Teaching Formal Methods for the Unconquered Territory

Nestor Catano[1] and Camilo Rueda[2]

[1] University of Madeira, Department of Mathematics and Engineering
Funchal, Portugal
ncatano@uma.pt
[2] Pontificia Universidad Javeriana, Department of Computer Science
Cali, Colombia
crueda@cic.puj.edu.co

Abstract. We summarise our experiences in teaching two formal methods courses at Pontificia Universidad Javeriana. The first course is a JML-based software engineering course. The second course is a model-driven software engineering course realised in the B method for software development. We explain how formal methods are promoted in Pontificia Universidad Javeriana, how we motivate students to embrace formal methods techniques, and how they are promoted through the presentation of motivating examples.

1 Introduction

In this paper we summarise our experiences in teaching formal methods to undergraduate students of Pontificia Universidad Javeriana. We strive to help students to build skills on formal methods, and to master formal tools they might use in their future IT software engineering jobs in the "unconquered territory", *e.g.*, traditional software engineering companies that are reluctant to adopt formal methods as part of their software development practices. The first author has been lecturing a JML [7] (short for Java Modeling Language) software engineering course during the past 3 years, and the second author a model-based formal software development course during the past 5 years. Prior to these courses, students must attend a standard software engineering course, as well as discrete mathematics related courses. That is, our students have enough software engineering and mathematical background to appreciate the benefits of using formal methods tools and techniques during software development. Formal methods offer a practical alternative to constructing correct software, and can be implanted along the several steps of the software development cycle, from requirements capture, by employing formal specification language notations such as B [1] or Z [16, 17], through the design and coding of systems [14]. But, formal methods will only be widely popular in software companies if formal methods tools exist that provide support for software engineering practices, and software engineers are properly trained in related techniques in universities and education centres.

We explain here two approaches undertaken to the teaching of formal methods in Pontificia Universidad Javeriana. (*i.*) Model-driven software engineering is realised in the B method for software development [1]. That is, from a B model of a system, obtained from the software requirements of an application, a code-level model in B is attained while applying successive *refinement steps*. Mathematical techniques are used along the way so as to guarantee the correctness of the refinement process. The code-level model is a concrete model of the more abstract initial model of the system. Section 5 presents the model-driven B-based course taught at Pontificia Universidad Javeriana, Cali. The course is illustrated with the formal software development of the control system structure of the recently inaugurated Cali mass transportation system (MIO). (*ii.*) Section 4 presents the structure of our JML-based formal software development course. The JML-based course places between the rather heavy, fully mathematically based, formal software development course in B, and the rather mathematically informal previous software engineering course, introduced in Section 3. The formal methods courses in JML and B are two complementary courses that give students two different insights of the use of formal methods in software development. In the following, Section 2 gives an overview on JML and B based formal methods techniques and tools.

2 Preliminaries on Formal Methods

2.1 The Java Modeling Language (JML)

JML is a specification language for Java that provides support for B. Meyer's design-by-contract principles [11]. The idea behind the design-by-contract methodology is that a contract between a class and its clients exists. The client must guarantee certain conditions, called pre-conditions, to be able to call a method of the class. In return, the class must guarantee certain conditions, called post-conditions, that will hold after the method is called.

JML specifications use Java syntax, and are embedded in Java code within special marked comments `/*@ ... @*/` or after `//@`. A simple JML specification for a Java class consists of pre- and post-conditions added to its methods, and class invariants restricting the possible states of class instances. Specifications for method pre- and post-conditions are embedded as comments immediately before method declarations. JML predicates are first-order logic predicates formed of side-effect free Java boolean expressions and several specification-only JML constructs. Because of this side-effect restriction, Java operators like `++` and `--` are not allowed in JML specifications. JML provides notations for forward and backward logical implications, `==>` and `<==`, for non-equivalence `<=!=>`, and for logical *or* and logical *and*, `||` and `&&`. The JML notations for the standard universal and existential quantifiers are `(\forall T x; E)` and `(\exists T x; E)`, where `T x;` declares a variable `x` of type `T`, and `E` is the expression that must hold for every (some) value of type `T`. The predicates `(\forall T x; P; Q)` and `(\exists T x; P; Q)` are equivalent to `(\forall T x; P ==> Q)` and `(\exists T x; P && Q)` respectively.

