests in the reification phase. Therefore, the step relies on an *adaptation layer* which is composed of a test translator that computes the tests into the test bench syntax (e.g. JUnit for a Java implementation). It also contains mapping tables that relate the abstract model values to their corresponding concrete values on the SUT. At this level, the user is required to input a correspondance table that maps the abstract names of the model items, more precisely constants and operation names, to concrete ones (e.g. Java constants and Java methods names for a JUnit reification).

The oracle of the test, that decides whether it passes or fails, is deduced by comparing the outputs of the test bench w.r.t. the expected results computed on the model. Here again, the adaptation layer provides the translation/correspondance between these two abstraction levels.

2.3 From BZ-TT to LEIRIOS Test Generator

The LEIRIOS[4] company has been created by Pr. Bruno Legeard, based on the experience and reputation acquired within the BZ-Testing-Tools project. Indeed, this framework has been used in various industrial partnerships, including cutting-edge industries such as Schlumberger/Parkeon (ticket and parking solutions), Axalto (now known as Gemalto) (smart cards) and Peugeot (automotive constructor). These studies have shown the efficiency of the automation of the test design, which provided a better coverage of the code than manual tests.

[3] These tests are said to be abstract because they were computed on the model. They are opposed to concrete tests that can be directly run on an implementation.

[4] http://www.leirios.com

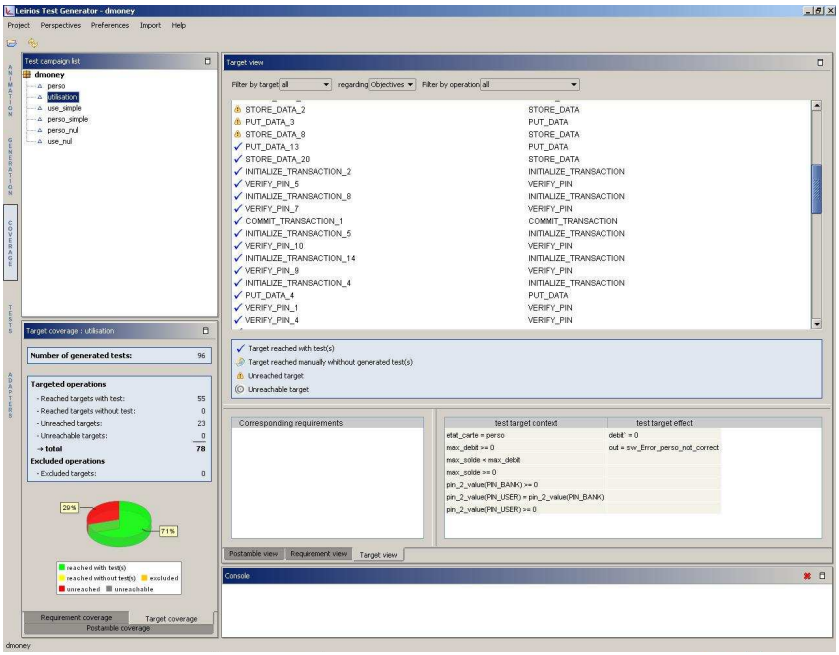


Fig. 3. Screenshot of the LEIRIOS Test Generator GUI

The industrialization and commercialization of the environment, renamed to LEIRIOS Test Generator, gave the opportunity to improve or specialize the approach to adapt it to the needs of industrials [5]. The main evolutions are the following:

- The symbolic animation feature disappeared. It is now replaced by a "classical" animation mechanism in which all the parameters have to instantiated by the user. The reason is that this feature may be interesting for researchers, but not for the industrials.
- Traceability issues of the test cases are taking a central piece. The model operations can now be annotated to introduce *requirements* [BJL+05] relating a piece of the model with an informal need expressed in the specifications. The produced test cases can be related to the original requirements.
- Reports on the coverage of the test targets and/or the requirements coverage are now displayed to the user.
- In order to deal with unreached targets, a special mechanism, named *preamble helpers* has been introduced to manually design execution sequences with the help of the animator. These sequences can be used as preambles for reaching a test target.

LTG is now a complete model based testing solution, of which a screenshot is displayed in Fig. 3. It is used by industrials, but it can also be used as a tool-support for teaching formal methods, as we will now describe.

[5] Notice that industrials are like students: they prefer a tool that works, is displayed in a colorful graphical user interface and provides a lot of interesting services, rather than a research prototype that may be more extensible but less attractive visually, and potentially error-prone.

3 Teaching Software Engineering with LTG

The software engineering course is focused on teaching formal methods, through the learning of the B method to masters' students. The theoretical course covers the scope of the B method, from the abstract machines to the implementation through refinement, proofs and machine composition. The tutorials are dedicated to the application of parts of the methodolgy on several specific exercises. We have decided to focus the content of the pratical sessions on very applied exercises that illustrate to the students a real-life use of formal methods, that also corresponds to our working topic as researchers.

During theoritical sessions and tutorials, our main objective is to teach the B method to the students. Nevertheless, for the practical sessions, our purpose is to encourage students to practice modelling, thus we choose a less reluctant way (for students), which is modelling for testing. This approach has two advantages. First, the modelling work is quickly concretized by using model animation and test generation. Second, it introduces the test to students in a practical way. This approach permit to motivate students to practice modelling and testing which are two notions that they will learn into details in the next year.

As stated in the introduction, the use of a test generation tool is recurring since 2002. At first, a single session, tool-supported by the BZ-Testing-Tools framework, was considered. Now, the maturity of the toolset, especially since the commercial version LTG exists, makes of it a reliable tool on which a complete class project can be led. Thus, we have defined a project that consists, for our students, in employing the full approach of model-based testing on a realistic context, that shows them one of the most widespread application of formal methods.

Before presenting the content of the project and to show the influence of LTG on this project, we first introduce the notion of test-based modelling, that is a different approach from designing a model for properties verification purposes.

3.1 How to Design a Model for the Test?

The test-based design of a model is different from modelling a system for verifying properties or generating code. We first present the differences between the two approaches, then we introduce the test-based modelling technique that is taught to our students. This notion is related to the concept of *design for testability* which consists in adding test-specific features in a system.

3.1.1 Singularity of Test-Based Modelling

As for all design activities, software modelling depends on its final goals. For instance, models can be used either to generate verified concrete code through a refinement and proof process (e.g. using the B method) or to test the conformance of an implementation w.r.t. its specification (e.g. using model-based testing). In this latter, we identify four modelling issues which need to be taken into account during the model design: (*i*) the scope of the test that determines the abstraction level of the model, (*ii*) the abstraction gap between the model and the implementation, (*iii*) the observability of the system under test (SUT), and (*iv*) the restrictions of the test generation technology.

The choice of the model abstraction level depends on what the user wants to test. For instance, consider the validation of a smart card. If the validation objective is to ensure the conformance of the PIN protection system w.r.t. the specification, then we do not need to model data encryption or the other authentication protocols (e.g. asymetric key generation,...).

If the choice of a high abstraction level simplifies the modelling of an application, it creates an abstraction gap between the model and the implementation. Thus, we have to bridge this gap in order to run any test generated from the model on the system under test. Basically, we use an adaptation layer which translates the abstract tests cases (generated from the model) into concrete tests cases (ready to be executed). But sometimes, the developement of an adaptation layer is not easy. For instance, the abstraction of the data encryption feature in the model of the smart card is problematic, because in order to run our test cases on the implementation, each message have to be encrypted before being sent. To solve this problem, we need that the adaptation layer recreates the mechanisms that have been abstracted during the model design. So, choosing an abstraction level for the model of a system is more difficult than choosing what we want to test. It also depends on how it will be possible to concretize the abstract tests.

Observation of the system under test is a crucial problem in the black-box testing domain. In the context, the verdicts only depend on observable elements, through the returned values of the operation calls. For instance, in smart card applications, all the operations can always be invoked (their precondition is true) and return either a success status word or an error code which indicates the cause of the error. An abstraction choice that would consider only two values, e.g. *error* and *success*, is not very relevant, because it reduces the accuracy of the tests verdicts. Indeed, it is not possible to distinguish the different an error from another. Thus, it is very important to identify the informations which can be used to improve the test verdict and to take them into account during the modelling of the system.

The proximity of the data model between the two abstraction levels is crucial for the observation issues. Since the verdict is obtained by comparing the results from the model and the system under test, it is mandatory to be able to compare them. This implies using "compatible" data structures that are the same in the two formalisms (e.g. integers) or that can be translated from one formalism to the other (e.g. functions vs. arrays).

Moreover, the test generation technology is limited by the complexity of the search algorithms. In order to deal with it, it is sometimes required to take into account this issue during the modelling step. Three modelling parts have some influence on this issue: (*i*) the parameters instantiation, (*ii*) the abstraction level, and (*iii*) the initialization of the system.

When modelling for testing, it is mandatory to instantiate all the parameters of the model. Indeed, in order to be able to design relevant tests to be run on the system, the model has to present the same "characteristics" that the system under test. For example, consider a model representing an elevator parameterized by the number of floors. In order to generate the tests for a real elevator (e.g. built in a skyscraper), it is necessary to precisely know how many floors are considered, since

the content of the test sequences may depend on it (e.g. in a test that requests to reach the next floor until the roof). As a consequence, the parameter has to be instantiated.

The abstraction level of the model plays an important role in the reduction of the combinatiorial explosion. This is due to the fact that a very low level of abstraction introduces a lot of unnecessary behaviors in the model w.r.t. the testing scope. For instance, in the smart cards domain, the modelling of the communication protocol between the card and the terminal introduce an unecessary complexity in the model, so it is a good choice to abstract it (as long as this part of the requirement can be restablished by the adaptation layer).

The choice of a good initial state for the model is also important to improve the test cases building. If it is possible to defined one (or more) initial state(s) of the system which reduces the length of the operation sequences to reach the targets. Then, it may reduce both the generation time and the number of unreached test targets. For example, consider a model of a file system; most of the generated tests will require that specific files and directories exist. If the initial state is an empty file system, then the preamble will have to call the operations that create the appropriate configurations for the different tests. Whereas an non-empty file system with existing files and directory will require less effort to the engine, improving its computation time and its chances to reach the most complicated test targets.

Finally, when the test generation is off-line, in order to establish the most accurate conformance relationship between the model and the SUT, the model must be deterministic. In the BZ-TT/LTG approach, a non-deterministic model is not problematic for the test generation, but it may become difficult to establish the verdict when the resulting behavior and values are not the ones that were expected. One solution is to resort to an on-line testing solution, by embedding the model through assertions inside the code of the program that are evaluated at runtime, e.g. by translating the B pre- and postconditions into JML [BCC+05] assertions in a Java program.

3.1.2 Specification Method

Consider software systems in which only the controllable services modify the handled informations and return the observable informations. We propose a method with five steps inspired of [UL06]: (*i*) choose the goal of the test, (*ii*) identify the controllable services and the observable informations, (*iii*) choose the operations of the model separating the controllable and the observable ones; design their signatures and choose how to represent the parameters; design the PROPERTIES clause, (*iv*) choose the state variables and design their types; describe the INVARIANT by expressing invariant properties that have to be verified by the model, (*v*) design the behaviors of the operations by pre- and postconditions; express them as conditions and generalized substitutions.

To choose the test goal many different situations can be considered, depending on the scope of the test and the hypothesis on the environment. For example, we distinguish the case of generating functional tests for the nominal cases from the case of robutness test generation. In the former case, the model contains only nominal cases of interactions with the system. In the latter case, additional elements have to

be taken into account in the model so that the relevant test data and test context can be targeted by the approach. For example, this may result in establishing a defensive model of the system, which adds the behaviors that were not described in the informal specification.

The controllable services will be modelled by operations whose one must define the signature from its definition in the SUT and from the abstraction level of the model. To an implemented service can be associated one or many services of the model. In general, as the model is more abstract than the SUT, a service of the model may be associated to a composition of several services of the SUT. This is the adaptation layer that re-establishes this mapping. The test driver determines the conformance relation between the expected results defined by the model and the results computed by the SUT.

The operation of the model being chosen, we then design their parameters' list and their results. We have to choose their types and how to represent them. Some will be constants to limit the combinatorial explosion if it is compatible with the goal of the test. The design of the state variables is guided by the abstraction level. When the state variables are chosen, we have to design their types and specify their properties in an invariant.

The preconditions of the operations are established according to the test purpose and the kind of operation (controllable or observable). When the robustness is targeted, the specification of the controllable operations is likely to be defensive and the preconditions are only typing conditions of the parameters. On the contrary, if the goal of the test is limited by assumptions on the environment of the system, then the preconditions are strengthened in order to specify the targeted cases. The preconditions of the observable operations express the conditions in which the environment triggers them. The postconditions are defined by the design of generalized substitutions that modify the state variables. The exceptional cases are ordered by the `IF` imbrications designing them in a deterministic manner. The multiple error cases have to be considered separately, or the conformance relationship has to be softened, otherwise, false negative tests can be reported, if the order of errors treatment differs between the implementation and the model.

Moreover, the design of the conditions and the coverage criteria may have an influence on the generated tests. These latter two can be combined in different ways to take the goal of the test into account. For example, condition $x \in \{1..3\}$ will require to correctly drive the test generation using the appropriate data coverage criterion in order to produce three test cases (one for each value of x), that would have been covered by writing $x = 1 \vee x = 2 \vee x = 3$ instead and selecting the appropriate condition coverage criterion.

3.2 Using LTG on a Realistic Application

In this part, we present the project we designed and proposed to the students, in order to reach two objectives. First, it aims at evaluating the students with a realistic case study and to lead them to consider the usefulness of the formal methods for software testing. We try to reach this goal by applying a full model-based testing approach in order to validate a realistic application. Second, our idea

is to introduce this project as a challenge for the students in order to motivate them and to force them to do their work seriously. To accomplish this objective, we ask them to detect faulty programs among a set of different implementations of the same specification. In this project, we rely on the previous modelling techniques to help them designing their model. LTG is used as a tool-support that makes it possible for them to generate the tests.

3.2.1 The Challenge and its Rules

Like any challenge, this project is governed by some rules. Here, we present these rules and the related workflow, given in Fig. 4, that the students have to follow in order to bring their project to fruition.

Each pair of students have to design a B model of the system w.r.t. the specification document provided. They also have to define some invariant properties over their model –some properties are explicitly described in the specifications, and some others are suggested. In order to ensure the correctness of their model, the students have to perform some proof using Click'n'Prove [AC03] to verify the preservation of the invariants properties. They also have to validate the behavior of their model by using the LTG animator. These two complementary approaches (proof and animation) provide an interesting tool-support to design a model, little by little, by successive round trips between the model, the prover and the animator.

Once their model is verified and validated, the students have to generate functional tests using the LTG tool. Here, they have to apply what they have seen previously about the choice of the coverage criteria parameters and the observation phase. These parameters are very important in order to generate tests which cover most of the interesting behaviors of the system and which enable to establish accurate verdicts. The abstract tests generated by the students (saved in XML files) need to be concretized in order to be executed over the real implementation (a Java program) of the system. We provide to them an adaptation layer which translate a XML test file into a JUnit test campaign. This translation is not fully automated: the students have to define a mapping between the operations (resp. parameters) names and types defined in the model and the operations (resp. parameters) defined in the implementation.

The challenge aspect of this project is introduced with the execution of the test cases. Each pair of students receives a different set of 20 implementations which contains 19 mutants and a correct program. Each mutant represent a wrong

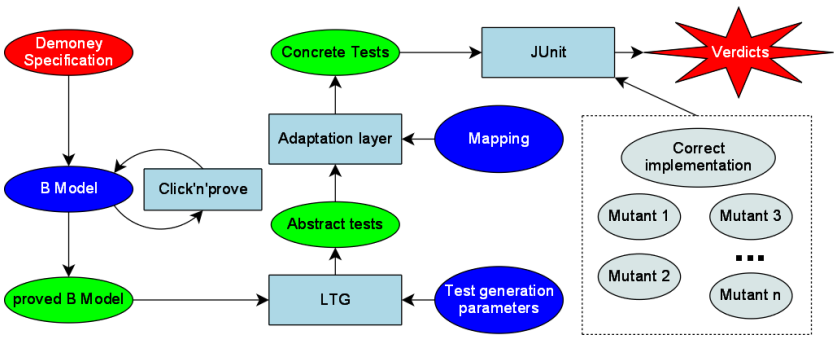


Fig. 4. Workflow of the project, from specification to test execution

implementation of one of the initial informal requirement. Thus, the students' objective is to kill (i.e. to detect) all the mutants and make all the tests pass on the correct program. This motivates them to design a model of the system which is both correct and complete.

3.2.2 Demoney: a Realistic System to be tested

The case study we propose to the student is based on the specification of *Demoney* (Demonstrative Electronic Purse) [MM] developed by *Trusted Logics* for research applications. No implementation of this specification is used in real life, but it is closed to credit card applets and thus, pertinent. Demoney is a good example of a critical system, a banking transaction, which require a lot of validation and verification, but it is also simple enough to be studied as a student project. Our specification describes a card with four life cycle states: *personalization*, *use*, *blocked* and *disabled*. The card is secured with two PIN objects –user and bank– which are associated to retry counters and values. Moreover, the card contains informations such as the current balance, the maximal balance and the maximal debit.

During the *personalization* phase, the PUT_DATA operation makes it possible to set the informations and the objects of the card. Then, the STORE_DATA operation terminates this phase if all the informations are setup and coherent with the security policy. Once the card has quit the personalization phase and started a use phase, it will never come back to the former state. During the *use* phase, the card holder can invoke some commands to gain authentication on PIN, using VERIFY_PIN, and to initialize a financial transaction, with INITIALIZE_TRANSACTION, that has to be completed by a call to the COMMIT_TRANSACTION command. If the user fails so much times to be authenticated that the retry counter of the user PIN reached zero then the card becomes *blocked*. When the card is *blocked*, the authenticated bank can unblock the user PIN and change its value. But if the card is blocked and the retry counter of the bank PIN reaches zero then the card become *disabled* and will never be used again. In addition, the implementation presents observation operations that can be used to retrieve the current balance, the maximal balance, the maximal debit and the authentication state of the PINs. An informal specification is given to the students that details the different behaviors of the system. They have to choose the most relevant design for their model so that interesting test cases are generated.

3.2.3 Model-based Testing of the Demoney Case Study

The students are asked to design a defensive model, that for each operation, considers its correct behavior and the potential errors. In order to be able to deal with multiple errors, the order in which the potential errors have to be checked is indicated in the specification document.

The interest of this specification is to provide some behaviors which require a non-trivial command chaining to be reached. It forces the students to target these behaviors and help the test generation engine. This specification raises the notion of life cycle which is important in security testing. Moreover, the specification contains important properties of the system. Some of these properties can be expressed as invariant properties, whereas some others, like operations chaining properties or

more generally temporal properties, must be respected by operations but can not be formalized as invariant properties. Thus, the students have to select what they are able to model as invariant properties or not. For instance, an invariant property over the system is: "If the card is not in the personalization phase then the balance can not be negative or higher than the maximal balance". At the opposite, the property: "All initialization of a transaction must be immediately followed by a transaction confirmation, otherwise the transaction is cancelled", can not be easily formalized as an invariant property. In this case, the use of the LTG animator can help the students to check that this property is correctly designed in their model.

Finally, in order to deal with the potential complexity of the model, that will slow down the computation time of the test cases, we require the students to produce two test campaigns. The first one is dedicated to the personalization phase, and the second considers a given personalization, that the students have to determine (and justify), so that these tests directly starts from the use phase of the card.

3.2.4 Feedbacks

The main objective of the approach, interesting students to formal methods by using a practical point of view, seems to be sucessfully fulfilled, according to the feedback they gave us on the course evaluation. Moreover the study of a smart card application was really interesting for them.

The challenge introduced by using some mutants maintained the students' interest until the end of the project, this is partially due to the fact that some mutants were very difficult to detect, thus, they spend the very last minutes to generate and run tests to kill the remaining mutants. If the use of faulty implementations to motivate the students and to evaluate their ability to produces quality tests, it also offer a good benchmark to them to evaluate their model and to help them to improve it, by using the correct model as a reference.

The tool chain that we have proposed worked perfectly and none of the students complained about bugs or strange features of the LTG tool. The animator was very useful in the early phases of model design. The test generator was used without any problems, even if some of the students had to perform many tries before understanding the proper use of some of the coverage criteria. Basically, the choice of the LTG tool was well-received by the students who appreciated the visual of the tool and the features it proposed.

4 Conclusion

We have presented in this paper the principles and the story of the BZ-Testing-Tools framework, initially developed in our research laboratory, and transferred into the industry. This tool consists of a model animator and a test generator that we use as a support for teaching formal methods to masters students. The maturity acquired by the tool makes a reliable tool-support for our students to learn the B notation, and to apply it into a realistic context. This year, we have designed a project [DT08] that makes it possible for the students to apply a complete model-based test generation approach, from the design of the model, to the execution and the analysis of the tests.

The design of the project offered a good support to practice formal methods in an application validation context. Furthermore, it provided an industrial-like model based testing approach to test the conformance of a system. The similarities between this class project and an industrial validation activity reside in both the method and the tools that were used. Moreover, the Demoney case study is very close to a "real" smart card application. Thus, all these facts motivated the students and their feedback on the project was excellent.

References

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic, August 2002. INRIA Technical Report.

[Abr96] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[AC03] J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In D.A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference (TPHOLs 2003)*, volume 2758, pages 1–24, Rome, Italy, 2003. Springer.

[BCC+05] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[Bei95] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[BJL+05] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting. Requirement traceability in automated test generation - Application to smart card software validation. In *Procs. of the ICSE Int. Workshop on Advances in Model-Based Software Testing (A-MOST'05)*, St. Louis, USA, May 2005. ACM Press.

[BLP04] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B: A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):143–157, August 2004.

[CLP04] S. Colin, B. Legeard, and F. Peureux. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3):213–235, 2004.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the International Conference on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[DT08] F. Dadeau and R. Tissot. Teaching Model-Based Testing with the Leirios Test Generator. In *Proceedings of the International Workshop on Formal Methods in Computer Science Eduction (FORMED'08)*, Budapest, Hungary, March 2008.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated Test Generation from B Models. In J. Julliand and O. Kouchnarenko, editors, *7th International Conference of B Users (B'2007), Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *LNCS*, pages 277–280. Springer, 2007.

[LB03] M. Leuschel and M. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *Proceedings of the International Conference on Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 21–40, Copenhagen, Denmark, July 2002. Springer Verlag.

[MM] R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse - card specification.

[OXL99] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5^{th} IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006. 550 pages.

[Utt00] Mark Utting. Data Structures for Z Testing Tools. In *4th workshop on Tools for System Design and Verification (FM-TOOLs 2000)*, Reisenburg Castle, Germany, July 2000.

Easy Graphical Animation and Formula Visualisation for Teaching B [1]

Michael Leuschel[2] Mireille Samia[3] Jens Bendisposto[4] Li Luo

Softwaretechnik und Programmiersprachen
Institut für Informatik, Heinrich-Heine-Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany

Abstract

ProB is being used for teaching the B-method. In this paper, we present two new features of ProB that we have introduced while teaching B. One feature allows a student (or an expert user) to graphically visualise any predicate as a tree. The underlying algorithm can deal with undefined subformulas and tries to provide useful feedback even for existentially quantified formulas which are false. This feature is especially useful to inspect unexpected invariant violations or operations which are unexpectedly enabled or disabled. The other feature enables a student or lecturer to easily and quickly write custom graphical state representations, to provide a better understanding of the model. With this method, one simply has to assemble a series of pictures and to write an animation function in B itself, which stipulates which pictures should be shown where depending on the current state of the model. As an additional side-benefit, writing the animation function in B itself is a good exercise for students.

Keywords: B-Method, Tool Support, Animation

1 Introduction

There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another. These activities are supported by tools, such as Atelier-B, B4Free, and the B-toolkit. In addition to the proof activities, it is increasingly being realised that validation of the initial specification is important to avoid deriving a "correct" implementation of an incorrect specification. This validation can come in the form of *animation*, e.g., to check that certain functionality is present in the specification. Another useful tool is *model checking* [6], whereby the specification can be systematically checked for properties expressed in a temporal logic. In previous work [9], the ProB animator

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

[2] Email: leuschel@cs.uni-duesseldorf.de

[3] Email: samia@cs.uni-duesseldorf.de

[4] Email: bendisposto@cs.uni-duesseldorf.de

and model checker has been presented to support those activities. The tool can also be used to complement proof activities, as it supports automated consistency and refinement checking of B machines.

ProB [9] is used at several universities for teaching the B-method. [5] Usually the feedback of students is very positive, and we feel that animation helps the students get a better understanding of the B formalism. The students have also provided useful feedback about the tool, and over the years we have added many new features to make the tool more useful. One such feature, which we present in Section 2, allows the student (but also the industrial user) to inspect formulas and graphically visualise them as a tree. It is especially useful to analyse unexpected invariant violations or operations which are unexpectedly enabled or disabled.

The possibility to see and to explore the behaviour of a formal model is often invaluable for students. ProB provides feedback about the current state, the enabled operations and can be used to visualise the statespace of a model [10]. However, sometimes a more graphical, domain-specific visualisation of the current state of a formal model is desirable, to give the student a better understanding of the model. In earlier work [3], we have presented a flash-based animation engine. However, this engine still requires careful setup and the development of gluing code together with Flash animations. It is thus usually too complicated for students to set up and use themselves, and even for lecturers the overhead can be prohibitive. In this paper, we provide a very simple but effective way of producing custom animations of a model. With this method, the student or lecturer simply has to assemble a series of pictures and has to write an animation function in B itself, which stipulates which pictures should be shown where depending on the current state of the system. As an additional side-benefit, writing the animation function in B itself is a good exercise for students.

Take for example the following B machine describing the well-known sliding 8-puzzle, where numbered tiles can be moved horizontally and vertically (into an empty square). The goal is to reach a configuration where all tiles are in order. On the left of Figure 1 you can see the non-graphical visualisation provided by ProB. It is not unreadable, but the graphical visualisation on the right is clearly much more inspiring and understandable. We now show how this animation can be achieved using our new version of ProB with very little effort.

```
MACHINE Puzzle8
DEFINITIONS INV == (board: ((1..dim)*(1..dim)) -->> 0..nmax);
GOAL == !(i,j).(i:1..dim & j:1..dim =>
          board(i|->j) = j-1+(i-1)*dim);
CONSTANTS dim, nmax
PROPERTIES dim:NATURAL1 &  dim=3 & nmax:NATURAL1 & nmax = dim*dim-1
VARIABLES board
INVARIANT INV
INITIALISATION board : (INV & GOAL)
OPERATIONS
   MoveDown(i,j,x) = PRE i:2..dim & j:1..dim &
      board(i|->j) = 0 & x:1..nmax & board(i-1|->j) = x
     THEN board := board <+ {(i|->j)|->x, (i-1|->j)|->0}
   END;
   MoveUp(i,j,x) = PRE i:1..dim-1 & j:1..dim &
      board(i|->j) = 0 & x:1..nmax & board(i+1|->j) = x
     THEN board := board <+ {(i|->j)|->x, (i+1|->j)|->0}
   END;
   MoveRight(i,j,x) = PRE i:1..dim & j:2..dim &
```

[5] For example, Besançon,Nantes in France; Southampton and Surrey in England; McMaster University, in Canada, Uppsala University in Sweden, and of course Düsseldorf in Germany.

```
      board(i|->j) = 0 & x:1..nmax & board(i|->j-1) = x
    THEN board := board <+ {(i|->j)|->x, (i|->j-1)|->0}
  END;
  MoveLeft(i,j,x) = PRE i:1..dim & j:1..dim-1 &
      board(i|->j) = 0 & x:1..nmax & board(i|->j+1) = x
    THEN board := board <+ {(i|->j)|->x, (i|->j+1)|->0}
  END
END
```

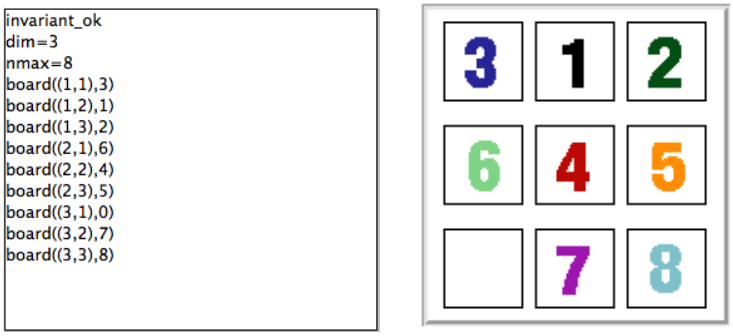


Fig. 1. Puzzle8 Non-Graphical and Graphical Visualisation

Another example is the scheduler from [8], whose code is repeated in Figure 2. Figure 3 shows both the non-graphical animation on the left, as well as the graphical animation obtained using our new tool, which clearly shows to the student and the user how the processes progress through the various stages.

```
MACHINE scheduler
SETS PID = {process1,process2,process3}
VARIABLES active, ready, waiting
INVARIANT
   active <: PID & ready <: PID &  waiting <: PID &
   (ready /\ waiting) = {} &
   active /\ (ready \/ waiting) = {} &
   card(active) <= 1 &
   ((active = {}) => (ready = {}))
INITIALISATION active := {} || ready := {} || waiting := {}
OPERATIONS
   new(pp) =
     SELECT pp : PID & pp /: active & pp /: (ready \/ waiting)
     THEN waiting := (waiting \/ { pp })
   END;
   del(pp) =
     SELECT pp : waiting
     THEN waiting := waiting - { pp }
   END;
   ready(rr) = SELECT rr : waiting
     THEN waiting := (waiting - {rr}) ||
      IF (active = {})
      THEN active := {rr}
      ELSE ready := ready \/ {rr}
      END
     END;
   swap = SELECT active /= {}
     THEN
      waiting := (waiting \/ active) ||
      IF (ready = {}) THEN active := {}
      ELSE ANY pp WHERE pp : ready
           THEN active := {pp} || ready := ready - {pp}
           END
      END
     END
END
```

Fig. 2. Scheduler Specification

We present this new feature for ProB in Section 3 along with typical examples from teaching, and apply it to a larger case study the size of a typical student project in Section 5.

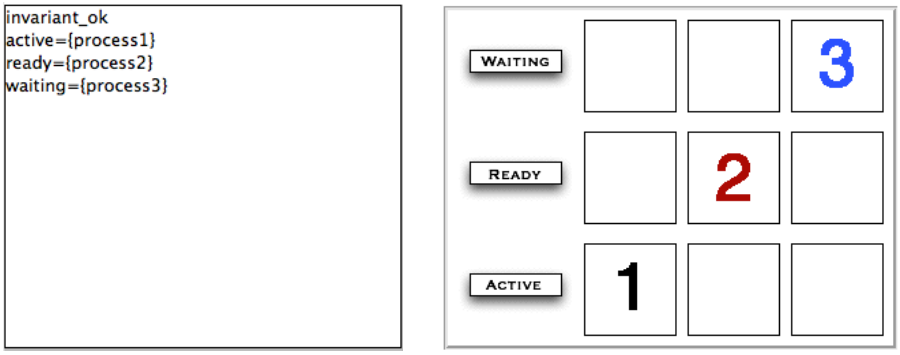


Fig. 3. Scheduler Non-Graphical and Graphical Visualisation

2 Visualising Formulas

Animation has the purpose of identifying unexpected behaviours of a model. If an unexpected behaviour does occur, the user often would like to know more about the source of the problem.

- If the invariant is violated, one would like to know exactly which part of the invariant is violated and why.
- If an operation is unexpectedly not enabled, or unexpectedly enabled, one would like to know the reason.
- If the animator cannot find values for the constants which satisfy the properties of a machine, one would like to be able to locate the problematic properties.

For this, ProB had for quite some time the ability to inspect the invariant and also to debug the properties. However, this view was often not precise enough, and we have now implemented an algorithm which can take any B predicate or expression and translates it into a graphical representation that can be inspected by the user. We believe this feature to be especially useful for students and newcomers to B, but we also believe it to be important when animating complex specifications.

The algorithm basically uses the ProB interpreter to compute the value of an expression or the truth-value of a predicate. It then tries to decompose the expression or predicate into sub-expressions or -predicates. These are in turn recursively evaluated, until we reach sub-expressions or -predicates which can no longer be decomposed. The whole is then assembled into a graphical tree representation and rendered using the GraphViz package [2]. For example, Figure 4 contains a visualisation of the invariant of the scheduler machine, for the state already depicted earlier in Figure 3. For each expression, we have two lines of text: the first indicates the type of the node, i.e., the top-level operator. The second line gives the value of evaluating the expression. For predicates, the situation is similar, except that there is a third line with the formula itself and that the nodes are coloured: true predicates are green and false predicates are red.

Note that our algorithm also deals with undefined predicates. Those are rendered in orange. For example, the formula $x \in dom(f) \wedge f(x) = 1$, where f is the empty function (and x some value aa) would be visualised as in Figure 5.

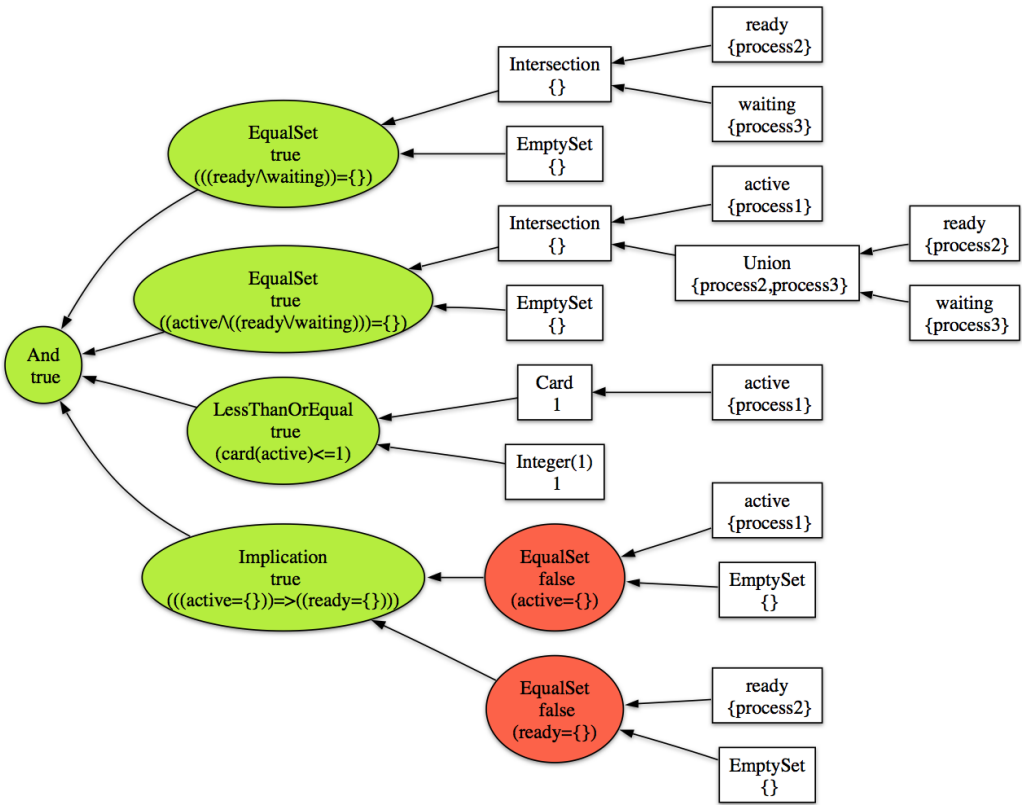


Fig. 4. Visualising the invariant of the scheduler for state of Figure 3

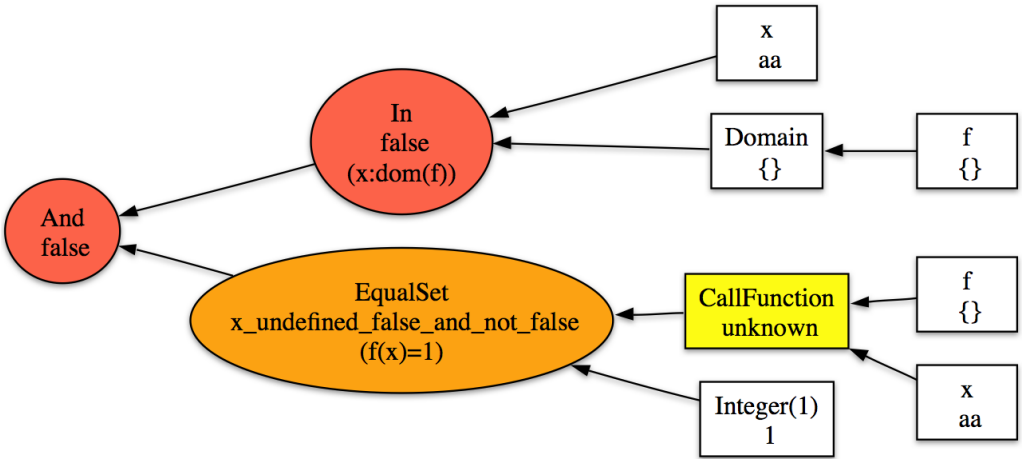


Fig. 5. Visualising a predicate with undefined sub-predicates

One interesting problem is what to do when an existentially quantified formula is false. In this case, our algorithm removes conjuncts from the end of the body of the quantified formula, until the formula becomes true. An example is shown in Figure 6, where we apply our algorithm to the guard of the `new` operation of the scheduler machine (for the state already depicted earlier in Figure 3). Note that the guard is modelled by an existential quantification over the parameters of the operation. As can be seen, the existential formula is false, but our algorithm has de-

tected that by removing the first condition $pp \notin active$, the formula would become satisfiable. This information can be very valuable in detecting exactly where an inconsistency arises inside a formula. In previous works [5,11], different algorithms based on SAT solvers were developed to find a Minimal Unsatisfiable Subformula (MUS). Our approach to find a short unsatisfiable formula is simple. We are currently interested in improving our algorithm by evaluating how the approaches in [5,11] can be applied to our Prolog constraint solving technique.

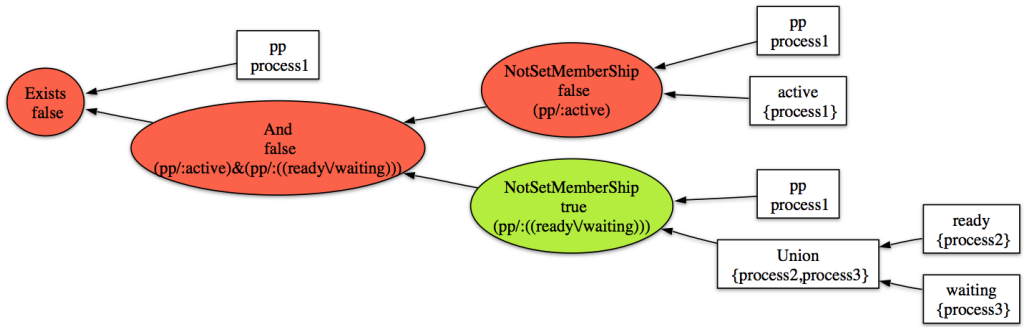


Fig. 6. Visualising the guard of `new` of the scheduler for the state of Figure 3

3 The new Graphical Animation Model

The animation model is very simple:

(i) The basic units are individual images. The images are given a number and their source file location is declared in the DEFINITIONS section of the animated machine. A definition `ANIMATION_IMG`x `==` "$filename$", defines the image with number x where $filename$ is the path to a gif image file.

(ii) The graphical visualisation consists of a two-dimensional grid, each cell in the grid can contain an image. The same image can appear multiple times in the grid.

(iii) The graphical visualisation is recomputed for every state, by evaluating a user-defined animation function f_a. The animation function f_a is declared by defining `ANIMATION_FUNCTION` in the DEFINITIONS section and must be of type `INTEGER * INTEGER +-> INTEGER`. If the function is defined for r and c, this means that the animator should display the image with number $f_a(r, c)$ at row r and column c. If f_a is undefined at r and c, then no image is displayed in that cell.

