Teaching Model-Based Testing with Leirios Test Generator

Frédéric Dadeau, Régis Tissot

Teaching Model-Based Testing with the Leirios Test Generator

Frédéric Dadeau, Régis Tissot [1,2]

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

Abstract

This paper proposes a technique to encourage the interest of students in learning formal methods. Our course is focused on the B method, involving basic knowledge of set theory, invariance proofs, refinement techniques and so on. While lectures and tutorials cover a large range of such concepts, the practical work is focused on applying the principles of a model-based approach in the context of test generation. This paper explains the pratical outcome of the course, through the Leirios Test Generator tool, that gives an interesting and playful use of the B method, by simulating the execution of the model through animation, and by generating tests –based on the B model– that can be run on an implementation. In order to make sure that students will be interested in applying these techniques, we challenge them to play a game consisting in detecting mutants of a program with their model-based tests. The feedback from the students is very positive here, and suggests that formal methods are more likely to be understood if their interest is shown through a concrete application.

Keywords: B notation, model-based testing, proof, animation, testing, mutation

1 Introduction

The B method [Abr96] is one of the most common system development method used in France for teaching formal methods. The reason that motivates this choice is twofold. First, the B notation is well tool-supported by provers (the AtelierB [Cle04] or its free version, B4free [Cle]), animators (BZ-Testing-Tools [ABC+02] or ProB [LB03]), platforms (Rodin [Abr06], for EventB) or test generators (BZ-Testing-Tools, Leirios Test Generator (LTG) [3] [JL07]). Second, the B notation requires minimal basic notions of first-order logic, and set theory, which makes it a well-suited specification language for the teaching of formal methods.

The B method starts by the writing of a formal specification, named **abstract machine**, that gives a functional view of the system. This abstract machine is

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

[2] This work is partially funded by the Région Franche-Comté.

[3] LTG is the commercial version of the BZ-Testing-Tools developed at the University of Besançon.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

spiced up with invariant properties that represent properties that have to hold at each state of the system execution. It means that (*i*) the initialization has to establish the invariant, (*ii*) the operations have to preserve the invariant (meaning that if the invariant is satisfied before the operation, then it also has to be satisfied after the execution of the operation). Operations are written in terms of *Generalized Substitutions* that are built on basic assignments, composed of more generalized and expressive structures, that may e.g., represent conditional substitutions (IF...THEN...ELSE...END) or non-deterministic buildings (CHOICE, ANY).

The **refinement** steps consist in increasing the detail level of the model. The additional elements of the refinement have to be linked to the abstract machine. The refinement step can be iterated over and over again, until the highest detail level is reached. Finally, when the refinements steps are completed, an **implementation** may be written, based on the last refinement, that describes a ready-to-execute system. From there concrete code (e.g. C or Ada) may be generated. The interest of specifying by refinement is that the properties that have been established w.r.t. the invariants at each level are preserved through the refinement relationship. This considerably simplifies the writing of a complete formal model "at once" which is source of numerous errors/defaults, and it is also an help for the proof of the model.

B is the starting point for studying formal methods in our University. It is the first modeling language that is introduced to our students. Even if the language seems to be easy-to-learn and easy-to-use, there exists several problems to make them practice formal methods. As researchers in the domain of formal methods, we exhort our students to practice formal methods in the software engineering process, even when they will be working in industry. The major problem is that they usually do not see the concrete application of writing a formal model.

This paper proposes an approach, based on a concrete application of the formal methods that helps the students to understand their usefulness. The idea is to force them to work with a model-based test generator that represents a concrete application of formal modelling. Section 2 presents the formal methods course to which the students attend. Then, we present the tools we use in practical work session in Section 3, namely B4free for the proof part, and LTG for the testing part. The description of what has to be done during the project is given in Section 4. We present the case study on which the students work as a practical session project in Section 5. Finally, Section 6 analyses the results obtained in this "teaching experiment", before concluding and presenting the future courses in Section 7.

2 Presentation of the Course

B is taught in our university to the master students of computer science; the main notions of first-order logic and set theory are seen two years before. The main problem with the B-method is its lack of concretization. Most of the students complain about the too high abstraction level of the models. Their greatest difficulty is to understand how models can be employed, and thus, they doubt of their usefulness. A corollary difficulty in learning the B-method is to see how do an abstract B model and a concrete code implementation relate to each other, due to the difference of abstraction level between them.

2.1 Theoretical Sessions

Lectures cover the basics of B notation, and B method. This include abstract machines, proof obligations generations, machine compositions, refinement and implementations. Tutorials concretize these concepts, through exercises.

