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

Marie-Laure Potet

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

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 I also present some unpublished examples and theoretical results.

1 Introduction

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

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

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

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

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

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

2 The B method studied as a whole

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

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

2.1 The objectives

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

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

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

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

2.2 Modelling aspects

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

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

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

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

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

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

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

$r \longleftarrow \ allocate =$

CHOICE

ANY v WHERE $v \in MEM - used$

THEN $used := used \cup \{v\} \ || \ r := v$

END

OR $r := null$

END

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

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

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

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

$AddRole(r, s) =$

PRE $r \in \ ROLE \wedge \ s \in SUBJECT \wedge \ r \notin \ conflict[subject2role[\{s\}]]$

THEN $subject2role := subject2role \cup \ \{s \mapsto \ r\}$

END

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

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

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

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

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

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

2.3 Formal development process

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

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

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

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

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

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

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

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

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

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

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

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

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

```
place ⟵ booking =
    VAR ind IN
        ind := 1;
        WHILE tab(ind) = TRUE DO
            ind := ind + 1;
            INVARIANT ...
            VARIANT maxseat − ind
        END;
        tab(ind) := TRUE ;
        place := ind ;
    END
```

The invariant part has to be completed by the formula:

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

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

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

2.4 Conclusion of the first period

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

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

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

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

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

3.1 Weakest precondition for exception features

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

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

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

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

$$F \in EXC \leftrightarrow Predicate$$

This calculus can be defined in the following way:

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

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

$wpe($
  BEGIN S
  CATCH   WHEN e_1 THEN S_1
          $\dots$
          WHEN e_n THEN S_n
  END, $F)$
	$wpe(S,$
	      $F \mathbin{<\!\!\!+} \{e_1 \mapsto wpe(S_1, F),$
	          $\dots,$
	          $e_n \mapsto wpe(S_n, F)\}$
	   $)$

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

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

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

```
BEGIN
  (F4)      {(y > 0 ⇒ true) ∧ (¬(y > 0) ⇒ true) }
  x := 1;
  (F3)      {no ↦ (y > 0 ⇒ x = 1) ∧ (¬(y > 0) ⇒ true), stop ↦ x = 1}
   IF y > 0 THEN RAISE stop END ;
  (F2)      {no ↦ true, stop ↦ x = 1}
  x := 2
  (F1)      {no ↦ x = 2, stop ↦ x = 1}
END
```

with:

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

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

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

and:

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

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

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

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

3.2 Semantics of operation calling

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

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

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

By reference semantics:

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

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

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

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

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

Theorem 1 *Invariant preservation by call*
Let $r \leftarrow op(p) \hat{=}$ PRE P THEN S END *be the definition of an operation op and let* $v \leftarrow op(e)$ *be an operation call, as defined before. Let* I *be a property on* x*, the set of variables of the component in which op is defined* ($x \cap v = \emptyset$)*. Then:*

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

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

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

3.3 Refinement theory

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

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

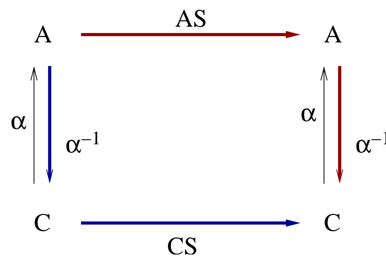
Semantic definition of refinement. Let M be a component refined by another component R. As defined in the B-Book (chapter 11, p 511) a substitution U is an external substitution for M and R if it contains no reference to the variables of components M and R (v_M or v_R). Internal variables can only be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

Definition 1 *Semantic characterization of data refinement.*

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

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

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



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

[4] Termination condition are not considered here.

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

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

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

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

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

MACHINE $CASINO1$ VARIABLES i INVARIANT $i \in 0..36$ INITIALISATION $i :\in 0..36$ OPERATIONS $r1 \longleftarrow spin \widehat{=} r1 := i \;\|\|\; i :\in 0..36$ END	MACHINE $CASINO2$ OPERATIONS $r2 \longleftarrow spin \widehat{=} r2 :\in 0..36$ END

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

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

CASINO1 $\not\sqsubseteq^{L}$ CASINO2:

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

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

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

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

4 Conclusion

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

4.1 Conclusion of my experiments

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

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

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

4.2 The future

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

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

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

References

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