The dimension of the grid is computed by looking at the minimum and maximum coordinates that occur in the animation function. More precisely, the rows are in the range $min(dom(dom(f_a)))..max(dom(dom(f_a)))$ and the columns are in the range $min(ran(dom(f_a)))..max(ran(dom(f_a)))$.

For our 8-puzzle example, one could thus write the following animation function, together with an image declaration list. The result of using this animation function can be seen on the right of Figure 1.

```
ANIMATION_FUNCTION == ( {r,c,i|r:1..dim & c:1..dim & i=0} <+ board);
ANIMATION_IMG0 == "images/sm_empty_box.gif";  /* empty square */
```

```
ANIMATION_IMG1 == "images/sm_1.gif";  /* square with 1 inside */
ANIMATION_IMG2 == "images/sm_2.gif";
ANIMATION_IMG3 == "images/sm_3.gif";
ANIMATION_IMG4 == "images/sm_4.gif";
ANIMATION_IMG5 == "images/sm_5.gif";
ANIMATION_IMG6 == "images/sm_6.gif";
ANIMATION_IMG7 == "images/sm_7.gif";
ANIMATION_IMG8 == "images/sm_8.gif"; /* square with lightblue 8 inside */
```

Each of the three integer types in the signature can be replaced by a deferred or enumerated set, in which case our tool translates elements of this set into numbers. In case of enumerated sets, the number is position of the element in the definition of the set in the SETS clause. Deferred set elements are numbered internally by ProB, and this number is used. (Note, however, that the whole animation function has to be of the same type; otherwise the animator will complain about a type error.)

To avoid having to produce images for simple strings, one can use a declaration `ANIMATION_STR`x `== "my string"` to define image with number x to be automatically generated from the given string.

Typical patterns for the animation function are as follows:

- A useful way to obtain a function of the required signature is to write a set comprehension of the following form:
  `{row,col,img | row:1..NrRow & col:1..NrCols & P}`, where `P` is a predicate which gives `img` a value depending on `row` and `col`.
- Another useful pattern is to write one function for default images, and then use the override operator to replace the default images only when needed: `DefaultImages <+ CurrentImages`. This results in much more concise and readable functions. This was used in the 8-Puzzle, by setting as default the empty square (image 0) overriden by the partially defined `board` function.
- Translation predicates between user sets and numbers (extension above can directly handle user sets, but does not work well if we need a special image for undefined,...)

4 Further Examples

Below we show three more examples, which illustrate how our new animation model can be used. The examples also show that, despite its simplicity, the model is powerful enough to provide interesting animations for a variety of models. This will be further corroborated in Section 5.

4.1 Towers of Hanoi

The Towers of Hanoi problem is widely used to teach recursion and problem solving. A B model of this problem is as follows. The animation function is surprisingly simple (although we needed to make it slightly more complicated to ensure that the stakes are not shown upside down). The graphical result can be seen in Figure 7.

```
MACHINE Hanoi
SETS Stakes
DEFINITIONS GOAL == (!s.(s:Stakes & s/=dest => on(s) = <>));
scope_Stakes == 1..3;
ANIMATION_FUNCTION == ({r,c,i|r:1..nrdiscs & c:Stakes & i=0} <+
  {r,c,i|r:1..nrdiscs &
   c:Stakes & r-5+size(on(c)): dom(on(c)) &
   i = on(c)(r-5+size(on(c)))});
```

```
ANIMATION_IMG0 == "images/Disc_empty.gif";
ANIMATION_IMG1 == "images/Disc1.gif";   /* the smallest disc */
ANIMATION_IMG2 == "images/Disc2.gif";
ANIMATION_IMG3 == "images/Disc3.gif";
ANIMATION_IMG4 == "images/Disc4.gif";
ANIMATION_IMG5 == "images/Disc5.gif";  /* the largest disc */
CONSTANTS orig,dest,nrdiscs
PROPERTIES orig: Stakes & dest:Stakes & orig /= dest & nrdiscs = 5
VARIABLES on
INVARIANT on : Stakes --> seq(INTEGER)
INITIALISATION
   on := %s.(s:Stakes & s /= orig | <>) \/
   {orig |-> %x.(x:1..nrdiscs|x)}
OPERATIONS
   Move(from,to,disc) =
     PRE
      from:Stakes & on(from) /= <> & to:Stakes & to /= from &
      disc:NATURAL1 & disc = first(on(from)) &
      (on(to) /= <> =>  first(on(to))> disc)
     THEN
      on := on <+ { from |-> tail(on(from)), to |-> (disc -> on(to))}
   END
END
```

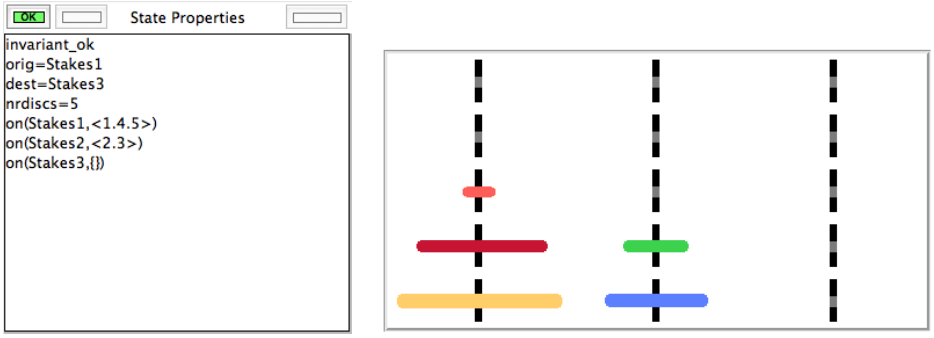


Fig. 7. Hanoi Non-Graphical and Graphical Visualisation

4.2 Scheduler

We return to the scheduler example from [8], whose code is in Figure 2. Here we require a more complicated animation function, because we have to map `PID` elements to image numbers. The result of the animation has already been shown in Figure 3.

```
IsPidNrci ==
 ((p=process1 & i=1) or (p=process2 & i=2) or (p=process3 & i=3));
ANIMATION_FUNCTION ==  ({1|->0|->5, 2|->0|->6, 3|->0|->7} \/
  {r,c,img|r:1..3 & img=4 & c:1..3} <+
  ({r,c,i| r=1 & i:INTEGER & c=i & #p.(p:waiting & IsPidNrci)} \/
   {r,c,i| r=2 & i:INTEGER & c=i & #p.(p:ready & IsPidNrci)} \/
   {r,c,i| r=3 & i:INTEGER & c=i & #p.(p:active & IsPidNrci)} ));
ANIMATION_IMG1 == "images/1.gif";
ANIMATION_IMG2 == "images/2.gif";
ANIMATION_IMG3 == "images/3.gif";
ANIMATION_IMG4 == "images/empty_box.gif";
ANIMATION_IMG5 == "images/Waiting.gif";
ANIMATION_IMG6 == "images/Ready.gif";
ANIMATION_IMG7 == "images/Active.gif"
```

It would have been more elegant to use subsidiary definitions with arguments, such as:

```
IsPidNr(c,i) ==
 ((c=process1 & i=1) or (c=process2 & i=2) or (c=process3 & i=3))
```

However, the current parser of ProB (derived from jbtools) cannot deal with definitions with arguments when used inside other definitions. We are currently deploying a new parser developed by Fabian Fritz using SableCC, which will overcome

this problem.

4.3 Sudoku

For our course, we have also developed a B model of Sudoku, and show students how they can use ProB to solve Sudoku puzzles. The machine has the variable `Sudoku9` of type `1..fullsize-->(1..fullsize+->NRS)`, where `NRS` is an enumerate set `{n1, n2, ...}` of cardinality `fullsize`. The animation function is as follows:

```
Nri == ((Sudoku9(r)(c)=n1 => i=1) & (Sudoku9(r)(c)=n2 => i=2) &
        (Sudoku9(r)(c)=n3 => i=3) & (Sudoku9(r)(c)=n4 => i=4) &
        (Sudoku9(r)(c)=n5 => i=5) & (Sudoku9(r)(c)=n6 => i=6) &
        (Sudoku9(r)(c)=n7 => i=7) & (Sudoku9(r)(c)=n8 => i=8) &
        (Sudoku9(r)(c)=n9 => i=9)  );
ANIMATION_FUNCTION == ( {r,c,i|r:1..fullsize & c:1..fullsize & i=0} <+
    {r,c,i|r:1..fullsize & c:1..fullsize & c:dom(Sudoku9(r)) &
           i:1.. fullsize & Nri} );
```

Figure 8 shows the non-graphical visualisation of a particular puzzle, then the graphical visualisation of the puzzle as well as the visualisation of the solution found by ProB (after a couple of seconds).

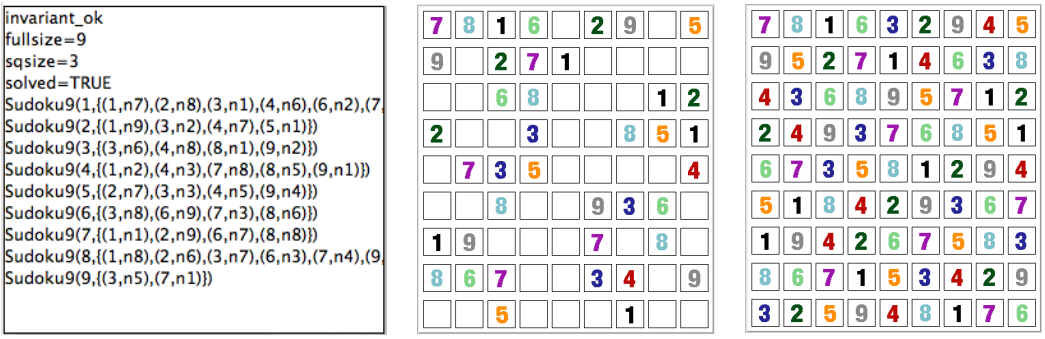


Fig. 8. Sudoku Non-Graphical and Graphical Visualisation

Note that it would have been nice to be able to replace `Nri` inside the animation function simply by `i = Sudoku9(r)(c)`. While our visualisation algorithm can automatically convert set elements to numbers, the problem is that there is a type error in the override: the left-hand side is a function of type `INTEGER*INTEGER+->INTEGER` while the right-hand side now becomes a function of type `INTEGER*INTEGER+->NRS`. One solution is to extend our animator to accept multiple definitions of the animation function. We have done so, and the user can, in addition to the standard animation function, optionally define a default background animation function. The standard animation function will override the default animation function, but the overriding is done within the graphical animator and not within a B formula. In this way, one can now rewrite the above animation as follows:

```
ANIMATION_FUNCTION_DEFAULT == ( {r,c,i|r:1..fullsize & c:1..fullsize & i=0} );
ANIMATION_FUNCTION == ( {r,c,i|r:1..fullsize & c:1..fullsize &
    c:dom(Sudoku9(r)) & i:1.. fullsize &  i = Sudoku9(r)(c)} )
```

The scheduler animation from Section 4.2 can now also be rewritten more elegantly as follows:

```
  ANIMATION_FUNCTION_DEFAULT ==
     ( {1|->0|->5, 2|->0|->6, 3|->0|->7} \/ {r,c,img|r:1..3 & img=4 & c:1..3} );
  ANIMATION_FUNCTION == ( {r,c,i| r=1 & i:PID & c=i & i:waiting} \/
                                {r,c,i| r=2 & i:PID & c=i & i:ready} \/
                                {r,c,i| r=3 & i:PID & c=i & i:active}
                              )
```

5 A more detailed case study: Formalising and Visualising a Lift

In this section, we provide a more detailed case study and apply our animation technology to a bigger specification, similar to a typical student project: the model of a lift along with a controller that ensures that requests are eventually served.

Our experience was very positive, it was relatively straightforward to provide an appealing graphical visualisation of the model. This greatly enhanced the understandability of the model, and allowed us to spot errors more quickly (e.g., in one version of the model, the lift could get stuck on the top floor, which was not that obvious to spot in the non-graphical visualisation).

In the following, we actually present two models of a lift. The first model makes no distinction between internal and external call buttons. The second model is a refinement of the first, and here there are separate panel buttons inside the lift and external call buttons on each floor. In both cases, the lift can move between a ground floor (constant `groundf`) and a top floor (constant `topf`). The state of both lift models consists of the current floor the lift is on (variable `cur_floor`), whether its door is open or not (variable `door_open`), whether it is currently moving up or down (variable `direction_up`) as well as the state of the buttons. In Figure 9, we present the graphics we used in the definition `ANIMATION_IMG`x `== "`$filename$`"` and describe their associated event. Note that the images used before refinement are from x=0 to 6.

5.1 The Lift Model before Refinement

Below are the constants and variables of the first lift model.

```
MODEL LiftM
CONSTANTS groundf,topf
PROPERTIES
   topf : INTEGER &
   groundf : INTEGER &
   groundf = -1 &
   topf = 2 &
   groundf < topf
VARIABLES  call_buttons,cur_floor,direction_up,do_count,
   door_open,inside_panel_buttons
INVARIANT
   cur_floor : groundf .. topf &
   door_open : BOOL &
   call_buttons : POW(groundf .. topf) &
   direction_up : BOOL &
   inside_panel_buttons : POW(groundf .. topf) &
   do_count : 0 .. 3
   ...
```

Below is our animation function. The left part of the relational overriding `<+` gives the default values (i.e., images), and the right part gives the actual images.

```
DEFINITIONS
  Rconv == topf-r+groundf;
  ANIMATION_FUNCTION ==
   ( {r,c,i|r:groundf..topf & ((c=2 & i=0) or (c=1 & i=2))} <+
    ({r,c,i|r:groundf..topf & Rconv:call_buttons & c=2 & i=1} \/
     {r,c,i|r:groundf..topf & Rconv=cur_floor & c=1 &
        ((door_open=TRUE & i=3) or (door_open=FALSE & i=4))}) \/
     {r,c,i| r=topf+1 & c=1 & ((direction_up=TRUE & i=5)
        or (direction_up=FALSE & i=6)) } );
```

Our default images are 0 and 2. Our current images are 1, 3, 4, 5 and 6. In our case, the lowest floor `groundf` and the highest floor `topf` are equal to -1

ANIMATION_IMG x == "filename"			x	"filename"	Image		Description of the Event
	Before Refinement		0	"images/lift/B_CallButtonOff.gif"		Outside the lift	A call button is off.
			1	"images/lift/B_CallButtonOn.gif"			A call button is on.
			2	"images/lift/LiftEmpty.gif"			The lift is not on a specific floor.
			3	"images/lift/B_LiftOpen.gif"			The lift's door is opened.
			4	"images/lift/B_LiftClosed.gif"			The lift's door is closed.
			5	"images/lift/B_up_arrow.gif"			The lift's direction up is on.
			6	"images/lift/B_down_arrow.gif"			The lift's direction down is off.
	After Refinement		7	"images/lift/B_up_arrow_off.gif"			The lift's direction up is off.
			8	"images/lift/B_floor_U1_off.gif"		Inside the lift	The panel button U1 is off.
			9	"images/lift/B_floor_U1_on.gif"			The panel button U1 is on.
			10	"images/lift/B_floor_E_off.gif"			The panel button E is off.
			11	"images/lift/B_floor_E_on.gif"			The panel button E is on.
			12	"images/lift/B_floor_1_off.gif"			The panel button 1 is off.
			13	"images/lift/B_floor_1_on.gif"			The panel button 1 is on.
			14	"images/lift/B_floor_2_off.gif"			The panel button 2 is off.
			15	"images/lift/B_floor_2_on.gif"			The panel button 2 is on.
			16	"images/lift/B_down_arrow_off.gif"			The lift's direction down is off.
			17	"images/lift/B_floor_U1.gif"		Outside the lift	Floor U1
			18	"images/lift/B_floor_E.gif"			Floor E
			19	"images/lift/B_floor_1.gif"			Floor 1
			20	"images/lift/B_floor_2.gif"			Floor 2

Fig. 9. The Definition ANIMATION_IMGx == "$filename$" and the Description of the Event associated to each Image

and 2, respectively. The minimum and maximum coordinates, which occur in the animation function, are 2 and 5, respectively. Then, the dimension of the grid is 3x2. To determine the value of r, we sometimes use `Rconv==topf-r+groundf`. The values of `Rconv` are in the ranges `groundf..topf`. `Rconv` depends on the values obtained, when a call button is pushed or when the lift is on the current floor. For instance, if the current floor `cur_floor` is equal to 2, then `Rconv` is also equal to 2. Consequently, applying `r==topf+groundf-Rconv`, the row r is equal to -1. If `door_open=TRUE`, then the image number `i=3` is displayed at column 1. Otherwise, if `door_open=FALSE`, then the image number `i=4` appears at column 1. More details are presented in Figure 10. Due to space restrictions we do not show the graphical visualisation of this model, only of the more refined model later in Figure 12.

ANIMATION_FUNCTION == DefautsImages <+ CurrentImages		r	c	i = x	Rconv= =topf-r+groundf	Image	Explanation
	DefaultImages	groundf..topf	2	0			When a call button is off, the default image is displayed at column 2.
			1	2			When the lift is not on a specific floor, the default image is displayed at column 1.
	CurrentImages	groundf..topf	2	1	Rconv:call_buttons (call_buttons are in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)		A call button is on. It is displayed at column 2. The row is determined according to the value of **Rconv**.
		groundf..topf	1	3	Rconv=cur_floor (The value of the current floor cur_floor is in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)		The image is displayed at column 1, when **door_open=TRUE** (the lift's door is opened).
				4			The image is displayed at column 1, when **door_open=FALSE** (the lift's door is closed).
		topf+1	1	5			The image is displayed at column 1 and row 3, when **direction_up=TRUE** (the lift's direction is up).
				6			The image is displayed at column 1 and row 3, when **direction_up=FALSE** (the lift's direction is down).

Fig. 10. The Definition ANIMATION_FUNCTION before Refinement

5.2 The Lift Model after Refinement

After refinement, the ProB animation function is as follows. Note that, due to the distinction between internal buttons and external call buttons, the graphical visualisation has become more sophisticated and the animation function thus more complicated.

```
DEFINITIONS
  pushed_buttons == (call_buttons \/ inside_panel_buttons);
  Rconv == topf-r+groundf;
  ANIMATION_FUNCTION ==
   ({r,c,i|r:groundf..topf & ((c=3 & i=0) or (c=2 & i=2))} <+
    ({r,c,i|r:groundf..topf & Rconv:call_buttons & (c=3 & i=1)} \/
    {r,c,i|r:groundf..topf & Rconv=cur_floor & c=2 &
      ((door_open=TRUE & i=3) or (door_open=FALSE & i=4))}) \/
    {r,c,i|c=1 & ((r=groundf & i=20) or (r=groundf+1 & i=19) or
      (r=topf-1 & i=18) or (r=topf & i=17)) } \/
    {r,c,i| r=topf+1 &
      ((direction_up=TRUE & ((c=1 & i=5) or (c=6 & i=16))) or
      (direction_up=FALSE & ((c=1 & i=7) or (c=6 & i=6)))) } \/
    {r,c,i|r=topf+1 & c=2 & (((-1):inside_panel_buttons & i=9) or
      ((-1)/:inside_panel_buttons & i=8))} \/
    {r,c,i|r=topf+1 & c=3 & ((0:inside_panel_buttons & i=11) or
      ((0/:inside_panel_buttons) & i=10))} \/
    {r,c,i|r=topf+1 & c=4 & ((1:inside_panel_buttons & i=13) or
      (1/:inside_panel_buttons & i=12))} \/
    {r,c,i|r=topf+1 & c=5 & ((2:inside_panel_buttons & i=15) or
      (2/:inside_panel_buttons & i=14))});
...
```

In the lift model `LiftR_1`, we use 21 gif images. Note that the first seven gif images were also used in the lift model `LiftM`. In the relational overriding `<+`, the default values (i.e., images) are 0 and 2 and our current images are 1 and from 3 to 20. As in the lift model `LiftM`, `groundf` and `topf` are equal to -1 and 2, respectively. The dimension of the grid is 3x6. Moreover, we sometimes use `Rconv` in order to compute the value of r. More details about the definition `ANIMATION_FUNCTION`; i.e, the position of each image in the graphical visualisation, the image's number and

ANIMATION_FUNCTION == DefautsImages <+ CurrentImages		r	c	i=x	Rconv= = topf-r+groundf	Image	Explanation
	DefaultImages	groundf..topf	3	0			When a call button is off, the default image is displayed at column 3.
			2	2			When the lift is not on a specific floor, the default image is displayed at column 2.
	CurrentImages	groundf..topf	3	1	Rconv:call_buttons (call_buttons are in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)		A call button is on. It is displayed at column 3. The row is determined according to the value of **Rconv**.
		groundf..topf	2	3	Rconv=cur_floor (The value of the current floor cur_floor is in the range groundf..topf. Then, Rconv is also in this range. r is computed as r=topf+groundf-Rconv)		The image is displayed at column 2, when **door_open=TRUE** (the lift's door is opened).
				4			The image is displayed at column 2, when **door_open=FALSE** (the lift's door is closed).
		topf+1	1	5			The image is displayed at column 1 and row 3, when **direction_up=TRUE** (the lift's direction is up).
				7			The image is displayed at column 1 and row 3, when **direction_up=FALSE**
			6	6			The image is displayed at column 6 and row 3, when **direction_up=FALSE** (the lift's direction is down).
				16			The image is displayed at column 6 and row 3, when **direction_up=TRUE**
		groundf	1	20		2	The image is displayed at column 1 and row -1.
		groundf+1		19		1	The image is displayed at column 1 and row 0.
		topf-1		18		E	The image is displayed at column 1 and row 1.
		topf		17		U1	The image is displayed at column 1 and row 2
		topf+1	2	8		U1	If (-1)/:inside_panel_buttons, the image is displayed at column 2 and row 3.
				9		U1	If (-1):inside_panel_buttons, the image is displayed at column 2 and row 3.
		topf+1	3	10		E	If 0/:inside_panel_buttons, the image is displayed at column 3 and row 3.
				11		E	If 0:inside_panel_buttons, the image is displayed at column 3 and row 3.
		topf+1	4	12		1	If 1/:inside_panel_buttons, the image is displayed at column 4 and row 3.
				13		1	If 1:inside_panel_buttons, the image is displayed at column 4 and row 3.
		topf+1	5	14		2	If 2/:inside_panel_buttons, the image is displayed at column 5 and row 3.
				15		2	If 2:inside_panel_buttons, the image is displayed at column 5 and row 3.

Fig. 11. The Definition ANIMATION_FUNCTION after Refinement

the image's associated event are provided in Figure 11.

Figure 12 contains an animation sequence, starting from the initial state (a1), where the lift is on the ground floor, with its door closed, no buttons pressed and the lift controller being in upwards mode. Between states (a1) and (a2), the operation `push_call_button(-1)` has been executed. The user can clearly see the effect of the operation. This operation now also enables the `open_door` operation, after whose execution we reach state (a3). This enables the `close_door` operation, whose execution leads us to state (a4). Here, we can clearly see that the call button to floor U1 has been turned off again. After (a4), the user has executed the operation `push_call_button(1)` leading to (a5). This enabled the `move_up` operation, after whose execution we reach state (b1), etc...

6 Related Work and Conclusions

As far as related work is concerned, we would like to mention the Possum animation tool [7] for Z. The latter is probably most related, as it allows the user to write custom TclTk code that can query the state of a Z specification in order to provide a custom graphical visualisation. The most closely related work on the B side is [4,1], which uses a special purpose constraint solver over sets (CLPS) to animate B and Z specifications using the so-called BZ-Testing-Tools. However, the focus of these tools is test-case generation and not verification, and the subset of B that is supported is comparatively smaller (e.g., no set comprehensions or lambda abstractions, constants and properties nor multiple machines are supported). To our knowledge no graphical visualisation for states is available.

In summary, we have provided two new graphical visualisation features for ProB: one to visualise predicates as a tree and the other to view the state of a B model in a domain-specific manner. We have shown the usefulness of these features on a series of examples. We hope that this provide further value to teaching B to students.

Acknowledgements

We would like to thank Daniel Plagge for his suggestions in Section 2. We also wish to extend our thanks to the reviewers for their helpful comments.

References

[1] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of FATES'02, Formal Approaches to Testing of Software*, pages 105–120, August 2002. Technical Report, INRIA.

[2] AT&T Labs-Research. Graphviz - open source graph drawing software. Obtainable at `http://www.research.att.com/sw/tools/graphviz/`.

[3] J. Bendisposto and M. Leuschel. A generic flash-based animation engine for ProB. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 266–269, Besancon, France, 2007. Springer-Verlag.

[4] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.

[5] R. Bruni and A. Sassano. Restoring Satisfiability or Maintaining Unsatisfiability by Finding Small Unsatisfiable Subformulae. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001) Boston (Massachusetts, USA), June 14-15, 2001, Proceedings*. Elsevier Science Pub., 2001.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. *Automated Software Engineering*, 00:302, 1998.

[8] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proceedings FME'02*, LNCS 2391, pages 21–40. Springer-Verlag, 2002.

[9] 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.

[10] M. Leuschel and E. Turner. Visualizing larger states spaces in ProB. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proceedings ZB'2005*, LNCS 3455, pages 6–23. Springer-Verlag, April 2005.

[11] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

A Sudoku Example

```
MACHINE SudokuSETS9
SETS
 NRS = {n1,n2,n3,n4,n5,n6,n7,n8,n9}
DEFINITIONS
  Column(jj) == %ii.(ii:1..fullsize & jj:dom(Sudoku9(ii)) | Sudoku9(ii)(jj));
  ColumnSol(jj) == %ii.(ii:1..fullsize | SudokuSol(ii)(jj));
 SUBSqIndx == {1..3, 4..6, 7..9}
CONSTANTS sqsize,fullsize
PROPERTIES
  fullsize = card(NRS) & sqsize:NAT1 & sqsize*sqsize = fullsize
VARIABLES Sudoku9, solved
INVARIANT
  Sudoku9: 1..fullsize --> (1..fullsize +-> NRS) & solved:BOOL
INITIALISATION
  Sudoku9:= %i1.(i1:1..fullsize|{}) || solved := TRUE
OPERATIONS
  Set(i,j,nr) =
   PRE
    i:1..fullsize & j:1..fullsize & j/: dom(Sudoku9(i)) &
    nr:NRS & nr/:ran(Sudoku9(i)) & nr/:ran(Column(j))
   THEN Sudoku9(i) := Sudoku9(i) \/ {j |-> nr}
  END;
  StartSolve =
   PRE
    solved=TRUE
   THEN solved := FALSE
  END;
  SetPuzzle1 =
   BEGIN
    Sudoku9 := { 1 |-> { 1|->n7, 2|->n8, 3|->n1, 4|->n6,   6|->n2, 7|->n9, 9|->n5 },
                 2 |-> { 1|-> n9, 3|->n2, 4|->n7, 5|->n1 },
                 3 |-> { 3|-> n6, 4|-> n8, 8|->n1, 9|->n2},
                 4 |-> { 1|-> n2, 4|->n3, 7|->n8, 8|->n5, 9|->n1} ,
                 5 |-> { 2|-> n7, 3|->n3, 4|->n5, 9|->n4} ,
                 6 |-> { 3|-> n8, 6|->n9, 7|->n3, 8|->n6}  ,
                 7 |-> { 1|-> n1, 2|->n9, 6|->n7, 8|->n8} ,
                 8 |-> { 1|-> n8, 2|->n6, 3|->n7, 6|->n3, 7|-> n4, 9|-> n9} ,
                 9 |-> { 3|-> n5, 7|->n1} }
   END;
  Solve =
   ANY
    SudokuSol
   WHERE
    solved=FALSE &
    SudokuSol: 1..fullsize --> (1..fullsize --> NRS) & /* all values are specified */
    !(i,j).(i:1..fullsize & j:1..fullsize & j:dom(Sudoku9(i))
     => SudokuSol(i)(j) = Sudoku9(i)(j))& /*all existing values copied from current board*/
    !(i,j1,j2).(i:1..fullsize & j1:1..fullsize & j2:1..fullsize &
    j2 > j1 => (SudokuSol(i)(j1) /= SudokuSol(i)(j2) &  /* all different on a row */
    SudokuSol(j1)(i) /= SudokuSol(j2)(i)    /* all diferent on a column */ )) &
    !(xi,yi).(xi: SUBSqIndx & yi: SUBSqIndx =>
     !(i1,j1,i2,j2).(i1:INTEGER & i2:INTEGER & j1:INTEGER & j2:INTEGER &
    i1:xi & i2:xi & i2 >= i1 & j1:yi & j2:yi &
    (i2=i1 => j2>j1)  /* (i1|->j1) /= (i2|->j2) */
     => SudokuSol(i1)(j1) /= SudokuSol(i2)(j2))
   THEN Sudoku9 := SudokuSol || solved := TRUE
   END
END
```

Fig. 12. Lift Graphical Visualisation

Teaching programming methodology using Event B

Dominique Méry[1,2]

LORIA
Université Henri Poincaré Nancy 1
Vandoeuvre lès Nancy,France

Abstract

Event B is supported by the RODIN platform and provides a framework for teaching programming methodology based on the famous pre/post specifications, together with the refinement. We illustrate a methodology based on Event B and the refinement by developing Floyd's algorithm for computing the shortest distances of a graph, which is based on an algorithm design technique called dynamic programming. The development is based on a paradigm identifying a non-deterministic event with a procedure call and by introducing control states. We discuss points related to our lectures at the university.

Keywords: Event B, refinement, seuential algorithm, pattern, proof, teaching, formal method, recursive procedure, development, correct by construction

1 Foreword

It is a great pleasure to thank Henri Habrias for his lectures on B at the University of Nantes. He understood the rôle of mathematics in the curriculum of computer scientists and the impact of the B methodology in industry and in education. Moreover, he was not only teaching notations but concepts that the young computer scientist, who has attended his lectures, will understand when the maturity will be there. He helps our community to propagate *the two mammels of computer science*, namely abstraction and refinement and is becoming now the King Henri. He was a solid support to B. It is a very modest exercise to discuss with Jean-Raymond, You and colleagues of our meeting in Nantes. Thanks Henri!

2 Introduction

Overview. Event B is supported by the RODIN platform and provides a framework for teaching programming methodology based on the famous pre/post specifications, together with the refinement. We illustrate a methodology based on Event B and the refinement by

[1] Email: mery@loria.fr

[2] Work of Dominique Méry is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

developing algorithms for computing the shortest distances of a graph, which is based on an algorithm design technique called dynamic programming. Floyd's algorithm is redeveloped and we add comments on the complexity of proofs and on the discovery of invariant; it should be considered as an illustration of a technique introduced in a joint paper with D. Cansell[8]. The development is based on a paradigm identifying a non-deterministic event with a procedure call and by introducing control states. We discuss points related to our lectures at different levels of the university. It is also a way to introduce a pattern used for developing sequential structured programs.

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

Proof-based Development. Proof-based development methods[5,1,14] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete events. The relationship between two successive models in this sequence is that of *refinement*[5,1]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination. A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine[4]. At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations. The goal of a event B development is to obtain a *proved model* and to implement the correctness-by-construction[13] paradigm. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited

by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity.

Organization of the paper. Section 2 introduces introduces definitions related to the methodology and details for representing the pattern used for designing the algorithm. Section 3 describes the development of the *shortest-path* problem and the relationship between models and programs; it illustrates the methodology for developing structured programs. Section 4 develops Floyd's algorithm using the same pattern and discusses issues related to the implementation in the C programming language. Finally, we conclude our work in the last section.

3 The modelling framework

We do not recall oncepts of the Event B modelling language developed by J.-R. Abrial[2,7]; we sketch the general methodology we are applying. The ingredients for describing the modelling process based on events and model refinement can be found in [2,7]. We assume that the goal is to solve a given problem described by a semi-formal mathematical text and we assume that the problem is defined by a precondition and a postcondition[14]. The modelling process starts by identifying the domain of the problem and it is expressed using the concept of **CONTEXT**. A **CONTEXT** PB (see Figure 1) states the theoretical notions required to be able to express the problem statement in a formal way. The **CONTEXT** PB declares

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

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

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

CONTEXT PB

SETS
D

CONSTANTS
x, P, Q

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

END

Fig. 1. Context for modelling the problem PB

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

Table 1
Set-theoretical notation for event B models

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

MACHINE PREPOST

SEES PB

VARIABLES
y

INVARIANTS
$inv1 : y \in D$

EVENTS

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

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

END

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

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

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

The next section illustrates the technique used for developing new algorithms. We think that it is a good way to teach the design of algorithms. HOARE logic[11] provides a very interesting framework for dealing with specifications an development and our work shows how the ingredients of HOARE logic can be used to provide a general framework for developing sequential programs correct by construction. Event B and the RODIN plateform can be used to teach basic notions like pre and postconditions, invaraint, verification and finally design-by-contract.

Teacher's note: *The challenge of the teacher is to relate the Event B notations to the notations of the programming language. We have used the Event B notations in lectures on fixed-point theory and on the explanation of sequential algorithms. It is then clear that we should provide more systematic rules for deriving algorithms. The management of definitions using a tool, like RODIN, helps students to understand why a function call like $f(x)$ generates conditions like $x \in dom(f)$. Nobody can cheat with the tool. Moreover, when a tool is available for a free download, it is really a teachermate.*

4 The Shortest Path Problem

4.1 Summary of the problem

Floyd's algorithm[10] computes the shortest distances of a graph and is based on an algorithmic design technique called dynamic programming: simpler subproblems are first solved before the full problem is solved. It computes a distance matrix from a cost matrix: the costs of the shortest path between each pair of vertices are in $O(|V|^3)$ time.

Teacher's note: *In the case of Floyd's algorithm, there is a mathematical definition of the matrix we have to compute from a starting state defining the initial basic link between nodes with cost. The function is called d and should be first defined in a context of the problem.*

The set of nodes N is $1..n$, where n is a constant value and the graph is simply represented by the distance function d ($d \in N \times N \times N \pfun \mathbb{N}$) and when the function is not defined, it means that there is no vertex between the two nodes. The relation of the graph is defined as the domain of the function d. n is clearly greater than 1 and it means that the set of nodes is not empty.

The distance function d is defined inductively from bottom to top a ccording to the dynamic programming principle and the next axioms define this function:

- $axm1 : d \in N \times N \times N \pfun \mathbb{N}$
- $axm5 : \forall i \cdot i \in N \limp 0 \mapsto i \mapsto i \in dom(d) \land\ d(0 \mapsto i \mapsto i) = 0$
- $$axm6 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land\ (k-1 \mapsto i \mapsto k \notin dom(d) \lor k-1 \mapsto k \mapsto j \notin dom(d)) \end{array} \right) \\ \limp \\ \left(k \mapsto i \mapsto j \in dom(d) \land\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \right) \end{array} \right)$$
- $$axm7 : \forall i, j, k \cdot \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land\ k-1 \mapsto i \mapsto k \in dom(d) \\ \land\ k-1 \mapsto k \mapsto j \in dom(d) \\ \land\ d(k-1 \mapsto i \mapsto j) \leq d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \limp \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \land\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \end{array} \right)$$

- $$axm8 : \forall i, j, k \cdot \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \wedge\ k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\ k-1 \mapsto k \mapsto j \in dom(d) \\ \wedge\ d(k-1 \mapsto i \mapsto j) > d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \Rightarrow \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \wedge\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right)$$

- $$axm9 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \notin dom(d) \\ \wedge\ k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\ k-1 \mapsto k \mapsto j \in dom(d) \end{array} \right) \\ \Rightarrow \\ k \mapsto i \mapsto j \in dom(d) \end{array} \right)$$

The optimality property is derived from the definition of d itself, since it starts by defining bottom elements and applies an optimal principle summarized as follows: $D_{i+1}(a, b) = Min(D_i(a, b), D_i(a, i + 1) + D_i(i + 1, b))$ and means that the distances in D_i represent paths with intermediate vertices smaller than i; D_{i+1} is defined by comparing new paths including $i + 1$. D_i is defined by a partial function over $N \times N \times N$. The partiality of d leads to some possible problems for computing the minimum and when at least one term is not defined, we should define a specific definition for the resulting term. Floyd's algorithm provides an algoithmic process for obtaining a matrix of all shortest possible paths with respect to a given initial matrix representing links between nodes together with their distance. Our first attempt was based on the computation of a shortest path between to given nodes a and b. The resulting matrix is called R and a boolean variable FD tells us if the shortest path exists. By the way, this first attempt is not the strict Floyd's algorithm but it will use the same principle of computation for the resulting matrix R.

The first step defines the *context* of the problem and the context is validated by the RODIN platform[15]. We decide to design an algorithm which is computing the value of the shortest path between two given nodes but using the same principle than Floyd's algorithm.

Teacher's note: *The validation of the context SHORTESTPATH0 helps us to define carefully the function d. The translation of mathematocal properties is made easier by the notion of partial function. The expression $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i+1)+ D_i(i+1,b))$ hides possible underfinedness and generally the non-existence of an edge between two nodes is defined by an extra value like ∞. We have to compute the following value $\lambda i,j \in N.d(l \mapsto i \mapsto j)$ but the λ notation is not directly usable in the B notations. However, we are computing in fact the value of d for the triple $l \mapsto i \mapsto j$ because it seems to be simpler to state.*

4.2 Writing the function call

The first model provides the declaration of the procedure shortestpath. Variables D and FD are *call-by-reference* parameters, constants l, a, b, D are *call-by-value* parameters:

procedure shortestpath$(l, a, b, G;$ **var** $D, FD)$

precondition $G = d_0 \wedge\ FD = FALSE \wedge\ l > 0 \wedge\ a \in N \wedge\ b \in N$

postcondition $(FD = true \Rightarrow D = d(l, a, b))$

We apply the *Call as Event principle* and we have to define a new model called SHORTESTPATH1 , which is defining an event corresponding to the action of calling the procedure.

Teacher's note: *The event is considered as a function call; we can explain at this time that the event is triggered because the guard is true. It is not a precondition.*

The new model SHORTESTPATH1 is using definitions of the context SHORTESTPATH0 . The event FLOYDKO models the fact that the call of `floyd` is returning a value FALSE for FD: there is no path between a and b. The event FLOYDOK returns the value TRUE for FDand the value of the minimal path from a to b. The two events are also interpreted by a procedure which is called with respect to the existence of a path.

MACHINE SHORTESTPATH1
SEES SHORTESTPATH0

VARIABLES
D
FD

INVARIANTS
$inv1 : D \in N \times N \pfun \mathbb{N}$
$inv2 : FD \in BOOL$

EVENTS

INITIALISATION
BEGIN
$act1 : D :| \left(\begin{array}{l} D' \in N \times N \pfun \mathbb{N} \\ \land\ (\forall i, j \cdot 0 \mapsto i \mapsto j \in dom(d) \Rightarrow i \mapsto j \in dom(D') \land\ D'(i \mapsto j) = d(0 \mapsto i \mapsto j))) \end{array} \right)$
$act2 : FD := FALSE$
END

EVENT shortestpathOK
WHEN
$grd1 : l \mapsto a \mapsto b \in dom(d)$
THEN
$act1 : D(a \mapsto b) := d(l \mapsto a \mapsto b)$
$act2 : FD := TRUE$
END

EVENT shortestpathKO
WHEN
$grd1 : l \mapsto a \mapsto b \notin dom(d)$
THEN
$act1 : FD := FALSE$
END

END

Now, we have two events really non-deterministic, since they are defined using the constant d which should be computed in fact!. The solution is to refine the model SHORTESTPATH1 into a new model SHORTESTPATH2 which reduces non-determinism.