Our objective is to use formal methods in a concrete way so as to ensure that the students understand their usefulness. Thus, practical sessions cover a different aspect of B method, by illustrating the use of formal models in the software development process.

2.2 Practical Sessions

Practical sessions start by learning the basics of B notation, using the JEdit editor improved with the B plug-in, which offers a toolbar for mathematical symbols and a typechecking feature. Then, students are asked to prove their models using the B4free prover. One of the first reaction of the students after having written and proved their model is: "ok, and so what? can it be executed?". This question is quite logical since the machine represents a system, or a program, and a natural reflex is to check whether it acts as expected.

Unfortunately, proof can hardly be used for that step. Thus, we use the Leirios Test Generator tool, which provides an user-friendly animator in order to validate the behaviour of the model.

2.3 Student's point of view

We noticed that it is quite complicated to teach students a refinement approach. It appears that they are expecting concrete lessons; doing the effort of writing one model is acceptable, but adopting a complete refinement-based approach represents for them a waste of time. Note that this vision is shared by most of our industry partners.

3 Tools

3.1 B4free/Click'n'prove

Click'n'prove [4] is a graphic interface (see Fig. 1), developed by J.-R. Abrial and D. Cansell, for the interactive prover B4free, which is a single-user version of the "Atelier B" developed by ClearSy System Engineering.

3.1.1 Interest of the tool

The main objective of using Click'n'prove is to provide to students a complete framework including formal modelling and verification. Typechecking and proof features of this framework make the students apply the notions they learned during the theorical sessions. Students are ask to produce a model which is both syntactically correct and consistent.

[4] http://www.loria.fr/~cansell/cnp.html

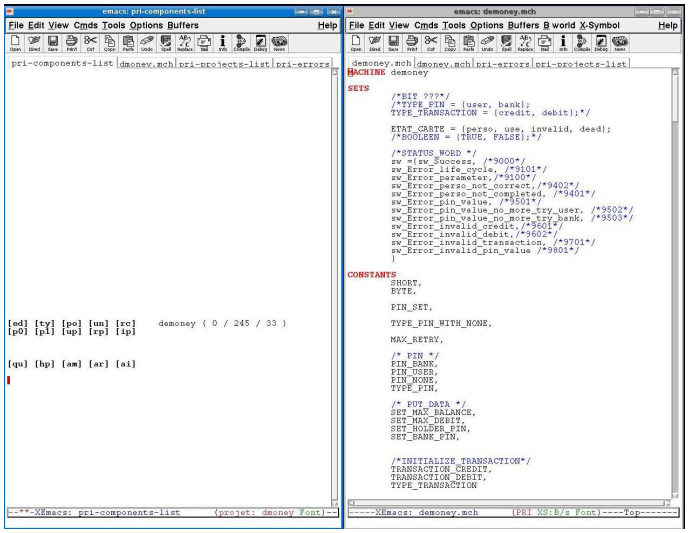

Fig. 1. A screenshot of the Click'n'Prove GUI

Working with this tool shows the students what is an interactive prover, the automatic proof features and their limitations. When automatic proof fails, the interactive proof mode forces the student to understand the proof obligations' definitions, and mechanisms. Then they have to correct their model or to solve by themselves the proof obligations with the interactive proof support.

3.1.2 Features of the tool

Click'n'prove is grounded on the B-method introduced by J.-R. Abrial [Abr96]. This methods aim to define the development of computer software process based on refinement. In order to support a B-method based development process, Click'n'prove provides two functionalities. The first one of these features is the *typecheck* functionality that makes it possible to check whether a B model is syntactically correct.

The second feature of Click'n'prove is the *proof support*. First, the software generate proof obligations have to be proved to ensure that all properties expressed in the invariant are still satisfied after each operation of the system and that all properties expressed over a more abstract level are still verified through its refinements.

The automatic prover solves most of the proof obligations. When some proof obligations are not automatically proved, the user enters the interactive proof mode. In this mode, the proof obligations are summarized as a list of hypothesis and goals to reach. The user disposes of some features to solve the remaining proof obligations, such as proof by case, proof by contradiction, selecting/removing hypothesis from the invariant, add assertions, etc.

3.1.3 Which features do we use ?

The type-checking confronts the students to their syntactic errors. Students realize that even though the specification of a system seems informal, the related model should be formal. Generally, syntactic errors relate to multiple variable assignment in one operation, incomplete initialization of variables, or mismatch between sets and elements.

