A Formal Method Teaching Experience: Why and How We Evolved From a Correct-by-Design Content to a More Pragmatic Content

Marie-Laure Potet[1][3], David Monniaux[2][3], and Cyril Prévé[4]

[1] Grenoble-INP (Ensimag)
[2] CNRS
[3] Vérimag, centre équation, 2 avenue de Vignate – 38610 Gières, France,
Marie-Laure.Potet@imag.fr, David.Monniaux@imag.fr
[4] The MathWorks, 100 C Allée Saint-Exupéry, 38330 Montbonnot Saint-Martin, France, cyril.preve@mathworks.fr

Abstract. Twelve years ago we introduced a course in formal methods, mainly based on the B method, at Ensimag, a renowned French school in mathematics and computer science and engineering. Due to the evolution of our curriculum and, above all, due to the evolution of the industrial context w.r.t. formal tools, this course has evolved towards a new content, that aims to train our students to understand and evaluate the variety of existing static analysis tools. In this article we present this evolution, the body of knowledge we have selected and some teaching resources.

1 Introduction

Twelve years ago we introduced a B course at Ensimag,[5] a renowned French school specialized in mathematics and computer science and engineering. [6]

During this long period, this course has been subjected to many evolutions, due first to our better understanding and knowledge of the domain and, more importantly, to our perception of the notions that students are really able to acquire and master. Furthermore, during the last two decades, the external context has changed: formal techniques have acquired a new status in industry. Due to safety and security requirements, a lot of tools and approaches have been developed, allowing to verify some dedicated classes of program properties. Teaching formal methods, and more importantly the underlying concepts, is thus no more considered an esoteric idea. We can even go further and affirm that future software engineers have to be somewhat educated in formal methods.

The aim of this article is threefold: first to share our teaching experience and exercises, secondly to explain why the B method supplies a well-adapted

[5] http://ensimag.grenoble-inp.fr/

[6] In the company of Didier Bert, Marie-Laure Potet also gave lectures to the "Ecole des jeunes chercheurs en programmation", dedicated to PhD students in the domain of programming. This school is supported by the French CNRS-GPL workgroup. http://ejcp2010.inria.fr/programme10.htm.

framework for a formal method course and finally to present the evolution of the content of this course from a top-down content (correct-by-design approach) to a bottom-up content (code verification with abstraction). This article extends and reuses [13], in which a fine presentation and evaluation of our B method course is detailed. In Sec. 2, we present our experience about the B method course, with an evaluation of how some notions have been introduced with which efficiency in regard of students. In Sec. 3 we describe the new course relative to semantics and program analysis, pointing out the objectives and how we try to reach them. We conclude by an evaluation of this first year of the new course and our planned improvements.

2 First Period: the B Correct-by-Design Method

The course dedicated to the formal aspects of programming takes place in the second year of Ensimag, corresponding to the first year of a Master degree (fourth year of higher education). It was initially optional, attracting students interested by theoretical aspects of computer science. The audience was composed of future engineers having a taste for technical and fundamental aspects and with some abilities with abstraction, due to a solid background in mathematics.[7] Our students thus have little difficulty with mathematical notions such as set theory. More important, they are also able to manipulate several levels of formalization such as syntax, semantics and reasoning about semantics without too much difficulty. In addition, Ensimag students have some background in logic, computability, compiling (implementation of a compiler for a small object-oriented language) and rigorous algorithmics.

2.1 Objectives

The first version of the B method course was proposed as the successor of lectures on formal specifications, in which many approaches were presented (algebraic specifications, model-based specifications, refinement notions as in VDM and Z, ...). At this stage the choice was to focus on the B method, which integrates the main steps of formal development into a single framework. Furthermore, since this method was supported by robust tools, the challenge was to conciliate practice with a fine understanding of theoretical concepts. The course had three ambitious objectives, met according to students' competences:

1. the ability to formalize behaviors and properties with the help of abstract languages;
2. the ability to understand the semantics of programming languages and to reason about programs in a formal way;
3. the ability to understand the methodological and theoretical concepts underlying the correct-by-design principle.

In the sequel we detail these three objectives, and how we try to reach them.

[7] Most students come from "classes préparatoires aux grandes écoles" emphasizing mathematics and preparing for competitive entrance examinations to e.g. Ensimag.