JML supports the use of several mathematical types such as sets, sequences, functions and relations, in specifications. JML specifications are inherited by subclasses – subclass objects must satisfy super-class invariants, and subclass methods must obey the specifications of all super-class methods that they override. This ensures behavioural sub-typing. That is, a subclass object can always be used (correctly) where a super-class object is expected. In the following, we briefly review JML specification constructs, and the JML common tools. The reader is invited to consult [9] for a full introduction to JML.

2.2 The JML Common Tools

The JML common tools [7, 6] is a suite of tools providing support to run-time assertion checking of JML-specified Java programs. The suite includes `jmlc`, `jmlunit` and `jmlrac`. The `jmlc` tool compiles JML-specified Java programs into a Java byte-code that includes instructions for checking JML specifications at run-time. The `jmlunit` tool generates JUnit [10] unit tests code from JML specifications and uses JML specifications processed by `jmlc` to determine whether the code being tested is correct or not. Test drivers are run by using the `jmlrac` tool, a modified version of the `java` command that refers to appropriate run-time assertion checking libraries.

The JML common tools make possible the automation of regression testing from the precise, and correct JML characterisation of a software system. The quality and the coverage of the testing carried out by JML depend on the quality of the JML specifications. The run-time assertion checking with JML is sound, *i.e.*, no false reports are generated. The checking is however incomplete cause users can write informal descriptions in JML specifications, e.g., `(* x is positive *)`. The completeness of the checking performed by JML depends on the quality of the specifications and the test data provided.

2.3 Refinement Calculus and the B Method

In the *refinement* calculus strategy for software development, the process of going from a system specification to its implementation in a machine goes through a series of stages. Each stage adds more details to the description of a system. Each stage can thus be seen as a model of the system at a particular level of abstraction. Models at each level serve different purposes. At higher levels models are used to state and verify key system properties. At lower levels models are used to simulate the system behaviour. It is crucial that models at each stage are coherent with the system specification, *i.e.*, that the simulation obeys the specification properties. A model M_{i+1} at stage $i+1$ is said to be a refinement of a model M_i at stage i when the states computed by M_i and M_{i+1} at each given step obey a so-called "gluing invariant" stating properties for the joint behaviour of both models. A refinement step generates *proof obligations* that must be formally verified in order to assert that a model M_{i+1} is indeed a refinement of a model M_i. These are necessary and sufficient conditions to guarantee that, although at

different levels of abstraction, both are models of the same system. Correctness of the whole development process is thus ensured ([4]).

In the B method ([1], [15]) models are so-called *machines* composed of a static part defining observations (variables, constants, parameters, etc) of the system and their invariant properties, and a dynamic part defining operations changing the state of the system. Each operation must maintain the invariant property. In B, the language for stating properties (essentially predicate logic plus set theory) and the language for specifying dynamic behaviour (*i.e.* programs) are seamlessly integrated.

A significant feature of the B system modelling approach is the availability of automatic verification tools such as B-tools (http://www.b-core.com/-btool.html), Atelier-B (http://www.atelierb.eu/index_en.html) or Rodin (http://-www.event-b.org/platform.html), and model-checking simulators such as ProB (http://users.ecs.soton.ac.uk/mal/systems/prob.html).

2.4 Event B Models

We introduce a derivative of the B method called event B [3]. Event B models are complete developments of discrete transition systems. They are composed of *machines* and *contexts*. These correspond, roughly, to a B method *machine* whose static part (except variables and their invariants) is transferred to a different module (the context). B method operations are replaced in event B machines by *events*. In B method machines, operations are *invoked*, either by a user or by another machine, whereas in event B, an event *occurs* when some condition (its *guard*) holds. Three basic relations are used to structure a model. A machine *sees* a context and can *refine* another machine. A context can *extend* another context. Events have two forms, as shown in table 1.