Through the proof obligations solving process, students are confronted to the incoherence of their model. When a property fails to be proved automatically, they have to discover the problem, and thus, they need to understand the proof obligations. First, they have to verify if the proof obligation can be proved. If not, they have to answer to some questions, e.g., "Are the invariant properties correct ?", "Are there conflicting properties in the invariant?" or "Do all the behaviours of the operations preserve the invariant?" in order to locate their mistake(s). Otherwise, if the proof obligation can be proved, they have to answer other questions, e.g., "Does the invariant contain all the properties needed to solve the proof obligation?", "How is it possible to help the prover solving this proof obligation? by case? by contradiction? etc.".

3.1.4 Feedback from the students

The main problem is that most of the student find that the emacs-based graphical interface is not intuitive. This essentially results from the following reasons. Click'n'prove does not have a graphical interface like the majority of the software they use generally. The tool suffers from a lack of flexibility: a model can not be modified out of the Click'n'prove editing window, otherwise the project is locked. Finally, the substitution of mathematical symbols is systematic (e.g. string "pin" is displayed "πn" in the editor window).
Nevertheless in practice, they quickly accommodate to this tool. Contrary to Click'n'prove, the second tool we consider, LTG, has a user-friendly graphical user interface as described below.

3.2 Leirios Test Generator

LTG is a software developed by LEIRIOS Technologies, it is a test generator based on constraint solving techniques (see Fig. 2).

3.2.1 Interest of the tool

In practice, the students do not refine their model to generate the code of an implementation. Thus, thanks to LTG, students confront their model to a real implementation. The model animator of LTG offer a complementary approach for the students to validate their model. The tests generator feature introduces them the notions of coverage, verdict and search heuristics.

3.2.2 Features of the tool

LTG includes a model animator which is used to validate the model or to manually define tests sequences. This animator verifies the preservation of invariant properties after operation calls. It makes it possible to check whether the model behaves as expected in the informal specifications.

For each test generation campaign, the user can set some parameters which are crucial for the results, namely: the initial state of the system, the state to reach by a test sequence –crucial if one wants to concatenate tests sequences–, the selection of the operations to cover, the condition coverage criteria, some other criteria e.g. transition pair coverage, the observation of the system after a test, the heuristic em-

ployed to reach the test targets, including *preambule helper* strategies (ie. operation sequences defined by the user to help the software to find tests sequences).

3.2.3 Which features do we use?

Each student's B model is given as an input to LTG. The animator helps the student to understand which behaviours are covered by the model. This feature can be used to "replay" a test sequence and analyse the state of the system in order to find more precisely why the test sequence fails. Then, LTG is used to derive test sequences based on the student's model analysis. In order to generate relevant tests, students have to precisely set the campaign generation parameters. During the practical work sessions, they have to produce test campaigns.

For each test campaign, students have to define a initial state and the set of operations which are tested. Then, for each tested operation, they select the coverage criteria for the conditions and the sequence of observation. The use of observation operations shows them the importance of having an accurate verdict for a test sequence –the more comparisons are done, the more conformance verdicts are precise.

In addition, students have to set up strategies for test target search. They can concatenate different automatic strategies –width-first, forward/backward search with best first, preamble helpers. The preamble helpers strategy forced them to forecast the system states which are the most difficult to reach.

3.2.4 Feedback from the students

Although LTG is user-friendly, beginners may experience problems for setting the parameters of the test campaign. For instance, setting the accurate model coverage criteria requires to find the trade-off between the number of tests, their relevance