Teacher's note: *It is very important to explain the difference between a flexible [12] variable and a rigid variable. Rigid variable like d denotes values which are defined as mathematical static objects and flexible variables denotes a name which is assigned to a value depending on the current state.*

4.3 Refining the procedure call

The main idea is to unfold the calls or to refine the events to get a model which is closer to an algorithm. We introduce several new variables:

- D and FD are both variables of the models SHORTESTPATH1 and SHORTESTPATH2 .
- c ($inv1 : c \in C$) expresses the control flow and the possible values of c are in the set C ($axm15 : C = \{start, end, step1, step2, step3, finalstep\}$).
- $D1$, $D2$ and $D3$ are three variables storing the values required for computing the next value of D at a given step; the values may be undefined and the undefinedness is controlled by the three variables $FD1$, $FD2$ and $FD3$. Variables are typed according to the following part of the invariant:
  - $inv2 : D1 \in \mathbb{Z}$
  - $inv3 : D2 \in \mathbb{Z}$

- $inv4 : D3 \in \mathbb{Z}$
- $inv5 : FD1 \in BOOL$
- $inv6 : FD2 \in BOOL$
- $inv7 : FD3 \in BOOL$

We do not give more details for the invariant and we will give later the details of the invariant of the current model. First we give the different events of the model SHORTESTPATH2 .

The event

INITIALISATION

is simply setting the variables as follows: $act1 : D := D0$, $act2 : FD := FALSE$, $act3 : FD1 := FALSE$, $act4 : FD2 := FALSE$, $act5 : FD3 := FALSE$, $act6 : D1 :\in \mathbb{Z}$, $act7 : D2 :\in \mathbb{Z}$, $act8 : D3 :\in \mathbb{Z}$, $act10 : c := start$.

Since $d(0 \mapsto i \mapsto j)$ models the existence of an elementary path from i to j, $D0$ is defined by the following axioms:

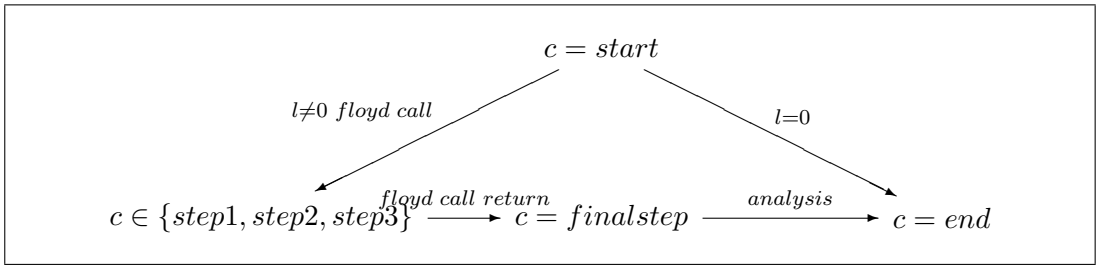
- $axm12 : D0 \in N \times N \nrightarrow \mathbb{N}$
- $axm13 : dom(D0) = \{i \mapsto j | 0 \mapsto i \mapsto j \in dom(d)\}$
- $axm14 : \forall i, j \cdot i \mapsto j \in dom(D0) \Rightarrow D0(i \mapsto j) = d(0 \mapsto i \mapsto j)$

Now, we can introduce refinement of existing events of SHORTESTPATH1 and new events which are not in the abstraction.

4.3.1 Refining events of SHORTESTPATH1

First, we give elements for competing the invariant; the typing informations can be completed as follows and they correspond to an analysis of the definition of d. We introduce a new variable c which is expressing the control state and whose possible values are given by the set C: $C = \{start, end, step1, step2, step3, finalstep\}$. We summarize the different steps for computing D.

Teacher's note: *Using a graphical notation helps to communicate the meaning of control assertions. The steps of the algorithm appear. Moreover, steps provide a guide for defining the invariant which is based on the construction of d.*



Teacher's note: *The invariant is based on the decomposition into steps and each step analyses the definition of values required for computing the minimum of $D1$ and $D2{+}D3$. The invariant should take into account th definedness of these values and the tool helps us to complete the invariant.*

The *analysis* step provides a decision depending on the values of $D1$, $D2$ and $D3$, if they are defined. The boolean expression $FD1 \wedge\ (FD2 \vee FD3)$ is the key for updating $D(a \mapsto b)$ and it is triggered , when the control is finalstep.

Teacher's note: *The expression $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i{+}1){+}D_i(i{+}1,b))$ should be carefully analysed and it allows us to derive specific conditions for structuring the algorithm.*

When the control is at start:

- when l is initially equal to 0, D and d are equal too; D is defined when d is defined and reciprocally:
  - $inv20 : c = start \wedge\ a \mapsto b \notin dom(D) \wedge\ l = 0 \Rightarrow 0 \mapsto a \mapsto b \notin dom(d)$
  - $$inv8 : \left(\begin{array}{l} \left(\begin{array}{l} c = start \\ \wedge\ a \mapsto b \in dom(D) \\ \wedge\ l = 0 \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} 0 \mapsto a \mapsto b \in dom(d) \\ \wedge\ D(a \mapsto b) = d(0 \mapsto a \mapsto b) \end{array} \right) \end{array} \right)$$
- when l is not equal to 0 and there is no path from a to b with intermadiate nodes whose numbers is smaller than $l - 1$, $a \mapsto b$ is not in D.

- $inv34 : \left(\begin{array}{l} \left(\begin{array}{l} c = start \\ \wedge\, l \neq 0 \\ \wedge\, l-1 \mapsto a \mapsto b \notin dom(d) \\ \wedge\, l-1 \mapsto l \mapsto b \notin dom(d) \end{array} \right) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array} \right)$
- $inv37 : \left(\begin{array}{l} \left(\begin{array}{l} c = start \\ \wedge\, l-1 \mapsto a \mapsto b \notin dom(d) \\ \wedge\, l-1 \mapsto a \mapsto l \notin dom(d) \end{array} \right) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array} \right)$

When the control is at end:

If the control is at end, the invariant enumerates the different cases for the resulting computation. The variable D should contain the values correspondin to l.

- $inv12 : \begin{array}{l} \left(\begin{array}{l} c = end \\ \wedge\, FD = TRUE \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} a \mapsto b \in dom(D) \\ \wedge\, l \mapsto a \mapsto b \in dom(d) \\ \wedge\, D(a \mapsto b) = d(l \mapsto a \mapsto b) \end{array} \right) \end{array}$
- $inv14 : c = end \wedge\ FD = FALSE \Rightarrow a \mapsto b \notin dom(D) \wedge\ l \mapsto a \mapsto b \notin dom(d)$
- $inv18 : c = end \wedge\ l \mapsto a \mapsto b \notin dom(d) \Rightarrow FD = FALSE$
- $inv19 : c = end \wedge\ a \mapsto b \notin dom(D) \Rightarrow FD = FALSE$
- $inv21 : c = end \wedge\ a \mapsto b \in dom(D) \Rightarrow FD = TRUE$

When the control is at finalstep:

The invariant states that the variables $FD1$, $FD2$ and $FD3$ are related to the definition of the expression $Min(D(a,b), D(a,l) + D(l,b))$. $Min(D(a,b), D(a,l) + D(l,b))$. is defined,if, end only, if $FD1 \wedge\ (FD2 \vee FD3)$. The invariant explores the different cases for the definition of D for the given pairs. Moreover, the values are stired in the variables $D1$, $D2$ and $D3$ when defined.

- $$\begin{array}{l} c = finalstep \wedge\ FD3 = TRUE \\ inv11 : \Rightarrow \\ l-1 \mapsto l \mapsto b \in dom(d) \wedge\ D3 = d(l-1 \mapsto l \mapsto b) \end{array}$$
- $$\begin{array}{l} c = finalstep \wedge\ FD1 = TRUE \\ inv15 : \Rightarrow \\ l-1 \mapsto a \mapsto b \in dom(d) \wedge\ D1 = d(l-1 \mapsto a \mapsto b) \end{array}$$
- $$\begin{array}{l} c = finalstep \wedge\ FD2 = TRUE \\ inv16 : \Rightarrow \\ l-1 \mapsto a \mapsto l \in dom(d) \wedge\ D2 = d(l-1 \mapsto a \mapsto l) \end{array}$$
- $$inv13 : \left(\begin{array}{l} \left(\begin{array}{l} c = finalstep \\ \wedge\ FD1 = FALSE \\ \wedge\ (FD2 = FALSE \vee FD3 = FALSE) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} l \mapsto a \mapsto b \notin dom(d) \\ \wedge\ a \mapsto b \notin dom(D) \end{array} \right) \end{array} \right)$$
- $inv24 : c = finalstep \wedge\ FD3 = FALSE \Rightarrow l-1 \mapsto l \mapsto b \notin dom(d)$
- $inv27 : c = finalstep \wedge\ FD2 = FALSE \Rightarrow l-1 \mapsto a \mapsto l \notin dom(d)$
- $inv29 : c = finalstep \wedge\ FD1 = FALSE \Rightarrow l-1 \mapsto a \mapsto b \notin dom(d)$

- $$inv38 : \left(\begin{array}{l} \left(\begin{array}{l} c = finalstep \\ \wedge\ FD1 = TRUE \\ \wedge\ (FD2 = FALSE \vee FD3 = FALSE) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} l \mapsto a \mapsto b \in dom(d) \\ \wedge\ d(l \mapsto a \mapsto b) = d(l-1 \mapsto a \mapsto b) \end{array} \right) \end{array} \right)$$

The diagram shows that `shortestpath` is made up of three steps.

$$\begin{array}{l} \qquad c = step1 \\ evaluation\ FD1,D1 \Big\downarrow \\ c = step2 \xrightarrow{evaluation\ FD2,D2} c = step3 \xrightarrow{evaluation\ FD3,D3} c = finalstep \end{array}$$

When the conrol is in $\{step1, step2, step3\}$:

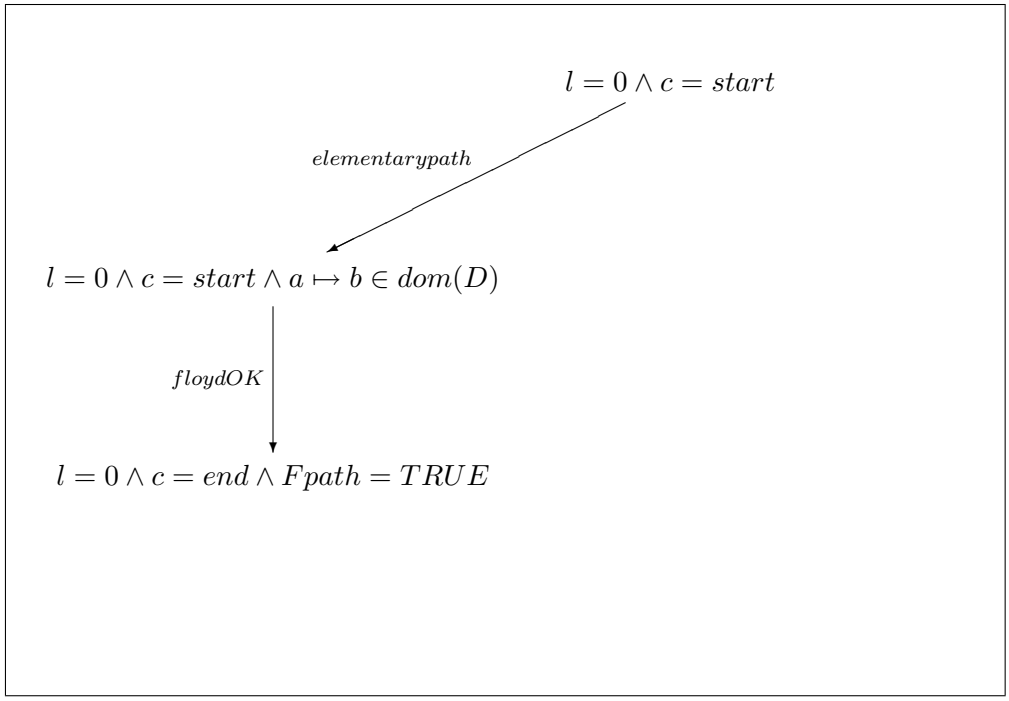
- When the control is in $\{step1, step2, step3\}$, since $inv28 : c \neq start \wedge\ c \neq end \Rightarrow l \neq 0$, l is not equal to 0.
- When the control is at $step1$, l is not equal to 0. There are two conditions for the undefinedness of D in relationship to d.
  - $$inv33 : \begin{array}{l} c = step1 \wedge\ l-1 \mapsto a \mapsto b \notin dom(d) \wedge\ l-1 \mapsto l \mapsto b \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array}$$
  - $$inv36 : \begin{array}{l} c = step1 \wedge\ l-1 \mapsto a \mapsto b \notin dom(d) \wedge\ l-1 \mapsto a \mapsto l \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array}$$
- When the control is in $step2$, either the evaluation of $D1$ is successful or not.
  - $inv9 : c = step2 \wedge\ FD1 = TRUE \Rightarrow l-1 \mapsto a \mapsto b \in dom(d) \wedge\ D1 = d(l-1 \mapsto a \mapsto b)$

- $inv22 : c = step2 \wedge\ FD1 = FALSE \Rightarrow l - 1 \mapsto a \mapsto b \notin dom(d)$
- $inv32 : c = step2 \wedge\ FD1 = FALSE \wedge\ l - 1 \mapsto l \mapsto b \notin dom(d) \Rightarrow a \mapsto b \notin dom(D)$
- $inv35 : c = step2 \wedge\ l - 1 \mapsto a \mapsto l \notin dom(d) \wedge\ FD1 = FALSE \Rightarrow a \mapsto b \notin dom(D)$

- When the control is in $step3$, either the evaluation of $D2$ is successful or not.
  - $inv10 : c = step3 \wedge\ FD2 = TRUE \Rightarrow l - 1 \mapsto a \mapsto l \in dom(d) \wedge\ D2 = d(l - 1 \mapsto a \mapsto l)$
  - $inv17 : c = step3 \wedge\ FD1 = TRUE \Rightarrow l - 1 \mapsto a \mapsto b \in dom(d) \wedge\ D1 = d(l - 1 \mapsto a \mapsto b)$
  - $inv23 : c = step3 \wedge\ FD2 = FALSE \Rightarrow l - 1 \mapsto a \mapsto l \notin dom(d)$
  - $inv25 : c = step3 \wedge\ FD1 = FALSE \Rightarrow l - 1 \mapsto a \mapsto b \notin dom(d)$
  - $inv26 : c \neq finalstep \wedge\ c \neq end \wedge\ 0 \mapsto a \mapsto b \notin dom(d) \Rightarrow a \mapsto b \notin dom(D)$
  - $inv30 : c = step3 \wedge\ FD1 = FALSE \wedge\ FD2 = FALSE \Rightarrow a \mapsto b \notin dom(D)$
  - $inv31 : c = step3 \wedge\ FD1 = FALSE \wedge\ l - 1 \mapsto l \mapsto b \notin dom(d) \Rightarrow a \mapsto b \notin dom(D)$

Refining shortestpathOK

Now, we define each transition between the different steps according to the invariant. We consider severall possible cases depending on l and other conditions. When the value of l is 0 and when D is defined for the pair $a \mapsto b$, it means that there is a path between a and b without any intermediate node. It is the basic case and one returns the value TRUE for FD. The control is set to end, since the procedure is completed:



EVENT shortestpathOK
REFINES shortestpathOK
WHEN
$grd2 : l = 0$
$grd1 : a \mapsto b \in dom(D)$
$grd3 : c = start$
THEN
$act2 : FD := TRUE$
$act3 : c := end$
END

When the control expresses the accessibility of the last control point ($c = finalstep$) and when the three values $D1$, $D2$ and $D3$ are defined and satisfy the condition $D1 \leq D2 + D3$, we can update D in $a \mapsto b$ by $D1$. In fact, the value is not modified. The control is set to the final control point called end. There is a path and FD is set to TRUE.

EVENT shortestpathcallOKmin
REFINES shortestpathOK

WHEN
$grd1 : FD1 = TRUE \wedge\ FD2 = TRUE \wedge\ FD3 = TRUE$
$grd2 : D1 \leq D2 + D3$
$grd3 : c = finalstep$
THEN
$act1 : D(a \mapsto b) := D1$
$act2 : FD := TRUE$
$act3 : c := end$
END

The next case is stating that there is a new path from a to b, which is shortest than the current one ($grd3 : D1 > D2 + D3$) and we should update D by the new value $D2 + D3$.

EVENT shortestpathcallOKmax
REFINES shortestpathOK
WHEN
$grd1 : FD1 = TRUE \wedge\ FD2 = TRUE \wedge\ FD3 = TRUE$
$grd2 : c = finalstep$
$grd3 : D1 > D2 + D3$
THEN
$act1 : D(a \mapsto b) := D2 + D3$
$act2 : c := end$
$act3 : FD := TRUE$
END

The next possible case is that the value $D1$ is not defined; it means that there is not yet a path from a to b and we have discovered that there is a node which can be reached from a and which can reach b. Hence, the variable D is defined in $a \mapsto b$ by the value $D2 + D2$.

EVENT shortestpathFD2FD3
REFINES shortestpathOK
WHEN
$grd1 : c = finalstep$
$grd2 : FD1 = FALSE \wedge\ FD2 = TRUE \wedge\ FD3 = TRUE$
THEN
$act1 : D(a \mapsto b) := D2 + D3$
$act2 : FD := TRUE$
$act3 : c := end$
END

Finally, when either $D2$ or $D3$ is not defined, the value of D is not modified and remains equal to $D1$.

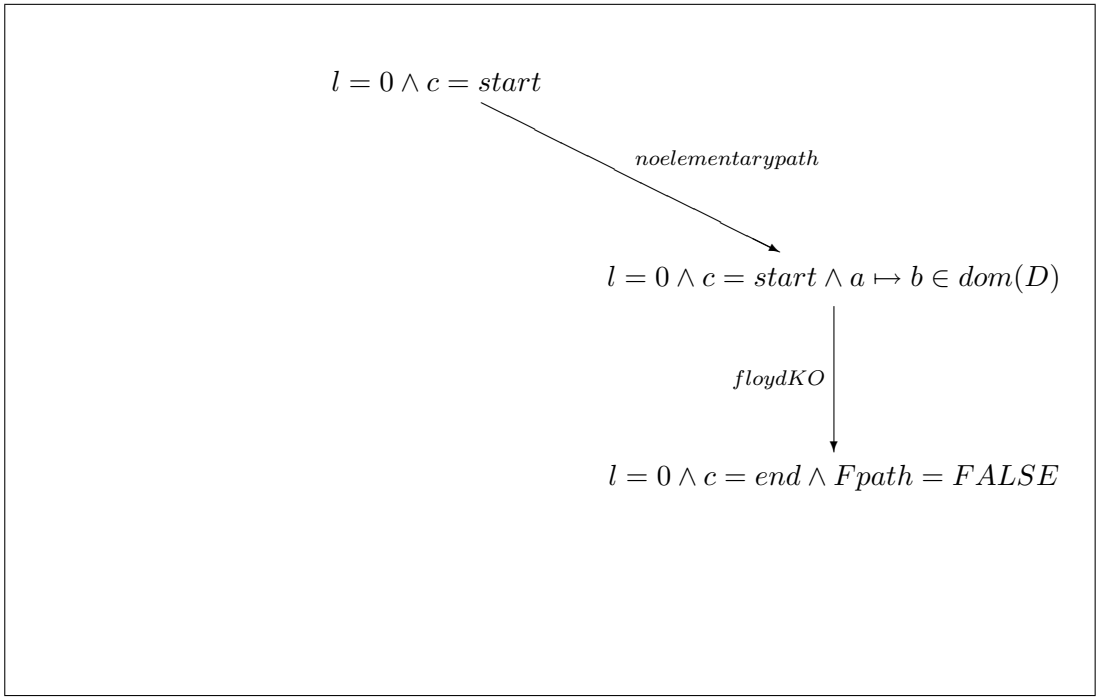
EVENT shortestpathFD1
REFINES shortestpathOK
WHEN
$grd1 : c = finalstep$
$grd2 : FD1 = TRUE \wedge\ (FD1 = FALSE \vee FD2 = FALSE)$
THEN
$act1 : D(a \mapsto b) := D1$
$act2 : c := end$
$act3 : FD := TRUE$
END

The refinement of abstract events should be completed by events which compute the values $D1$, $D2$ and $D3$.

Refining shortestpathKO

We consider severall possible cases depending on l and other conditions.

When the value of l is 0 and when D is not defined for the pair $a \mapsto b$, it means that there is no elementary path between a and b. It is the basic case and one returns the value FALSE for FD. The control is set to end, since the procedure is completed:



EVENT shortestpathKO
REFINES shortestpathKO
WHEN
$grd2 : l = 0$
$grd1 : a \mapsto b \notin dom(D)$
$grd3 : c = start$
THEN
$act1 : FD := FALSE$
$act2 : c := end$
END

When the value of l is not 0 and when $D1$ is not defined and either $D2$ is not defined, or $D3$ is not defined, for the pair $a \mapsto b$, it means that there is no path between a and b. One returns the value FALSE for FD. The control is set to end, since the procedure is completed:

EVENT shortestpathKOelse
REFINES shortestpathKO
WHEN
$grd1 : c = finalstep$
$grd2 : FD1 = FALSE \wedge (FD2 = FALSE \vee FD2 = FALSE)$
THEN
$act1 : c := end$
$act2 : FD := FALSE$
END

4.3.2 Introducing new events in SHORTESTPATH2

The first new event models the calling step of the procedure `floyd` and it transfers the control to the control point $step1$.

EVENT shortestpathcallone
WHEN
$grd1 : l > 0$
$grd2 : c = start$
THEN
$act1 : c := step1$
END

Now, we consider the three steps for computing $D1$, $D2$ and $D3$.

Calling the procedure `floyd` for evaluating $D1$ and $FD1$

The event $shortestpathcalltwook$ simulates the procedure for computing $D1$, which is $d(l-1 \mapsto a \mapsto b)$ and which is successfully computed, since $FD1$ is TRUE. The event $shortestpathcalltwoko$ simulates the procedure for computing $D1$, which is $d(l-1 \mapsto a \mapsto b)$ and which is unsuccessfully computed, since $FD1$ is FALSE.

EVENT
floydcalltwook
WHEN
$grd1 : c = step1$
$grd2 : l-1 \mapsto a \mapsto b \in dom(d)$
THEN
$act1 : D1 := d(l-1 \mapsto a \mapsto b)$
$act2 : FD1 := TRUE$
$act3 : c := step2$
END

EVENT shortestpathcalltwoko
WHEN
$grd1 : l-1 \mapsto a \mapsto b \notin dom(d)$
$grd2 : c = step1$
THEN
$act1 : FD1 := FALSE$
$act2 : c := step2$
END

Calling the procedure `floyd` for evaluating $D2$ and $FD2$

The event $shortestpathcallthreeok$ simulates the procedure for computing $D2$, which is $d(l-1 \mapsto a \mapsto l)$ and which is successfully computed, since $FD2$ is TRUE. The event $shortestpathcallthreeko$ simulates the procedure for computing $D2$, which is $d(l-1 \mapsto a \mapsto l)$ and which is unsuccessfully computed, since $FD2$ is FALSE.

EVENT shortestpathcallthreeok
WHEN
$grd1 : c = step2$
$grd2 : l-1 \mapsto a \mapsto l \in dom(d)$
THEN
$act1 : D2 := d(l-1 \mapsto a \mapsto l)$
$act2 : FD2 := TRUE$
$act3 : c := step3$
END

EVENT shortestpathcallthreeko
WHEN
$grd1 : c = step2$
$grd2 : l-1 \mapsto a \mapsto l \notin dom(d)$
THEN
$act1 : c := step3$
$act2 : FD2 := FALSE$
END

Calling the procedure `floyd` for evaluating $D3$ and $FD3$

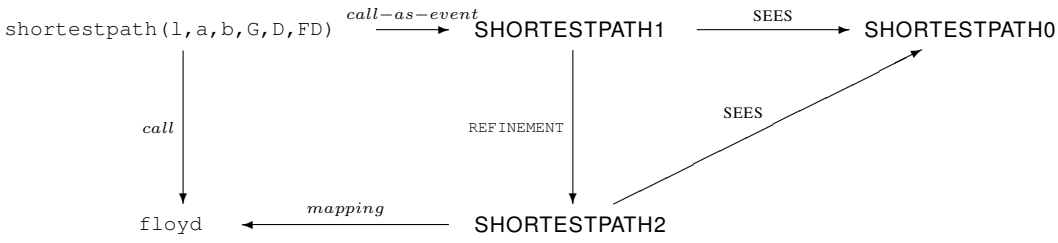
The event $shortestpathcallfourok$ simulates the procedure for computing $D3$, which is $d(l-1 \mapsto l \mapsto b)$ and which is successfully computed, since $FD3$ is TRUE. The event $shortestpathcallfourko$ simulates the procedure for computing $D3$, which is $d(l-1 \mapsto l \mapsto b)$ and which is unsuccessfully computed, since $FD3$ is FALSE.

EVENT shortestpathcallfourok
WHEN
$grd1 : c = step3$
$grd2 : l - 1 \mapsto l \mapsto b \in dom(d)$
THEN
$act1 : c := finalstep$
$act2 : D3 := d(l - 1 \mapsto l \mapsto b)$
$act3 : FD3 := TRUE$
END

EVENT shortestpathcallfourko
WHEN
$grd1 : c = step3$
$grd2 : l - 1 \mapsto l \mapsto b \notin dom(d)$
THEN
$act1 : FD3 := FALSE$
$act2 : c := finalstep$
END

4.4 Producing the shortestpath procedure

The shortestpath procedure can be derived from the list of events of the model SHORTESTPATH2 and we structure events into conventional programming structures like `while` or `if` statements. J.-R. Abrial[3] has proposed several rules for producing algorithmic statements. The next diagram gives the complete description of the process we have followed:



The procedure header is `shortestpath(l,a,b,G,D,FD)` and the text of the procedure is given by the algorithms 1 and 2.

Algorithm 1: Algorithm Version 1

precondition : $l \in 1..n \wedge$
postcondition : D, FD
local variables: $FD1, FD2, FD3 \in BOOL$

$FD := FALSE;$
$FD1 := FALSE;$
$FD2 := FALSE;$
$FD3 := FALSE;$
if $l = 0$ **then**
  if $(a, b) \in dom(D)$ **then**
    $FD := TRUE;$
    $R := D[a, b];$
  else
    $FD := FALSE;$
else
  $floyd(l - 1, a, b, D1, FD1); floyd(l - 1, a, l, D2, FD2); floyd(l - 1, l, b, D3, FD3);$
  case $FD1 \wedge FD2 \wedge FD3$
    if $D1 < D2 + D3$ **then**
      $R := D1;$
    else
      $R := D2 + D3;$
    ;
    $FD := TRUE;$
  ;
  case $FD1 \wedge (\neg FD2 \vee \neg FD3)$
    $R := D1;$
    $FD := TRUE;$
  ;
  case $\neg FD1 \wedge (FD2 \wedge FD3)$
    $R := D2 + D3;$
    $FD := TRUE;$
  ;
  case $\neg FD1 \wedge (\neg FD2 \vee \neg FD3)$
    $FD := FALSE;$
  ;
;

The two next frames are containing C codes produced for the two algorithms 4.4 and 4.4; we have produced the C codes by hand and we have forgotten that C arrays starts by 0 and it means that our initial calls were wrongly written. It is clear that we need a way to produce codes in a mechanized way. Moreover, there are some conditions to check and

some interactions to manage with the user to help in choices.

Teacher's note: *It is the time to recall that we are planning to use a real programming language and that we should represent abstract objects by concrete objects. It would be better to add informations on the integers of computer scientists and it is easy to add the constraint.*

```
/*  N = 1..n-1 */
void shortestpath (int l, int a, int b,int g[][n], int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;
  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       shortestpath(l-1,a,b,g,&D1,&FD1); shortestpath(l-1,a,l,g,&D2,&FD2);
       shortestpath(l-1,l,b,g,&D3,&FD3);
      if (FD1 == 1 && (FD2==1 &&  FD3==1))
        {  if (D1 < D2+D3)
             {*D= D1;}
           else
             {*D=D2+D3;};
           *FD = 1;
        }
      else if (FD1==1 && ( FD2==0 ||  FD3==0))
        {*D= D1;*FD = 1;}
      else if (FD1==0 &&  ( FD2 == 1 && FD3==1)) {*D=D2+D3; *FD=1;}
      else /* (FD1==0 && ( FD2==0 || FD3==0)) */ { *FD = 0;}
    }
}
```

```
/*  N = 1..n-1 */
void shortestpath (int l, int a, int b,int g[][n], int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;

  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       shortestpath(l-1,a,b,g,&D1,&FD1);
      if (FD1 == 1) {
         shortestpath(l-1,a,l,g,&D2,&FD2);
         if (FD2==1) {
           shortestpath(l-1,l,b,g,&D3,&FD3);
           if (FD3==1) {
           if (D1 < D2+D3)
             {*D= D1;}
           else
             {*D=D2+D3;};
           *FD = 1;}
           else
             {*D=D1;*FD=1;}}
         else
           {*D=D1;*FD=1;}}
      else
        {
          if  ( FD2 == 1 && FD3==1) {*D=D2+D3; *FD=1;}
      else
        {*FD=0;};}
    }}
```

The complete development has a cost related to proof obligations. The refinement generates 493 proof obligations and 328 proof obligations were automaically discharged. 165 proof obligations were manually discharged with minor interactions.

Algorithm 2: Algorithm Version 2

precondition : $l \in 1..n \wedge a, b \in \mathbb{N} \wedge G \in N \times N \nrightarrow N$
postcondition : D, FD
local variables: $FD1, FD2, FD3 \in BOOL$

$FD := FALSE$;
$FD1 := FALSE$;
$FD2 := FALSE$;
$FD3 := FALSE$;
if $l = 0$ **then**
  if $(a, b) \in dom(D)$ **then**
    $FD := TRUE$;
    $R := D[a, b]$;
  else
    $FD := FALSE$;
else
  $shortestpath(l - 1, a, b, D1, FD1)$;
  if *FD1* **then**
    $shortestpath(l - 1, a, l, D2, FD2)$;
    if *FD2* **then**
      $shortestpath(l - 1, l, b, D3, FD3)$;
      if *FD3* **then**
        if $D1 < D2 + D3$ **then**
          $R := D1$;
        else
          $R := D2 + D3$;
        ;
        $FD := TRUE$;
      else
        $R := D1$;
        $FD := TRUE$;
      ;
    else
      $R := D1$;
      $FD := TRUE$;
    ;
  else
    if $FD2 \wedge FD3$ **then**
      $R := D2 + D3$;
      $FD := TRUE$;
    else
      $FD := FALSE$;
    ;
;

model	Total	Auto	Manual	Reviewed	Undischarged
SHORTESTPATH0	8	8	0	0	0
SHORTESTPATH1	5	4	1	0	0
SHORTESTPATH2	493	328	165	0	0
Global	506	340	166	0	0

Teacher's note: *Proof obligations are not very difficult to discharge;there were based on the properties of d and it was boring to click the tool for discharging mechanichally them. Efforts were made on the definition of d.*

Now, it turns that our goal was to get Floyd's algorithm and we have an algorithm for computing the existence or the non existence of a shortest path between two nodes. The next section address the question.

5 Floyd's algorithm

We can use the developed algorithm to produce a result equivalent to Floyd's execution. In fact, we apply our algorithm on each pair of possible nodes and we store it in a matrix. The algorithm 5 describes the real algorithm which can be found in any lecture notes.

Algorithm 3: Floyd's Algorithm Wikipedia

precondition : $l \in 1..n \wedge matrix \in N \times N \rightarrow\!\!\!\!\!\!\rightarrow N$
postcondition : $matrix \in N \times N \rightarrow\!\!\!\!\!\!\rightarrow N \wedge$
local variables: $FD1, FD2, FD3 \in BOOL$

foreach $k = 1; k <= n; k++$ **do**
  foreach $i = 1; i <= n; i++$ **do**
    foreach $j = 1; j <= n; j++$ **do**
      if $matrix[i][j] > (matrix[i][k] + matrix[k][j])$ **then**
        $matrix[i,j] = matrix[i][k] + matrix[k][j]$

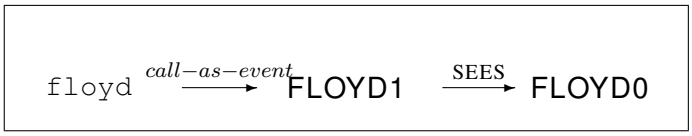
Now, we are considering the problem of derivation of this solution. In fact, the development starts from the same context. Two new constants are defined namely DF and Daf. Df is the final value of the matrix D correponding to d for the value l. C is simpler and is defined as follows: $axm15 : C = \{start, end, call, finalstep\}$.

New axioms define new constants:

- $axm39 : Df \in N \times N \rightarrow\!\!\!\!\!\!\rightarrow N$
- $axm40 : dom(Df) = \{u \mapsto v | l \mapsto u \mapsto v \in dom(d)\}$

- $axm41 : \forall u, v \cdot u \mapsto v \in dom(Df) \Rightarrow Df(u \mapsto v) = d(l \mapsto u \mapsto v)$
- $axm42 : Daf \in N \times N \nrightarrow N$
- $axm43 : l \neq 0 \Rightarrow dom(Daf) = \{u \mapsto v | l - 1 \mapsto u \mapsto v \in dom(d)\}$
- $axm44 : l \neq 0 \Rightarrow (\forall u, v \cdot u \mapsto v \in dom(Daf) \Rightarrow Daf(u \mapsto v) = d(l - 1 \mapsto u \mapsto v))$
- $axm22 : l = 0 \Rightarrow Df = D0$
- $axm23 : l \neq 0 \Rightarrow D0 \subseteq Daf$
- $axm24 : l \neq 0 \Rightarrow Daf \subseteq Df$
- $axm25 : l \neq 0 \Rightarrow (\forall u, v \cdot u \mapsto v \in dom(Daf) \Rightarrow Daf(u \mapsto v) = d(l - 1 \mapsto u \mapsto v))$
- $axm26 : \forall u, v \cdot u \mapsto v \in dom(Df) \Rightarrow Df(u \mapsto v) = d(l \mapsto u \mapsto v)$
- $axm27 : \forall u, v \cdot u \mapsto v \in dom(D0) \Rightarrow D0(u \mapsto v) = d(0 \mapsto u \mapsto v)$
- $$axm28 : \begin{array}{l} \Big(l \neq 0 \Big) \\ \Rightarrow \\ \forall u, v, w \cdot \left(\begin{array}{l} w \mapsto v \in dom(Df) \\ \wedge\ w \mapsto u \in dom(Daf) \\ \wedge\ u \mapsto v \in dom(Daf) \\ \wedge\ Daf(w \mapsto v) > Daf(w \mapsto u) + Daf(u \mapsto v) \end{array} \right) \end{array}$$

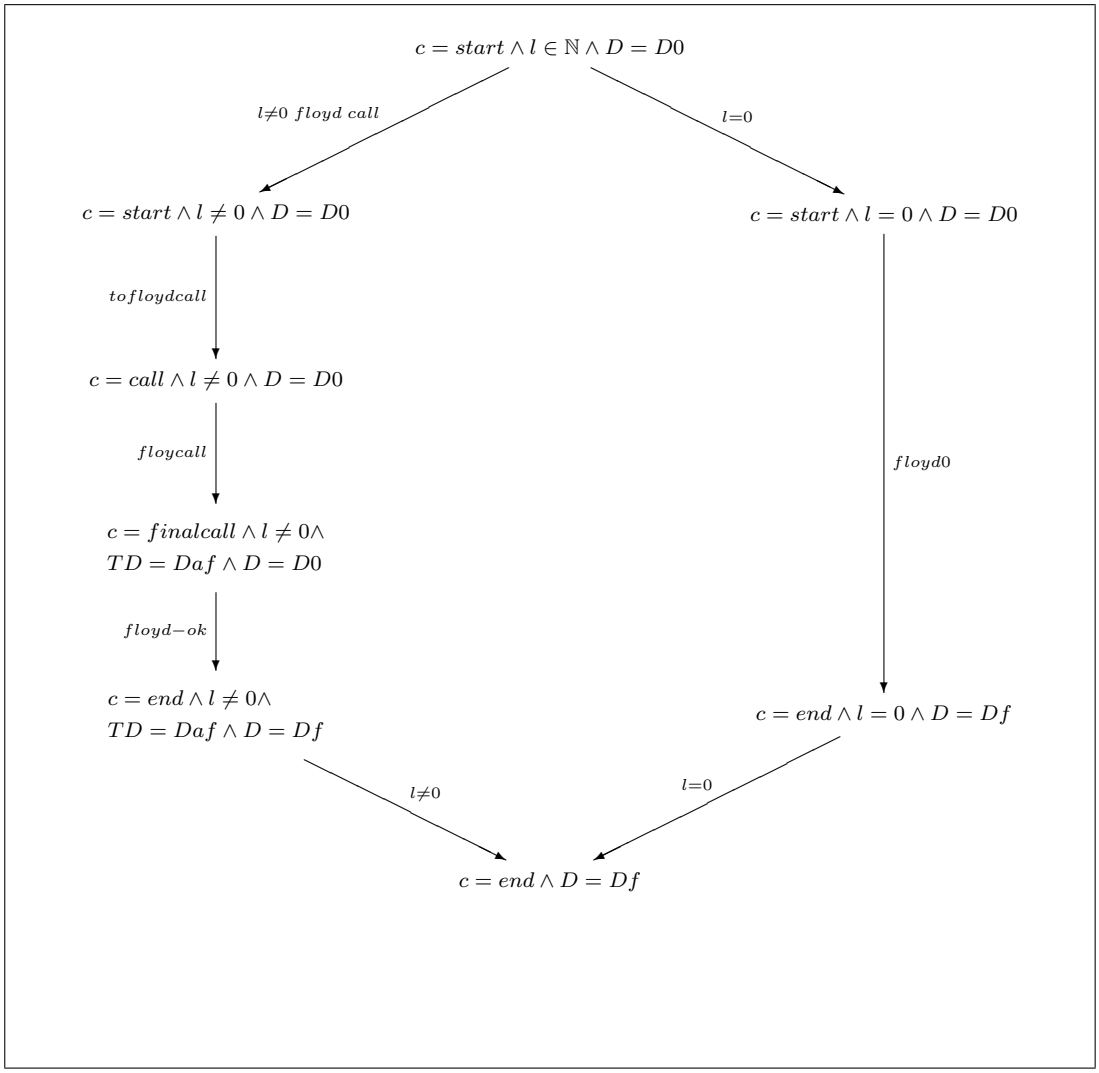
The new model FLOYD1 assigns the value Df to D. The new relationship between models and call is given by the next diagram:



The problem is to refine the model FLOYD1 to get a list of events which lead to an algorithm. The two constants Df and Daf are used to state the final step and the intermediate step:

- Daf is the result of the call of the under construction algorithm for $l-1$
- Df is the final value which is computed from Daf.

We obtain the following diagram for expressing events corresponding to Floyd's algorithm:



The new model has three variables: c, D, TD.