Table 1. Events

any x **where** $G_2(x, v)$ **then** $A_2(x, v)$ **end**	**when** $G_1(v)$ **then** $A_1(v)$ **end**

The "when" form of event executes the action A_1 when the current value of the system variables v satisfies the guard G_1. The "any" form of event executes action A_2 when there exists some value of x satisfying the guard G_2. Proofs obligations require invariants to hold after executing the actions. A simple example, modelling a parking lot, is shown in table 2. Variable C keeps track of the cars in the parking lot. Set $CARS$ defines a type. Constant n is the maximum capacity of the parking lot. Events model entrance and exit from the parking.

Table 2. A parking lot model

machine
Parking
sees
ctx
variables
C
invariants
$C \subseteq CARS \wedge card(C) \leq n$
events
$arrives =$
any car
where
$car \in CARS \wedge car \notin C \wedge \ card(C) < n$
then
$C := C \cup \{car\}$
end
$leaves =$
any car **where** $car \in C$
then
$C := C \backslash \{car\}$
end
end

context
ctx
sets
$CARS$
constants
n
axioms
$n \in \mathbb{N} \wedge n > 0$
end

3 The Software Engineering Programme at Pontificia Universidad Javeriana

Engineering in computer science at Pontificia Universidad Javeriana (called systems engineering for historical reasons) is a 5 year program, structured per semesters, comprising a 2 years' common trunk in physics and mathematics, shared by all other engineering programs, including 2 courses in discrete mathematics and logic. This is followed by a 3 years' program in computer science, with about a 30% course charge in economics and humanities. The discrete mathematics, as well as all the basic computer related courses comply with the ACM/IEEE undergraduate computer science curriculum (`http://sites.computer.org/ccse/`). There are two basic courses in software engineering:

1. Software processes dealing with fundamental design principles, API design, tool design, software life cycle and capability models.
2. Software engineering and management, dealing with software requirements specifications (SRS), project management, validation and verification, and software evolution.

These courses are scheduled for sixth and seventh semester undergraduate students respectively. A formal software development course is given in the sixth semester, at the same time as the software processes course. Additionally, there

is a case-based software engineering workshop scheduled for eighth semester students. Topics for this workshop vary, yet it has often been the case that the workshop has been dedicated to software verification with JML.

Formal model courses are not part of the core in the ACM/IEEE curriculum. We decided to include formal model courses in Pontificia Universidad Javeriana engineering program mostly for two reasons: (*i.*) a survey among recruiting executives of companies in Cali revealed they consider the ability to clearly reason about a software design as a key (usually lacking) competence in young professionals, and (*ii.*) the economic development plan of Cali pointed at software production as a key strategy and increasing software quality as the most pressing need in this realm.

Pontificia Universidad Javeriana computer science department keeps a close relations with ParqueSoft, the biggest Colombian technological cluster, with more than 200 software companies and 800 software developers (`http://www.-parquesoft.com`). ParqueSoft software companies are typically launched by students from the three biggest universities in Cali. ParqueSoft's total sales are about US $47 million a year. About half of these companies have achieved standard quality assurance certifications. Most computer science students at Pontificia Universidad Javeriana conduct internships in these and other companies based in Cali during their third year of studies. Pontificia Universidad Javeriana Computer Science department encourages students to initiate software start-ups at ParqueSoft through an entrepreneurship joint educational program. Students in this program substitute their engineering degree final thesis report with a technical report on their proposed software venture.

4 The JML-Based Software Engineering Course

This course covers conceptual underpinnings of formal software development and software verification with JML [7, 6]. We evolve our course through several software development examples that illustrate our methodology. We employ The JML common tools, and the Prototype Verification System (PVS) [13] to validate the developments (see Section 2). We start this course with an overview of first order logic and proof systems. Then, program correctness using Dijkstra's weakest pre-condition calculus is introduced, followed by the design-by-contract, and software verification using the JML common tools. In the end of the course, the PVS specification and verification system is presented with the aid of the electronic phone book example that comes with the PVS standard documentation. The course outline follows.