2.2 Modeling Aspects

Here the aim is that students should be able to write formal specifications and expected properties. With respect to this aim, the B method offers a very pleasant framework, because students do not have to learn wholly new languages. First order logic and set theory are expressive enough and, above all, well known by our students. Abstract behaviors are described using the generalized substitution language, which is not much more difficult to comprehend. Contrary to assertion languages or pre/postconditions, two difficulties are avoided: we do not have to introduce the notion of before-after variables (variables and their primed versions as in Z or TLA for instance), and we stay within an imperative specification style, natural for our students. Variables that are not assigned are implicitly left unmodified.

The difficulties met by our students are intrinsically related to the specification activity, in particular nondeterministic behaviors in specifications, an unnatural concept for programmers. It is also not obvious for them to see programming data structures as mathematical objects. Let us see here examples illustrating data structure modeling, non determinism and properties (see the Ref. column for references).

Examples	Illustrated notions	Ref.
a memory allocator	nondeterministic behaviors	[8]
a dynamic class loader or a RBAC security policy	specification based on relations and hierarchy as trees	[17] [13]
a simple elevator	expected properties formalized by invariants and dynamic behaviors	[8]
arrays, lists, ...	data structures are mathematical objects	[1]

The allocator example introduces two sources of non-determinism: we do not know whether the allocator can or cannot supply a memory cell, and we do not know which address will be returned.

2.3 Reasoning about Properties and Programs

The second objective was that students should be able to reason about behaviors (specifications or programs) in order to prove some properties. Here we introduce the weakest precondition calculus [5] and the notion of proof obligations (or verification conditions). For the generalized substitution language used in B, the weakest precondition is not too complex to comprehend. but richer than the classical while language because it includes nondeterministic specification constructs. It is an opportunity to discuss the different forms of non determinism (angelic versus demoniac).

Even though our students have some knowledge about decidability vs undecidability, it is very hard for them to be confronted with this notion: if my property is not proved, is it due to my specification or to the weakness of the automatic prover? To alleviate these difficulties, we have introduced exercises

tailored for lab classes: components are partially specified and, if students state the right formulae, proofs are automatically established[8] (see sources in [7]).

Examples	Notions that are used
gcd program	proofs of iteration and absence of overflows
finding a given element in an array	a sophisticated iteration invariant

2.4 Semantic Modeling

Another aim is that students should be able to specify and understand some fine aspects of programming language semantics and to reason about them. Two new features are considered: function calls and exceptions.

The first extension, not detailed here, consists in formally defining function calls for two modes of parameter passing and in studying these definitions w.r.t. invariant preservation. We prove that call-by-value preserves invariant, contrary to call-by-reference (see [2, 13] for details). In the second example, following Lilian Burdy's work [3], we extend the generalized substitution language with exceptions and two new primitives (raise and catch). We also illustrate the notion of correctness by proving that this extension is consistent with the classical weakest precondition calculus, up to a program transformation, as proposed in [13].

First, students have to define a new weakest precondition calculus $wpe(S, F)$ with F a function mapping exceptional exits to postconditions, i.e. $F \in EXC \nrightarrow Predicate$ with EXC the set of exception names, including a distinguished constant no corresponding to normal behaviors. Secondly, they have to define a translation $\mathcal{C}(S)$ from programs with exceptions into equivalent programs without any. Let here some very simple examples (with exc a new variable):

$\mathcal{C}(x := v) \mathrel{\hat{=}} x := v \ ; \ exc := no \qquad \mathcal{C}(\text{RAISE } e) \mathrel{\hat{=}} exc := e$

Finally they have to prove the correctness of their wpe calculus, w.r.t. the $\mathcal{C}$ translation, in establishing the following equivalence, for some substitutions as assignment and sequencing (see [8] for these proofs):

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

2.5 Correct-by-Design Development Process

The third objective was to provide students with an initiation to the correct-by-design principle, using refinement. We focused on the following notions:

- refinement: its intuitive definition in term of proof obligations
- refinement properties like transitivity and monotonicity allowing us to use refinement in a development process (stepwise and partwise refinement)

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

– correctness and completeness of refinement proof obligations w.r.t. a semantic characterization of refinement.