- $\boxed{inv6 : TD \in N \times N \nrightarrow N}$
- $\boxed{inv1 : c \in C}$
- $\boxed{inv2 : c = start \Rightarrow TD = D0 \wedge\ D = D0}$

- $\boxed{inv3 : c = end \Rightarrow D = Df}$
- $\boxed{inv4 : c = call \Rightarrow TD = D0 \wedge\ l \neq 0}$
- $\boxed{inv5 : c = finalstep \Rightarrow TD = Daf \wedge\ l \neq 0}$

Initial conditions over variables are defined by $act1 : D := D0$, $act2 : c := start$, $act3 : TD := D0$. Events are very simple to write from the diagram:

EVENT floyd-ok
REFINES floyd
WHEN
$grd1 : c = finalstep$
THEN

$$
act1 : D, c : |
\left(
\begin{array}{l}
\Big(D' \in N \times N \nrightarrow N \wedge\ c' = end \Big) \\
\wedge \left(\forall w, v \cdot \left(\begin{array}{l} \left(\begin{array}{l} w \in N \wedge\ v \in N \\ \wedge\ w \mapsto v \in dom(TD) \\ \wedge\ w \mapsto l \in dom(TD) \\ \wedge\ l \mapsto v \in dom(TD) \\ \wedge\ TD(w \mapsto v) > TD(w \mapsto l) + TD(l \mapsto v) \end{array} \right) \\ \Rightarrow \\ D'(w \mapsto v) = TD(w \mapsto l) + TD(l \mapsto v) \end{array} \right) \right) \\
\wedge \left(\forall w, v \cdot \left(\begin{array}{l} \left(\begin{array}{l} w \mapsto v \in dom(TD) \\ \wedge\ w \mapsto l \in dom(TD) \\ \wedge\ l \mapsto v \in dom(TD) \\ \wedge\ TD(w \mapsto v) \leq TD(w \mapsto l) + TD(l \mapsto v) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} w \mapsto v \in dom(D') \\ \wedge\ D'(w \mapsto v) = TD(w \mapsto v) \end{array} \right) \end{array} \right) \right) \\
\wedge \left(\forall u, v \cdot \left(\begin{array}{l} \left(\begin{array}{l} u \mapsto v \in dom(TD) \\ \wedge\ (u \mapsto l \notin dom(TD) \vee l \mapsto v \notin dom(TD)) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} u \mapsto v \in dom(D') \\ \wedge\ D'(u \mapsto v) = TD(u \mapsto v) \end{array} \right) \end{array} \right) \right) \\
\wedge \left(\forall u, v \cdot \left(\begin{array}{l} \left(\begin{array}{l} u \mapsto v \notin dom(TD) \\ \wedge\ (u \mapsto l \in dom(TD) \\ \wedge\ l \mapsto v \in dom(TD)) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} u \mapsto v \in dom(D') \\ \wedge\ D'(u \mapsto v) = TD(u \mapsto l) + TD(l \mapsto v) \end{array} \right) \end{array} \right) \right) \\
\wedge \left(\forall u, v \cdot \left(\begin{array}{l} \left(\begin{array}{l} u \mapsto v \notin dom(TD) \\ \wedge\ (u \mapsto l \notin dom(TD) \vee l \mapsto v \notin dom(TD)) \end{array} \right) \\ \Rightarrow \\ u \mapsto v \notin dom(D') \end{array} \right) \right)
\end{array}
\right)
$$

END

The event
EVENT floyd-ok
uses a structure of Event B, which is assigning a value to variables and values are in a set. The set can be either empty, a singleton or a general set. In our case, the statement defines only one possible singleton and then the statement is clearly deterministic. However, we can subtitute the event by a call of a new procedure and we should starts a new development in another development using the same principle. We get the nestesd loops.

The two algorithms 5 and 5 are produced from the set of events. The recursive version is simply derived using the control points. The second algorithm is the iterative version which is produced by applying the classical transformations over recursive algorithms. The function nld is derived from an independant development by applying the same pattern.

EVENT floyd0
REFINES floyd
WHEN
$grd1 : c = start \wedge\ l = 0$
THEN
$act2 : c := end$
END

EVENT tofloydcall
WHEN
$grd1 : c = start \wedge\ l > 0$
THEN
$act1 : c := call$
END

EVENT floydcall
WHEN
$grd1 : c = call$
THEN
$act1 : c := finalstep$
$act2 : TD := Daf$
END

Algorithm 4: Recursive algorithm floyd

precondition : $l \in 1..n \wedge G$
postcondition : D
local variables: TD

$TD := D0$;
if $l \neq 0$ **then**
  $floyd(l-1, G, TD)$;
  $D := nld(TD)$;
else
  $D := TD$;
;

Algorithm 5: Non-recursive algorithm floyd

precondition : $l \in 1..n \wedge G$
postcondition : D
local variables: TD

$TD := D0$;
$l := 0$;
while $l \neq 0$ **do**
  $TD := d(TD)$;
;
$D := TD$;

6 Concluding Remarks and Perspectives

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

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

(i) Developing a first abstract one-shot model using pre/post-condition. It provides the declarations part of the procedure (method) related to the one-shot model. The basic structure to express is the function d which the key of the problem. Constants of the model are defined as *call-by-value* parameters and variable of the model are *call-by-reference* parameters, The context SHORTESTPATH0 is clearly reusable and we

have reused it for the effective algorithm of Floyd.

(ii) Refining the abstract model to obtain the body of procedure. New variables are defined as *local variables*. The refinement introduces control states which provide a way to structure the body of the procedure. We have clearly the first control point namely *start* and the last control point namely *end*. The diagram helps to decompose the procedure into steps of the call and a special control point called *call* is introduced. The main question is to obtain a deterministic transition system in the new refinement model.

(iii) If there are still remaining non-deterministic events, we can eliminate the non-deterministic events by developing each non-deterministic event in a specific B development starting by the statement of a new problem expressed by the non-deterministic event itself. In fact, it is what is done with the last version of Floyd's algorithm and the event computing D' from TD is clearly refined to get two nested loops.

(iv) Proof obligations are relatively easy to check because the invariant is written by a list of properties of d according to d. Evene if the number of *manual* proof obligations is high, it was very easy to discharge them using the prover and to reuse former intercative ones.

(v) The translation of Event B model into a C program was carried out by hand and we did a mistake. We forgot that C arrays are starting the index by 0 and it leads to a bad call. We should mechanize this step to avoid this mistake.

Now, if we have to teach concepts, it is easier to teach how to write concepts and definitions using notations provided by Event B (see for instance the table1). You will get a way to check definitions and the type checker is sometimes cruel. We can discuss on many questions using this methodology: coding of numbers, preconditions, postconditions, invariant, assertions, proofs, ldots and questions can lead to replies which are pertinent because of the proof tool.

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

References

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

[2] J.-R. Abrial. B$^{\#}$: Toward a synthesis between z and b. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer, June 2003.

[3] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME 2003*, pages 51–74, 2003.

[4] J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOL 2003*, pages 1–24, 2003.

[5] R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[6] Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages*. EATCS Textbook in Computer Science. Springer, 2007.

[7] Dominique Cansell and Dominique Méry. *The event-B Modelling Method: Concepts and Case Studies*, pages 33–140. Springer, 2007. See [6].

[8] Dominique Cansell and Dominique Méry. Proved-patterns-based development for structured programs. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *CSR*, volume 4649 of *Lecture Notes in Computer Science*, pages 104–114. Springer, 2007.

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

[10] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[12] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[13] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, October 2006.

[14] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[15] project RODIN. Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/, 2004. 2004–2007.

First balance sheet of a formal approach in the teaching of Data Structures

Marc Guyomard[1,2]

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

P. Alain, A. Hadjali, H. Jaudoin[3]

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

G. Smits[3]

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

Abstract

A formal method, EB, inspired by B method, has been used for the first time for teaching "data structures". Starting on one hand from a set oriented specification of an abstract type and on the other from the specification of an implementation, method EB favours the *calculation* of the representation of concrete operations. We present the principle of this approach as well as several examples. The context of its application in a data structures teaching module is then detailed. A first balance sheet is drawn up.

Keywords: formal methods, teaching data structures, specification, refinement, homomorphism, B method, functional methods, program derivation, induction, sets, stacks, queues, priority queues, flexible arrays.

1 Introduction

Considered recently as an autonomic discipline, and taught as such [2], most often in the last year of the course in computer science, formal methods, and notably B [1], influence more and more the teaching of other modules in computer science. This shows itself either by the use of a formal method as cross support for the specification [16], or as a methodology support for a more rigorous approach to traditional modules [15].

[1] Thanks to Henry Hicks

[2] Email: Marc.Guyomard@univ-rennes1.fr

[3] Email: {Pierre.Alain,Helene.Jaudoin,Allel.Hadjali,Gregory.Smits}@univ-rennes1.fr

In this article we relate an experiment of this type in which a formal approach, EB, inspired in particular from B method, has been used in teaching "data structures and algorithms". This experiment was carried out at Enssat, a school for engineers housed in Lannion, France, which is part of the University of Rennes 1.

The second section presents the principles of the approach using a deliberately simple example. Section 3 describes the module and its context: objectives, duration, population, content, etc. Section 4 draws the first conclusions from the experiment, emphasising the point of view of the teacher. That of the students, still being analysed, will be developed during the oral presentation. Finally section 5 contains our conclusions.

2 EB: a tutorial

2.1 The principle of the approach

The idea of formally stating the specifications before developing from them an algorithmic solution has not been the subject for debate for a long time. Its translation in the domain of data structures or of data types has itself given rise to numerous works. Those which have led to algebraic data types are worth citing [13]. The same is true for the step proposed by T. Hoare in [17] which is presented as a method of verification based on the proof of the correctness and of the completeness of the concrete model compared with the simulated one. Attempts to bring the two steps together have been carried out [5].

If none of these approaches has become an accepted standard it is in our opinion because at least one of the two following criteria has not been met:

- The language used for specifying the data must be at a "reasonable level". Neither too high (as in algebraic abstract types where the semantic gap to the objective is a problem), nor too low (as in predicate calculus which, in particular, leads to verbose wordings).
- The process of refinement (or of transformation) must be guided to be sufficiently constructive.

The EB approach defended here results from the worry of satisfying these two criteria in searching for, firstly from the B method [1] a satisfactory specification language for data and secondly from the Dutch school [19,22,8], a well identified succession of steps of refinement.

More precisely, the EB approach is based on an abstract specification and one refinement step. The abstract specification defines the support (the domain of possible values) as well as the operations, each of them in a functional way. The refinement consists of:

(i) Formally specifying the concrete support.
(ii) Formally specifying an "abstraction function" the source of which is the concrete support and the destination the abstract support. This abstraction function specifies which abstract value corresponds to which given concrete value.
(iii) Formally specifying each operation in the form of an equation which expresses the relations which maintain the concrete operation, the abstract operation

and the abstraction function.

(iv) Calculate the representation of each concrete operation.

Out of this development we have available a functional [4] representation of each operation. It is then possible (and generally easy) to obtain a more classical version of type "modification *in situ*" (MIS).

2.2 Specification of abstract type NATABST

The example which is the linking thread in this section is that of natural numbers, on which we allow two operations: addition (*plus*) and the relational operation $<$ (inf). It is important to remember that the operations are always *functions*.

In a first stage we specify the type NATABST in question. Figure 1 presents this specification. The heading gives the type name NATABST, the name of the support (*natAbst*, which represents all the values which can take an entity of type NATABST), the list of internal operations (that is the list of operations which produce a value of type under consideration) limited here to *plus*, and finally the list of external operations (reduced here to inf). The heading **support** gives a definition of the support set. Here we affirm that "to say that n is a natural numbers is the same as saying that n belongs to the set *natAbst*". The heading **operations** re-lists the operations in the heading and gives their name, those of their parameters and the category of the represented function (thus *plus* is a total surjective function from $natAbst \times natAbst$ to $natAbst$: $\twoheadrightarrow$), the *precondition,* which includes the typing of the arguments (*plus* demands that the object values of the operation both belong to the support *natAbst*) and finally the definition properly said, expressed by a "functional equation" of which the right hand side is an expression of the same type as the range of the function.

abstractType NATABST$= (natAbst, (plus), (inf))$
uses
$bool, \mathbb{N}$
support
$n \in \mathbb{N} \quad \Leftrightarrow \quad n \in natAbst$
operations
1) **name:** $plus(x, y) \twoheadrightarrow natAbst$
pre: $x, y \in natAbst \times natAbst$
spec: $plus(x, y) \quad = \quad x + y$

2) **name:** $inf(x, y) \twoheadrightarrow bool$
pre: $x, y \in natAbst \times natAbst$
spec: $inf(x, y) \quad = \quad (x < y)$
end

Figure 1: Specification of the abstract type NATABST

2.3 Specification of concrete type NATP

Once the specification has been created, we can envisage one (or several) implementation(s). This implementation cannot always represent faithfully the abstract type [5]. In this section we propose an implementation of abstract type NATABST by

[4] The functional character implies that an existing structure is never changed.

[5] This is the case if the abstraction function is not a bijection. Such restrictions can be the result of physical limitations of the concrete type.

the concrete type NATP (cf. fig. 2).

The heading of NATP includes information similar to figure 1. The lists of abstract and concrete operations have a biunique relationship. The rubric **support** describes (in this example in an inductive manner) the concrete support. The third clause, that of closure, specifies that every element of $natP$ is made up by applying the first two clauses a finite number of times. For example, $zero \in natP$ and $suc(suc(zero)) \in natP$. The heading **abstractionFunction** defines the abstraction function $\mathcal{A}$. $\mathcal{A}$ specifies how each element of $natP$ can be considered as an element of $natAbst$. For example $\mathcal{A}(zero) = 0$, $\mathcal{A}(suc(suc(zero))) = plus(plus(0,1),1) = 2$. Here $\mathcal{A}$ is a bijection [6].

type NATP= $(natP, (plus\_p), (inf\_p))$
refines NATABST
uses
  bool
support
  1) $n = zero \Rightarrow n \in natP$
  2) $n \in natP \Rightarrow suc(n) \in natP$
  3) closure
abstractionFunction
  heading: $\mathcal{A}(n) \rightarrowtail\!\!\!\!\!\rightarrow natAbst$
  pre: $n \in natP$
  rep: $\mathcal{A}(n) =$
    if $n = zero \rightarrow$
      0
    | $n \neq zero \rightarrow$
      Let $n = suc(n')$
      $plus(\mathcal{A}(n'), 1)$
    fi
operations
  1) **name:** $plus\_p(x, y) \twoheadrightarrow natP$
    pre: $x, y \in natP \times natP$
    spec: $\mathcal{A}(plus\_p(x, y)) = plus(\mathcal{A}(x), \mathcal{A}(y))$

  2) **name:** $inf\_p(x, y) \twoheadrightarrow bool$
    pre: $x, y \in natP \times natP$
    spec: $inf\_p(x, y) = inf(\mathcal{A}(x), \mathcal{A}(y))$
end

Figure 2: Specification of the concrete type NATP

The heading **operations** takes up the different operations which are listed in the heading **type**. As far as the heading **spec** is concerned, we must distinguish between internal and external operations although in both cases this specification is systematic. The specification of the internal operation $plus\_p$ tells us that the abstract value which corresponds, by the abstraction function, to the concrete value obtained by use of the concrete function $plus\_p$ to two concrete arguments x and y is identical to the abstract value obtained by the application of the abstract function $plus$ to two arguments obtained by application of the abstraction function to concrete arguments x and y. In the same way the specification of the external operation inf_p teaches us that the result obtained by application of the function inf_p to two concrete arguments x and y is identical to the result obtained by application of the abstract

[6] Which means that we define in this way an isomorphism between the abstract structure and the concrete one.

operation inf to the two abstract arguments resulting from the application of the abstraction function to x and y.

2.4 Calculation of the operations of concrete type NATP

The next step consists in calculating a functional representation of each operation. This calculation is based on the use of Leibniz's axiom [12] which states that if f is a function and if $x, e \in \mathrm{dom}(f) \times \mathrm{dom}(f)$ then

$$x = e \;\Rightarrow\; f(x) = f(e)$$

The application of this axiom to our problem consists in identifying x to the internal concrete operation (io_c), considered as the unknown, and the function f to the abstraction function $\mathcal{A}$. If we can produce an expression e such that $\mathcal{A}(io\_c(\ldots)) = \mathcal{A}(e)$ then a possible solution for $io\_c(\ldots)$ is e. In general the search for the expression e demands that we proceed by induction on the arguments of the operation io_c or by a case analysis aiming to take into account the different possible values for these arguments. The result of this is an iterative step, in which each stage gives a solution guarded by the condition which acts as an hypothesis for the calculation. When all cases have been taken into account it is necessary to group the different results in order to make up the functional representation of the operation under consideration.

2.4.1 Calculation of the operation plus_p

Let us show how this step leads us to calculate a representation of the operation $plus\_p$. We start from the expression $\mathcal{A}(plus\_p(x, y))$:

$$\begin{array}{ll} & \mathcal{A}(plus\_p(x,y)) \\ = & \quad \text{- - specification of the operation } plus\_p\text{, figure 2, page 4} \\ & plus(\mathcal{A}(x), \mathcal{A}(y)) \\ = & \quad \text{- - definition of the operation } plus\text{, figure 1, page 3} \\ & \mathcal{A}(x) + \mathcal{A}(y) \end{array}$$

It is, however, difficult to follow on without an hypothesis on the structure of x and/or y. We choose to proceed by structural induction on x. We must consider two cases: $x = zero$ and $x \neq zero$. We start with $x = zero$.

$$\begin{array}{ll} & \mathcal{A}(x) + \mathcal{A}(y) \\ = & \quad \text{- - hypothesis } (x = zero) \\ & \mathcal{A}(zero) + \mathcal{A}(y) \\ = & \quad \text{- - definition of the abstraction function } \mathcal{A}\text{, figure 2, page 4} \\ & 0 + \mathcal{A}(y) \\ = & \quad \text{- - arithmetic} \\ & \mathcal{A}(y) \end{array}$$

Thus for $x = zero$ we have:

$$\mathcal{A}(plus\_p(x, y)) = \mathcal{A}(y)$$

function $plus\_p(x, y) \twoheadrightarrow natP \ \widehat{=}$
pre
  $x, y \in natP \times natP$
then
  if $x = zero \ \rightarrow$
    y
  $| \ x \neq zero \ \rightarrow$
    Let $x = suc(x')$
    $suc(plus\_p(x', y))$
  fi
end

Figure 3. Fonctionnal representation of the operation *plus_p*

Now

$$\begin{array}{ll} & \mathcal{A}(plus\_p(x, y)) = \mathcal{A}(y) \\ \Leftarrow & \quad \text{- - Leibniz} \\ & plus\_p(x, y) = y \end{array}$$

This gives us the first guarded equation of the function *plus_p*:

$$x = zero \ \rightarrow \ plus\_p(x, y) = y$$

The induction case ($x = suc(x')$) is treated as follows:

$$\begin{array}{ll} & \mathcal{A}(x) + \mathcal{A}(y) \\ = & \quad \text{- - hypothesis } (x = suc(x')) \\ & \mathcal{A}(suc(x')) + \mathcal{A}(y) \\ = & \quad \text{- - definition of the abstraction function } \mathcal{A}\text{, figure 2, page 4} \\ & plus(\mathcal{A}(x'), 1) + \mathcal{A}(y) \\ = & \quad \text{- - definition of the operation } plus\text{, figure 1, page 3} \\ & \mathcal{A}(x') + 1 + \mathcal{A}(y) \\ = & \quad \text{- - arithmetic} \\ & \mathcal{A}(x') + \mathcal{A}(y) + 1 \\ = & \quad \text{- - definition of the operation } plus\text{, figure 1, page 3} \\ & plus(\mathcal{A}(x'), \mathcal{A}(y)) + 1 \\ = & \quad \text{- - specification of the operation } plus\_p\text{, figure 2, page 4} \\ & \mathcal{A}(plus\_p(x', y)) + 1 \\ = & \quad \text{- - definition of the operation } plus\text{, figure 1, page 3} \\ & plus(\mathcal{A}(plus\_p(x', y)), 1) \\ = & \quad \text{- - definition of the abstraction function } \mathcal{A}\text{, figure 2, page 4} \\ & \mathcal{A}(suc(plus\_p(x', y))) \end{array}$$

From Leibniz we obtain the second guarded equation for operation *plus_p*:

$$\begin{array}{l} x \neq zero \ \rightarrow \\ \quad \text{Let } x = suc(x') \\ \quad plus\_p(x, y) = suc(plus\_p(x', y)) \end{array}$$

We have calculated in all the representation of figure 3 for the operation *plus_p*.

function $inf\_p(x, y) \twoheadrightarrow bool \ \widehat{=}$
pre
    $x, y \in natP \times natP$
then
    if $y = zero \ \rightarrow$
        false
    | $y \neq zero \rightarrow$
        if $x = zero \rightarrow$
            true
        | $x \neq zero \rightarrow$
            Let $y = suc(y')$ et $x = suc(x')$
            $inf\_p(x', y')$
        fi
    fi
end

Figure 4. Functional representation of the operation inf_p

2.4.2 Calculation of the operation inf_p

External concrete operations (eo_c) are easier to deal with since Leibniz' axiom is not needed. All that is needed is to calculate an expression e such that $eo\_c(\ldots) = e$. For the operation inf_p, starting from $inf\_p(x, y)$ and carrying out a first structural induction on y then a second on x we obtain the solution of figure 4.

2.5 Transformation of the operations

We now have available a definite type NATP to represent natural numbers. This type can, with the appropriate notation, be included as it is in most programming languages (and especially in functional languages). However, for more efficiency (cf. [18]), we can transform the existing operations to allow sequencing, variables and the consequences which flow from them (especially iteration) while at the same time suppressing – if possible – recursion. At this stage in the development, we are ready to make a choice for the low level representation of the support of the type under consideration. We will be directly making the different transformations which will lead us to the final version of the type.

As regards support we have chosen a solution using a chained list such that the length of the list represents the number under consideration. Thus, using an ADA-like formalism:

type box;
type lnk **is** **access** box;
type box **is**
record
    next: lnk;
end record ;

Let $\mathcal{A}'$ be the abstraction function for this stage:

heading: $\mathcal{A}'(x) \rightarrowtail\!\!\!\!\!\rightarrow natP$
pre: $x \in lnk$
rep: $\mathcal{A}'(x)\ =$
  if $x =$ **null** $\rightarrow$
    $zero$
  | $x \neq$ **null** $\rightarrow$
    Let $x = box'(next \Rightarrow x')$
    $suc(\mathcal{A}'(x'))$
  fi

As regards the transformation of the operation $plus\_p$, the first stage consists in obtaining a procedure by adding an output parameter res:

procedure $plus\_p'(x, y, res : \textbf{out})\ \widehat{=}$
pre
  $x, y, res \in natP \times natP \times natP$
then
  var
    $aux \in natP$
  in
    if $x = zero \rightarrow$
      $res := y$
    | $x \neq zero \rightarrow$
      Let $x = suc(x')$
      $plus\_p'(x', y, aux)$;
      $res := suc(aux)$
    fi
  end
end

It is then easy to use the implantation type lnk in order to obtain a non-destructive version $plus\_p''$. This is what can be found in figure 5.

The transformation of the operation inf_p goes through an intermediate stage where a version satisfying the conditions of removing tail recursion is obtained (cf. for example [6]). The final, iterative, version is as it is shown in figure 6.

3 Application to the teaching of data structures

In this section we present the teaching module, its contents, organisation and development.

3.1 The teaching module proper

This module is aimed at first year student engineers, specialising in information technology (equivalent, in England, to the last year of a Bachelor of Science course and in the US to the third year). It takes place during the second semester.

The origin of the population is twofold. Some of the students come from the "preparatory classes for the Grandes Écoles" and have had little experience in programming before. On the other hand, these students are supposed to be good at

```
procedure plus_p''(x, y, res : out) ≙
pre
    x, y, res ∈ lnk × lnk × lnk
then
    var
        aux ∈ lnk
    in
        if x = null →
            res := y
        | x ≠ null →
            plus_p''(x.all.next, y, aux);
            res := new box'(next ⇒ aux)
        fi
    end
end
```

Figure 5. Final version of the operation *plus_p*

```
function inf_p'(x, y) ↠ bool ≙
pre
    x, y ∈ lnk × lnk
then
    var
        a, b ∈ lnk × lnk
    in
        a, b := x, y;
        do a ≠ null ∧ b ≠ null →
            a, b := a.all.next, b.all.next
        variant
            A(A'(a)) + A(A'(b))
        od;
        if a = null →
            b ≠ null
        | a ≠ null ∧ b = null →
            false
        fi
    end
end
```

Figure 6. Final iterative version of the operation *inf_p*

mathematics. The rest of the target group have a DUT (Diplôme Universitaire de Technologie) in computers, which is a professional qualification in IT. These students have, in general, taken a course with the same title as this module. But their level in mathematics is lower.

On arrival at ENSSAT all students take a basic course in programming. This

follows a semi-formal approach [15]. The students from the preparatory classes then follow complementary modules in programming design to familiarise them with the ideas of elementary data structures, of pointers and dynamic memory allocation.

The teaching of the data structures' module, out of a total of 66 hours, is made up of 22 hours of lectures, 18 hours of supervised activities, 24 hours of practicals followed by an exam lasting 2 hours, where personal documentation is allowed.

Up until this scholastic year the teaching of the module was more traditional, quite similar to what is recommended in [3]: basic data structures were presented using algebraic abstract data types [11,13,7] followed by a semi-formal refinement, by structural induction, on data structures at a basic level.

3.2 Lectures

The plan that we had anticipated is shown in figure 7. In section 4, page 23, we will see that we have had to limit our ambitions. Section 1.2 of figure 7 develops some simple examples, in the style of the example shown in section 1 above. Section 1.3 of figure 7 makes the students aware of two types of step (functional and MIS), at the same time showing how a classical programming language can be used as much as in one style as in another. Section 1.4 of figure 7 has been little developed because it is dealt with specifically in a module "Discrete Mathematics oriented toward Computer Science". Section 2 of figure 7 presents the principal set notations of B method as well as the extensions such as array shifting [7], conditional expressions, nondeterministic expressions [8], inductive structures and multiset (bag) notations. The data structure "list" is also studied which include catenation $l1 \sim l2$, inversion l^{-1} and casting to set [9]. The main properties (as, for example, $(a \sim b)^{-1} = (b^{-1} \sim a^{-1})$) are proved or presented.

The choice of data structures studied comes from [26]. The partition data structure has been put to one side for reasons of time. On the other hand the data structure "graphs", also absent from this module, is the object of a particular module in the second year.

In section 3 of the plan of figure 7 we introduce informally the data structures selected before presenting them formally. Section 4, "implementation of fundamental data structures" develops some traditional implementations from among the most efficient. At this time the students are warned that the quality of a data structure is intimately linked to the group of operations selected: adding a new abstract operation could bring to naught the research attempts for efficiency of a concrete support.

We propose at present a synthesis of the contents of sections 3 and 4 from the plan of figure 7 resuming each of the data structures studied while limiting ourselves, for the abstract specification, to the heading and to the support, and for the concrete specification, to the heading, to the support and to the abstraction

[7] For that we use the abbreviation $f \gg v$ defined for $v \in \mathbb{Z}$ and for any function f such that $\mathrm{dom}(f) \subset \mathbb{Z}$. By definition $f \gg v = (\lambda j \cdot (j \in \mathrm{dom}(f) \mid j + v))^{-1}; f$. In other words, if f is an array, $f \gg v$ returns an array of which the domain of definition is shifted by v compared to f, while retaining the values of array f.

[8] This notation modifies the unbounded choice substitution of method B to make it into an unbounded choice of *expressions*.

[9] $set(l)$ returns the set of the elements present in list l.

Figure 7. The plan of the module "Data Structure"

function. We assume the reader is familiar with B notations as well as with the abstract and concrete structures developed. We will not go into them unless we think it is necessary.

3.3 Sets

Historically, sets of (totally ordered) scalars were the first abstract data structures which interested computer scientists. They naturally concentrated on efficient implementations such as binary search trees, AVLs, 2-3 trees, B-trees, hash tables and so on. In the lectures we limit ourselves to the implementation by characteristic vectors, by binary search trees and by B-trees. However, see [14] for a formal development of two other implementations: by array and by AVL.

3.3.1 Sets of scalars: abstract specification

The abstract support identifies the type *setAbst* to the set of finite subsets of natural numbers.

abstractType SETABST= $(setAbst, (clear, insert, remove), (isIn, isEmpty))$
...
support
$e \in \mathbb{F}(\mathbb{N}) \quad \Leftrightarrow \quad e \in setAbst$

3.3.2 Sets of scalars: concrete specification by characteristic vectors

Several types of implementation by arrays are conceivable, few of them are efficient. But when the elements to be represented belong to a reasonable interval, an efficient solution exists for most of the operations. This consists of using an array of Booleans such that the value subscripted by i indicates the presence, or not, of i in the subset under consideration. This technique is known as the "characteristic vector method". The abstraction function $\mathcal{A}$ identifies the set represented to the set of subscripts corresponding to the value **true** in the array used as support.

type SETCV(n)=
$(setCV, (clear\_cv, insert\_cv, remove\_cv), (isIn\_cv, isEmpty\_cv))$
...
support
$a \in 1 \,..\, n \rightarrow bool \quad \Leftrightarrow \quad a \in setCV(n)$
abstractionFunction
heading: $\mathcal{A}(a) \rightarrow setAbst$
pre: $a \in setCV(n)$
rep: $\mathcal{A}(a) \;=\; \mathrm{dom}(a \rhd \{\mathbf{true}\})$

This solution is in general very efficient, save for the operation *clear_cv* which demands a traverse of the array.

3.3.3 Sets of scalars: concrete specification by binary search trees

The support *setBst* is described without difficulty by induction. The abstraction function $\mathcal{A}$ which gives the set of values present in the tree is defined by structural induction on its argument.

type SETBST=
$(setBst, (clear\_st, insert\_st, remove\_st), (isIn\_st, isEmpty\_st))$
...
support
1) $e = \langle\rangle \;\Rightarrow\; e \in setBst$
2) $n \in \mathbb{N} \wedge\; l \in setBst \wedge\; r \in setBst \wedge\; \max(\mathcal{A}(l)) < n \;\wedge\; \min(\mathcal{A}(r)) > n$
$\Rightarrow$
$\langle l, n, r\rangle \in setBst$
3) closure
abstractionFunction
heading: $\mathcal{A}(x) \twoheadrightarrow setAbst$
pre: $x \in setBst$
rep: $\mathcal{A}(x) \;=$
if $x = \langle\rangle \rightarrow$
$\varnothing$
$\mid x \neq \langle\rangle \rightarrow$
Let $x = \langle l, n, r\rangle$
$\mathcal{A}(l) \cup \{n\} \cup \mathcal{A}(r)$
fi

The operation *insert_st* is stated in two forms: insertion at the leaves and at the root. For this last version two techniques are studied: by cutting and by rotations. Random insertion is simply referred to.

3.3.4 Sets of scalars: concrete specification by B-trees

B-trees have first been described in an article by R. Bayer and E. McCreight in 1972 [4]. Since then, numerous variants of B-trees have been suggested. We will consider B-trees of order n ($n > 0$) which are characterised by the following properties:

1. For any given node, all sub-trees have the same height.

2. Each node has between n and $2.n$ values, except the root.

2 b. The root has between 1 and $2.n$ values.

3. The values at the nodes are sorted in strict ascending order.

4. Each node made up of p values refers to $p + 1$ B-trees, all of the same height.

4 b. Each leaf made up of p values is linked to $p + 1$ empty B-trees.

5. A B-tree is a search tree: the inorder traversal encounters values in strict ascending order.

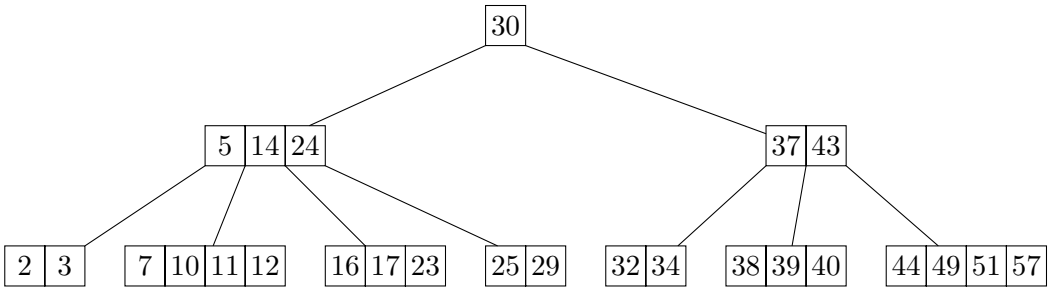


Figure 8. An example of 2-ordered B-tree

The main difficulties in the definition of the support come from the fact that (1) the root of a B-tree could have only one element, (2) at the time of an update (an insertion, for example) the structure could not be a strict B-tree: a node can momentarily become overflowed. The parameters of type NBT take into account these particularities: n is the order of the B-tree, *low* the minimum number of elements of a node, *up* the maximum number. *tt* is an auxiliary type which allows the description of nodes. $h(t)$ is the function which produces the height of the tree t. In the rest of the paper, contrary to B, we admit, together with [10], that if $s \subset \mathbb{N} \wedge s = \varnothing$ then $\max(s) = 0$ and $\min(s) = +\infty$.

The abstraction function, on the other hand, presents no difficulties: it is defined by structural induction on the support nBT.

type NBT(n, low, up) =
$(nBT, (clear\_bt, insert\_bt, remove\_bt), (isIn\_bt, isEmpty\_bt))$
constraints
$n \in \mathbb{N}_1 \wedge$
$low \mathinner{..} up \in \{1 \mathinner{..} 2.n, n \mathinner{..} 2.n, 1 \mathinner{..} 2.n+1, n \mathinner{..} 2.n+1\}$
$\ldots$
support
$a \in 1 \mathinner{..} 2.n+1 \rightarrow \mathbb{N} \wedge sz \in low \mathinner{..} up \wedge \forall i \cdot (i \in 1 \mathinner{..} sz-1 \Rightarrow a(i) < a(i+1))$
$\Leftrightarrow$
$a \in tt(sz)$
1) $t = \langle\rangle \;\Rightarrow\; t \in nBT(n, low, up)$
2) $t = \langle te, tl, sz\rangle \;\wedge$
$sz \in low \mathinner{..} up \;\wedge$
$te \in tt(sz) \;\wedge$
$tl \in 0 \mathinner{..} 2.n+1 \rightarrow nBT(n, n, 2.n+1) \;\wedge$
$\forall i \cdot (i \in 1 \mathinner{..} sz \Rightarrow h(tl(i)) = h(tl(0))) \;\wedge$
$\forall i \cdot (i \in 1 \mathinner{..} sz \Rightarrow \max(\mathcal{A}(tl(i-1))) < te(i) \;\wedge\; \min(\mathcal{A}(tl(i))) > te(i))$
$\Rightarrow$
$t \in nBT(n, low, up)$
3) closure
abstractionFunction
heading: $\mathcal{A}(x) \twoheadrightarrow setAbst$
pre: $x \in nBT(n, low, up)$
rep: $\mathcal{A}(x) \;=$
if $x = \langle\rangle \rightarrow$
$\varnothing$
$\mid x \neq \langle\rangle \rightarrow$
Let $x = \langle te, tl, sz\rangle$
$te[1 \mathinner{..} sz] \cup \bigcup i \cdot (i \in 0 \mathinner{..} sz \mid \mathcal{A}(tl(i)))$
fi

In the lectures we limit ourselves to the calculation of the operation *insert_bt* in its ascending version [10]. The possibility of descending versions or of exploiting rotations is only mentioned.

3.3.5 Sets of strings

Sets of strings are also broached. The implementation used is that of tries. However, Patricia tries [24] are not explained.

3.4 Stacks

For an experienced computer scientist the choice of stacks as a fundamental data structure is unavoidable. This is not the case for beginners for whom the experience that they have of recursive procedures is often not enough for them to understand the link between embedded structures and stacks.

[10] Which means the possible splitting of a node happens when ascending.

3.4.1 Stacks: abstract specification

A stack is specified as a pair, the first element of which is a function whose upper limit is not restricted and the second is a pointer to the last element recorded. The domain restriction on the interval going as far as the pointer is an array.

abstractType STACKABST(T)=
$(stackAbst, (clear, push, pull), (topVal, isEmpty))$
...
support
$t \in \mathbb{N}_1 \nrightarrow T \wedge ip \in \mathbb{N} \ \wedge \ 1 \,..\, ip \lhd t \ \in 1 \,..\, ip \rightarrow T \quad \Leftrightarrow \quad (t, ip) \in stackAbst(T)$

3.4.2 Stacks: concrete specification by lists

The implementation by arrays is very close to the abstract specification. For this reason its development has no great interest, it is simply mentioned in the lectures. The only implementation studied is that based on lists. Its characteristics are shown below.

type STACKL(T)=
$(stackL, (clear\_l, push\_l, pull\_l), (topVal\_l, isEmpty\_l))$
...
support
1) $a = \langle\rangle \quad \Rightarrow \quad a \in stackL(T)$
2) $v \in T \wedge p \in stackL(T) \quad \Rightarrow \quad \langle v, p\rangle \in stackL(T)$
3) closure
abstractionFunction
heading: $\mathcal{A}(s) \rightarrowtail\!\!\!\!\!\rightarrow stackAbst(T)$
pre: $s \in stackL(T)$
rep: $\mathcal{A}(s) \ =$
if $s = \langle\rangle \rightarrow$
any t **where** $t \in \mathbb{N}_1 \nrightarrow T$ **then** $(t, 0)$ **end**
$\mid s \neq \langle\rangle \rightarrow$
Let $s = \langle v, p\rangle$ and $\mathcal{A}(p) = (t, ip)$
$(t \lhd\!\!\!\!- \{(ip + 1, v)\}, ip + 1)$
fi

The students are told that this implementation is very efficient, but as soon as one tries to extend the range of operations, for example by allowing access to a value other than the one at the top of the stack, the solution by list has to be abandoned.

3.5 Queues

3.5.1 Queues: abstract specification

The justification of the choice of queues as a fundamental data structure is easy to understand, examples from daily life are sufficiently illustrative.

Below, the abstract specification is made on the lists' base. The elements of queues are keys, consequently an element cannot appear several times in the list.

This property materialises in the support by the existence of the conjunct $v \notin set(l)$ which prohibits duplicates in the list. From this support it is easy to specify the operations, thus the insertion (at the end of the queue q) of the value v is expressed by $\langle v, q^{-1}\rangle^{-1}$.

abstractType QUEUEABST(T) =
$(queueAbst, (clear, insert, remove), (head, isEmpty))$
...
support
1) $a = \langle\rangle \;\Rightarrow\; a \in queueAbst(T)$
2) $v \in T \wedge l \in queueAbst(T) \wedge v \notin set(l) \;\Rightarrow\; \langle v, l\rangle \in queueAbst(T)$
3) closure

There are at least three different representations of queues: (1) the representation by array, with a pointer to the head and another to the tail, (2) the representation by list with insertion at the head of the list and suppression from the tail, and, less often taught, (3) representation by double lists. We have decided to select solution (1), to put to one side solution (2) (solutions using pointers or by *deque* are however possible, cf. [8]), and to detail solution (3) which has the added advantage of a nice introduction to the theory of amortized complexity.

3.5.2 Queues: concrete specification by arrays

Conceptually simple, this solution has, however, technical difficulties. The first being that it is necessary, in the definition of support, to distinguish the case where the pointer to the head is less than or equal to the pointer to the tail from the complementary case. The second difficulty comes from the fact that the nature of the abstraction function $\mathcal{A}$ demands that the reverse function $\mathcal{A}^{-1}$ is also specified.

type QUEUEA(T, n) =
$(queueA, (clear\_a, insert\_a, remove\_a), (head\_a, isEmpty\_a))$
...
support
$f \in 1\,..\,n \to T \wedge h \in 1\,..\,n \wedge t \in 1\,..\,n \wedge$
$(h \leq t \;\Rightarrow\; h\,..\,t-1 \lhd f \in h\,..\,t-1 \rightarrowtail T) \wedge$
$(h > t \;\Rightarrow\; t\,..\,h-1 \lhd\!\!\!- f \in (1\,..\,t-1 \cup h\,..\,n) \rightarrowtail T)$
$\Leftrightarrow$
$(f, h, t) \in queueA(T, n)$
abstractionFunction
heading: $\mathcal{A}((f, h, t)) \rightarrowtail queueAbst(T)$
pre: $(f, h, t) \in queueA(T, n)$
rep: $\mathcal{A}((f, h, t)) =$
if $h = t \to$
$\langle\rangle$
$\mid h \neq t \to$
$\langle f(h), \mathcal{A}((f, h+1 \bmod n, t))\rangle$
fi

rep: $\mathcal{A}^{-1}(l)$ =
  if $l = \langle\rangle \rightarrow$
    any f, h, t **where**
      $(f, h, t) \in queueA(T, n) \wedge h = t$
    then
      (f, h, t)
    end
  | $l \neq \langle\rangle \rightarrow$
    Let $l = \langle v, b\rangle$
    any f, h, t **where**
      $(f, h, t) \in queueA(T, n) \wedge f(h) = v \wedge b = \mathcal{A}((f, h + 1 \bmod n, t))$
    then
      (f, h, t)
    end
  fi

3.5.3 Queues: concrete specification by double lists

To implement directly the specification of a queue using one list would lead, for the insertion operation, to traversing the list to its end before carrying out the insertion itself. The cost of such an operation is prohibitive. An other solution consists in managing two lists, one for suppression and the other for insertion. Manipulation is always done at the head: suppression is done at the head of the first list and addition at the head of the second. However, at the time of a suppression, if the first list is empty it is necessary to invert the second into the first as it is shown in the following example:

Operations	Concrete representation	abstract queue
	`([4,8],[7,3])`	→ `[4,8,3,7[`
Insertion of 2		
	`([4,8],[2,7,3])`	→ `[4,8,3,7,2[`
Removal		
	`([8],[2,7,3])`	→ `[8,3,7,2[`
Removal		
	`([],[2,7,3])`	→ `[3,7,2[`
Removal		
	`([3,7,2],[])`	
	`([7,2],[])`	→ `[7,2[`

In supposing the data structure "list" is available, the concrete specification is simple. The abstraction function defines the abstract queue as the catenation of the first list and the inverse of the second.

type QUEUEDL(T) =
$(queueDL, (clear\_dl, insert\_dl, remove\_dl), (head\_dl, isEmpty\_dl))$
$\ldots$
support
$a \in queueAbst(T) \wedge b \in queueAbst(T) \wedge set(a) \cap set(b) = \varnothing$
$\Leftrightarrow$
$(a, b) \in queueDL(T)$
abstractionFunction
heading: $\mathcal{A}((a, b)) \twoheadrightarrow queueAbst(T)$
pre: $(a, b) \in queueDL(T)$
rep: $\mathcal{A}((a, b)) \ = a \sim b^{-1}$

3.6 Priority queues

Priority queues are the object of much scientific literature. Several implementations have been suggested. They are distinguished one from another by the existence or not of the merging operation. If merging is not part of the set of operations, a simple solution exists: perfect heaps. In the opposite case efficient solutions are based on sophisticated data structures. In the lectures we study one from among them: binomial queues [25,26].