1. An Introduction to Formal Methods **(3 hours)**
2. First Order Logic **(6 hours)**
   (a) Syntax and Semantics
   (b) Expressiveness, Models and Tautologies
3. The Java Modelling Language (JML) **(24 hours)**
   (a) Program Correctness and Weakest Pre-condition Calculus

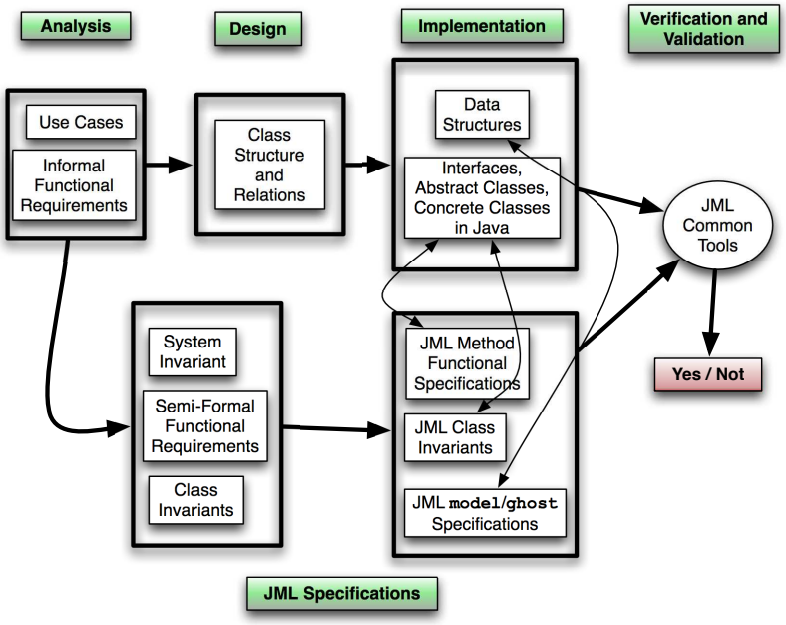

Fig. 1. JML-based Software Engineering

(b) Design-by-Contract
(c) JML and The Design-by-Contract
(d) JML's Advanced Features
(e) The JML Common Tools
(f) Formal Development Examples

4. Proof Systems **(3 hours)**
   (a) Decidability, Soundness, Completeness
5. The PVS Specification and Verification System **(15 hours)**
   (a) The Logic of PVS
   (b) An Electronic Phone Book Example in PVS
   (c) Types, Declarations, Induction, Recursion
   (d) Encoding Abstract State Machines in PVS
   (e) JML-Checked Java Implementation of a PVS Specification of B-Trees

The JML-based software development methodology introduced in the course is based on Meyer's object oriented methodology presented in [12] (see Chapter 28), while using the JML common tools for validating the developments. The methodology is illustrated in Figure 1. The software development cycle consists of four steps, namely, analysis, design, implementation, and verification. In the same spirit of the methodology introduced by Meyer, we do not restrict any

step of the software development cycle to occur before or after any other step (as opposed to the Waterfall model introduced in 1970's), so that the arrows in Figure 1 convey information on usage rather than on precedence in time. JML specifications (lower part of the figure) are written in parallel to the Java application itself (upper part of the figure). The informal functional requirements (functional requirements written in English) of the step of analysis originate three documents that will later serve to write the JML specifications describing the application, namely, the semi-formal functional requirements, the invariant of each class, and the invariant of the whole system. The semi-formal functional requirements, though written in English, are expressed in a more mathematical style, suitable to be then ported into JML specifications. Class invariants and the system invariant are naturally expressed as JML invariants. And the semi-formal functional requirements are expressed as JML method pre- and post-conditions. At the same time, JML invariants and the JML method specifications reflect the class structure of the Java code.

Additionally, in order to have a higher level of abstraction in specifications, JML allows one to declare so-called `model` variables. These are variables that exist only at the level of the specifications. Model variables can be related to concrete variables (*i.e.*, variables declared in Java code) by the use of `represents` and `depends` clauses, specifying how the value of a model variable can be calculated from the values of the concrete variables. JML model variables allow one to describe in full detail the data structures used in a Java program, and how these structures evolve through class inheritance.