Our students do not have particular difficulty writing refinement examples with their associated invariants, because they are familiar with abstraction and data representations. Nevertheless, the definition of refinement proof obligations is intrinsically abstruse for them (in particular in the case of data refinement) and general properties (transitivity, correctness ...) seem very very abstract and meaningless. Here are examples we used as laboratory exercises:

Examples	Notions that are used	Ref.
modeling and controlling a lock	a simple data refinement	[7]
a booking service example	a global development	

The last example is developed in several documents[9] (in French). It is also used to illustrate proof obligations relative to iteration and how these proof obligations depend on the initial specification. More concretely, depending on whether the specification only imposes to choose a free seat or the free seat with the smallest number, the concrete invariant differs (see [13] for more details).

Correctness and completeness properties w.r.t. B refinement proof obligations are interesting for several reasons: first they are based on a semantic definition of refinement in term of component substitution principle (the contract metaphor) and secondly they allow to illustrate the classical notions of correctness and completeness of an operational procedure with respect to a semantic definition. Teaching material can be founded in [13] and is inspired by [14]. Incompleteness of B refinement proof obligations (equivalent to L-simulation) is illustrated by the casino example, borrowed from Steve Dunne [6].

2.6 Conclusions Relative to the Correct-by-Design Course

In a top-down design process (from abstract models until implementations) students become comfortable when implementations are tackled. They rediscover known notions and a formalization of their usual practice. Furthermore, it seems difficult for them to integrate in the same time many theoretical notions such as a specification language, the weakest precondition calculus and the refinement theory, in addition to the methodological aspects of formal development process. Then a seemingly more appealing approach for students is to focus first on a proof of program approach, starting from their background and knowledge. The fact remains that some non trivial notions, such as formal semantics and reasoning about programming data structures as mathematical objects, are not easy to master. Furthermore, because we are at the program level, students are expecting that we can deal with real programs, that make the semantics definition more complex.

[9] `http://www-verimag.imag.fr/~potet/presentationB.acc.ps`.

The B framework has proved to be a very nice framework for teaching and has been reused for the new draft of this course. In particular the use of a unique language (the generalized substitution language) to write both specification and program in an imperative style is really an advantage, as well as the free AtelierB tool, which is very easy to download and install.

3 Current Period: a Pragmatic Static Analysis Course

3.1 The New Context

The curriculum of Ensimag is being modified in order to be closer of the "LMD" format.[10] In the Information System Engineering curriculum, it was decided to impose a compulsory course with some formal contents. The audience should now be around 50-60 students.

Furthermore, as pointed out in the introduction part, formal methods have acquired a new status in industry. Due to safety and security constraints, a lot of tools and approaches have been developed, allowing to verify some dedicated classes of program properties. Because our future engineers have to know the state-of-the-art technologies, we have decided to study a larger set of formal verification techniques, from rapid and very approximate tools to precise but interactive approaches, depending on the goals of the verification process (bug finding, verification of some particular form of properties, proof of assertions ...). *Our aim is that, at the end of this course, our students must be able to evaluate the appropriateness of a given tool in regards of some needs or, equivalently, which techniques can be deployed for a given problem.* That means that they should become acquainted to:

- advanced formal verification analysis techniques;
- undecidability and complexity results (Rice's theorem for instance);
- theoretical and methodological difficulties (aliases, compiler or processor dependent analysis ...).

3.2 Motivating examples

Here is a motivating example for understanding the possible levels of verification accuracy. In Java, for security reasons, variable initialization is mandatory. Local variables are not initialized by default, then conforming compilers must verify that these variables are initialized according to simple dataflow rules. For instance the following example is rejected at the compile time (generally students do not believe it!):

```
class Security1 {
  public static void main (String args[])
  {int i = 1 ; O1 o1 ; O2 o2 ;
```

[10] Following the Bologna Accords, higher education should be structured in 3 years of Bachelor ("licence"), 2 years for master and 3 for doctorate.

```
  if (i==1) o1=new O1(); else o2 = new O2();
  if (i==1) System.out.println(o1.f1());
  else System.out.println(o2.f2());}}
```

On the contrary, compilers must accept the following code, including typecasts that could be proved correct (they never result in failed dynamic checks) with more sophisticated techniques:

```
class Security2 {
  public static void main (String args[])
  {int i = 1 ; Object o ;
   if (i==1) o=new O1(); else o = new O2();
   if (i==1) System.out.println(((O1)o).f1());
   else System.out.println(((O2)o).f2());}}
```