3.6.1 Priority queues: abstract specification

Conceptually a priority queue is similar to a bag (multiset). In order to avoid an implementation by a double refinement, we have decided to consider that the everyday notations on the bags are available in the language used for specification. To make them easier to remember by the students we have used the following convention: a multiset symbol is the "squared" counterpart of the set symbol (thus $\boxslash$ corresponds to $\varnothing$, $\sqcup$ corresponds to $\cup$, [ ] corresponds to { }, etc.). As regards $\mathbb{B}(\mathbb{N})$ it represents the finite set of bags of natural numbers.

abstractType PQABST=
$(pqAbst, (clear, insert, remove, merge), (prioVal, isEmpty))$
$\ldots$
support
$q \in \mathbb{B}(\mathbb{N}) \ \Leftrightarrow \ q \in pqAbst$

3.6.2 Priority queues: concrete specification by binomial queues

Binomial queues are defined by cross recursion with binomial heaps. A binomial heap (of support bH) is a couple with one part made up by a natural number value and the other by a binomial queue. The whole makes a heap. h is the function which produces the height of the tree.

The use of a binomial heap demands the use of the function bh which returns the bag of values present in the tree. A binomial queue is defined as a list (possibly empty) of binomial heaps in strict height descending order. The example in figure 9, page 20, shows a binomial queue made up of 4 binomial heaps of heights 4, 3, 2,

and 1 respectively. It is defined inductively, using the function bh to convert the binomial heap at the head of the queue.

type BH(n)= $(bH, (), (bh, min, h))$
...
support
$a \in \mathbb{N} \wedge$
$f \in pqBq(i) \wedge$
$a \leq min(f) \wedge$
$\#(f) = i$
$\Leftrightarrow$
$(a, f) \in bH(i+1)$
operations
1) **name:** $bh((r,l)) \rightarrow bqAbst$
pre: $(r,l) \in bH(i)$
spec: $bh((r,l)) = [r] \sqcup \mathcal{A}(l)$
...

The merging operation plays a central role in the calculation of operations. In fact, insertion can be defined as the merging of the queue with the value to be added, converted to a binomial queue. Suppression can in the same way be defined as the suppression of the smallest value followed by a merging of 1) the initial queue deprived of the heap affected by the suppression, and 2) of the heap deprived of its root. The operation $merge\_bq$ obtained by calculation is slightly different from that supplied in [25]. Moreover it makes explicit use of the function h. This solution is only viable under the condition that, at the time of looking for the eventual implantation data structure, the calls to h can be replaced by the consultation of a field in the data structure. This is the case since the function h is decomposable (cf. [8]). We thus obtain a very efficient solution for all operations.

type PQBQ(n)= $(pqBq, (clear\_bq, insert\_bq, remove\_bq, merge\_bq),$
$(prioVal\_bq, isEmpty\_bq))$
...
support
1) $f = \langle\rangle \Rightarrow f \in pqBq(0)$
2) $m \in bH(i) \wedge$
$f \in pqBq(j) \wedge$
$i > j$
$\Rightarrow$
$\langle m, f\rangle \in pqBq(i)$
3) closure
abstractionFunction
heading: $\mathcal{A}(f) \twoheadrightarrow pqAbst$
pre: $f \in pqBq(i)$
rep: $\mathcal{A}(f) =$
if $f = \langle\rangle \rightarrow$
$\boxslash$
$\mid f \neq \langle\rangle \rightarrow$
Let $f = \langle h, t\rangle$
$bh(h) \sqcup \mathcal{A}(t)$
fi

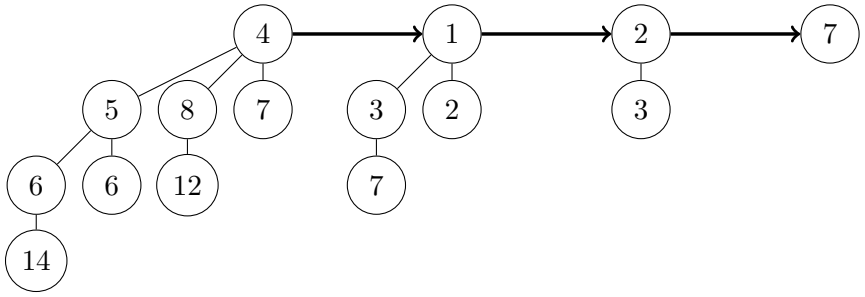


Figure 9. Example of a binomial queue

3.7 Flexible arrays

In its standard version a flexible array is a data structure which, following the example of a single dimensional array, is defined as a total function on the domain $1..n$ ($n \geq 0$), but with the difference that this one allows an insertion in the interval $1..n+1$ or a removal in the interval $1..n$, bringing with it a variation of the upper limit n at run time. Our specification is inspired by [26] but restricted interesting forms of flexibility exist such as [19,8].

3.7.1 Flexible arrays: abstract specification

The operation $insert(v, i, a)$ inserts the value v in the flexible array a at position i; the values initially in position after $i-1$ have their subscripts increased by one. The array's length is increased by one. Inversely the operation $remove(i, a)$ deletes the element in position i and the values placed initially after i have their subscripts reduced by one. The array's length decreases by one. The operation $val(i, a)$ returns the value in position i.

abstractType FAABST(T, s) =
$(faAbst, (clear, insert, remove, catenate), (val, \#))$
...
support
$t \in \mathbb{N}_1 \nrightarrow T \ \wedge \ s \in \mathbb{N} \ \wedge \ 1..s \lhd t \ \in \ 1..s \rightarrow T \ \Leftrightarrow \ t \in faAbst(T, s)$

A flexible array is specified as a total function defined on the domain $1...s$.

3.7.2 Flexible arrays: concrete specification by lists

The implementation of flexible arrays by lists defines the abstraction function $\mathcal{A}$ by induction on the rest of the list using the shifting operation $\gg$.

type FALS(T) =
  $(faLs, (clear\_ls, insert\_ls, remove\_ls, catenate\_ls), (val\_ls, \#\_ls))$
  ...
support
  1) $l \in \langle\rangle \;\Rightarrow\; l \in faLs(T)$
  2) $v \in T \;\wedge\; l \in faLs(T) \;\Rightarrow\; \langle v, l\rangle \in faLs(T)$
  3) closure
abstractionFunction
  heading: $\mathcal{A}(l) \twoheadrightarrow faAbst(T, s)$
  pre: $l \in faLs(T) \wedge s \in \mathbb{N} \wedge s = \#(l)$
  rep: $\mathcal{A}(l) \;=$
    if $l = \langle\rangle \rightarrow$
      $1 \,..\, 0 \lhd \varnothing$
    | $l \neq \langle\rangle \rightarrow$
      Let $l = \langle v', l'\rangle$
      $\{(1, v')\} \;\Leftarrow\; \mathcal{A}(l') \gg 1$
    fi

This solution is obviously costly in so far as all the direct access to any position in the array is obtained by sequential access to the list.

3.7.3 Flexible arrays: concrete specification by heaps

An acceptable solution is obtained by adopting as support a heap memorising all the values in the array.

type FAHP(T) =
  $(faHp, (clear\_hp, insert\_hp, remove\_hp, catenate\_hp), (val\_hp, \#\_hp))$
  ...
support
  1) $a = \langle\rangle \;\Rightarrow\; a \in faHp(T)$
  2) $n \in T \;\wedge\; l, r \in faHp(T) \times faHp(T) \;\wedge$
    $n \leq \max(set(l)) \;\wedge\; n \leq \max(set(r))$
      $\Rightarrow$
    $\langle l, n, r\rangle \in faHp(T)$
  3) closure
abstractionFunction
  heading: $\mathcal{A}(a) \twoheadrightarrow faAbst(T, s)$
  pre: $a \in faHp(T) \wedge s \in \mathbb{N} \wedge s = w(a)$
  rep: $\mathcal{A}(a) \;=$
    if $a = \langle\rangle \rightarrow$
      $1 \,..\, 0 \lhd \varnothing$
    | $a \neq \langle\rangle \rightarrow$
      Let $a = \langle l, n, r\rangle$
      $\mathcal{A}(l) \;\Leftarrow\; \{(w(l) + 1, n)\} \;\Leftarrow\; \mathcal{A}(r) \gg w(l) + 1$
    fi

```
1 Set oriented specification
2 inductive structures and specification
3 Refinement:  specification and operation calculation
4 Sets
5 Stacks
6 Queues
7 Priority queues
8 Flexible arrays
```

Figure 10. Plan of supervised work in Data Structures

Formalising the support uses the operation *set* on the tree in order to express the heap condition. $set(t)$ provides the set of the values belonging to the heap t. The abstraction function is defined by structural induction on the heap. It uses the function w which returns the weight of the heap passed as an argument.

This method avoids having to memorise explicitly the absolute value of each subscript, a solution which brings with it a prohibitive cost for updating operations. The weight w ensures the relation between the subscript of an element in the flexible array and its position in the heap. The call to function w appears explicitly in the operations as they are derived. The remark on page 19 about the function h applies here to the function w: the result is only viable on the condition that the call to the function w could be replaced by an equivalent field in the final data structure. It is possible since the function w is, for a heap, separable. We thus obtain a solution which is efficient on average but which could degenerate into $O(n)$ in the worst case.

The teaching support offered to the students for the lectures are, on the one hand, a hand-out containing the essential points of the course (abstract and concrete specifications, functional representation of operations) and on the other hand a Web access to the development of operations, allowing them to go over the course at home.

3.8 Supervised work

The 18 hours of supervised work have, as an objective, allowing the student to understand in depth the lectures by the intermediary of assimilation and application exercises. The 18 hours are made up as shown in figure 10.

Here is an example of an exercise suggested in section 1 of the supervised works:

> Let $n \in \mathbb{N}$ and tab the support defined by:
>
> $$t \in 1\,..\,n \rightarrow \mathbb{N} \quad \Leftrightarrow \quad t \in tab.$$
>
> Let $s \in tab$. Specify the operation $sevenBefore23$ which returns an array holding the same bag of values as s but such that all the 7s come before all the 23s.

In section 2, figure 10, we are studying lists and we show some of their properties.

We also make up some simple exercises on binary trees. In sections 4 to 8 we perform exercises which complete the developments carried out in the lectures (calculation of a new version of an operation, or calculation of a yet undeveloped operation). We also suggest new data structures as, for example, sets of couples, or queues with tokens.

3.9 Practical work

The objective of the practical work is to familiarise the students with the practical aspects of data structures. A typical session would consist of, starting from a data structure in which all the operations have been calculated, supplying the development necessary in order to obtain either a functional solution or an MIS solution for this data structure. In the first two sessions simple solutions are developed in parallel using the two methods. The last sessions are allotted to projects in which all the conception and the development are to be done following a given specification. As an example, the students have to develop a key based priority queue where an already present client could take place in the queue with a different priority. The students work on their own.

4 First appraisal

As we have said in the introduction, the appraisal which we present here reflects principally the observations and the points of view of the teachers of the module. The general feeling of the contributors is that this method has re-awakened interest in a scientific area which has suffered from routine. The following points are those most mentioned:

(i) The amount of time allowed for lectures was enough for about 60% of the course previously covered. Replying off the cuff, our strategy has been to maintain the depth of knowledge at the expense of the width. Consequently, the implementation of certain data structures has had to be transferred to supervised work (as for example is the case for stacks or queues by arrays). Others have purely and simply disappeared (as is the case for sets of strings by tries).

(ii) One classical consequence of the use of a formal approach is the ease with which properties are expressed. Using EB, this advantage is accentuated by the functional character of the operations as far as it is possible to formulate a property bearing on the "new" structure as well as on the "old", the two existing together. Thus, if one wishes to state that the insertion in an AVL increases the height of the tree of in most one, it is possible to simply write

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

(iii) The EB approach looks fairly useless in the discovery of efficient data structures. This is a privilege which remains the sole right of the human being. On the other hand, we have noticed that the operations obtained by calculation can bring something new compared with known solutions. Thus, we have noticed that the operation of merging binomial queues obtained by calculation

is different from those presented in [25]. The latter, in adopting the metaphor of addition suggests a solution which goes from right to left. Our solution produces a calculation which advances from left to right.

This is also the case in the implementation of sets by Avl (cf. [14]): the insertion version obtained by calculation does not demand, in contrast to traditional versions, to convey a parameter indicating if the height has changed or not, since the functional character of the operation means that we have access to the height of the new tree as well as that of the old one [11] .

(iv) The use of formal approaches leads to a beneficial change in the way of thinking. This has often been observed (cf. [6,9,20,10]). eb does not depart from it. This is due, according to us, to the fact that, although starting from the concrete space for finally coming back to it, the main reasoning is done in the abstract space. Let us illustrate our statement with an example. Consider the implementation of sets of scalars by binary search trees and more exactly the remove operation *remove_st* (cf. § 3.3.3, page 12). Removing the value at the root can be formulated in eb in the following manner: *If* $l \neq \langle\rangle$ *and* $r \neq \langle\rangle$ *removing the value* n *of* $\mathcal{A}(\langle l, n, r\rangle)$ *leads to the expression* $\mathcal{A}(l) \cup \mathcal{A}(r)$. *If we were able to transform this expression so that it was in the form* $\mathcal{A}(l') \cup \{m\} \cup \mathcal{A}(r')$ *we would be able to apply the reverse of the abstraction function (on condition that moreover* $\max(\mathcal{A}(l')) < m \ \wedge \ m < \min(\mathcal{A}(r'))$*) in order to make it take the form* $\mathcal{A}(\ldots)$. *A solution consists in searching for a couple* (m, l') *such that* $\mathcal{A}(l') = \mathcal{A}(l) - \{m\}$ *and that* $m = \max(\mathcal{A}(l))$ *and in taking* $r' = r$. *This couple* (m, l') *exists since* $l \neq \langle\rangle$.

This reasoning is to be put in parallel with, for example, that carried out in [23] [12] : *"However, if the node we want to delete has two children, then we need to find a place for the two pointers. Let B be a node with two children whose key we want to delete. In the first step we exchange the key of B with a key of another node X, such that (1) X has at most one child, and (2) deleting X will leave the tree consistent. In the second step, we delete X, which now has the key of B which we wanted to delete. We can easily delete X, because it has at most one child. To preserve the consistency of the tree, the key of X must be at least as large as all the keys in the left subtree of B, and must be smaller than all the keys in the right subtree of B... X cannot have a right child, since otherwise it would not have the largest key in that subtree."* We note in particular that the reasoning is informal and that it is sited entirely in the concrete space (which does not guarantee that the operation has really had the effect wanted at the abstract level).

Concerning an appraisal which would reflect the students' point of view, a questionnaire will be provided for all at the end of the module and the results will be analysed carefully. Some discussions at the end of the course have allowed a glimpse of their impressions. Some of these points are:

- Difficulty in understanding the nature of and the interest in a specification.

[11] Of course, this advantage disappears if the operation loses its functional character.

[12] An excellent book which bases the construction of programs on reasoning by induction.

- A feeling of surprise that programs can be calculated ("but what do you mean by calculate programs ?").
- Some difficulty in the beginning to absorb the formalisms and the derivational style, but for most a fairly quick assimilation (about 6 hours of supervised work).
- Hermetism in the face of the difficulties and of the issues of the discipline. Let us illustrate with a dialogue showing what happens at the end of a course where the teacher had spent about one hour constructing and producing a program of about 15 lines:
  Student: Does it always take that long to develop a program ?
  Teacher: Yes, a program is the often the result of complex reasoning. It is necessary to explain the different aspects of its construction.
  Student: But I can do that in 5 minutes!

5 Conclusion

The scientific character of our discipline is a reality, every computer scientist feels this intimately. It is however a facet of our activity which is only too rarely shown. The main reason is the excessive room offered to descriptive approaches, notably for those to do with the teaching of data structures. We have tried to show that there is no inevitability in this state of affairs. We have applied the formal approach EB for the first time in the teaching of data structures aimed at student engineers starting a course of three years. The principle chapter headings of an existing course have been preserved, the contents have been adapted to EB, offering an homogenous presentation of each data structure under the form:

(i) Informal presentation of each abstract data structure.

(ii) Formal specification of abstract data structures (support, operations).

(iii) Informal presentation of each concrete implementation.

(iv) Formal specification of concrete data structures (support, abstraction function, operations).

(v) Calculation of the functional representation of each operation.

The response of the teaching team is very positive. Not one contributor wishes to go back to the previous situation. The students' point of view will be presented at the conference.

References

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

[2] Abrial, J.-R., *Teaching Formal Methods: an Experience with Event-B.* Formal Methods in Computer Science Education, Satellite workshop of ETAPS 2008, Budapest Hungary, March 29, 2008, 1-4.

[3] Aho, A.V., J.D. Ullman, "Foundations of Computer Science," C Ed., MIT Press, W.H. Freeman company ISBN-13: 978-0716782841 1992.

[4] Bayer R., E. McCreight, *Organization and Maintenance of Large Ordered indexes.* Acta Informatica, Vol. 1 Fasc. 3 1972. p. 173-189.

[5] Chabernaud, V., C. Marteau, *Mini projet de génie logiciel : les spécifications algébriques.* DESS Quassi 2001-2002. http://www.univ-angers.fr/docs/etudquassi/Specifications_algebriques.pdf.

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

[7] Derniame, J.-C., J.-P. Finance, "Types abstraits de données : spécification, utilisation et réalisation," CRIN. Nancy. École d'été de l'AFCET. Monastir. 79.E.57. 1979.

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

[9] Dijkstra, E.W. "A discipline of programming". Prentice-Hall, 1976. ISBN 0-13-215871-X.

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

[11] Dufourd J.-F., D. Bechman, Y. Bertrand, "Spécifications algébriques, algorithmique et programmation". InterEditions. ISBN 2 7296 0581 9. 1995.

[12] Gries, D., F.B. Schneider, "A Logical Approach to Discrete Math". Springer-Verlag. 1994.

[13] Guttag, J.V., Horning, J.J. *The Algebraic Specification of Abstract Data Types.* Acta Informatica. **10** (1978), 27-52.

[14] Guyomard, M. eb*: A constructive approach for the teaching of data structures.* Formal Methods in Computer Science Education, Satellite workshop of ETAPS 2008, Budapest Hungary, March 29, 2008, 25-36.

[15] Guyomard, M. *Spécification et programmation : Le cas de la construction de boucles.* Enssat, 2006. Available on the web at http://www.irisa.fr/cordial/mguyomar/guyomard-15-09-05.htm.

[16] Habrias, H. *Teaching specifications, hands on.* Formal Methods in Computer Science Education, Satellite workshop of ETAPS 2008, Budapest Hungary, March 29, 2008, 5-14.

[17] Hoare, C.A.R. *Proof of Correctness of Data Representations.* Acta Informatica, 1, p. 271-281, 1972.

[18] Hoare, C.A.R. *An Overview of Some Formal Methods for Program Design.* IEEE Computer, September 1987, p. 85-91.

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

[20] Kaldewaij A. "Programming: the Derivation of Algorithms". Prentice-Hall, 1990. ISBN 0-13-204108-1.

[21] Kaldewaij A. "Programming: the Derivation of Algorithms. Teacher's Manual". Prentice-Hall, 1991. ISBN 0-13-678046-6.

[22] Kaldewaij A., V.J. Dielissen, *Decomposable Functions and leaf Trees: A Systematic Approach.* Programmaing Concepts, Methods and Calculi (A-50). E-R Olderog (Editor). Elsevier Science B.V. (North Holland), 1994 IFIP.

[23] Manber U. "Introduction to algorithmes – A Creative Approach", Addison-Wesley, 1989.

[24] Morrison D.R. patricia – *Practical Algorithm To Retreive Information Codes in Alphanumeric.* Journal of the Association for Computing Machinery, Vol 15, No. 4, October 1968, p. 514-534.

[25] Vuillemin J. *A Data Structure for manipulating Priority Queues.* Communications of the ACM. Vol. 21, Number 4, p. 309-315. April 1978.

[26] Vuillemin J. "Structures de Données". Notes de cours. INRIA 1981.

Modelling Role-based Access Control in B

Steve Dunne and Anthony Howitt

School of Computing, University of Teesside
Middlesbrough, TS1 3BA, UK
s.e.dunne@tees.ac.uk

Abstract

We report our experience of recasting the mathematically-based ANSI RBAC standard model for access control into the notation of the classical B method. We show how our use of the B method enabled us to discover a number of anomalies in the original ANSI standard. We also explain how it enabled us to simplify and clarify the standard in several important respects, as well as enhancing its reliability by type-checking it and verifying its logical consistency.

1 Introduction

Our purpose in this paper is to report our experience of recasting the mathematically-based ANSI RBAC [1] standard model for access control into the notation of the classical B method. Certainly, the B method did prove to lend itself very readily to such an endeavour, but we are anxious to assure the reader that what we undertook was much more than a mere mechanistic transcription exercise. Indeed, we aim to show how our use of the B method enabled us to discover a number of anomalies in the original ANSI standard, and we also seek to show how it enabled us to simplify and clarify the RBAC standard model in several important respects. Moreover, we report how the B method has enabled us formally to verify our recast model's consistency by generating the necessary consistency proof obligations and then discharging them with the autoprover and interactive prover of B's computer-based tool support, thus further enhancing the reliability of our RBAC model over that of the original ANSI standard. On the other hand we would also stress that our intention was in no sense to attempt to "improve" on the RBAC standard model by modifying it in any substantial way, but rather simply to clarify it as it stands. The only modifications we have imposed on the existing model, therefore, have been those necessary to remove redundancies or rectify actual inconsistencies in it. In each of these cases we have striven to remain faithful to the intentions of the model's original architects as we have perceived them.

[1] **R**ole-**B**ased **A**ccess **C**ontrol

In Section 2 we give a short apprisal of the origins and salient features of the RBAC standard, while in Section 3 we describe its three constituent reference models. In Section 4 we present our recasting of the RBAC standard in B and in Section 5 we describe how we were subsequently able to check our recast model for logical consistency. In Section 6 we go on to assess the benefits of recasting the RBAC standard in B and report on the deficiencies we discovered in the original standard while undertaking this exercise. Finally, in Section 7 we draw some general conclusions about the suitability of the B method for the sort of exercise reported in this paper, namely that of building and subsequently consistency-checking a structured formal state-based model intended to serve as a reference model in some area of systems engineering.

2 Role-based Access Control

Access control has existed as long as human beings have had assets they wanted to protect. Locks, gates, and guards are all forms of access control. Access management systems are usually perceived as consisting of three parts: **authentication**, for establishing the identity of the user; **authorisation**, for determining the resources that the user is permitted to use; and **administration**, for recording and maintaining the desired configurations of privileges and prohibitions.

Since the 1970s computer-based access control and security have become increasingly important with the increase in large computer-based resource-sharing systems in both government and commerce. An **authorisation system** is concerned with ways in which users can access resources within such a resource-sharing system. Informally, one might say it addresses the question of *who* is allowed to do *what* to *which*. The users can be human end-users or other computer systems. The challenge for an authorisation system designer is essentially that of maximising administrative efficiency whilst allowing sufficient granularity in the granting of permissions. Over the last two decades the so called **role-based access control** (RBAC) model of authorisation has emerged as the predominant architecture for authorisation systems. In this model administrative efficiency is achieved by assigning permissions to groups of users. Permissions are allocated to *roles*, and then individual users are assigned these various roles as appropriate. In this way, a user can acquire a complex collection of permissions. Furthermore, *role inheritance* can allow a user also to acquire permissions allocated to any ancestors of the roles he has been explicitly assigned.

2.1 Basic Entities of Role-based Access Control

As has already been indicated, the concept of role-based access control is predicated on our recognition of several basic entities, which we shall now characterise in more detail.

A **user** is usually thought of as a human being. However, the concept of a user can be extended to include machines, networks, or intelligent autonomous agents. A **role** is a job function within the context of an organization with some associated

semantics regarding the authority and responsibility conferred on the users to which it is assigned. A **permission** is an approval associated with one or more roles to perform an operation on a protected **object**, where an operation is an executable image of a program which upon invocation executes some function for the user.

A **session** is a series of authorised interactions with a system undertaken by an authenticated user, which is initiated by the authentification of the user and activation of one or more of their assigned roles, and subsequently terminated either by explicit action of the user or autonomously by the system itself.

2.2 Background to the ANSI RBAC Standard

Various RBAC-oriented models were originally proposed sharing similar fundamental features while also exhibiting some significant differences. However, the growing importance of role-based authorisation systems led to the desirability of the RBAC model's standardisation to secure a stable foundation for product development, evaluation, and procurement specification. One particular candidate RBAC model [5,4] developed under the auspices of the American National Institute for Standards and Technology (NIST) eventually secured widespread endorsement within the access-control community. After successfully negotiating all the necessary stages of the adoption process of the American National Standards Institute (ANSI), this model was duly adopted as the ANSI RBAC standard [6] on the recommendation of ANSI's International Committee for Information Technology Standards (INCITS).

ANSI

$$AssignUser(user, role: NAME) \lhd$$
$$user \in USERS; role \in ROLES; (user \mapsto role) \notin UA$$
$$\forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(ssd) \\ |subset| = ssd\_card(ssd) \\ us = \text{if } r = role \text{ then } \{user\} \text{ else } \varnothing}} (assigned\_users(r) \cup us) = \varnothing$$
$$UA' = UA \cup \{user \mapsto role\}$$
$$assigned\_users' = assigned\_users \setminus \{role \mapsto assigned\_users(role)\} \cup \{role \mapsto (assigned\_users(role) \cup \{user\})\} \rhd$$

Fig. 1. The AssignUser Operation in the RBAC Standard

As one might expect, this RBAC standard expresses its key concepts formally in terms of discrete mathematics and first-order predicate logic. Indeed, for this purpose it employs a mathematical notation loosely based on Z [9]. Unlike orthodox Z, though, the RBAC standard's formal mathematical notation is unfortunately very challenging for even the relatively mathematically-sophisticated reader to decipher, as the operation specification in Fig 1 taken from [6] clearly demonstrates.

2.3 Classification of RBAC Operations

The ANSI RBAC standard usefully divides the actual operations by which both an RBAC system's administrator and its clients interact with an RBAC authorisation system into three distinct categories. These are

(i) **administrative** operations, by which the administrator maintains his authorisation system's persistent information about users, assigned roles, role inheritances and permissions;

(ii) **supporting system** operations, by which transitory operational information about sessions and activated roles is created, managed and subsequently discarded;

(iii) **review** operations, by which the authorisation system can be interrogated about both its transitory and persistent information.

3 The ANSI RBAC Reference Models

The ANSI RBAC standard [6] actually defines a sequence of reference models, where each successive model in the sequence inherits all the significant features of its predecessors as well as introducing one or more new features of its own. We start by considering the core RBAC reference model, then we consider the hierarchical and finally the constrained RBAC reference models.

3.1 Core RBAC

The sets and relations of the **core** RBAC reference model [2] are shown in Fig 2. They includes five basic set types called USER, ROLE, OPN (operation), OBJ (object), and SESSION. The set type PRM (permission), on the other hand, clearly need not be introduced as a separate type as the RBAC standard as represented in [6] appears to do, since it can be formulated quite adequately simply as the cartesian product OPN x OBJ of existing types OPN and OBJ.

The core RBAC model imposes the constraint that a role can be activated within a session only if it has already been explicitly assigned to that session's user.

3.2 Hierarchical RBAC

The **hierarchical** RBAC reference model introduces role hierarchies through a parent-child relationship pc between roles as indicated in Fig 2. In this model a user acquires the permissions allocated to the ancestors in the parent-child role hierarchy of the roles which have been explicitly assigned to him, in addition to the permissions of those explicitly assigned roles. For example, a user assigned the role

[2] This is also sometimes called the "**flat**" RBAC model, as for example in [2].

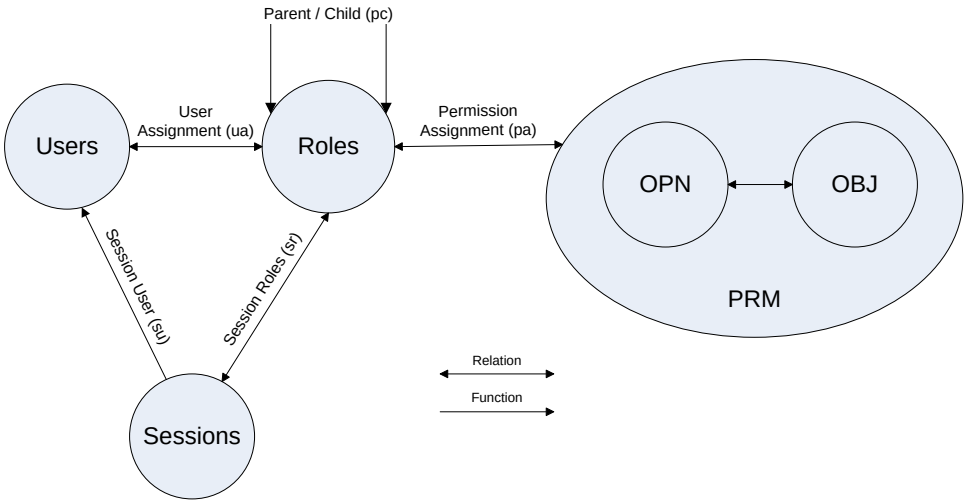


Fig. 2. RBAC Schematic

of *paediatrician* will, in addition to being entitled to exercise all the permissions specifically allocated to that role, also be entitled to exercise all the permissions allocated in general to its ancestor role of *doctor*.

The hierarchical RBAC model therefore relaxes the core RBAC model's constraint that every role activated in a session must previously have been explicitly assigned to that session's user. Instead it requires only that each role activated within a session is an ancestor within the role hierarchy of some role explicitly assigned to that session's user.

3.3 Constrained RBAC

The so called **constrained** RBAC reference model places further constraints on permissions which can be assigned to a user. Two sorts of constraints which are most commonly applied are the so called **static separation of duties** (ssd) in which the number of roles which can be authorised to a user out of each of certain specified sets of roles is limited, and **dynamic separation of duties** (dsd) in which the number of roles which an be concurrently exercised by a user in a session out of each of certain specified sets of roles is limited.

4 RBAC Specification in B

The basic building block of specification in B [1] is the **abstract machine**. An abstract machine is usually a specification of part of a system. A machine essentially expresses an encapsulated data type. It has a name, some internal state and a set of operations. Machines can be included in other machines. The B notation provides an extremely useful distinction between what we might call *primary* state variables, which are declared in the VARIABLES clause of an abstract machine and are subject to update as the system evolves during use, and what we can call *secondary* state variables, which are declared in the DEFINITIONS clause of an abstract machine.

Such secondary variables can be referenced (read) but are not subject to explicit update since their values change only in accordance with those of the primary state variables in terms of which they are defined.

Fig 3 shows the dependency graph for the B machines we have developed to express the ANSI RBAC standard.

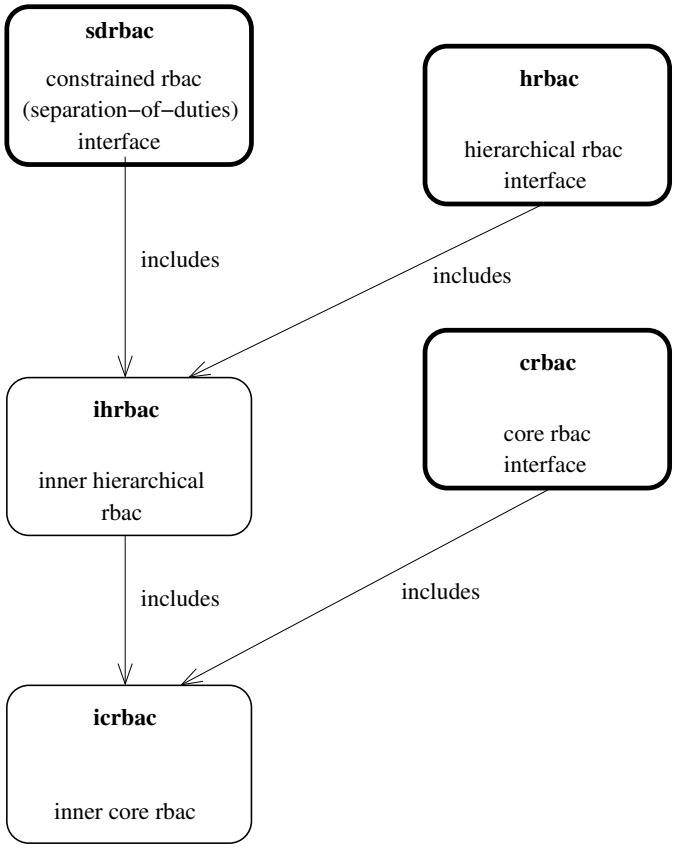


Fig. 3. The RBAC B machines dependency graph

We have adopted the convention that operations in these machines whose names begin with X are *internal* operations: such operations in an included machine may be invoked by the operations of the including machine, but do not feature as part of the external interface of any of the RBAC reference models.

Because of limitations of space all the abstract machines in this paper are displayed, as in Figs 4, 5, 6, 7 and 8, in abbreviated form showing only a representative sample of two specified operations in each machine. Full listings of the same machines with all their operations are available in [3].

4.1 The Inner-core RBAC Machine

The inner-core RBAC **icrbac** machine shown in Fig 4 captures the essential functionality of the core RBAC part of the ANSI RBAC standard described in Section 3.1. In contrast to the latter, however, which merely introduces a single ground type NAME from which all the fundamental entities such as users, roles, operations,

MACHINE *icrbac*
SEES *Bool_TYPE*
SETS *USER* ; *ROLE* ; *OPN* ; *OBJ* ; *SESSION*
VARIABLES *Users* , *Roles* , *ua* , *pa* , *su* , *sr*
DEFINITIONS
$PRM \mathrel{\widehat{=}} OPN \times OBJ$;
$Sessions \mathrel{\widehat{=}} \mathsf{dom}(su)$; $UserSessions(u) \mathrel{\widehat{=}} su^{-1}[\{u\}]$;
$Activeroles(s) \mathrel{\widehat{=}} sr[\{s\}]$; $Rolesactive(u) \mathrel{\widehat{=}} sr[su^{-1}[\{u\}]]$;
$Assignedroles(u) \mathrel{\widehat{=}} ua[\{u\}]$; $Assignedusers(r) \mathrel{\widehat{=}} ua^{-1}[\{r\}]$;
$Assignedperms(r) \mathrel{\widehat{=}} pa^{-1}[\{r\}]$; $Permsassigned(s) \mathrel{\widehat{=}} pa^{-1}[s]$
INVARIANT
$Users \subseteq USER \wedge Roles \subseteq ROLE \wedge ua \in Users \leftrightarrow Roles \wedge pa \in PRM \leftrightarrow Roles \wedge$
$su \in SESSION \nrightarrow Users \wedge sr \in Sessions \leftrightarrow Roles$
INITIALISATION
$Users , Roles , ua , pa , su , sr := \{\} , \{\} , \{\} , \{\} , \{\} , \{\}$
OPERATIONS
XAssignUser (*uu* , *rr*) $\mathrel{\widehat{=}}$
PRE $uu \in Users \wedge rr \in Roles - Assignedroles(uu)$
THEN $ua := ua \cup \{uu \mapsto rr\}$
END ;
$bb \longleftarrow$ **CheckAccess** (*ss* , *op* , *ob*) $\mathrel{\widehat{=}}$
PRE $ss \in Sessions \wedge op \in OPN \wedge ob \in OBJ$
THEN $bb := \mathsf{bool}(op \mapsto ob \in Permsassigned(Activeroles(ss)))$
END ;
...
END

Fig. 4. The inner-core RBAC machine

objects and sessions must take their values, our icrbac machine introduces quite distinct deferred type sets for each of these entities. This immediately brings improved clarity to the model and provides an appropriate foundation for its much more extensive consistency checking. We treat a permission as an ordered pair of an operation and an object, and therefore derive a non-basic type set PRM for permissions, defined as the cartesian product OBJ x OPN of our object and operation type sets.

The primary state variables of the icrbac machine, which directly reflect all the entities and relations of Fig 2 other than the parent-child relation between roles which doesn't feature in core RBAC, are as follows:

Users the set of users currently known to the system;

MACHINE *crbac*
INCLUDES *icrbac*
PROMOTES
AddUser , *DeleteUser* , *AddRole* , *GrantPermission* , *RevokePermission* , *SessionPermissions* , *DeactivateRole* , *CheckAccess* , *AssignedUsers* , *AssignedRoles* , *SessionRoles* , *DeleteSession*
INVARIANT
$sr \subseteq (su ; ua)$
OPERATIONS
AssignUser (*uu* , *rr*) $\widehat{=}$
PRE $uu \in Users \wedge rr \in Roles - Assignedroles (uu)$
THEN *XAssignUser* (*uu* , *rr*)
END ;
$Rp \longleftarrow$ **RolePermissions** (*rr*) $\widehat{=}$
PRE $rr \in Roles$
THEN $Rp := Assignedperms (rr)$
END ;
...
END

Fig. 5. The core RBAC interface machine

Roles	the set of roles currently known to the system;
ua	the relation between users and their currently assigned roles;
pa	the relation between permissions and the roles to which these have been assigned;
su	the function associating each current session with its user;
sr	the relation between current sessions and their activated roles.

No constraint is imposed by the icrbac machine to reconcile the activated roles of a session with the assigned roles of the session's user. Such a constraint will be imposed, however, by the relevant including machine when the icrbac machine is included either in the crbac external interface machine or the inner-hierarchical rbac ihrbac machine.

4.2 The Core RBAC Interface Machine

The purpose of the core RBAC **crbac** interface machine shown in Fig 5 is simply to provide an external shell for the inner-core RBAC icrbac machine, thereby providing developers with a functional specification of the core or "flat" RBAC standard model without role hierarchies or separation of duties.

The crbac machine includes the icrbac machine but introduces no further primary or

secondary state variables of its own. In its INVARIANT clause it does however impose the important further constraint $sr \subseteq (su; ua)$ on the variables of the included icrbac machine. This ensures that only roles which are already assigned to a session's user can be activated in that session. Most of the operations offered by the crbac machine are directly promoted from those of the included icrbac machine, while the remainder are mainly just renamings of various internal operations of the latter.