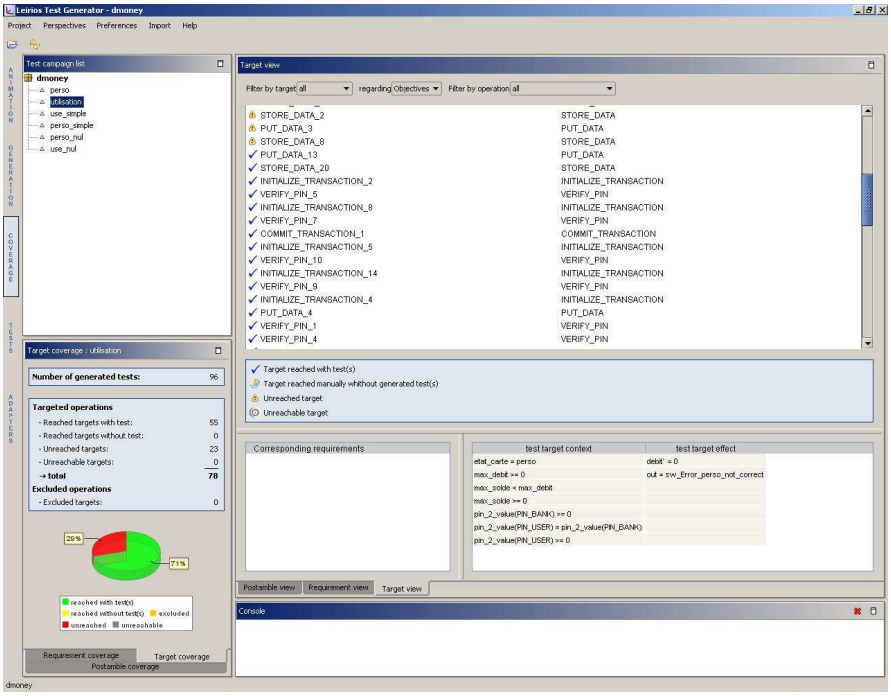

Fig. 2. A screenshot of the Leirios Test Generator GUI

and the capabilities of the test generator. The other main problems for the students are the addition of system observations after the tests and the configuration of the heuristics applied to reach the test targets.

4 An (Supposedly) Interesting Project

Before going into the detail of the project assigned to the students, let us first introduce the notions that we want to illustrate with the project.

4.1 Promoting Interest

First, we have to find a theme that will catch the students' interest. Being involved in research projects in embedded software, and especially smart cards, we have noticed that these represent a great challenge in the software industry, and one of the only domains in which significant efforts are made to use formal methods. We plan to make students model a system of a banking application, namely an electronic purse. In order to make it sound real, we use a simplified version of the Demoney applet [MM], a demonstrative electronic purse, that describes a realistic behaviour of smart cards, and contains enough details to be quite difficult.

Then, we have to define the goal of the project, that will guarantee that the students spend time on their models. Our first thought was to make them successively refine their models and generate an implementation. Nevertheless, this does not seem to be very motivating: smart card applets do not necessary require a refinement approach to be modelled, and the time available for the project would have been a bit too short.

4.2 Maintaining Interest

One of the greatest difficulty in writing a formal model is to know what degree of precision has to be put in it. This fact is a crucial point that has to be well understood by the students. In general, the accuracy and precision of the model is highly related to the degree of motivation shown by the students, including various external factors, e.g. the time they devote to the project.

Our objective is therefore to ensure that students will be putting efforts writing an efficient model that is both complete and coherent. For this purpose, we propose a new approach which consists in confronting their model to a real world application. This constitutes a validation challenge that requires them to write the most accurate possible model. To do so, an informal specification is given to them. This latter is based on the original specification of Demoney, and thus, it teaches the students to decode such documents.

Following a model-based testing approach [Bei95], our idea is to ask the students to build a model that they first have to prove, in order to ensure its coherence, and from which they have to derive test cases. This exercice is already practiced in the practical work sessions, during which the tests are run on a correct implementation that is used to help them making sure that their model is correct. Once this step is done, they are asked to run their test suites on several mutations of the original implementation, in order to check the accuracy of their model.

In our context, the task is different. The correct implementation is given to them, along with the mutants, but without any identification of the correct and faulty implementations. The challenge for the students is to find, among all the implementations, the one that is correct and explain why the others are wrong, ie. which part of the informal specification has not been respected by the implementation, based on the analysis of their tests.

In order to avoid copying out results on mutant detection, each team recieves a different archive, that contains a "personalized" set of 20 implementations. In order to increase the challenge, each team is given a different set of mutants and a diffferent file name relating to the correct implementation.

5 Case Study

The study case we propose to the student is based on the specification of *Demoney* –Demonstrative Electronic Purse– developed by *Trusted Logics* for research applications. No implementation of this specification is used in the real world, but it is closed to credit card applets and therefore pertinent. Demoney is a good example of a critical system, a banking transaction, which requires extensive validation and verification, simple enough to be used by students.

Our specification describes a card with four life cycle states: a personalization state, a use state, an blocked state and a disabled state. The card is secured with two PIN objects –user and bank– which are associated to retry counters values. Moreover, the card contains information e.g. the current balance, the maximal balance and the maximal debit.

During the *personalization* phase, the PUT_DATA operation makes it possible to set the informations and the objects of the card. Then, the STORE_DATA operation terminates this phase if and only if all the informations are setup and coherent with the security policy. Once the card has quit the personalization phase and started a use phase, it will never come back to the former state.