We also used the MISRA standard (Motor Industry Software Reliability Association) [10] to illustrate coding guidelines that require different analysis algorithms (from lexical constraints to undecidable ones, through control flow analysis). Here is some MISRA rules associated to pointers and arrays:

1. pointer arithmetic shall not be used (rule 101) or no more than 2 levels of pointer indirection should be used (rule 102)
2. Relational operators shall not be applied to pointers except where both operands are of the same type and point to the same array, structure or union (rule 103)
3. the null pointer shall not be dereferenced (rule 107)

Rules 101 and 102, on the one hand, can be implemented with simple syntactic and typing rules. Rules 103 and 107, on the other hand, are undecidable in general; thus checking them could require more sophisticated algorithms, based on value analysis and possibly a fine-grained memory model. Furthermore, such techniques are bound to be incomplete (they reject correct programs) and/or unsound (they accept incorrect programs).

The MISRA standard is a good example of the confusion faced by students. The standard mixes indistinctly stylistic rules and rules with serious semantic impact, decidable requirements and undecidable ones. It is thus impossible to write an exact checker for conformance to MISRA-C; at best, one can perform exact checks for the decidable rules, and conservative checks for the undecidable rules: a warning is given unless the tool can prove that the program abides by the rule. Thus, one objective of the course is to impact the knowledge of such distinctions to the students.

3.3 Course content and sessions

The two challenges related to our objectives are:

1. To present a variety of static analysis approaches (from simple analyzes such as dependency analysis, to abstract interpretation and even interactive

proofs) as a continuum and, as much as possible, within the same framework, in order to minimize notations. So we present forward and backward assertion propagation analyzes (with their pros and cons) and introduce abstract interpretation technique as a restricted form of assertions for which proper algorithms are tailored.
2. To convince as much as possible our public. Some students feel content with theory, but many have to practice to be convinced. We therefore have a series of exercises that have to be developed using tools.

The duration of this course is 36 hours including exercises and the first edition takes place in 2010. Here is the precise content of this course (lesson resources are available at [15]):

Session 1	Introduction
Sessions 2-4	Assertions, WP, proof obligations
Session 5	AtelierB exercises: examples presented section 2.3
Session 6-7	Model-checking, predicate abstraction principle
Session 8	Intervals and polyhedra
Session 9	Polyspace Verifier from MathWorks company
Session 10	TP Polyspace Verifier: an exercises suit and student codes
Session 11	Beyond simple programs: thread, pointer, floating numbers ...
Session 12	Back on static analysis techniques and code verification

First part reuses a significant part of resources presented in the previous sections. The abstract interpretation part does not introduce too much theory but mainly focuses on automation (fixed point iteration) and accuracy (see subsection 3.4). Session 11 is dedicated to problems introduced by non trivial programming language features. In particular we detail alias analysis (Steensgaard's algorithm [16]) and several form of memory models. In Session 12 we come back to the notion of approximation (false positive/false negative), how notion of correctness and completeness can be defined (applied here to run-time error detection) and which analysis can be chosen, depending on the verification objectives. For instance some tools, such as Coverity, prioritize efficiency and speed over soundness; a tool may for instance skip parts of programs that it cannot analyze precisely. This is very reasonable for finding bugs, but of limited use for certification.

Certain features of the C language and their common implementations are often largely misunderstood. For instance, the following C program may, depending on the architecture,[11] the compiler and optimization options [11], produce an assertion failure on the z != 0 condition, which is surprising given that z is not modified between the two z != 0 tests; in fact, a prover based on Hoare logic will typically assume that the two z != 0 expressions have the same truth value. The reason is that between the two tests, z, originally a 80-bit floating point

[11] The problem occurs on Intel x86 processors with 32-bit Linux or Windows operating systems.

value, may be spilled to a 64-bit memory location and rounded to zero, depending on register allocation. Thus, one should be careful about results obtained on high-level languages, since their implementation semantics may be different from expected.

```
void do_nothing(double *x) { }
int main(void) {
  double x = 0x1p-1022, y = 0x1p100, z;
  do_nothing(&y);
  z = x / y;
  if (z != 0) { do_nothing(&z); assert(z != 0); } }
```

3.4 Session and laboratory exercises

We present here an example, in C, that allows comparing several verification techniques. We recall that global variables are initialized to 0.

```
int t[20];
int main() {
  int i;
  for(i=0; i<20; i++) { t[i] = 1; } }
```

1. What is the strongest postcondition of function main ? Give an invariant for the loop allowing us to prove this postcondition.
2. We now apply abstract interpretation over intervals. All cells in the array t are represented as a single abstract variable t (and thus a single intervals). What is the best interval we can infer?
3. Same question if now we represent each cell t[n] by a variable t_n.
4. In the previous question we have a very fine-grained memory model, the price being the number of variables and computations. One solution, as proposed in [9], is to consider array slices. Propose some slices suitable for obtaining an accurate invariant.