4.3 The Inner-hierarchical RBAC Machine

The inner-hierarchical RBAC **ihrbac** machine shown in Fig 6 captures the essential functionality of the hierarchical RBAC part of the ANSI RBAC standard described in Section 3.2. It inherits all the essential features of the core RBAC model by including the inner-core icrbac machine, and in its VARIABLES clause it introduces the new primary state variable ***pc*** denoting the parent-child relation between roles which realises the role hierarchy. Also, the icrbac machine in its DEFINITIONS introduces the following secondary state variables which are based on the *pc* parent-child primary variable:

- *ad* — the ancestor-descendant relation between roles, defined as the reflexive transitive closure pc^* of the parent-child relation *pc*; we note that this definition implies that every role is considered to be one of its own descendants;
- *da* — the descendant-ancestor relation between roles, defined as the relational inverse of the ancestor-descendant relation *ad*; we note that this definition implies that every role is considered to be one of its own ancestors;
- *uauth* — the relation between users and their *authorised* roles, *i.e.* the ancestors of their assigned roles, defined as the relational composition $(ua; da)$ of the user assignment relation *ua* with the descendant-ancestor relation *da*.

To avoid the nonsensical situation of a role being its own non-reflexive descendant the parent-child role hierarchy relation *pc* clearly must be non-cyclic in the sense that it contains no non-trivial cycles. To express this constraint the icrbac machine introduces in its DEFINITIONS clause the generic function ***cycles*** which maps any homogeneous relation[3] r to the cyclic core of its non-reflexive relational transitive closure. This is defined by

$$cycles(r) \;\widehat{=}\; \mathsf{id}\,(\mathsf{dom}\,(r)\,) \cap \bigcup ii \;.\; (\; ii \in \mathbb{N}_1 \;\mid\; (r)^{ii})$$

The icrbac machine then ensures that the parent-child role hierarchy relation *pc* is non-cyclic in this sense by constraining its transitive cyclic core to be empty by means of the assertion in its INVARIANT clause that $cycles(pc) = \{\}$.

[3] A relation is *homogeneous* if its source and target sets are the same.

MACHINE *ihrbac*
SEES *Bool_TYPE*
INCLUDES *icrbac*
PROMOTES
AddUser , *DeleteUser* , *AddRole* , *XAssignUser* , *GrantPermission* ,
RevokePermission , *DeleteSession* , *DeactivateRole* , *CheckAccess* ,
AssignedUsers , *AssignedRoles* , *SessionRoles* , *SessionPermissions*
VARIABLES *pc*
DEFINITIONS
$ad \mathrel{\widehat{=}} pc^* \;;\; da \mathrel{\widehat{=}} pc^{*\,-1} \;;\; uauth \mathrel{\widehat{=}} (ua \,;\, pc^{*\,-1}) \;;$
$Authorisedroles (u) \mathrel{\widehat{=}} ua \,;\, pc^{*\,-1} [\{ u \}] \;;$
$Authorisedusers (r) \mathrel{\widehat{=}} pc^* \,;\, ua^{\,-1} [\{ r \}] \;;$
$Authorisedperms (r) \mathrel{\widehat{=}} (pa \,;\, pc^*)^{-1} [\{ r \}] \;;$
$Permsauthorised (s) \mathrel{\widehat{=}} (pa \,;\, pc^*)^{-1} [s] \;;$
$cycles (r) \mathrel{\widehat{=}} \mathsf{id} (\mathsf{dom} (r)) \cap \bigcup ii \,.\, (ii \in \mathbb{N}_1 \mid (r)^{ii})$
INVARIANT
$pc \in Roles \leftrightarrow Roles \wedge cycles (pc) = \{\} \wedge sr \subseteq (su \,;\, uauth)$
INITIALISATION
$pc := \{\}$
OPERATIONS
AddDescendant (*rp* , *rc*) $\mathrel{\widehat{=}}$
PRE $rp \in Roles \wedge rc \in ROLE - Roles$
THEN $AddRole (rc) \parallel pc := pc \cup \{ rp \mapsto rc \}$
END ;
$Rp \longleftarrow$ **RolePermissions** (*rr*) $\mathrel{\widehat{=}}$
PRE $rr \in Roles$
THEN $Rp := Authorisedperms (rr)$
END ;
. . .
END

Fig. 6. The inner-hierarchical RBAC machine

4.4 The Hierarchical RBAC Interface Machine

The hierarchical RBAC **hrbac** interface machine shown in Fig 7 is an external shell for the inner-hierarchical RBAC ihrbac machine, thereby providing developers with a functional specification of the role-hierarchical RBAC standard model without separation of duties, as described in Section 3.3.

The hrbac machine includes the inner-hierarchical RBAC ihrbac machine, which in

MACHINE *hrbac*
SEES *Bool_TYPE*
INCLUDES *ihrbac*
PROMOTES
AddUser , *DeleteUser* , *AddRole* , *DeassignUser* , *GrantPermission* , *RevokePermission* , *DeleteInheritance* , *AddAscendant* , *AddDescendant* , *DeleteSession* , *DeactivateRole* , *CheckAccess* , *AssignedUsers* , *AssignedRoles* , *AuthorisedUsers* , *AuthorisedRoles* , *RolePermissions* , *UserPermissions* , *SessionRoles* , *SessionPermissions* , *RoleOperationsOnObject* , *UserOperationsOnObject*
OPERATIONS
DeleteRole (*rr*) $\widehat{=}$
PRE $rr \in Roles - \mathsf{ran}\ (\ pc\ )$
THEN *XXDeleteRole* (*rr*)
END ;
AddInheritance (*rp* , *rc*) $\widehat{=}$
PRE $rp \in Roles \wedge rc \in Roles \wedge rp \mapsto rc \notin pc \wedge rp \mapsto rc \notin da$
THEN *XAddInheritance* (*rp* , *rc*)
END ;
...
END

Fig. 7. The hierarchical RBAC interface machine

turn includes inner-core RBAC icrbac machine. The hrbac machine introduces no further primary or secondary state variables of its own, and most of the operations it offers are directly promoted from those of the included ihrbac machine, while the remainder are mainly just renamings of various internal operations of the latter.

4.5 The Separation-of-duties RBAC Machine

The separation-of-duties RBAC **sdrbac** machine shown in Fig 8 captures and expresses all the features of the full "constrained" RBAC model in the ANSI standard described in Section 3.3.

The sdrbac machine inherits all the essential features of the role-hierarchical RBAC model by including the inner-hierarchical ihrbac machine. Also in its VARIABLES clause it introduces two new primary state variables ***ssd*** and ***dsd*** needed to characterise the static and dynamic separation-of-duties constraints. These denote the following:

ssd a partial function which associates with various finite sets of roles a limit on the number from each of those sets which may be

MACHINE *sdrbac*
INCLUDES *ihrbac*
PROMOTES
AddUser , *DeleteUser* , *AddRole* , *DeassignUser* , *GrantPermission* , *RevokePermission* , *DeleteInheritance* , *AddAscendant* , *AddDescendant* , *DeleteSession* , *DeactivateRole* , *CheckAccess* , *AssignedUsers* , *AssignedRoles* , *AuthorisedUsers* , *AuthorisedRoles* , *RolePermissions* , *UserPermissions* , *SessionRoles* , *SessionPermissions* , *RoleOperationsOnObject* , *UserOperationsOnObject*
VARIABLES *ssd* , *dsd*
INVARIANT
$ssd \in \mathbb{F}_1 \ (\ Roles \ ) \pfun \mathbb{N}_1 \ \land \forall \ rs \ . \ (\ rs \in \mathsf{dom} \ (\ ssd \ ) \Rightarrow ssd \ (\ rs \ ) < \mathsf{card} \ (\ rs \ ) \ ) \land$
$\forall \ (\ uu \ , \ rs \ ) \ . \ (\ uu \in Users \land rs \in \mathsf{dom} \ (\ ssd \ ) \Rightarrow$
$\mathsf{card} \ (\ \{ \ uu \ \} \lhd uauth \rhd rs \ ) \leq ssd \ (\ rs \ ) \ ) \land$
$dsd \in \mathbb{F}_1 \ (\ Roles \ ) \pfun \mathbb{N}_1 \ \land \forall \ rs \ . \ (\ rs \in \mathsf{dom} \ (\ dsd \ ) \Rightarrow dsd \ (\ rs \ ) < \mathsf{card} \ (\ rs \ ) \ ) \land$
$\forall \ (\ ss \ , \ rs \ ) \ . \ (\ ss \in Sessions \land rs \in \mathsf{dom} \ (\ dsd \ ) \Rightarrow$
$\mathsf{card} \ (\ \{ \ ss \ \} \lhd sr \rhd rs \ ) \leq dsd \ (\ rs \ ) \ )$
INITIALISATION
$ssd \ , \ dsd := \{\} \ , \ \{\}$
OPERATIONS
AssignUser (*uu* , *rr*) $\widehat{=}$
PRE $uu \in Users \land rr \in Roles - Assignedroles \ (\ uu \ ) \land$
$\forall \ rs \ . \ (\ rs \in \mathsf{dom} \ (\ ssd \ ) \Rightarrow$
$\mathsf{card} \ (\ \{ \ uu \ \} \lhd (\ ua \cup \{ \ uu \mapsto rr \ \} \ ; \ da \ ) \rhd rs \ ) \leq ssd \ (\ rs \ ) \ )$
THEN *XAssignUser* (*uu* , *rr*)
END ;
CreateDSDSet (*rs* , *nn*) $\widehat{=}$
PRE $rs \in \mathbb{F}_1 \ (\ Roles \ ) \land nn \in \mathbb{N}_1 \ \land rs \notin \mathsf{dom} \ (\ dsd \ ) \land nn < \mathsf{card} \ (\ rs \ ) \land$
$\forall \ ss \ . \ (\ ss \in Sessions \Rightarrow \mathsf{card} \ (\ \{ \ ss \ \} \lhd sr \rhd rs \ ) \leq nn \ )$
THEN $dsd \ (\ rs \ ) := nn$
END ;
...
END

Fig. 8. The separation-of-duties RBAC machine

simultaneously assigned to the same user;

dsd a partial function which associates with various finite sets of roles a limit on the number from each of those sets which can be simultaneously activated in the same user session.

5 Consistency Checking

Each abstract machine in our RBAC B specification is a mathematically precise construct exhibiting both **static** features, notably the invariant properties of its state variables as expressed in its INVARIANT clause, and **dynamic** features as expressed in its INITIALISATION and OPERATIONS clauses. When we create such a machine we incur an obligation to show that these dynamic features are consistent with its static ones. Specifically, we must show that the machine's initialisation establishes its invariant, and that its operations, when invoked within their stated preconditions, preserve its invariant.

```
/* the empty relation is cycle-free */

      cycles({})={} ;

/* any reduction of a cycle-free relation is cycle-free */

      cycles(r)={}
     =>
      cycles(r-t)={} ;

/* Three safe extensions of a cycle-free relation */

      cycles(r)={} &
      not(y:dom(r))
     =>
      cycles(r\/{x|->y})={} ;

      cycles(r)={} &
      not(x:ran(r))
     =>
      cycles(r\/{x|->y})={} ;

      cycles(r)={} &
      not(x|->y : closure(r)~)
     =>
      cycles(r\/{x|->y})={}
```

Fig. 9. Proof rules for reasoning about cycles

The B method's supporting tools include both a **proof-obligation generator**, which will generate an appropriate set of first-order **proof obligations**, and automated and interactive **provers** for discharging these. These provers' in-built library of inference rules can be augmented by further user-supplied inference rules, called **user proof rules**, to capture some specific theory of the mathematical structures employed in the specification in order to facilitate reasoning about them.

For example, the user proof rules shown in Fig 9 embody certain general properties of non-cyclic relations which were used to discharge the proof obligations of the

inner-hierarchical ihrbac machine of Section 4.3. Such user proof rules are usually verified manually, although since each of them stands on its own quite independently of any B specification it is being used to verify (indeed quite independently of the B Method at all), they are themselves in principle each amenable to mechanical verification by a general-purpose theorem-prover such as Isabelle/HOL [7] or PVS [8].

6 Benefits of Recasting the RBAC Standard in B

In comparing our RBAC B specification which we have presented in the previous sections of this paper with the Z-like presentation of the ANSI RBAC standard in [6], several significant advantages of the B method are evident. We summarise these in the following list.

- B is almost unique among formal methods in providing an intuitively appealing imperative style of expressing operations using generalised substitutions without forgoing any of the power of abstraction associated with the relational style of operation specification employed by other formal methods such as Z. This is born out by the fact that we found it quite straightforward to express as B operations all the operations we encountered in the ANSI RBAC standard in [6] without introducing any implementation bias whatsoever. We contend that B's imperative style of operation specification should be much more readily comprehensible to product developers and other users of the RBAC standard than the Z relational style encountered in [6].

- The distinction in B between *primary* state variables, which are declared in an abstract machine's VARIABLES clause, and *secondary* state variables, declared in its DEFINITIONS clause, affords us major simplifications in expressing operations, since secondary variables never need to be explicitly updated. For example, in the inner-core RBAC icrbac machine, the secondary variables $Assignedusers$, $Assignedperms$, $Sessions$ and $UserSessions$ are never explicitly updated. This is in significant contrast to the situation in [6], where the need to update the corresponding variables there explicitly significantly complicates many operation specifications;

- The same point is perhaps even more forcefully evident in the inner-hierarchical RBAC ihrbac machine. Here all those operations which modify the role hierarchy only have to explicitly update the immediate parent-child relation denoted by the promary state variable pc. The associated secondary state variables ad, da and $uauth$ are all then automatically modified accordingly without need of explicit update.

- The B method's specification-structuring contruct INCLUDES has proved ideal in allowing us to express our sequence of three RBAC reference models incrementally.

- The B method's tool support facilitates both the generation and subsequent discharge, through either automatic or interactive proof, of the proof obligations which guarantee the mathematical consistency of our specification of each of the RBAC reference models. In contrast the reprentation of the ANSI RBAC standard in [6] does not lend itself nearly so readily to such machine-assisted verification. Sure enough, our attempts to prove the consistency of our first versions of our B specifications, which were essentially direct translations into B of the Z-like descriptions in [6], did indeed reveal certain significant inconsistencies and inadequacies in those descriptions which we will describe in the next section.

6.1 Anomalies in the ANSI RBAC Standard

As already indicated, a significant number of inconsistencies and other anomalies in the ANSI RBAC standard as represented in [6] came to light in the course of our reformulation of it in B. The most commonly occurring cause of these was failure to update all the state variables affected by an operation. We list here some of the more important anomalies we encountered.

- In the ANSI RBAC standard the variable $sessions\_users$ is sometimes used in the core RBAC specification although it is $user\_sessions$ which features in all diagrams and is updated in all its operation specifications. Either all references to $session\_users$ should be discarded in favour of $user\_sessions$ or else this variable should be updated in the operations along with $user\_sessions$.

- The ANSI RBAC standard is confused in how it handles role inheritance: first a general partial order on roles is introduced called the **inheritance relation** –corresponding, we can safely assume, to our derived ancestor-descendant (ad) relation between roles– but then an immediate inheritance relationship is also introduced, corresponding obviously to our parent-child (pc) relation between roles; some operations affecting role inheritances, such as AddInheritance then go on to update the general inheritance relation but not the immediate one, while confusingly others such as DeleteInheritance do update the latter.

- Although the DeleteRole operation is valid as given in the core RBAC ANSI standard, it becomes invalid when it is subsequently promoted into the hierarchical RBAC standard, since there it is necessary to ensure a role no longer features in role inheritances before it can safely be deleted.

- Similarly, the DeassignUser operation in the core RBAC ANSI standard, which deassigns a user from one of his assigned roles, becomes invalid when promoted into the hierarchical RBAC standard in the absence of a further precondition to ensure that the user has no currently active role whose authorisation comes solely from being an ancestor of the one now being deassigned.

- In the constrained (separation-of-duties) RBAC standard it is if not strictly mathematically essential then at least highly desirable on grounds of comprehensibility

that a role being deleted should also be expunged from all the static and dynamic separation-of-duties sets in which it appears, but this doesn't appear to happen in the RBAC standard when a role is deleted.

There are further indisputable errors in the ANSI RBAC standard. For example the definition of *authorized_permissions* is the wrong way round in regard to the direction of the role inheritance relation. Also, the operations DeleteSession and DropActiveRole should require only *session* as their input parameter, rather than both *user* and *session*. Similarly, the operation CreateSession requires only *user*, *session* and the active role set *ars* as its input parameters.

6.2 Other Clarifications of the RBAC Standard

It has already been explained how the B method's support for secondary state variables via an abstract machine's DEFINITIONS clause significantly simplifies the specification of many RBAC operations by relieving the specifier of responsibility explicitly to express how such variables are changed by the operation. Similarly, our grounding of the basic RBAC entities in distinct underlying type sets as described in Section 4.1, in contrast to the approach in [6] which grounds them all on the same basic type set NAME, greatly enhances the degree of type-checking to which our RBAC B specification can be subjected with the B method's automatic type-checker.

A further substantial simplification has been achieved in our RBAC specification in B *vis-a-vis* the RBAC standard in [6] in the way separation-of-duties information is represented. We encode each of the static and dynamic separation-of-duties constraints by a positive-integer-valued partial function from non-empty finite subsets of roles. This is significantly simpler than in [6] where the static case is encoded as a collection SSD on which the functions *dsd_set* and *dsd_card* are defined respectively to sets of roles and positive integers, while the dynamic case is correspondingly encoded as a collection DSD with a similar pair of functions *dsd_set* and *dsd_card*.

7 Conclusion

When we embarked on our recasting of the RBAC standard we of course already had considerable cause to expect that the B method would prove appropriate for such an exercise. Even so, we were agreeably surprised to discover through this work the actual degree to which the B method did indeed suit our purpose. Unquestionably, our own understanding of the RBAC standard has been greatly enhanced by the exercise. We have been able to improve the standard both by rectifying all the anomalies we discovered which we have described and by simplifying various aspects of its internal state representation to good effect. We believe four particular aspects of the B method have contributed perhaps more than any other to this satisfactory outcome:

(i) B's INCLUDES specification-structuring construct for abstract machines, which

proved very appropriate for the structuring our sequence of RBAC models;

(ii) B's conventional set-theoretic notation for expressing the static features a model, so well complemented by its less conventional but no less powerful imperative generalised-substitution notation for expressing the dynamic features of a model without forgoing any of the powers of abstraction usually associated only with relational notations;

(iii) B's DEFINITIONS facility allowing the introduction of secondary state variables for ease of expressibility and comprehensibility of both static and dynamic descriptions without incurring any obligation to describe how they should be updated;

(iv) B's powerful tool support which so greatly facilitates full type-checking and logical-consistency checking of our B machines.

Furthermore, we believe our experience in the exercise we have reported in this paper suggests that the B method should be more actively promoted as a standard modelling method in any engineering domain at all which features discrete-event models with state. That would of course open up for the B method a very wide potential range of applications indeed.

References

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

[2] Elisa Bertino. RBAC models – concepts and trends. *Computers & Security*, 22(6):511–514, 2003.

[3] S.E. Dunne and A. Howitt. Modelling role-based access control in B. 2008. Available as http://www-scm.tees.ac.uk/s.e.dunne/brbac.pdf.

[4] David F. Ferraiolo, Rick Kuhn, and Ravi Sandhu. RBAC Standard Rationale: comments on 'A Critique of the ANSI Standard on Role-Based Access Control'. *IEEE Security and Privacy*, 5(6):51–53, 2007.

[5] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[6] ANSI INCITS. Information technology - Role Based Access Control. (Standard Document 359-2004), 2004.

[7] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[8] S. Owre, J.M. Rushby, and N. Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *Automatic Deduction–CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[9] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.

On Teaching the Concept of Refinement with B

Joanna Tomasik Guy Vidal-Naquet

Computer Science Department
SUPELEC
Gif-sur-Yvette, France

Abstract

The concept of refinement is central to the development of software. It appears in various forms in the different methodologies taught to students. A key point in the B method is the validation of the refinement step. The B methodology exhibits mathematical properties of correct refinements, and also automatically checkable conditions that ensure those properties. Some of the main pedagogical difficulties that the present authors found in teaching B centered around the notions linked to refinement, at the conceptual level, and at the tool level. Many papers have been published on the general benefits of the B method. This paper will focus on the specific concepts linked to refinements, and on the ones which need special care. We argue that, although B presents a complete mathematical analysis, it is beneficial to put the concept of refinement in perspective with other theories that come from formal methods, namely, in this paper, coalgebra and bisimulation.

Keywords: Coalgebra, Gluing invariant, Morphism, Refinement.

1 Introduction: pedagogical aspects linked to positioning B in a software engineering cursus

The necessity of a methodology for producing quality software is not questioned nowadays. Such was not the case a few years ago, as shown by a study of the Software Engineering Institute (SEI) [12]. To our knowledge, this is the last available diagram that the SEI published which presents the evolution per year. It shows that during the years 1987–1991 more than eighty percent of software companies were at the initial or "chaotic" level of maturity. As one can see on the diagram in Fig. 1, in 2005, a few companies were still at this level. Nevertheless this diagram shows that the maturity level of firms has increased. One of the acknowledged reasons is the fact that engineers who have followed a cursus in software engineering and its different methods, including formal methods, are working now in the software development industry.

[1] Email: Joanna.Tomasik@supelec.fr
[2] Email: Guy.Vidal-Naquet@supelec.fr

We think that the B method is an adequate answer to the old (but still mentioned) joke:

Formal methods have been, are, and will always be, the future of software development.

and that it will help increase the figures in the rightmost sections of this diagram.

One of the first questions that arises with teaching B is whether it should be integrated into a general course on software engineering, or be taught in a separate course. In view of the conceptual richness of the method, and of the need for having practical development by students in order to show the usefulness of this methodology, we definitely opted for the second solution.

UML occupies a central position in the teaching of software engineering. Some semantic meanings can be associated with the graphical syntax of UML. Rational Rose is a tool that is often associated with RUP (Rational Unified Process) and UML. A recurrent problem encountered when teaching UML with Rose is that of coherence between the different phases of processes of development (for instance, with Rose, between Case View and Logical View). Students ask questions about possible contradictions which may occur in a developed system seen on different process stages without being detected. They are truly disappointed when learning that there is not any formal method for detecting inconsistencies. They are also perturbed by the lack of a rigorous methodology for the construction of UML diagrams.

In addition, the use of mathematically founded methods, such as Hoare logic, and the corresponding system of proof seems to students to be distant from real applications. Students are supposed to make Hoare proofs manually. The conclusion they draw from this activity, however, is that these proofs cannot be performed automatically and are unapplicable in real cases.

For these reasons, software engineering teachers are confronted with a quandary:

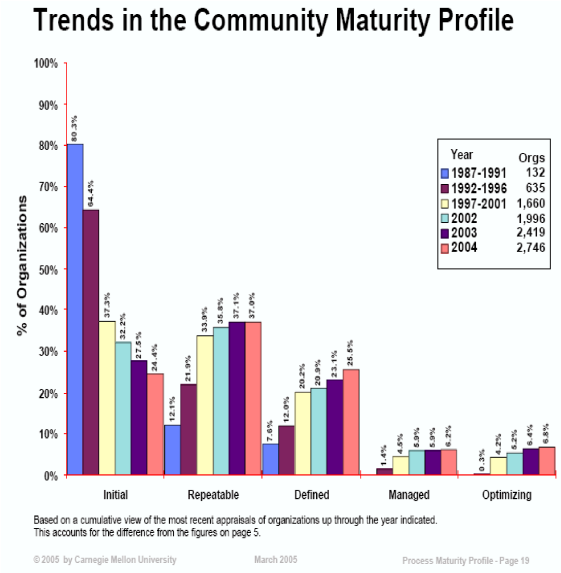


Fig. 1. Evolution of the maturity level

the need to use formal methods, and a lack of pedagogically effective ones. One important aspect of teaching B is that this apparent paradox disappears because the B methodology presents many concepts which have direct applications for the practical development of software.

In order to obtain a coherent curriculum, the teaching of B has to be related to the other courses on software engineering and formal methods. The common and specific aspects of B must be addressed not only in the course on B, but also in other courses. This means that in other courses mention of the way that B deals with specific questions, i.e. encapsulation, should be brought out, and in the specific course on B, the relationship with other methodologies should be examined.

Teaching the B method illustrates how the utilisation of formal methods contributes to software development's being an industrial activity. While important concepts, such as preconditions and offensive programming, are easily understood by students, we found that the real difficulties center around the notion of refinement and the meaning of the obligations of proof that are generated. In this paper, we will concentrate on the questions raised by refinement - not the concepts themselves, but on those aspects students find difficult and/or useful and on how to address these difficulties.

In our teaching, we use the AtelierB tool. We chose this tool, as opposed to B4free — available on the Internet without any licence — for several reasons. We want students to become familiar with the software that is used in the industry, but the most important reason for our choice is the possibility of generating, compiling and executing code in a programming language. We teach in an engineering school and our students are conscious of their future professional career development. They want to "see the things working".

We observed that students work seriously during classes concerning abstract machine definitions, refinement and project structure. The class in which students obtain their first executable programs leads to real enthusiasm. The difference in the attitude of the students before and after the obtention of executable code is striking. They recognize that all their efforts to reach an appropriate expression of logical properties lead to an executable program. They see this as a concrete outcome of the B development process. This observation makes them strongly motivated to progress in the B method.

Our students work on their individual projects, which have to be seen in an overall context. Students are asked to write a specification of a proposed system and follow the B development process to the end, i.e. an executable program, while paying attention to reducing the number of proof obligations that are not automatically proven. Due to the number of hours devoted to B, we cannot present the interactive prover in detail, but we insist that the students comment on the origin of the proof obligations generated by AtelierB. All of our students show a deep interest in having the best possible project and they invest a lot of time and energy in order to succeed. We are therefore convinced that the goal of obtaining the best code for an executable program stimulates them.

One of the aspects that students appreciate in the B methodology is the continuity in the development process, and, as mentioned before, the integration of formal methods. Predicate logic and the theory of proof system blend nicely with

their concrete development activities. Some residual questioning about the interest of the method and negative feedback are linked to the refinement step.

We found that beneficial effects result from spending more time on explaining the relationship between the states of a machine and the states of a refinement machine, as well as from formalizing the concept expressed informally by "a machine behaves like another". In order to do that in depth, we propose to make the links and differences between refinement and coalgebra theory explicit in the teaching cursus. We firmly believe that *THE* method does not exist, and that it is the duty of a teacher to show the relationship of what she/he teaches with the other aspects of a field.

Much of what we say is a rewriting in coalgebraic terminology of definitions and results present in the B methodology. This fact strengthen the ideas of the existence of links between the two approaches, and we believe that these links should be made explicit to the students.

The paper is organized as follows after this introduction. In Section 2, we present the fundamental concepts of coalgebas and morphisms between coalgebras. We do not enter into mathematical details, and we hope that this section will show the interest of this emerging field. In Section 3, we present the links between B machines and coalgebra. In Section 4, we study the link between refinement, gluing invariants and morphisms of coalgebras. Finally, in Section 5, we discuss methods for teaching coalgebra theory in relation with the B methodology.

2 Basic concepts on coalgebras: observation versus construction

Recently, a considerable amount of work on coalgebras was done, in order to provide a mathematical basis for the concept of observation in state based systems. The study of coalgebra was spurred by works on bisimulation for parallel systems. See [10] for a seminal paper on the use of coalgebras and their relationship with objects. The papers [7] and [8] give good, pedagogical introductions to the concepts and main results on coalgebras, with a short survey on relevant definitions of the theory of categories. In [3] coalgebras are used for defining the semantic of the Rosetta language. We found the concepts on coalgebras very well suited for teaching the different notions of refinement in a unified way.

Algebraic definitions will tell how a system is *built*, while coalgebraic definitions will tell how a system is *observed*. Quite often these two aspects are mixed. For example consider the specifications for lists. There will be

- a constuctive part: how to build a list from the empty list, adding an element,
- an observation part: i.e. test for emptiness, length, last element.

A basic concept for a system that is used (but has not necessarily been developed by the user) is that one does not know how the possible states are obtained, but from a given state one can get some information. It is the set of states and the correspondence between a state and the information on this state that defines a given coalgebra. This is also the approach followed for object-oriented software engineering.

For example, let us consider a traffic light system. The internal mechanism can consist only of a timer, so the light changes at regular time interval, or, in addition, it can use a car detection system. The user of the system ignores its internal mechanisms and can just observe the color of the light. In this example, we associate to a set of internal states S (which is not specified) a simple structure consisting of the set $L = \{\text{green}, \text{yellow}, \text{red}\}$ and we associate to each element of S an element of L.

More generally, we can associate to any set X of states of a system — *the carrier set* — a more complex structure $F(X)$ that possibly involves X. The structure $F(X)$ indicates what we can know about a state in X. The operation F that associates to any carrier set the structure of what can be observed is called a *signature*. Mathematically F is a functor, but this can be ignored for our purpose. A coalgebra $\langle S, c\rangle$ is a realisation of a signature F. S is a set which is a concrete "instance" of X and c is the observation function from S to $F(S)$.

Definition 2.1 Given a signature F, a coalgebra for F is a couple $\langle S, c\rangle$ where S is a carrier set of states and c is an observation function from S into $F(S)$.

Example 2.2 Consider the following signatures

(i) $F_1(X) = L$, with $L = \{\text{green}, \text{yellow}, \text{red}\}$.
A coalgebra $\langle S, c\rangle$ for F_1 is defined by $S = \mathbb{N}$ and c in defined as follows:
- $\forall n, 0 \leq n \bmod 30 \leq 15$, $c(n) = \text{green}$
- $\forall n, 16 \leq n \bmod 30 \leq 18$, $c(n) = \text{yellow}$
- $\forall n, 19 \leq n \bmod 30 \leq 29$, $c(n) = \text{red}$

How the state is determined is irrelevant here. It could be done with the help of an egg timer, an atomic watch or a deck of cards.

(ii) $F_2(X) = X$.
For a coalgebra $\langle S, c\rangle$, the observation of a state s is a state $c(s)$.

Generally, the interpretation of c is the next state, but it could be the preceding state, or a state that we want to avoid, or a state that we want to reach.

For example, $S = \{u, v, t, w\}$ the observation function c is given by $c(u) = v$, $c(v) = w$, $c(t) = t$, $c(w) = v$.

(iii) $F_3(X) = \{\bot\} \cup X$.
For a given coalgebra $\langle S, c\rangle$, the observation of a state s is a state $c(s)$, or the element $\bot$. Generally, the interpretation of $c(s)$ is the next state, if $c(s) \in S$. When s is a state that cannot evolve (a final state), $c(s) = \bot$.

(iv) $F_4(X) = \{\bot\} \cup (X \times A)$.
For a coalgebra $\langle S, c\rangle$, the observation of a state s is a set of pairs composed of a state $c(s)$ and a label in A, or the terminal element $\bot$. The usual interpretation is that if the system can evolve, then it is possible to determine in which state it will be and also the label attached to the transition from one state to another. This coalgebra describes a deterministic labelled transition system.

(v) $F_5(X) = \{\bot\} \cup P(X \times A)$,
where $P(X)$ is the set of all subsets of X. For a coalgebra $\langle S, c\rangle$, the observation of a state s is a set of pairs composed of a state $c(s)$ and a label in A, or

the terminal element $\perp$. This coalgebra describes a non-deterministic labelled transition system.

(vi) $F_6(X) = \{0,1\} \times X^A$.
It is the signature associated with automata: given a state, the function of A into S is given, and one can also know if the state is final or not.

The notion of morphism is central in mathematics. It captures the intuitive ideas of an operation that preserves structures. For coalgebras, we have the following definitions:

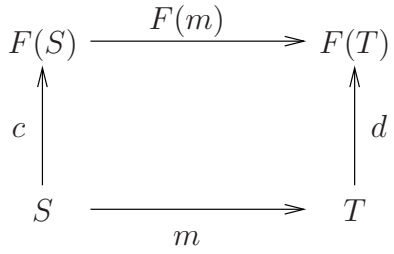
Definition 2.3 Let $\langle S, c\rangle$ and $\langle T, d\rangle$ be two coalgebras for the same signature F.

If m is a function from S into T, $F(m)$ is the function that extends m to an function from $F(S)$ into $F(T)$. Informally $F(m)$ is obtained by replacing elements of S by their image in the expression defining an element of $F(S)$.

For example, if — for two carrier sets $S = \{s, s'\}$, $T = \{t, t'\}$ and two label sets A, B such that $a \in A$, $b \in B$ — F is the signature defined by $F(S) = (S \times A) \times (S \times B)$, and $m(s) = t$, $m(s') = t'$, then $F(m)(((s, a), (s', b))) = ((t, a), (t', b))$.

A function m is a morphism from S into T, when it respects the observation functions: $d \circ m = F(m) \circ c$.

In other words, the function m is a morphism when the following diagram commutes:



Example 2.4 We follow the usual mathematical notation, $P(X)$ is the set of all subsets of X.
$F(X) = P(X \times \{a, b\})$
Let $S = \{s_1, s_2, s_3\}$, $c(s_1) = \{(s_2, a), (s_3, a)\}$, $c(s_2) = \{(s_3, b)\}$, $c(s_3) = \{(s_3, b)\}$,
$T = \{t_1, t_2\}$, $d(t_1) = \{(t_2, a)\}$, $d(t_2) = \{(t_2, b)\}$.

$\langle S, c\rangle$, and $\langle T, d\rangle$, can be graphically represented by the labelled transition systems, with m represented by the dotted line arcs (Figure 2). It is easy to check that the function $m(s_1) = t_1$, $m(s_2) = m(s_3) = t_2$ is a morphism. For example, starting from s_1 we have: $m(s_1) = t_1$, $d(m(s_1)) = \{(t_2, a)\}$. On the other hand: $c(s_1) = \{(s_2, a), (s_3, a)\}$, $F(m)(c(s_1)) = \{(t_2, a)\}$.

A main result which we will not develop here, but which is central to the coalgebra theory, is that if F is a "reasonable" signature then there exists a *final coalgebra* $\langle T, b\rangle$ for F, meaning that for any coalgebra $\langle S, c\rangle$ for F, there is a unique morphism from S into T. Intuitively, $\langle T, b\rangle$ represents all observations that can be made with the signature F

We refer to [8] or [7] for more details.

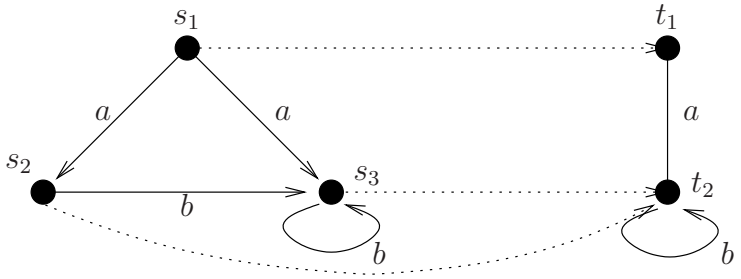


Fig. 2. Example of morphism between two coalgebras

A morphism from S to T corresponds to a deterministic way to establish a correspondence from S to T: given a state in S, one state in T is obtained. In general such a correspondence is non deterministic, i.e. to one object in S there correspond many objects in T. For example consider a LIFO queue implemented with an array and a number (the top of the stack). Intuitively a relation R between two sets S and T is a bisimulation for the signature functor F, when R "is carried through" by F. The formal definition is:

Definition 2.5 Let F be a signature, $\langle S, c\rangle$, $\langle T, d\rangle$ two coalgebreas for F, and $c : S \longrightarrow F(S)$, $d : T \longrightarrow F(T)$. A relation $R \subseteq S \times T$ is a *bisimulation* when there exists a coalgebra $\langle R, g\rangle$ such that the two projections $\pi_1 : R \longrightarrow S$ and $\pi_2 : R \longrightarrow T$ are coalgebra morphisms i.e. the following diagram commutes (Figure 3).

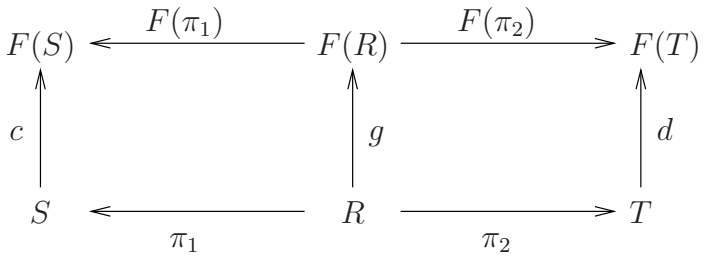


Fig. 3. Bisimulation of coalgebras

Example 2.6 Let $S = \{s_1, s_2, s_3\}$, $c(s_1) = \{(s_3, a)\}$, $c(s_2) = \{(s_3, a)\}$, $c(s_3) = \{(s_3, b)\}$, and $T = \{t_1, t_2, t_3\}$, $d(t_1) = \{(t_2, a), (t_3, a)\}$, $d(t_2) = \{(t_3, b)\}$, $d(t_3) = \{(t_3, b)\}$ (Figure 4).

The relation defined by $R = \{(s_1, t_1), (s_2, t_1), (s_3, t_2), (s_3, t_3)\}$ is a bisimulation.

Consider the function g from R into $F(R)$ defined by

$$g((s_1, t_1)) = \{((s_3, t_2), a), ((s_3, t_3), a)\},\ g((s_2, t_1)) = \{((s_3, t_2), a), ((s_3, t_3), a)\},$$
$$g((s_3, t_2)) = \{((s_3, t_3), b)\}, \qquad g((s_3, t_3)) = \{((s_3, t_3), b)\}.$$

It is easy but tedious to check that π_1 is a morphism from $\langle R, g\rangle$ into $\langle S, c\rangle$ and π_2 is a morphism from $\langle R, g\rangle$ into $\langle T, d\rangle$. In Figure 4, $\langle S, c\rangle$ and $\langle T, d\rangle$ can be viewed as two labelled transition systems, with the relation R shown by the dotted line. The notion of bisimulation in coalgebras is a generalization of the equivalent notions for CCS [9] and labelled transition systems [2]. The notion of simulation will play a central role in the way we present refinements.

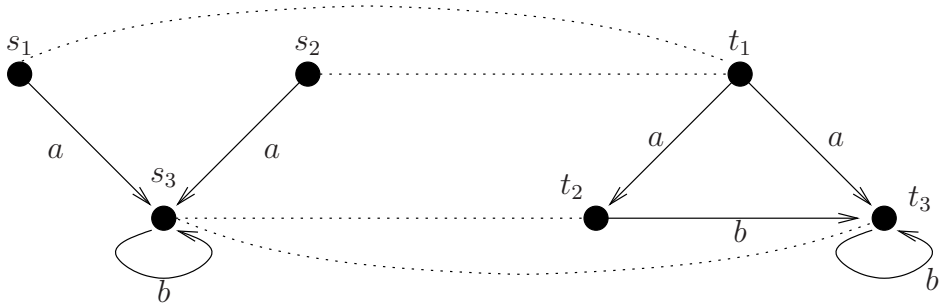


Fig. 4. Example of bisimulation between two coalgebras

When coalgebras correspond to non-deterministic labelled transitions systems (cf [2]), we have the notation $s \xrightarrow{\alpha} s'$ when $(\alpha, s') \in c(s)$ ($(\alpha, s') = c(s)$ in a deterministic case). For labelled transition systems, bisimulation corresponds to the fact that: for two labelled transition systems LS and LT with carrier states S and T, a relation R between S an T is a bisimulation when

If sRt then

(i) for all s' such that $s \xrightarrow{\alpha} s'$, there exists t', such that $t \xrightarrow{\alpha} t'$ and $s'Rt'$,

(ii) for all t' such that $t \xrightarrow{\alpha} t'$, there exists s', such that $s \xrightarrow{\alpha} s'$ and $s'Rt'$.

When only condition (i) holds, we say that R is a simulation of LS by LT. Thus we can see a simulation as "half of a bisimulation". This notion of simulation versus bisimulation can be extended to coalgebras. We will not go into the mathematical details about this and we refer to [11].

3 Links between B machines and coalgebras

The correspondence between B machines and signatures of coalgebras is quite natural. The notion of signature exists in B. We find it worthwhile to mention to the students the links between the B methodology and coalgebras. It helps convincing them that formal methods adress real problems and that there is not a unique way for decribing a method in mathematical terms.