During the *use* phase, the card holder can invoke some commands to gain authentication on PIN –VERIFY_PIN– and to initialize a financial transaction –INITIALIZE_TRANSACTION– that has to be completed –COMMIT_TRANSACTION. If the user fails so much times to be authenticated that the retry counter of the user PIN reached zero then the card become *bloked*.

When the card is *blocked*, the authenticated bank can unblock the user PIN and change its value. But if the card is blocked and the retry counter of the bank PIN reaches zero then the card become *disabled* and will never be used. In addition, the implementation presents observation operations to retrieve the current balance, the maximal balance, the maximal debit and the state of authentication of the PINs.

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

but can not be formalized as invariant properties. Thus, the students have to select what they are able to model as invariant properties or not. For instance, an invariant property over the system is: "If the card is not in the personalization phase then the balance can not be negative or higher than the maximal balance". On the contrary, the property:"All initialization of a transaction must be immediately followed by a transaction confirmation, otherwise the transaction is cancelled", can not be formalized as an invariant property.

6 Results Obtained

6.1 Evaluation of the students' work

In order to evaluate the students' work, we have defined scoring criteria based on the modelling choices, the understanding of the proof obligations and the coverage of tests. In order to evaluate the modelling choices as well as the understanding of the proof obligations, we ask the student to write a report. This latter includes explanations about the data strutures of the model, the behaviours of the operations and the invariant properties they define in order to ensure the conformance with the specification.

In order to take into account the coverage of the model and the quality of generated tests –number of tests, behaviour coverage and observations– we have based a part of the evaluation on the detection of mutants. Mutants are classified into three sets: the easily detectable ones with basic behaviours coverage criteria, those detectable with observations, and those that can only be detected with improved observations and behaviours coverage criteria selection. This simplifies the evaluation of the work: the harder the mutants are to be killed, the best is the students' score.

6.2 Feedback on the project

First of all the idea of a project appears to be a good idea. Since most of the students are only motivated by graduating, the aim is to make them learn without having noticed it. One interesting point is that the informal specification of the system they have to model is very dense, and thus, they are forced to conscienciously read the document in order to extract the pertinent informations.

The fact that the project is inspired from a real world application is also a good point, since they can really see how formal methods can be applied in the industry.

We believe this is very important that the students' first contact with formal methods links with concrete application. Thus, they understand better the usefulness of writing a model. In the years after, it will then be possible to teach them more abstract notion, that can not necessarily be concretized.

7 Conclusion and Future Courses

We have presented in this paper an approach for motivating students to learn formal methods. Our idea is based on showing the concrete application of formal methods instead of teaching it at a very abstract level, and without any link to real-life

concretization. Our solution is to use a playful and practical approach, that makes the students write a model for something interesting. We use an academic tool, named B4free, to perform the proofs of the B models we consider. We also rely on a commercial tool, named LTG, to provide a complete framework for the animation of the model. These tools are further used in a project inspired from a real world application of the formal methods in the Smart Card industry.

Since LTG is a commercial tool, universities are supposed to buy licences to run the tool. Nevertheless, a free alternative can be used. The ProB tool can be used instead of LTG in order to perform the animation. In this case, its flash-based plug-in for animation can be employed to improve the attractiveness of the interface. Concerning the test generation part, recent evolutions of the ProB authors [SLB05] let us think that the same mechanisms as LTG can be recreated at a minimal cost.

We regret in this project that the time was too short for making the students use a refinement-based approach for writing the model given to LTG. Nevertheless, such a change in the course planning is currently under study. One other improvement for the project is to adapt our course to the Event-B formalism and especially the Rodin plateform [Abr06] which appears to be the best current support for the Event-B systems. We are also looking forward to use some JML annotations [BCC+05], in order to improve the test verdict. Moreover, JML presents several possibilities for modelling programs with a few efforts. Finally, the use of real cards onto which the program is embedded, in order to increase the realism of the project, is currently under study.

References

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

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

[Abr06] Jean-Raymond Abrial. Tools for developing large systems (a proposal). In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, volume 4157 of *Lecture Notes in Computer Science*, pages 387–390. Springer, 2006.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

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

[Cle] ClearSy. B4free web site. `http://www.b4free.com`.

[Cle04] ClearSy. B reference manual v.1.8.5. 2004.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated Test Generation from B Models. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 277–281, Besançon, France, January 2007. Springer.

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

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

[SLB05] Manoranjan Satpathy, Michael Leuschel, and Michael Butler. ProTest: An Automatic Test Environment for B Specifications. *Electronic Notes in Theroretical Computer Science*, (111):113–136, January 2005.