Laboratory exercises use the AtelierB tool [4] (see section 2.3) and the Polyspace verifier product developped by MathWorks [12]. We provide a set of programs with some errors pointed by the tool that have to be corrected (see [15], session 11). Examples are tailored such that simple corrections to the code lead to no warnings from the verifier ("green" in the Polyspace interface). We also proposed that students may use their own programs. But in general this experiment was not conclusive: they use some non standard primitives or libraries and programs written by students generally do not take into account arithmetic overflows. One student came with an implementation of the RSA asymmetric encryption method. The Polyspace verifier revealed several potential important errors (prime numbers generation with potential overflows, uninitialized variables ...) that led to important improvements in the code.

4 Conclusion and perspectives

We presented a new evolution of formal method teaching at Ensimag, a mathematics and computer engineering school. We explained why we have evolved

from a correct-by-design course to a more pragmatic course, with the objective to give to our future engineers an initiation to the state of the art in the theoretical and practical aspects of static analysis techniques. We evaluated this new course: polled students were convinced that these techniques can no longer be ignored. Laboratory exercises, using industrial tools, largely contributed to this success.

For the next year, one objective is to better integrate the presented techniques and notations. We also plan to test a general platform allowing to experiment several types of analysis within the same framework (as the Frama-C platform). Finally, when we work with tools flirting with undecidability or approximation, an important activity is to build appropriate examples (errors are pointed and corrected in an exact manner).

A course with the same objectives but a different format is to be started in 2010-2011 at École polytechnique in Paris.

References

1. Abrial, J.: The B-Book. Cambridge University Press (1996)
2. Bert, D., Boulmé, S., Potet, M.L., Requet, A., Voisin, L.: Adaptable Translator of B Specifications to Embedded C programs. In: FME 2003: Formal Methods. LNCS, vol. 2805. Springer (2003)
3. Burdy, L., Requet, A.: Extending B with Control Flow Breaks. In: ZB 2003: Formal Specification and Development in Z and B. LNCS, vol. 2651, pp. 513–527 (2003)
4. Clearsy website, `http://www.clearsy.com/`
5. Dijkstra, E.: A discipline of Programming. Prentice-Hall (1976)
6. Dunne, S.: Introducing Backward Refinement into B. In: ZB 2003: Formal Specification and Development in Z and B. LNCS, vol. 2651, pp. 178–196 (2003), `http://link.springer.de/link/service/series/0558/bibs/2651/26510178.htm`
7. EJCP laboratory exercices (2010), `http://www-verimag.imag.fr/~potet/EJCP-PageB/`
8. EJCP tutorial (2010), `http://www-verimag.imag.fr/~potet/ejcp-expose.pdf`
9. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI. pp. 339–348 (2008)
10. MIRA Ltd: MISRA-C: 2004 Guidelines for the use of the C language in critical systems (Oct 2004), `http://www.misra-c2.com/`
11. Monniaux, D.: The pitfalls of verifying floating-point computations. TOPLAS 30(3), 12 (May 2008), `http://hal.archives-ouvertes.fr/hal-00128124/en/`
12. Polyspace website, `http://www.mathworks.com/products/polyspaceclientc/`
13. Potet, M.L.: Twelve years of B Teaching in an engineer school: from a correct by design approach to analysis techniques and tools. In: TFM'09, Teaching Formal Methods (2009), `http://www-verimag.imag.fr/~potet/publi.html/tfm09.pdf`
14. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press (1998)
15. SAP lessons (2010), `http://www-verimag.imag.fr/~potet/SAP-exterieur/`
16. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL. pp. 32–41. ACM, New York, NY, USA (1996)
17. Stouls, N.: Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés. Thèse de doctorat, Institut Polytechnique de Grenoble (2007)