We give an example with a machine with two operations `op1` and `op2`, where `op1` has a parameter and produces a result, while `op2` just changes the internal states which can easily be generalized. Consider the machine:

```
MACHINE MA
VARIABLES XX
INVARIANT IXX
INTIALISATION Z OPERATIONS
res <-- op1(NN) = PRE INN & C1 THEN S1 END;
op2 = PRE C2 THEN S2 END
END
```

The expression IXX is the invariant that defines the type `TXX` of the variable `XX`, INN is the expression that defines the type `TNN` of the parameter `NN` and `TRes` is the type of the result `res` of the operation. The signature of the coalgebra asssociated with the machine `MA` will be $F(X) = (\{\texttt{op1}\} \times (\{\bot\} \cup P(\texttt{TRes} \times X)^{\texttt{Tpar}})) \cup (\{\texttt{op2}\} \times$

$(\{\bot\} \cup P(X)))$. This signature allows us to observe sequences of operation executions and its final colgebra is a string. We consider $P(\texttt{TRes} \times X)^{\texttt{Tpar}}$ and $P(X)$ instead of $(\texttt{TRes} \times X)^{\texttt{Tpar}}$ and X, respectively, since *S1* and *S2* can be non deterministic.

Another possible signature could be $F(X) = (\{\texttt{op1}\} \times (\{\bot\} \cup P(\texttt{TRes} \times X)^{\texttt{Tpar}})) \times (\{\texttt{op2}\} \times (\{\bot\} \cup P(X)))$. This signature allows us to observe results of every operation for a given state and its final coalgebra is a tree.

A coalgebra for F is obtained when S corresponding to a set of the states defined by the invariant *IXX* and the observations is defined by the machine.

If a given state `XX` of the machine does not satisfy the precondition *C1*, one can observe that `op1` cannot be called and $c(\texttt{XX}) = (\texttt{op1}, \bot) \times \cdots$. If `op1`, whose parameter is in `Tpar`, is executed for a state `XX` then the machine passes to a new state i.e. the execution of *S1* attributes a new value to `XX` (which can be the same as the previous one) and the result is returned.

Let note that a signature does not give complete information on what the machine does, just the structure of what is produced. It depends on the observation one wants and the degree of knowledge on the process that one wants to observe. For example, if we are not interested in which operation is called, then for a machine having the two operations

```
res1 <-- op1(NN) = ...
res2 <-- op2(MM) = ...
```

If these two operation are deterministic, return results of the same type `TRes`, and their arguments have the same type `Tpar` then instead of the signature $F(X) = (\{\texttt{op1}\} \times (\{\bot\} \cup (\texttt{TRes} \times X)^{\texttt{Tpar}})) \cup (\{\texttt{op2}\} \times (\{\bot\} \cup (\texttt{TRes} \times X)^{\texttt{Tpar}}))$, it is possible to consider the signature $F(X) = \{\bot\} \cup (\texttt{TRes} \times X)^{\texttt{Tpar}}$. If an operation has no precondition (or a precondition equivalent to `TRUE`), then it is possible to have a signature of the form $F(X) = (\texttt{TRes} \times X)^{\texttt{Tpar}}$.

In the case of non-determinism of these opertations we replace $(\texttt{TRes} \times X)^{\texttt{Tpar}}$ by $P(\texttt{TRes} \times X)^{\texttt{Tpar}}$.

These exemples show that signature of coalgebras expresses what one **chooses** to observe.

Similarly to choosing the coalgebra associated with the machine, one can choose the set defined by the typing invariant and/or the set of reachable states induced by the initialisation and the operations.

4 Refinement, morphisms, and bisimulations

Consider the machine taken from [1]:

```
MACHINE M1
VARIABLES yy
INVARIANT yy:FIN(NAT1)
INITIALISATION yy:={}
OPERATIONS
enter(nn) = PRE nn:NAT1 THEN yy:=yy\/nn END;
mm <-- maximum = PRE yy /= {} THEN mm:=max(yy) END
END
```

The carrier set, defined by the invariant, is equal to $S = \texttt{FIN(NAT1)}$. The signature that is naturally associated with this machine is $F(X) = (\{\texttt{enter}\} \times X^{\texttt{NAT1}}) \times (\{\texttt{maximum}\} \times \texttt{NAT1})$. The coalgebra associated is $\langle S, c\rangle$ with $S = \texttt{FIN(NAT1)}$ and $c(s) = \{\texttt{enter}\} \times f_s \times \{\texttt{maximum}\} \times \texttt{max}(s)$, if $s \neq \emptyset$), where $f_s : \text{NAT1} \longrightarrow S$ defined by $f_s(n) = s \cup \{n\}$.

Now, consider this second machine also taken from [1]:

```
REFINEMENT R2
REFINES M1 VARIABLES zz
INVARIANT zz:NAT
INITIALISATION zz:=0

OPERATIONS
enter(nn) = PRE nn:NAT1 THEN zz:=max({nn,zz}) END;
mm <-- maximum = PRE zz /= 0 THEN mm:=zz END
END
```

We can associate the same signature F to this machine and the coalgebra $\langle T, d\rangle$ with $T = \texttt{NAT}$ and $d(z) = (\{\texttt{enter}\} \times g_z) \times (\{\texttt{maximum}\} \times (\bot \text{ if } z = 0,\ z \text{ if } z \neq 0)$, where $g_z : \texttt{NAT1} \longrightarrow T$ is defined by $g_z(n) = \max(n, z)$.

Consider the function m from the states S of M1 into the states T of R2 defined by $m(s) = 0$ if $s = \emptyset$ and $m(s) = \max(s)$ if $s \neq \emptyset$.

The initialisation of M1 produces the state where $\texttt{yy} = \emptyset$ and initialization of R2 produces the state where $\texttt{zz} = 0$. We have $m(\emptyset) = 0$. It is easy to see that m is a morphism from $\langle S, c\rangle$ into $\langle T, d\rangle$. In fact m is a bijection and m^{-1} is a morphism, therefore m is an isomorphism.

As a consequence of the fact that m is a morphism, the executions of the operations enter and maximum from corresponding states will produce corresponding states and the same outputs. This give a precise mathematical sense to the sentences "R2 behaves like M1", "M1 and R2 cannot be distinguished by their outputs".

A specific feature of this example is that the substitutions in both machines are deterministic. We will mention briefly some issues.

In the following we will suppose that all proof obligations concerning machines and their refinements have been discharged.

Consider a machine MM with a state space S defining an operation op and its refinement RR with a state space T. The gluing invariant R_{GI} establishes a relation R between S and T. The difference between the notion of refinement and the notion of bismulation can be expressed as follows:
For a refinement:
Given a state $s, s \in S$ such that op can be executed from state s and a state $t, t \in T$ such that $sR_{GI}t$, there exists s' and t' such that $\texttt{op}_{\texttt{MM}}(s) = s'$, $\texttt{op}_{\texttt{RR}}(t) = t'$, and $s'R_{GI}t'$.
For a simulation relation R_S:
Given state s and s' such that MM can execute op for state s and $s' \in \texttt{op}_{\texttt{MM}}(\texttt{s})$ and a state t such that sR_St, there exists t' such that $t' \in \texttt{op}_{\texttt{RR}}(\texttt{t})$ and $s'R_St'$.

Therefore, if RR refines MM, then there exists a simulation relation between the state spaces $T' \subseteq T$ and S. The relation R_{GI} induced by the gluing invariant

is contained in the simulation induced by the gluing invariant, with T' such that $T' = \texttt{dom}(\texttt{R}_{GI})$. The relaxation of the precondition for a refinement ensures that $S = \texttt{ran}(\texttt{R}_{GI})$. Intuitively, the relaxation of the precondition ensures that no state that should be taking into account by R_S. Operations returning values give the same results for corresponding states in S and T. We found that the gluing invariant plays a role analogous to a loop invariant in the Hoare proof system: When a developer writes a refinement of a machine he has the gluing invariant in his mind, in the same way as when he writes a loop, he has the loop invariant in his mind. Expliciting the invariant is one of the difficult points in the B methodology. The gluing invariant can be seen as indications (which can be a complete indication as in the example concerning machine `M1` and `R2`) about the simulation relation.

Finally let us say that we do not claim that one concept can take into account all aspects of the other, for example the question of the conservation of properties is not dealt with by coalgebra theory, but that there exists a lot of similarities that should be mentioned.

5 Conclusions

It is a banality to say that formal methods are indispensable. It is quite another matter to convince students of this fact. One of our goals is that students should become familiar with several usual formal modelling methods and their usage. It is with concrete examples of applications that students become aware of the use of such methods. Let us cite algebraic specification [4], first order logic and proof systems, model checking, CCS, Petri nets [5], Performance Evaluation Process Algebra (PEPA) [6]. All these models are taught with their applications, meaning that we integrate them with the courses dedicated to the studies of different systems and their associated software, when possible. These formalisms are used to treat different aspects of the modelled systems: feasibility, performance, correctness and observability, respectively. One of our goals is to show students that it is necessary to use these formal methods. The mathematics for observation of systems have reached a maturity level such that we find it necessary to introduce them in our curriculum for computer sciences. It is a natural complement to algebraic specifications, and we find that they help unify concepts for parallelism and distribution as well as concepts linked to refinement in B.

A question that still has to be answered in our school is at which point these mathematics should be taught. In an ideal teaching environment, it should be taught at the same time as algebraic specifications. Because of the time frame, this seems difficult, and a specific introduction to the subject can be given independently, before the course on B and on parallel systems.

A complete treatment of the relationships between coalgebra, bisimulation, and B machines, would be beyond the scope of this conference. We hope that we have given enough arguments to show that these relationships should be treated in a curriculum on software engineering.

References

[1] Jean-Raymond Abrial, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1966.

[2] Luca Aceto, Anna Ingòlfsdòttir, Kim Guldstrand Larsen, Jiri Srba, *Reactive Systems Modelling, Specification and Verification*, Cambridge University Press, 2007.

[3] Alexander, Cindy Kong, *Defining a Formal Coalgebraic Semantics for The Rosetta Specification Language*, Journal of Universal Computer Science, Vol. 9, no. 11 (2003), pp. 1322–1349.

[4] Marie-Claude Gaudel, Gille Bernot, Bruno Marre, Françoise Schlienger, *Précis de Génie Logicel*, Masson,1996.

[5] Annie Choquet-Geniet, *Les réseaux de Petri: Un outil de modélisation*, Dunod 2006.

[6] Jane Hillston. *A Compositional Approach to Performance Modelling*, Cambridge University Press, 1996.

[7] Bart Jacobs, *Draft: Introduction to Coalgebra. Towards Mathematics of States and Observations*, Institute for Computing and Information Sciences, Radboud University Nijmegen.

[8] Bart Jacobs, Jan Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, Bulletin of the European Association for Theoretical Computer Science, Vol. 62, pp. 222–259, 1997.

[9] Robin Milner, *A Calculus of Communicating Systems*, Springer Verlag, 1980.

[10] Horst Reichel, *Behavioural equivalence – a unifying concept for initial and final specifications*. Third Hugarian Computer Science Conference. Akademiai Kiado, Budapest, 1981.

[11] Davide Sangiogi, *On the origins of bisimulation, coinduction and fixed points*, Universita of Bologna, Italy, 2007.

[12] SEI, *Progress Maturity Profile*, March 2006.

A Cluedo Case Study

Bill Stoddart, Keerthi Rajendren and Simon Lynch

School of Computing
University of Teesside, UK

Abstract

We use the design of agents playing Cluedo as a case study to promote discussion between colleagues with interests in AI, Games, BDI agents, agent platform IDE's, and Formal Methods. We take an event based approach to modelling and a layered, refinement based approach to design. Topics considered include: design patterns for pattern language, obtaining agent software specifications from event based system models, design and non-determinism, and formal and informal approaches to ensuring invariant presentation in multi-agent systems. **Keywords.** Event B, Agency, Agent IDEs

1 Introduction

The material in this paper has arisen from discussions between colleagues with interests in AI, Games, BDI [1] agents [9], agent platform IDE's, Formal Methods, and Design Patterns. These discussions have centered round approaches to the design and deployment of agents playing Cluedo. Cluedo (the name is an amalgam of *clue* and *Ludo*, the name of an older board game and Latin for "I play") is a board game released in the UK in 1948. In the game, players (or agents) are expected to co-operate with other agents according to the rules of the game, whilst their objectives directly conflict. An agent's intentions are governed by the goals of obtaining competitive advantage by rationalising about the beliefs of other agents for the purpose of manipulating, misleading or deceiving them.

Cluedo is played with a board and a pack of cards. Each card represents a suspect, a room, or a weapon. At the start of the game one suspect card, one room card and one weapon card are removed at random from the pack and put aside. These three cards comprise the solution to the actual game. The remaining cards are dealt out and the game begins. Players move about the Cluedo board, which represents the ground floor of a mansion containing the rooms named in the game. The number of steps a player can take is given by the throw of a pair of dice. The board represents rooms linked by corridors. When a player enters a room, she must select another player and publicly voice a suspicion that the crime was committed

[1] BDI: belief, desire, intention

in that room by a certain suspect and with a certain weapon. If the player to whom the suspicion is addressed holds one or more of the named cards, she must refute the suspicion by privately showing such a card to the requesting player. Otherwise the responsibility for refuting the suspicion passes to the next player. The move is terminated when the suspicion is refuted, or when there are no more players to ask. A player who voices a suspicion is not required to believe it is true. She may, for example, include in it one or more cards that she herself holds.

Information is explicitly passed between players by one player showing another a particular card. However, information is also passed implicitly, by conclusions that can be drawn from observing a *lack of a response* or by observing the existence of a response without being able to observe any further details. This requires us to use techniques for modelling agent knowledge that are applicable in security analysis applications.

A player may voice an accusation rather than a suspicion. In this case she inspects the three cards that have been set aside. If the accusation is correct, she has won, otherwise she plays a passive role for the rest of the game, only responding to requests for information.

In describing the game of Cluedo we wish to make a distinction between a level of description which captures the rules of the game, and more detailed levels of description which capture the recording of information and the tactics, based on that information, which result in effective game play. The conceptual tool we use for this purpose is non-determinism. In describing the rules of the game, but not yet describing considerations that determine what is a good move, we provide a non-deterministic specification. That description is refined by adding variables to record the knowledge (definitive information) and beliefs (information based on assumptions regarding other players game play) that can be recorded by each agent. This information is used to reduce non-determinism by restricting agents to making moves that maximise their own information whilst not unnecessarily divulging information to other agents. From the perspective of deployment on a multi-agent platform, the case study provides an example of how an event based model is used to support specification and implementation of the software that is to run on individual agents. During the refinement of our models, a number of global invariants emerge. Within the B Method these invariants would be validated by proof, but we are also interested in how they might be handled when using a more lightweight approach such as a use of design by contract in which the mathematical descriptions of our model serve a descriptive purpose and are used to construct assertions. In this case we need to provide additional support for distributing information, which we envisage doing through extending a modular IDE [7] we have developed for use with multi-agent platforms.

The Cluedo case study requires the solution of many problems that are frequently encountered in game and software design, and this has prompted us to frame our discussion, in part, in terms of "design patterns" and "pattern language", the phrases coined by the architect Christopher Alexander to express his approach to correcting the soulless excesses of modern architecture, and up to now largely monopolised, within a computing context, by adherents of OO languages.

The paper is structured as follows. In Section 2 we review the ideas associated

with design patters and design languages. In Section 3 we identify some common game and event based design patterns and formalise one for dealing cards. In Section 4 we model the question and answer exchanges of Cluedo, and non-determinism is used to allow any legal move. In Section 5 we consider how to record the knowledge and beliefs of agents and refine play that allow any legal behaviour into tactical play. In Section 6 we consider implementation on a multi-agent platform. In Section 7 we conclude and outline future work.

2 Design Patterns and Pattern Language

The phrase "design pattern" was coined in the 1970's by the architect Christopher Alexander. He associates the term with re-occurring problems in design and the associated techniques for solving them. Such problems would include the setting of a building in a landscape, the design of an entrance (gate, door, portico..) and so on. Each problem should be considered in the context of the overall design requirements of the wider project, and addressed at a certain stage within the design process. This leads to the idea of a "pattern language". Alexander was concerned that modern architectural practice has lost the ability to produce the humanly pleasing constructs often associated with traditional and classical approaches, and his approach to design is an attempt to remedy this. In his book *The Timeless Way of Building* he explains that he regards patterns as generative: "These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules which define the patterns in the world. However, in one respect they are very different. The patterns in the world merely exist. But the same patterns in our minds are dynamic.. They are generative. They tell us what to do.." [3]

The ideas of Alexander were first cited in a computing context by Ward Cunningham and Kent Beck in *Using Pattern Languages for Object-Oriented Programs* OOPSLA-87. They have largely been appropriated by the OO community and OO approach to patterns was systematised by Erich Gamma et al in *Design Patterns: Elements of Reusable Object-Oriented Software.*

Design patterns emerge at a certain stage in the maturity of an art or technology. In this paper we are also concerned with potential or emerging patterns. We refer to the original work on design patterns because design patterns in formal methods will not correspond with OO design patterns: the latter attempts to achieve correctness through restrictive interfaces to data whilst the former wishes to present different levels of abstraction and to have a global view of data, ensuring correctness through proving or checking properties

We consider design patterns associated with both game design and system design, such as how an event model of a distributed system can give rise to a software specifications for operations that will be performed by a particular agent.

3 Game patterns and software design patterns

In the context of game design and implementation we see three abstractions we can apply to Cluedo which apply to many other games. One of these is card dealing,

concerned with the selection of some special cards and the random distribution of the rest of the cards to the players; another is scheduling, controlling which player is currently active; the third is navigation, concerned with the initial positioning and and subsequent movement of pieces on the board. We now consider card dealing in more detail.

We use an event based notation based on B-GSL. $G \Longrightarrow S$ is an event which may fire if the guard (predicate) G is true, and will perform the update described by S.

We use the constant $CARD$ to represent the set of Cluedo cards. Given enumerations of each set, which we do not write out here, we have the following context information:

$$SUSPECT \subset CARD \land ROOM \subset CARD \land WEAPON \subset CARD$$
$$\land\ SUSPECT \cup ROOM \cup WEAPON = CARD \land$$
$$disjoint(\{SUSPECT, ROOM, WEAPON\})$$

We characterise a constant set *allsoln* which holds all possible solutions to a Cluedo game.

$$allsoln = \{s, r, w \mid s \in SUSPECT \land r \in ROOM \land w \in WEAPON \bullet \{s, r, w\}\}$$

A solution must be chosen before the remaining cards are dealt. The variable *cards* will denote cards remaining in the pack. We have a variable $phase \in \{start, deal, play\}$ which controls the order of certain operations, and a variable *thesoln*, which holds the solution chosen for the current game.

In passing, we ask the reader whether the choice in the following event would adequately specify the selection of a solution:

$$SelectSoln \mathrel{\widehat{=}} phase = start \Longrightarrow \text{ Any } s \cdot s \in allsoln \Longrightarrow$$
$$thesoln := s \parallel phase := deal \parallel cards := cards \setminus s$$

and in fact we claim this description is *not specific enough*, as it leaves choice of solution to the implementor, who could choose to select the same solution each time. We need to specify that the choice is made randomly. for this we will make use of random choice from a set, denoted by $:\oplus$.

$$SelectSoln \mathrel{\widehat{=}} phase = start \Longrightarrow$$
$$thesoln :\oplus allsoln \parallel state := deal;\ cards := cards \setminus thesoln$$

We can model the dealing of cards as a classical action system containing a deal event and a termination event. As each card is dealt it is removed from the set *cards*, and this provides the variant that ensures termination.

Cluedo specifies a circular order amongst the players. We assume a function $next(p)$ to identify the next player after p, and a variable *player*, assumed to be suitably initialised, which is used to hold the identity of the next player to receive a card. Our dealing event, distributing a single randomly chosen card to the next player, is then:

$$Deal \mathrel{\widehat{=}} phase = deal \land cards \neq \{\} \Longrightarrow$$
$$\text{Any } c \cdot c \in cards \Longrightarrow c :\oplus cards\ ;$$

$hand(player) := hand(player) \cup \{c\} \parallel cards := cards \setminus \{c\} \parallel$
$player := next(player)$

Our termination event, which enables play to begin, is enabled when the set *card* has become empty.

$StartPlay \mathrel{\widehat{=}} cards = \{\} \wedge phase = deal \Longrightarrow phase := play$

This completes our event model of card dealing. Our selection of special cards was specified as a "one shot" event. Assuming a future implementation which will call a procedure to select the solution cards, we can obtain the specification of this procedure from the *SelectSoln* event by converting the guard (firing condition) of *SelectSoln* into a pre-condition (assumption about what is true when the procedure is called). In effect, the responsibility for firing the event will be taken by a calling program, and we are able to assume the guard holds when the program is called. This is one design pattern emerging from event based models, which produces classical software specifications for procedures that will form part of the eventual implementation.

Our event model of dealing, on the other hand, was specified in terms of events that can be directly related to the commands used *within* an implementation procedure. The *Deal* event must fire repeatedly until it is no longer enabled, at which point the *StartPlay* event will fire. This gives us a design pattern for the construction of a loop.

4 Questioning and Eavesdropping

The importance of non-determinism, which is universally accepted within the formal methods community, can be difficult to explain to students used to solving simple problems directly at the concrete level of programming. The rules of games provide an excellent example of how non-determinism can be used, since they specify what players of a game may do without specifying what they should do. We apply this separation of concerns to our Cluedo case study, and here we consider the rules for part of the game in which a player communicates a suspicion and other players must refute it if they can.

A player p_1 may propose to another player p_2 the suspicion that suspect s committed the crime in room r and with weapon w. If p_2 holds one of the cards in $\{s, r, w\}$, she will be in a position to refute the suspicion, and must do so by showing p_1 such a card. Other players are able to observe that the suspicion has been refuted, but cannot see any further details.

This interaction is modelled by events *Suspect* which describes the posing of the question; *CanHelp* which describes the reply, *Eave*1, which records the observation of the exchange made by each player not directly involved, firing once for each such player; and *Refuted*, which models the arrival of the refutation at the asking player, and ends the current move.

The other possible scenario is that p_2 does not hold any card which could be used to refute the suspicion voiced by p_1. In this case the question passes on to the next player (omitting p_1 of course). In this case each of the other players see that the responding player has been unable to give a refutation from the cards it

holds, and their observation of this fact is modelled by firings of the event *Eave*2. The event *NextResponse* will fire, and enables the next player after p_2 to make a response.

This pattern is repeated until either a refutation event fires, or until all players have been asked and failed. In the latter case, the event *EndResponses* will fire and terminate the move.

A significant aspect of this model is how the leakage of information to eavesdropping players is described. To capture the fact that an eavesdropping player can distinguish different types of answer but cannot see details of how the suspicion is refuted, we have two different eavesdropping events. These events do not, at the current level of abstraction (aimed at modelling the rules of this part of Cluedo) collect any useful information. That will change when we subsequently refine them.

We now sketch the model in more detail. We have a set of player states:

$$STATE = \{wait, startmove, accuse, navigate, ask, listen, respond, \}$$

The variable *player* denotes the set of players, between 3 and 6 in number. The players are drawn from the set of suspects. The variable *active* denotes the set of active players (a player ceases to be active after making an unwarranted accusation).

$$player \subseteq SUSPECT \land \text{card}(player) \in 3..6$$
$$active \subseteq player$$

The function *state* maps each player to her current state. The player whose turn it is will be accusing, moving, asking or listening. If a player is listening, there will be exactly one other player who is responding. Players who are not active can only be in a *wait* state or a *respond* state.

$$state \in player \mapsto STATE$$
$$\exists_1\, p.p \in player \land state(p) \in \{accuse, navigate, ask, listen\}$$
$$\exists\, p_1.p_1 \in player \land state(p_1) = listen \Rightarrow$$
$$\quad \exists_1\, p_2.p_2 \in player \land state(p_2) = respond$$
$$\forall\, p \;.\; p \notin active \Rightarrow state(p) \in \{wait, respond\}$$

An asking player voices a suspicion that the crime was committed by a particular suspect, in a particular room, with a particular weapon. This suspicion is visible to all players and is stored in the variable *suspect*

$$suspect \in allsolns$$

A player who is suspected of committing the crime in a particular room is moved into that room, and we record this via the variable *moved*. A player who is moved is allowed to ask questions from a room without moving into it (via a dice throw) first, but these details will not be modelled in this paper.

$$moved \in player \rightarrow BOOL$$

Channels are used to communicate responses from one player to another. There is a channel from p_2 to p_1 if p_2 and p_1 are different players. The channel may contain no message, an empty message, or a message identifying one card from the hand of p_2.

$$
\begin{array}{l}
channel \in player \times player \nrightarrow \mathbb{P}\,(CARD) \\
\mathrm{dom}(channel) = \{p1, p2 \mid p_1 \in player \wedge p_2 \in player \wedge p_1 \neq p_2\} \\
\forall\, p_1, p_2 \cdot (p_2, p_1) \in \mathrm{dom}(channel) \Rightarrow \\
\quad (\neg\, (state(p_1) = listen \wedge state(p_2) = respond) \Rightarrow channel(p_2, p_1) = \{\}) \\
\qquad \wedge \\
\quad (state(p_1) = listen \wedge state(p_2) = respond \wedge channel(p_2, p_1) \neq \{\} \Rightarrow \\
\qquad (\exists\, c.c \in hand(p_2) \cap suspect \wedge channel(p_2, p1) = \{\{c\}\}) \\
\qquad\quad \vee \\
\qquad \neg\, (\exists\, c.c \in hand(p_2) \cap suspect) \wedge channel(p_2, p1) = \{\{\}\})
\end{array}
$$

We have variables to identify the player who is asking the question and the first player to whom the question was posed. This latter is needed so we can determine when everyone has had a chance to give a response.

$$asking \in active \wedge first\_asked \in player \wedge asking \neq first\_asked$$

We have functions to identify the next active player (who will have the next move) and the next responding player. We depart from B custom here in allowing ourselves to name variables in function bodies.

$$
\begin{array}{l}
next\_active(p) \mathrel{\widehat{=}} \text{ if } next(p) \in active \text{ then } next(p) \\
\quad \text{else } next\_active(next(p)) \text{ end} \\
next\_respondent(p) \mathrel{\widehat{=}} \text{ if } next(p) = asking \text{ then } next(next(p)) \\
\quad \text{else } next(p) \text{ end}
\end{array}
$$

We have a function to record the position of each player.

$$posn \in player \rightarrow ROOM$$

Finally we have a variable to record the number of currently eavesdropping players.

$$eavesdropping \subset active \wedge \mathrm{card}(eavesdropping) \in 0..\mathrm{card}(active) - 2$$

We now consider the events of our model. A player in the *ask* state may address another player with a suspicion that the crime was committed by a certain person, in a certain room, and with a certain weapon. The asking player enters the *listen* state and the responding player the *respond* state. If the named suspect is not in the room he or she is moved there. The details of the question are available to all players.

$$
\begin{array}{l}
Suspect \mathrel{\widehat{=}} \text{ Any } p_1, p_2, s, r, w \ . \\
\quad state(p_1) = ask \wedge p_2 \in players \setminus \{p_1\} \wedge \\
\quad \{s, r, w\} \in allsoln \wedge room(p_1) = r \Longrightarrow \\
\qquad asking := p_1 \parallel first\_asked := p_2 \parallel state(p_1) := listen \parallel \\
\qquad state(p_2) := respond \parallel suspect := \{s, r, w\} \\
\qquad \text{if } posn(s) \neq r \text{ then } posn(s) := r \parallel moved(s) := true \text{ end}
\end{array}
$$

A player to whom such a suspicion is directed is obliged to respond. If they are able to refute the suspicion they must do so by showing the asking player a suitable card, as modelled by the event *CanHelp*. The response is made placing a message in the appropriate channel.

$$
\begin{array}{l}
CanHelp \mathrel{\widehat{=}} \text{ Any } p_1, p_2 \ . \\
\quad state(p_1) = listen \wedge state(p_2) = respond \wedge suspect \cap hand(p_2) \neq \{\} \Longrightarrow \\
\qquad \text{Any } c \ . \ c \in suspect \cap hand(p_2) \Longrightarrow channel(p_2, p_1) := \{\{c\}\} \\
\qquad\quad \parallel \\
\qquad eavesdropping := active \setminus \{p_1, p_2\}
\end{array}
$$

Otherwise they must say they are unable to help, done by placing an empty message in the channel. We distinguish an empty message in a channel, modelled as a channel state of $\{\{\}\}$, from an empty channel, modelled as a channel state of $\{\}$

$$
\begin{array}{l}
CanTHelp \mathrel{\widehat{=}} \text{ Any } p_1, p_2 \ . \\
\quad state(p_1) = listen \wedge state(p_2) = respond \wedge suspect \cap hand(p_2) = \{\} \Longrightarrow \\
\qquad channel(p_2, p_1) := \{\{\}\} \\
\qquad\quad \parallel \\
\qquad eavesdropping := active \setminus \{p_1, p_2\}
\end{array}
$$

Third parties are able to observe some aspects of the exchange of information described above. They can see the suspicion, and whether it was refuted, but cannot see how it was refuted. We model this binary level of visibility by two eavesdropping events, one for the case where the suspicion was refuted and one for where it was not. Either *eave*1 will fire once for each eavesdropping player, or *eave*2 will fire once for each eavesdropping player.

$$
\begin{array}{l}
Eave1 \mathrel{\widehat{=}} \text{ Any } p, p_1, p_2 \ . \ p \in active \wedge state(p) = wait \wedge \\
\quad state(p_1) = listen \wedge state(p_2) = respond \wedge \\
\quad channel(p_2, p_1) \neq \{\{\}\} \wedge p \in eavesdropping \Longrightarrow \\
\qquad eavesdropping := eavesdropping \setminus \{p\}
\end{array}
$$

$$
\begin{array}{l}
Eave2 \mathrel{\widehat{=}} \text{ Any } p, p_1, p_2 \ . \ p \in active \wedge state(p) = wait \wedge \\
\quad state(p_1) = listen \wedge state(p_2) = respond \wedge \\
\quad channel(p_2, p_1) = \{\{\}\} \wedge p \in eavesdropping \Longrightarrow \\
\qquad eavesdropping := eavesdropping \setminus \{p\}
\end{array}
$$

The event *Refuted* models the arrival of refutation information at the asking player. The current move is then at an end.

$$\begin{array}{l} \mathit{Refuted} \mathrel{\widehat{=}} \text{ Any } p_1, p_2 \cdot \mathit{state}(p_1) = \mathit{listen} \wedge \mathit{state}(p_2) = \mathit{respond} \\ \quad \wedge \mathit{eavesdropping} = \{\} \wedge \exists\, c. \mathit{channel}(p_2, p_1) = \{\{c\{\} \Longrightarrow \\ \qquad \mathit{channel}(p_2, p_1) := \{\} \parallel \mathit{state}(p_1) := \mathit{wait} \parallel \\ \qquad \mathit{state}(\mathit{nextactive}(p_1)) := \mathit{startmove} \parallel \\ \qquad \text{if } p_2 \neq \mathit{nextactive}(p_1) \text{ then } \mathit{state}(p_2) := \mathit{wait} \text{ end} \end{array}$$

Where the suspicion is not refuted, the next player on from the responding player is enabled to make a response, modelled by event *NextResponse*.

$$\begin{array}{l} \mathit{NextResponse} \mathrel{\widehat{=}} \text{ Any } p, p_1, p_2 \cdot \mathit{state}(p_1) = \mathit{listen} \wedge \mathit{state}(p_2) = \mathit{respond} \\ \quad \wedge \mathit{channel}(p_2, p_1) = \{\{\}\} \wedge \mathit{next\_respondent}(p_2) \neq \mathit{first\_asked} \\ \quad \wedge \mathit{eavesdropping} = \{\} \wedge p = \mathit{next\_respondent}(p_2) \Longrightarrow \\ \qquad \mathit{state}(p_2) := \mathit{wait} \parallel \mathit{state}(p) := \mathit{respond} \parallel \\ \qquad \mathit{eavesdropping} := \mathit{active} \setminus \{p_1, p\} \end{array}$$

If there are no more players, i.e. no player has been able to refute the suspicion posed, the move is at an end, as modelled by event *EndResponses*

$$\begin{array}{l} \mathit{EndResponses} \mathrel{\widehat{=}} \text{ Any } p_1, p_2 \cdot \mathit{state}(p_1) = \mathit{listen} \wedge \mathit{state}(p_2) = \mathit{respond} \\ \quad \wedge \mathit{next\_respondent}(p_2) = \mathit{first\_asked} \wedge \mathit{channel}(p_2, p_1) = \{\{\}\} \\ \quad \wedge \mathit{eavesdropping} = \{\} \Longrightarrow \\ \qquad \mathit{state}(p_1) := \mathit{wait} \parallel \mathit{state}(\mathit{next\_active}(p_1)) := \mathit{startmove} \parallel \\ \qquad \text{if } p_2 \neq \mathit{next\_active}(p_1) \text{ then } \mathit{state}(p_2) := \mathit{wait} \text{ end} \end{array}$$

5 Agent Knowledge, Belief, and Strategy

In this section we discuss the refinement of the operations described in the previous section. These refinements reduce non-determinism by selecting moves and responses that increase the acquisition of information whilst revealing less information to competing players.

Consider a scenario where player p_1 addresses the suspicion {s,r,w} to player $p2$ and receives a null response. $p1$ can deduce that none of the cards in {s,r,w} are held by p_2. An onlooking player can make the same deduction from his eavesdropping activities. This information may not directly reduce the number of possible solutions, but it may do so when combined with other information.

To record it we use a variable n, where $n(p_1, p_2)$ records the cards that p_1 knows are not held by p_2.

We have the invariant property:

$$\forall\, p_1, p_2 \cdot n(p_1, p_2) \cap \mathit{hand}(p_2) = \{\}$$

which is a global invariant in the sense that, when we later refine for implementation on a multi-agent platform, no one agent will be able to see all the values required to check this property via an assertion based approach. That illustrates an advantage of the proof based approach over assertion checking, but we will still be interested in attempting an assertion based approach to such a development, and we will return to this topic later.

If p_2 shows a card c say from its hand, to refute the suspicion posed, p_1 will know that $c \in hand(p_2)$. This is recorded in the variable *shown*, where $shown(p_1, p_2)$ is the set of cards p_1 has chown to p_2. This information (assuming it is new information for p_1) will directly reduce the number of possible solutions to the game by eliminating any that contain the card c.

A player p_1 may deduce that a certain hand is in the hand of another player p_2, even if p_2 has not shown her that card directly. This information is recorded in the variable k, where $k(p_1, p_2)$ denotes the cards p_1 knows are in the hand of p_2, and we have the invariant property:

$$shown(p_2, p_1) \subseteq known(p_1, p_2)$$

When a player announces a suspicion, she can use her knowledge of the hands of other players to avoid a suspicion that could be refuted by being shown a card she already knows to be held by another player. With this in mind we refine the *Suspect* event to:

$$\begin{array}{l} Suspect \mathrel{\widehat{=}} \text{ Any } p_1, p_2, s, r, w \ . \\ \quad state(p_1) = ask \wedge p_2 \in players \setminus \{p_1\} \wedge \\ \quad \{s, r, w\} \in allsolutions \wedge room(p_1) = r \wedge \\ \quad \forall\, p.p \in player \wedge p \neq p1 \Rightarrow \\ \quad \{s, r, w\} \cap k(p_1, p) = \{\}/^* \text{ New Condition } */ \Longrightarrow \\ \qquad asking := p_1 \parallel first\_asked := p_2 \parallel state(p_1) := listen \parallel \\ \qquad state(p_2) := respond \parallel suspect := \{s, r, w\} \\ \qquad \text{if } posn(s) \neq r \text{ then } posn(s) := r \parallel moved(s) := true \text{ end} \end{array}$$

This refinement avoids a player posing a suspicion which a potential respondent is able to refute by showing a card which the asking player already knows they hold.

We now consider a refinement for the responding player event *CanHelp* which we will refine by a collection of events $CanHelp_1$, $CanHelp_2$ and $CanHelp3$, guarded by different tactics. Each, individually, refines *CanHelp* and collectively they provide the required preservation of feasibility that ensures we do not refine to the point of system deadlock. $CanHelp_1$ will fire if the enquiry can be answered by showing a card already shown to the enquiring player. $CanHelp_2$ will fire if there is no such card but it is possible to show a card which can be indirectly inferred to belong to responding player. $CanHelp3$ will fire otherwise.

As a simple example of inferred information, suppose player p has been unable to refute the suspicion $\{s, r, w_1\}$ but has refuted the suspicion $\{s, r, w_2\}$. p has then revealed that she holds the card w_2. To deduce this we must record information about the responses p has given to enquiries. This is global knowledge, in the sense that we record what every player can see of each response. We have variables gi and gn, where $gi(p)$ is the set of suspicions which are globally known to intersect with $hand(p)$ (i.e. the suspicions which p has refuted) and $gn(p)$ is the set of cards $gn(p)$ is the set of cards p has revealed are not in $hand(p)$. Where a suspicion is refuted we update gi.

$$\begin{array}{l} \mathit{CanHelp}_1 \mathrel{\widehat{=}} \text{ Any } p_1, p_2, c \cdot \mathit{state}(p_1) = \mathit{listen} \wedge \mathit{state}(p_2) = \mathit{respond} \wedge \\ \quad c \in \mathit{suspect} \cap \mathit{shown}(p_1, p_2) \Longrightarrow \\ \qquad \mathit{channel}(p_2, p_1) := \{\{c\}\} \parallel \mathit{gi}(p_2) := \mathit{gi}(p_2) \cup \{\mathit{suspect}\} \parallel \\ \qquad \mathit{eavesdropping} := \mathit{active} \setminus \{p_1, p_2\} \end{array}$$

Where a player p is unable to respond by presenting a card that has already been shown, she attempts to respond with a card that the enquiring player should be able to deduce she holds. This set of cards is:

$$\{s, a \mid s \in \mathit{gi}(p) \wedge s \setminus \mathit{gn}(p) = \{a\} \bullet a\}$$

we have an associated invariant property:

$$\forall\, p \cdot \{s, s \mid s \in \mathit{gi}(p) \wedge s \setminus \mathit{gn}(p) = \{a\} \bullet a\} \subseteq \mathit{hand}(p)$$

and the event is:

$$\begin{array}{l} \mathit{CanHelp}_2 \mathrel{\widehat{=}} \text{ Any } p_1, p_2, c \cdot \mathit{state}(p_1) = \mathit{listen} \wedge \mathit{state}(p_2) = \mathit{respond} \wedge \\ \quad \mathit{suspect} \cap \mathit{shown}(p_1, p_2) = \{\} \Longrightarrow \\ \quad \text{Any } c \cdot c \in \{s, c \mid s \in \mathit{gi}(p_2) \wedge s \setminus \mathit{gn}(p_2) = \{c\} \bullet c\} \Longrightarrow \\ \qquad \mathit{channel}(p_2, p_1) := \{\{c\}\} \parallel \mathit{gi}(p_2) := \mathit{gi}(p_2) \cup \{\mathit{suspect}\} \parallel \\ \qquad \mathit{eavesdropping} := \mathit{active} \setminus \{p_1, p_2\} \end{array}$$

Finally, where there is no way for the responding player to avoid revealing new information, the event is:

$$\begin{array}{l} \mathit{CanHelp}_3 \mathrel{\widehat{=}} \text{ Any } p_1, p_2, c \cdot \mathit{state}(p_1) = \mathit{listen} \wedge \mathit{state}(p_2) = \mathit{respond} \wedge \\ \quad \mathit{suspect} \cap (\mathit{shown}(p_2, p_1) \cup \{s, a \mid s \in \mathit{gi}(p_2) \wedge s \setminus \mathit{gn}(p_2) = \{a\} \bullet a\}) \\ \qquad = \{\} \\ \quad \wedge\ c \in \mathit{suspect} \cap \mathit{hand}(p_2) \Longrightarrow \\ \qquad \mathit{channel}(p_2, p_1) := \{\{c\}\} \parallel \mathit{eavesdropping} := \mathit{active} \setminus \{p_1, p_2\} \end{array}$$

Where the responding player is unable to refute the suspicion, the event *CanTHelp* is triggered. A note is made that the responding player has revealed that she does not possess any of the *suspect* cards.

$$\begin{array}{l} \mathit{CanTHelp} \mathrel{\widehat{=}} \text{ Any } p_1, p_2 \cdot \mathit{state}(p_1) = \mathit{listen} \wedge \mathit{state}(p_2) = \mathit{respond} \wedge \\ \quad \mathit{suspect} \cap \mathit{hand}(p_2) = \{\} \Longrightarrow \\ \qquad \mathit{channel}(p_2, p_1) := \{\{\}\} \parallel \mathit{gn}(p_2) := \mathit{gn}(p_2) \cup \mathit{suspect} \\ \qquad \parallel \mathit{eavesdropping} := \mathit{active} \setminus \{p_1, p_2\} \end{array}$$

In the case where a suspicion is refuted, an eavesdropping player p does not get access to all information. They do, however know that the there must be at least one card from the suspicion that is held by the responding player p_2. By eliminating the cards p knows p_2 does not hold, it may be possible to determine which card this is, add this to the cards known to be held by p_2 and eliminate any solutions involving this card. Otherwise the information is recorded in the intersects function and the current suspicion is removed from p's set of possible solutions.

$$
\begin{array}{l}
Eave1 \mathrel{\widehat{=}} \text{ Any } p, p_1, p_2 \cdot p \in active \wedge state(p) = wait \wedge \\
\quad state(p_1) = listen \wedge state(p_2) = respond \wedge \\
\quad channel(p_2, p_1) \neq \{\{\}\} \wedge p \in eavesdropping \Longrightarrow \\
\quad \text{Any } v \cdot v = suspect \setminus n(p, p_2) \Longrightarrow \\
\quad \text{if} \\
\qquad card(v) = 1 \\
\quad \text{then} \\
\qquad k(p, p_2) := k(p, p_2) \cup v \parallel poss(p) := \{s \mid s \in poss(p) \wedge s \cap v = \{\}\} \\
\quad \text{else} \\
\qquad i(p, p_2) := i(p, p_2) \cup v \parallel poss(p) = poss(p) \setminus \{suspect\} \\
\quad \text{end} \parallel \\
\quad eavesdropping := eavesdropping \setminus \{p\}
\end{array}
$$

In the case where a suspicion is not refuted, an eavesdropping player p knows that no card from the announced suspicion is held by the responding player p_2. This information is used to revise the set of cards p knows p_2 does not hold, and also to revise the set of sets known to intersect with the hand of p_2, possibly revealing some cards that must be in p_2's hand and thus cannot be in the solution. This eavesdropping event is modelled as $Eave2$.

$$
\begin{array}{l}
Eave2 \mathrel{\widehat{=}} \text{ Any } p, p_1, p_2 \cdot p \in active \wedge state(p) = wait \wedge \\
\quad state(p_1) = listen \wedge state(p_2) = respond \wedge \\
\quad channel(p_2, p_1) = \{\{\}\} \wedge eavesdrop \neq \{\} \Longrightarrow \\
\quad n(p, p_2) := n(p, p_2) \cup suspect \parallel \\
\quad i(p, p2) := \{s \mid s \in i(p, p2) \wedge \text{ card}(s \setminus suspect) > 1 \bullet s \setminus suspect\} \parallel \\
\quad \text{Any } v \cdot v = \{s \mid s \in i(p, p_2) \wedge \text{ card}(s \setminus suspect) = 1 \\
\qquad \bullet choice(s \setminus suspect)\} \Longrightarrow (\\
\qquad k(p, p_2) := k(p, p_2) \cup v \parallel poss(p) := \{s \mid s \in poss(p) \wedge s \cap v = \{\}\} \\
\qquad) \parallel \\
\quad eavesdropping := eavesdropping \setminus \{p\}
\end{array}
$$

We provide a refinement of the *Refuted* event, showing the information recorded by an enquiring player p_1 when its suspicion has been refuted by p_2. In this case, p_1 will be shown a card c say. She can now, of course, eliminate any solution containing c from her set of possible solutions, but it is possible to do much more with the given information than this.

Taking a concrete example, suppose Bill has previously refuted Simon's suspicion "Colonel Mustard, in the ballroom, with the rope", then Keerthi will know that Bill holds one of the cards in the set $\{Mustard, Ballroom, Rope\}$. Suppose that Simon subsequently shows Keerthi the card for Colonel Mustard. Keerthi will now know that Bill does not hold that card, and therefore that the set $\{Ballroom, Rope\}$ has a non empty intersection with Bill's hand. And if Keethi also knows that $\{Mustard, Kitchen\}$ had a non-empty intersect with Bill's hand, she will now know that *Kitchen* is in Bill's hand.

In the following event we update the functions i, k and n in parallel, then employ sequential composition so that the new information recorded in k can be used to update the update the list of possible solutions. For some of the assignments in this

event we use the form $v :| Q$ meaning the variables in the list v take new values as described in the before-after-predicate Q, and any other variables are unchanged.

$$
\begin{array}{l}
\mathit{Refuted} \mathrel{\widehat{=}} \text{ Any } p_1, p_2 \cdot \mathit{state}(p_1) = \mathit{listen} \land \mathit{state}(p_2) = \mathit{respond} \\
\quad \land\ \mathit{eavesdropping} = \{\} \land \exists\, c.\mathit{channel}(p_2, p_1) = \{\{c\}\} \Longrightarrow \\
\qquad \mathit{channel}(p_2, p_1) := \{\} \parallel \mathit{state}(p_1) := \mathit{wait} \parallel \\
\qquad \mathit{state}(\mathit{nextactive}(p_1)) := \mathit{startmove} \parallel \\
\qquad \text{if } p_2 \neq \mathit{nextactive}(p_1) \text{ then } \mathit{state}(p_2) := \mathit{wait} \text{ end} \parallel \\
\qquad i, k, n :| \forall\, p, q \cdot p \in \mathit{player} \land q \in \mathit{player} \land p \neq q \cdot \\
\qquad\quad (p \neq p_1 \Rightarrow i'(p,q) = i(p,q) \land k'(p,q) = k(p,q) \land n'(p,q) = n(p,q)) \\
\qquad \land \\
\qquad\quad (p = p_1 \land q = p_2 \Rightarrow (\\
\qquad\qquad i'(p,q) = \{s \mid s \in i(p,q) \land c \notin s\} \land \\
\qquad\qquad k'(p,q) = k(p,q) \cup \{c\} \land \\
\qquad\qquad n'(p,q) = n(p,q) \\
\qquad\quad) \land \\
\qquad\quad p = p_1 \land q \neq p_2 \Rightarrow (\\
\qquad\qquad i'(p,q) = \{s \mid s \in i(p,q) \land \mathrm{card}(s \setminus \{c\}) > 1 \bullet s \setminus \{c\}\} \land \\
\qquad\qquad k'(p,q) = k(p,q) \cup \{s, a \mid s \in i(p,q) \land s \setminus \{c\} = \{a\} \bullet a\} \land \\
\qquad\qquad n'(p,q) = n(p,q) \cup \{c\} \\
\qquad\quad) ; \\
\qquad \mathit{poss} \ |: \ \forall\, p \cdot p \in \mathit{player} \Rightarrow \\
\qquad\quad (p \neq p_1 \Rightarrow \mathit{poss}'(p) = \mathit{poss}(p)) \land \\
\qquad\quad (p = p_1 \Rightarrow \mathit{poss}'(p) = \\
\qquad\qquad \{s \mid s \in \mathit{poss}(p) \land \nexists\, q \cdot q \in \mathit{player} \land q \neq p \land k(p,q) \cap s \neq \{\}\})
\end{array}
$$

Under the assumption that players are competent and do not cheat, the information recorded in the above model is reliable. We refer to this kind of information as *knowledge*. We can also make use of a less certain level of information, which we refer to as *belief*, based on assumptions about other players behaviour.

We might assume that any suspicion announced by player p contains at least one card that is not in $\mathit{hand}(p)$. If this were not the case, p would be unable to derive any information from her enquiry.

We might also assume that p will not announce a suspicion containing any cards she knows are held by another player, for should she do so, she would risk having the suspicion refuted by an answer that repeats information she already has. To see how such an assumption leads to a belief, suppose Keerthi has suspicion $\{\mathit{Mustard}, \mathit{Ballroom}, \mathit{Rope}\}$ refuted by Bill and subsequently proposes $\{\mathit{Mustard}, \mathit{Ballroom}, \mathit{Gun}\}$, then this would lead Simon to believe that *Rope* is in Bill's hand.

Use of beliefs can lead a player more quickly to a situation where she believes she has only one possible solution. However, relying on belief is risky, as it is possible a player might violate the above assumptions, either by accident or to confuse her opponents.

A belief might be used, however, to guide the choice of an enquiry. E.g. suppose player p_1 believes (or knows) that the set $\{c_1, c_2\}$ has an intersection with $\mathit{hand}(p_2)$. Suppose $\{c_1, c_3, c_4\}$ is a possible suspicion where $\{c_3, c_4\} \subset \mathit{hand}(p_1)$. Then a good

move is to pose the suspicion $\{c_1, c_3, c_4\}$ to $next\_respondent(p_2)$. This gives as many players as possible the opportunity to refute the suspicion before it comes round to p_2. If some player $p \neq p_2$ refutes with c_1, we obtain the knowledge that $c_1 \in hand(p)$ and also the belief that $c_2 \in hand(p_2)$.

A further level of refinement is required for the incorporation of beliefs and the incorporation of tactics to select a specific suspicion and the player to whom it is first directed, but that is not attempted here, and we move now to the problem of extracting a specification for each individual players behaviour.

6 Refinement for Deployment on a Multi-Agent Platform

Our implementation environments are the multi-agent platform BORIS[6], and the MIT Galaxy Communicator Platform[4]. We have developed a simple but extensible modular IDE[7], itself a multi-agent application, which works with both these platforms. This monitors and can display messages passed between agents, and can replay agent interactions in slow motion. Communications are based on TCP/IP sockets and are normally asynchronous "send and continue".

The target architecture has one agent for each player and a control agent which performs the initial solution selection and dealing tasks and subsequently takes on a co-ordinating role. There will also be additional agents which handle spoken dialogues and optical card recognition.

Our next level of refinement is to adapt our model to this deployment scenario. The result will still be an event model, but each event will now be informally associated by name with a particular agent, there will be an informal indication of which agent encapsulates which data, and where more than one agent needs to see the same data we model its communication over a channel.

We recall that players in this model are of the same type as cards, each player being one of the suspects. In this refinement we begin to conceptualise agents which correspond to players. We informally identify these agents as w, x, y.. Data which, in a future implementation will be encapsulated by agent x will be accordingly named, for example $xhand$ will be the hand of player x, and $xk(q)$ will be the cards x knows are in the hand of player q.

Each agent requires a variable of type $CARD$ which records his player identity, and these will be wis, xis, yis, zis. We have the glueing invariant clause:

$$\{wis, xis, yis, zis\} = player$$

We note that our previous specification was scalable in terms of the number of players, but that our approach forces us now to refine it to a model with a certain fixed number of players.

Events and data of our refinement are associated with players by name, so that $xRefuted$ becomes the event that player x has a suspicion refuted, and $xhand$ is the hand of player x. For data, this naming convention replaces one level of indexing over players, so that we have, as part of the glueing invariant:

$$hand(wis) = whand \land hand(xis) = xhand \land ..$$
$$gn(wis) = wgn \land gn(xis) = xgn \land ...$$

the pattern for functions whose domains range over a pair of players is illustrated by the following glueing invariant clauses:

$$\forall\, p \;.$$
$$p \in player \setminus \{wis\} \Rightarrow k(wis, p) = wk(p) \land$$
$$p \in player \setminus \{xis\} \Rightarrow k(xis, p) = xk(p) \land ..$$

Note that the agents w, x.. do not exist as identifiers in our formal model. They appear only through a naming convention which provides an informal association between informal conceptual agents and the elements of our model.

In addition to player agents we also conceptualise a control agent which manages the scheduling of game events and through which all messages between game agents will pass.

We sketch the refinement of the *Refuted* event. We now have events *wRefuted*, *xRefuted* etc which fire when an agent is listening for a response to its posed suspicion, and we have some control agent events, not given in detail, which communicate the response, receive an acknowledgment, communicate to the responding player and the player who will have the next move. The last of these events, which we might call *cRefuted* to show it is associated with the control agent, is the event that refines the *Refuted* event from our previous model, as it is only when *cRefuted* has fired that all concrete updates corresponding to abstract updates in *Refuted* will have taken place.

The event *xRefuted* can fire when agent x is in a listening state and the channel from the control agent contains information on the card that is being shown and the identity of the responding player. Each concrete channels holds (or, more formally, *is*) a sequence of strings. A total injection *name* maps each element of *CARD* to the string which is its name.

$$xRefuted \mathrel{\widehat{=}} \text{Any } c, p, other \;.\; xstate = listen \land$$
$$channelcx = \langle name(c), name(p) \rangle \land other = xplayers \setminus \{xis, p\} \Longrightarrow$$
$$channelcx := \langle\rangle \parallel xstate := wait \parallel$$
$$xn, xi, xk \;:|$$
$$xn'(p) = xn(p) \land \forall\, q \;.\; q \in other \Rightarrow xn'(q) = xn(q) \cup \{c\} \land$$
$$xi'(p) = \{s \mid s \in xi(p) \land c \notin s\} \land$$
$$\forall\, q \;.\; q \in other \Rightarrow xi'(q) = \{s \mid s \in xi(q) \land \text{card}(s \setminus \{c\} > 1 \bullet s \setminus \{c\}\}$$
$$\land\, xk'(p) = xk(p) \cup \{c\} \land$$
$$\forall\, q \;.\; q \in other \Rightarrow xk'(q) = xk(q) \cup \{s, a \mid s \in xk(q) \land s \setminus \{c\} = a \bullet a\}$$
$$;$$
$$poss := \{s \mid s \in poss \land (\textstyle\bigcup_{q \in xplayer \setminus \{xis\})} \cap s = \{\}\}$$
$$\parallel channelxc := \langle \text{“}ok''\rangle$$

Although this is still and event rather than an operation, we are moving closer to the specification of operations within individual agents. We conceive of each agent running a program which selects operations that correspond to the events of

our model. This program has the responsibility of selecting each of these operations only when the guard of its corresponding event is true. The specification of such an operations can assume the condition under which the program selects it for execution, and is obtained, grosso modo, by converting the guard of the corresponding event to a pre-condition.

7 Conclusions and Future Work

We began this paper with a discussion of design patterns and mentioned three that are commonly found in games. During the paper we have encountered a number of design patterns that have become familiar as part of the Event-B Method. These include the modelling of data using sets, the description of a system in terms of guarded events which act on those events. The reconciliation of state (data) and dynamic (event) views in terms of state invariants and their preservation, the use of appropriate levels of abstraction during a development, with the ability to reconcile descriptions at different levels of abstraction through refinement, and the use of non-determinism to obtain abstraction, this last illustrated by the non-determinism inherently present in the description of the rules for a game.

The refinement of an abstract event model into code specification for the component processes of a distributed application remains, we believe, an emerging pattern rather then established practice that can be reliably re-applied over changing application domains. In particular the existence of appropriate structuring mechanisms remains an issue. In a forthcoming paper, Mike Poppleton notes, with respect to the Rodin Platform [1] that "The classical B compositional mechanisms have been excised from Event-B to make way for their re-invention in future"[8].

Cluedo seems an excellent application with which to explore the emerging design patterns for Event-B and related methods, and their underlying techniques such as refinement, decomposition, and instantiation[2].

The pattern language of the B Method organises activities into a certain order, imposed by tool support. The discharge of proofs occurs at several stages, and the requirement to discharge all such obligations can appear onerous to practitioners who will, nevertheless, accept the value of invariants and assertions. Tony Hoare has suggested that increased use of assertions is a way to introduce formalism to programmers resistant to a full frontal approach[5]. A hybrid approach in which proof obligations which look plausible but are not proved are checked by assertions would seem worth investigating, but raises the practical difficulty that we will have to check global assertions which are predicated over variables encapsulated within different agents. The best approach to this needs further investigation, and we are keen to experiment with including this functionality within our multi-agent IDE, which already sees all messages that pass between agents, and furthermore has been designed from the outset to be extensible.

The Cluedo Case Study has provided a stimulating basis for discussion between colleagues from different areas of Computer Science. The most significant aspect of the modelling described in this paper is perhaps the technique used to capture the partial visibility of information collected by eavesdropping players, who are only able to distinguish some details of a communication between two other parties. Fu-

ture work will continue to explore the manner in which code specifications emerge from event driven models. We will also be looking more closely at the nature of belief in competitive and strategic game playing. We will be doing further work on the scenario Cluedo provides, and in which agents are expected to co-operate with other agents according to the rules imposed on them, yet their objectives directly conflict with those of other agents. Agent intentions are potentially governed by the goals of gaining competitive advantage by reasoning about the beliefs of other agents for the purpose of manipulating or misleading them. This situation is present in many societal scenarios where governing rules must be followed and there is a requirement to cooperate and exchange information, yet conflicting goals lead to hidden sub-behaviours. From a formal perspective we will explore the relationship between what an agent knows to be true about another agent, and what is judged to be true (e.g. on the assumption that the other agent is employing a certain policy).

Acknowledgement. We thank the referees for their comments.

References

[1] J-R Abrial, M Butler, S Hallerstede, and L Voisin. An open extensible tool environment for Event-B. In *ICFEM 2006*, number 4260 in Lecture Notes in Computer Science, 2006.

[2] J-R Abrial and S Hallerstede. Refinement, Decomposition and Instantiation of Discrete Event Models. *Fundamenta Informaticae*, (77(1-2)), 2007.

[3] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[4] S Bayer, C Doran, and B George. Exploring speech-enabled dialogue with the Galaxy Communicator infrastructure. In *Proceedings of the First International Conference on Human Language Technology Research*, 2001.

[5] C A R Hoare. Assertions. In W Grieskamp, T Santen, and Stoddart W J, editors, *The Second International Workshop on Integrated Formal Methods*, number 1945 in Lecture Notes in Computer Science, 2000.

[6] S Lynch and K Rajendran. Boris - A Framework for Constructing Multi-Agent Systems in Lisp and Java. In *International Lisp Conference.*, 2003. Available from www.agent-domain.org.

[7] S Lynch and K Rajendran. Breaking into industry: tool support for multiagent systems. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2007.

[8] M Poppleton. The Composition of Event-B Models. In M Butler et al, editor, *ABZ2008 The First International Symposium on Unifying Theories of Programming*, number To appear in Lecture Notes in Computer Science, 2008.

[9] A. Rao and M Georgeff. BDI Agents: from Theory to Practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, 1995.

Proof and Model-checking, two complementary Approaches [1] [2]

Henri Habrias

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

Abstract

We show with a small example, a two-places buffer, how proof, refinement and model-checking can be used in a complementary manner. Our software tools are *Atelier B* and *LTSA*.

Keywords: Formal specifications, Teaching, B method, Model-checking

"Thus some people do not listen to a speaker unless he speaks mathematically, others unless he gives instances, while others expect him to cite a poet as witness. And some want to have everything done accurately, while others are annoyed by accuracy, either because they cannot follow the connexion of thought or because they regard it as pettifoggery."(Aristotle, Metaphysics, Book 1 Part 3)

1 Introduction

It exists different paradigms and specification languages. A presentation through a same case study is done in [5]. We quote [8]: "We can distinguish two verification approaches: - One putting into practice an inference system, *i.e.* a well defined and formalised set of reasoning rules issued of the logic (modus ponens, recurrence, equal values substitution, etc.); the support tools are proof assistants (PVS, HOL, Coq, AtelierB, LP,...) - The other is based on the construction of a model generally finite representing the transitions system representing the behaviour of the modelised application. The verification of the properties is done by an exhaustive exploration of this model. The support tools are based on the principle of the evaluation of the properties on a model, known under the name of *model checking* (MEC, CAESAR-Aldebaran, Auto, Spin, Cospan, SMV, ...)"

[1] The exercice on the buffer is taken from [3]. We want to thank Steve Dunne and Frank Zeida, from the University of Teesside who presented this exercise during their visits in IUT de Nantes on Erasmus exchanges.

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

We show in this paper, using an example, how to use the two approaches: the proof and model-checking to verify (check) a specification. We put them into practice using two tools: Atelier B and *LTSA* [6]. This example is used in our teaching for first year students in computer science [5]. Today the functionalities of these tools are available under the form of plug-in for Eclipse (for B, see Projet Rodin : *www.inria.fr/recherche/equipes/rodin.fr.html*)). We expose our small example, and then we deal with it with the B method and Atelier-B (using the proof) and with the *FSP* language and the *LTSA* workshop (using model-checking). The use of *ProB* is presented in another paper of this conference. A presentation of the two specification approaches comes before their use.

2 The two places buffer

We start this study considering the behaviour of a two places buffer. We do not take into account the data going through this buffer. We give to the students some observations of traces:
$[in, out, in, out]$
$[in, in, out, in, out, out, in, out]$

And we precise that the following traces are never observed:
$[in, in, in]$
$[out, in, out]$
$[in, in, out, out, out]$

Using the notation for the regular expressions, we can modelized the behaviour of the buffer in such a manner:

$$M = (in.(in.out)^* out)^*$$

We can also use the Jackson trees (Fig: 1) which have the advantage to permit the naming of the nodes and the adding of information on the leaves.

3 Using the B method

We do not detail the B method here ([1], [2]).

3.1 Classical B vs Event B

In event B, it is very convenient to construct the automaton and to refine it (the Harel automata are very practical for that) and at the same time to construct the B specification and its refinement.

Very often the operations of classical B (Figure 2) are not understood by the students. They use *preconditions* stronger than what the invariant necessitates to be respected. A symptomatic example is the one of the door. Even if they know that if the before state is $state = closed$ then $state := closed$ respects the invariant

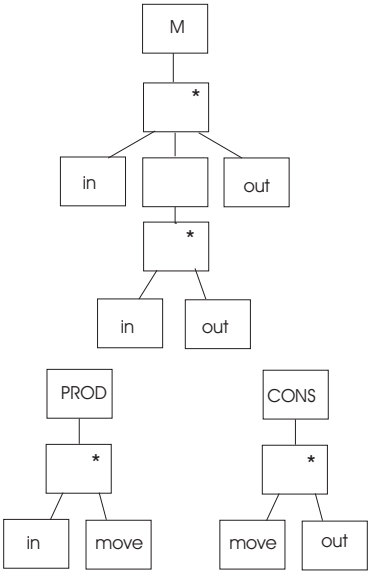


Fig. 1. The two places buffer, Jackson trees

$state \in \{closed, open\}$, they try to specify that is specified in Event B. So when we present classical B, we wait that some student said that our specification is bad, to speak about Event B (Figure 4).

We use the figures 3,5 to show that in Event B, we are considering the observation of a closed system. Nobody (excuses for E. Dijkstra [3]) asks an event to be executed ! An event occurs and the occurring of an event can be observed. A specified event (type) may occur. On the contrary somebody can ask to do an operation even an operation doing nothing. We have to pay attention to our discourse to the students: somebody can observe what the call of an operation does. If we call the operation *open* when the *doorState* is *open*, we can observe that the operation has been called and that the operation did not change the state of the door. So we concluded that it is better to say to the students in a first approach that an event occurs and an operation is called, than to say that an event is observed.

3.2 The parallel composition in B and the diamond

The parallel composition of two processes uses more often what we call the diamond semantic: to execute the sequence $P1; P2$ gives the same result than to execute the sequence $P2; P1$. With the definition of the parallel composition used in B, it is not necessary the case as it is shown on the following example. Considering the state $x = 3 \wedge y = 1$ before the composition of the two operations $op1 \mathrel{\widehat{=}} x := x + 2$ and $op2 \mathrel{\widehat{=}} y := x + 2$.

In B, the execution of these two operations composed in parallel ($op1 \parallel op2$) gives the after state $x = 5 \wedge y = 5$.
The execution of these two operations composed in sequence in such a manner $x := x + 2; y := x + 2$, gives the following after state $x = 5 \wedge y = 7$. The execution of

[3] *"The use of anthropomorphic terminology when dealing with computing systems is a symptom of professional immaturity"*, E.W. Dijkstra

MACHINE
$ClassicalDoor$
SETS
$STATE = \{open, closed\}$
VARIABLES
$doorState$
INVARIANT
$doorState \in STATE$
INITIALISATION
$doorState :\in ETAT$
OPERATIONS
open $\widehat{=}$ $doorState := open$ END;
fermer $\widehat{=}$ $doorState := closed$ END;
$st \leftarrow whatState \widehat{=} st := doorState$END;
END

Fig. 2. The door in classical B

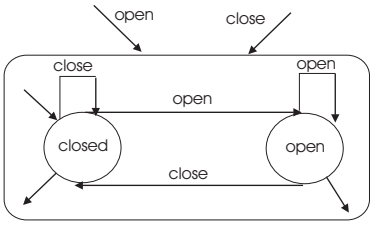


Fig. 3. The door in classical B

these two operations composed in sequence in such a manner $y := x + 2; x := x + 2$, gives the following after state $x = 5 \wedge y = 5$. The diamond semantic is not respected. The two sequences do not give the same state.

3.3 The buffer with the B method

3.3.1 A first solution

We modelise the buffer by a machine $M1$ (Fig. 9) which encapsules a variable giving the number of data in the buffer. We remark that we do not modelise the values of these data. The guards are modelised in B by a SELECT. Now, we analyse the buffer and it appears that it is composed of two cells (see Figure 6). One is considered as feeding the second. We name the first $PROD$ and the second $CONS$. For each one, we specify an abstract machine (Fig. 7, 8).

These two machines are included in a third machine $M2$ (Fig. 12). An including machine can use in the definition of its operations, the operations of the included machines. But the operations of the included machines are not part of the repertoire of the operations of the including machine. Nevertheless, it is possible to promote operations of the included machines, operations then becoming elements of the repertoire of the operations of the including machine. It is what is done in the machine $M2$ (Fig. 12). Every one of these operations being proved, we specify an

MACHINE
$EventDoor$
SETS
$STATE = \{open, closed\}$
VARIABLES
$doorState$
INVARIANT
$doorState \in STATE$
INITIALISATION
$doorState :\in STATE$
OPERATIONS
opening $\widehat{=}$
SELECT $doorState = closed$ THEN $doorState := open$END;
$closing \widehat{=}$ SELECT $doorState = open$ THEN $doorState := closed$
END;
$st \leftarrow whatState \widehat{=} st := doorState$ END
END

Fig. 4. The door in Event B

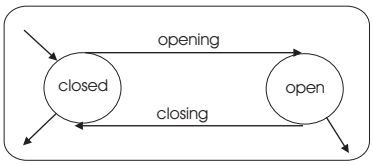


Fig. 5. The door in Event B

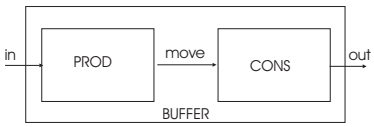


Fig. 6. The two places buffer

implementation machine which will permit us to prove that $M2$ simulates $M1$.

3.3.2 An other solution
We give a variante of the first solution.

4 Use of *LTSA*

4.1 Labelled transitions system and LTSA

The language *FSP* of *LTSA* is very simple. In the following we give the translation into *FSP* of some specifications written in the language of regular expressions.

4.2 From regular expressions to FSP

$P = aa^*$

MACHINE
PROD
VARIABLES
pstate
INVARIANT
pstate $\in 0..1$
INITIALISATION
pstate $:= 0$
OPERATIONS
in $\widehat{=}$
SELECT *pstate* $= 0$ THEN
pstate $:= 1$ END;
pmove $\widehat{=}$
SELECT *pstate* $= 1$ THEN
pstate $:= 0$ END
END

Fig. 7. Machine Producer

MACHINE
CONS
VARIABLES
cstate
INVARIANT
cstate $\in 0..1$
INITIALISATION
cstate $:= 0$
OPERATIONS
out $\widehat{=}$ SELECT *cstate* $= 1$
THEN *cstate* $:= 0$
END;
cmove $\widehat{=}$
SELECT *cstate* $= 0$ THEN
cstate $:= 1$
END
END

Fig. 8. Machine Consummer

```
P= (a -> STOP | a -> STATE),
STATE = (a -> STATE).
```

$P = (a, b, c)^*$

```
P= (a -> P |b -> P | c -> P).
```

$P = (a(cd)^*, b(cd)^*)^*$

MACHINE
$M1$
VARIABLES
$state$
INVARIANT
$state \in 0..2$ /* The state is the number of data in the buffer */
INITIALISATION
$state := 0$
OPERATIONS
$in \mathrel{\widehat{=}}$ SELECT $state = 0$ THEN $state := 1$
WHEN $state = 1$ THEN $state := 2$ END;
$out \mathrel{\widehat{=}}$ SELECT $state = 1$ THEN $state := 0$
WHEN $state = 2$ THEN$state := 1$ END;
move $\mathrel{\widehat{=}}$ **skip**
END

Fig. 9. Machine M1

IMPLEMENTATION $M1\_I$
* machine to prove that M2 simulates M1 *
REFINES $M1$
IMPORTS $M2$
PROMOTES $in, out, move$
END

Fig. 10. Implementation of $M1$

MACHINE $M1$
VARIABLES
$NumberOfData$
INVARIANT
$NumberOfData \in 0..2$
INITIALISATION
$NumberOfData := 0$
OPERATIONS
$in \mathrel{\widehat{=}}$ SELECT$NumberOfData < 2$
THEN $NumberOfData := NumberOfData + 1$ END;
$out \mathrel{\widehat{=}}$ SELECT $NumberOfData > 1$
THEN $NumberOfData := NumberOfData - 1$ END
END

Fig. 11. Another solution, machine M1

```
P= (a -> STATE1 | b -> STATE1),
STATE1 = (a -> STATE1 | b -> STATE1 | c -> STATE2),
```

REFINEMENT $M2$
REFINES $M1$
VARIABLES
$NumberOfData, cell1, cell2$
INVARIANT
$NumberOfData \in 0..2 \wedge$
$cell1 \in \{0,1\} \wedge$ /* number of data in cell 1 */
$cell2 \in \{0,1\} \wedge$
$NumberOfData = cell1 + cell2$ /* gluing invariant*/
INITIALISATION
$NumberOfData := 0$
OPERATIONS
in $\widehat{=}$ SELECT $cell1 = 0$
THEN $cell1 := 1 \parallel NumberOfData := NumberOfData + 1$
END;
$out \widehat{=}$ SELECT $cell2 = 1$
THEN $cell2 := 0 \parallel NumberOfData := NumberOfData - 1$ END;
$move \widehat{=}$ SELECT $cell1 = 1 \wedge \ cell2 = 0$
THEN $cell1, cell2 := 0, 1$ END
END

Fig. 12. Another solution : $M2$ refinement of $M1$

```
STATE2 = (d -> STATE1).
```

FSP is less powerful than the language of the regular expressions. Indeed, we cannot specify the case of the empty sequence. $P{=}a^*$ cannot be expressed in *FSP*.

4.2.1 Communication in FSP

FSP is very close to *CCS* of Milner ([7]. Whereas the communication between two processes in *CCS* is done by complementary actions - we note them by a prime - , in *FSP* it is done by actions (or events) of same name. So the following *CCS* specification

```
C1 = in.m. C1
C2 = m'.out.C2
System = C1 | C2
```

is written in LTSA :

```
C1 =(in -> m -> C1).
C2 =(m -> out -> C1).
||SYSTEM = (C1 || C2).
```

4.3 The absence of variables in FSP

We can qualify *FSP* of pure as we qualify *CCS* of pure relatively to "value passing CCS". In *FSP*, as in pure *CCS*, a variable is seen as a process. By example, a counter whose the values belong to the set 0..3 will be modelised in such a manner:

```
ZERO = (plus -> ONE),
ONE = (minus -> ZERO | plus -> TWO),
TWO = (minus -> UN | plus -> THREE),
THREE = (minus -> TWO).
```

If we want to modelise the input values, we have to consider as much channels as considered values. So a one-place buffer which can receive as input values, values belonging to the set 0..3 can be specified in *FSP* in such a manner:

```
range T=0..3
BUFFER = (in[i:T] -> out[i] -> BUFFER).
```

4.3.1 The guard in FSP

A guarded event is specified in *FSP* by a *when*.

4.3.2 From sequence diagrams to the labelled transitions system

MSC (Messages Sequences Charts) have been defined with an abstract semantic in terms of transitions system and of parallel composition. A tool [9] permits to translate a specification of scenarios into diagrams of states transitions. *LTSA* permits to detect the presence of "non implied scenarios". We do not deals with this aspect here.

4.4 Specification of the buffer

The figure 13 illustrates this specification.

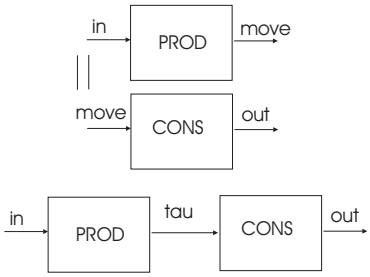


Fig. 13. The two places buffer, the parallel composition and the τ.

4.4.1 Specification of high level

```
BUFFER = STATE0,
STATE0 = (in -> STATE1),
STATE1 = (in -> STATE2 | out -> STATE0),
STATE2 = (out -> STATE1).
```

```
BUFFER(N=2) = STATE[0],
STATE[i:0..N] = (when (i<N) in -> STATE[i+1] |
                 when (i>0) out -> STATE[i-1]).
```

4.4.2 Specification of low level

```
PROD = (in -> move -> PROD).
```

```
CONS = (move -> out -> CONS).
||BUFF_2 = (PROD || CONS).
```

If now we hide the silent action, the *move*, this action is replaced by a τ. If we ask the *minimisation* of the automaton, we obtain then the same automaton as the one of the high level specification. So we have an equivalence between the two behaviours. In *FSP*, for the *minimisation*, two equivalences are considered. The *strong equivalence* considers that two systems are equal if they have the same behaviour when the occurrence of all their actions can be observed included the occurrence of the silent action. The *minimisation* uses this equivalence when there is no silent action. The *weak equivalence* considers that two systems are equal if they exhibit the same behaviour for an external observer who cannot detect the occurrence of the τ actions.

```
PROD = (in -> move -> PROD).
CONS = (move -> out -> CONS).
||BUFF_2 = (PROD || CONS)\{move}.
```

LTSA permits also to verify safety properties, as the absence of deadlock. A safety property defines a deterministic process asserting that this same process accepts every trace including the actions of the alphabet of this process. It permits also to verify the liveness properties. Such properties assert that something good will arrive eventually. In *FSP*, the temporal logic is not used and it is interesting for me because I have no time teach it. In *FSP* we are limited to a class of liveness property called progress, which is the opposite to the property of starvation. Such a property asserts that, whatever will be the state of the system, it is always the case where a specified action will eventually be executed.

5 Conclusion

In this short paper, we wanted to demonstrate how two approaches can be used to specify, refine and verify software. The B approach permits to manipulate variables as it is done classically in programming. The approach by Process Algebras as *CCS* does not use variables. A state is modelised by a behaviour. It is the possible behaviour when we are in this state. A state is an agent behaviour, following the vocabulary of Milner [7][3]. Some people can say that the approach is more abstract than the B approach, other people, on the contrary, considering that naming the states is an abstraction activity that is not useful in the process approach, can say that the approach is less abstract. It exists process algebras which use variables (they are named mixed approaches). Thanks to *LTSA*, we are obliged to modelise a variable by a behaviour, and it complicates the specification comparatively to what will be done with B. But otherwise with B the automaton is not explicit whereas *LTSA* products the automaton. Nevertheless the experience shows that very often to construct the specification in *FSP* or in *CCS*, we use the drawing of a transition diagram (see Fig. 14) which helps to structure the specification in *FSP*. So the two approaches are complementary. In B we are guided by the proof, in *FSP* we are guided by the execution generated by the tool and also by the automaton generated and his minimisation. It is possible, considering the automaton, restructurate the

specification in*FSP*. With *ProB*, we have a tool which is made to work from a B specification. It gives us much more functionalities than *LTSA* and allows us to make model-checking before the proof. When a proof fails we can see the reason why it failed using model-checking. So the proof and the model-checking techniques as those of refinement in event B and those of refinement by decomposition in processes are well complementary.

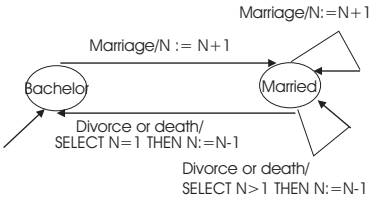


Fig. 14. Polygamy without polyandry, Harel Automata with guards for men

Our teaching is divided into two "modules". In the first one, we emphasize on the static aspects of the specification. *Classical B* and the *n-ary relational* model are used. In the second one, we emphasize on the dynamic aspects and *FSP* and *Event-B* are used. The connection between the two is done by exercises as the following.

We ask the students to modelise the different types of conjugal relationships using B or the notation of Codd for the *n-ary relational model*, and using automata and *FSP*. We show that *classical B* is not sufficient to represent the different cases. Here is an example of a question of one of our examinations :

Consider the following specification written in *FSP*. By MEN we want to denote "the behaviour" of the physical persons of sex masculine. To avoid an combinatory explosion of the number of states, we limited to 2 the number of spouses in course. To simplify, we do not have considered the death of a man who was never married. So our specification is not a good model of the reality.

```
MEN(N=3) = BACHELOR[0],
BACHELOR[spouses_nb : 0..N] = (marriage -> MARRIED[1]),
MARRIED[spouses_nb : 0..N] =
(when (spouses_nb == 1) divorce -> MEN |
when (spouses_nb > 1) divorce -> MARRIED[spouses_nb -1] |
when (spouses_nb == 3) it_is_enough_for_a_man ->
divorce -> MARRIED[spouses_nb - 1] |
when (spouses_nb < 3) marriage -> MARRIED[spouses_nb + 1]).
```

Say what are the following relational schemas (without null values) that are not in contradiction with this specification ?

- Schema 1
  Mariages_in_course (id_married_men, id_married_women)
- Schema 2
  Mariages_in_course (id_married_men, id_married_women)
- Schema 3
  Mariages_in_course (id_married_men, id_married_women)

- Schema 4
  Mariages_in_course (id_married_men, id_married_women)

For every above schema, what B specification says the same thing?

SETS
$PERSON$
VARIABLES
$men, women$
INVARIANT
$men \subseteq PERSON \wedge$
$women \subseteq PERSON \wedge$
* B1 *
$isTodayMarriedWith \in married\_women \twoheadrightarrow married\_men$
* B2 *
$isTodayMarriedWith \in married\_men \twoheadrightarrow married\_women$
* B3 *
$isTodayMarriedWith \in married\_women \rightarrowtail\!\!\!\!\!\twoheadrightarrow married\_men$
* B4 *
$isTodayMarriedWith \in married\_women \leftrightarrow married\_men$
$\wedge\ dom(isTodayMarriedWith) = married\_women$
$\wedge\ ran(isTodayMarriedWith) = married\_men$

The answer is schemas 1 and 4.

References

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

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

[3] Fencott C. (1996) *Formal Methods for Concurrency*, International Thomson Computer Press

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

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

[6] Magee, J. Kramer, J. (1999),*Concurrency, State Models & Java Programs*, Wiley

[7] Milner R. (1989),*Communication and Concurrency*,Prentice Hall

[8] Monin J.F., Sifakis J. (1994), Eléments de classification des méthodes formelles, ARAGO, *Applications des techniques formelles au logiciel*, Vol. 20, pp. 765-795

[9] Uschitel, S., Kramer, J., Magee, J. (2003), Synthesis of Behavioral Models from Scenarios, *IEEE Transactions on Software Engineering*, IEEE", volume 29, number 2

Journées scientifiques - Université de Nantes (FR)

The B Method: from Research to Teaching

publié par APCB, juin 2008
ISBN 2-9512461-2-9
EAN 9782951246126