4.1 Formal Software Development of Ax-LIMS with JML

In the following, we explain the JML-based software development methodology described in Figure 1 by presenting key aspects in the development of Ax-LIMS [8], a project management Java plug-in wrapping up in Java most services and features provided by LIMS (Laboratory Information Management System), a project management application, developed in PHP, specially designed for the planning, organisation and resource management of biotechnology projects (see Figure 2). Biotechnology projects manage laboratory processes and tasks. A process manages laboratory experiments. Tasks are administrative tasks that need to be carried out between the project start and completion dates. Information about projects is stored in a PostGreSQL database. Although LIMS business logic is written in PHP, a REST (Representation State Transfer) API interface exists that supports communication between Ax-LIMS and LIMS. Ax-LIMS implementation relies on the project and process features provided by LIMS. The Ax-LIMS example is adapted from a formal software development project involving ParqueSoft (`http://www.parquesoft.com`), the International Centre for Tropical Agriculture (CIAT, `http://www.ciat.cgiar.org/`), and Pontificia Universidad Javeriana faculty.

Students in our course are asked to formally develop the project manager plug-in that directly connects to LIMS. Students are given a software requirements document, describing the functional requirements of the Ax-LIMS plug-in,

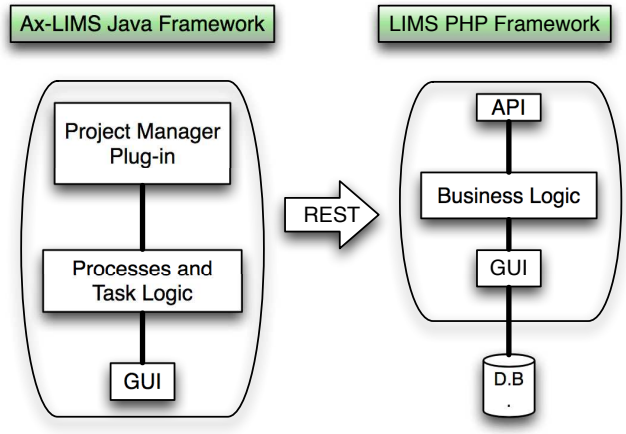

Fig. 2. Architecture of Ax-LIMS

as well as a set of use cases describing the interaction of the user with the plug-in. We call these requirements informal software requirements, since they are written in plain English. Students are asked to turn this informal requirements document into a semi-formal document, that is, one that expresses informal functional requirements in terms of pre-conditions, post-conditions and invariants, suitable to later be ported into JML formal specifications. Modelling these invariants in JML guarantees that any class method implementing any functionality of the Ax-LIMS manager plug-in must adhere to Ax-LIMS' software requirements. This is checked with the JML common tools.

Additionally, students are asked to derive a class structure for the plug-in from the use cases and the informal functional requirements. A typical class structure might include classes `Project`, `Task`, and `State`. The `Project` class might declare a field `tasks` of type `List`[3] for managing project tasks. The `State` might define all possible states for projects and tasks. In classes `Task` and `Project`, fields `psd`, `pcd`, `asd` and `acd`, representing planned start date, planned completion date, actual start date and actual completion date are declared.

From this basic implementation, semi-formal requirements such as "The planned start date for a task is earlier than its planned completion date", and "If a task has an actual completion date, then it also has an actual start date" can be specified in JML as two invariants in class `Task` (see below). The method `compareTo` in the Java standard class `Date` returns a positive number whenever the first date is bigger than the second one, a negative number when the second one is bigger than the first one, otherwise returns `0`.

[3] Field `tasks` ought to be declared of type `ArrayList<Task>` indeed, but the Java version under which the latest version of JML runs does not allow parametric types.

```
//@ invariant pcd.compareTo(psd) > 0;
//@ invariant acd != null ==> asd != null;
```

Finally, students are taught to write and evolve Java code that adheres to the JML invariant properties obtained from the functional requirements. Students are shown how to iteratively specify invariants in JML, write Java code that respects these specifications, check the code against the JML specifications using the JML common tools, and evolve the specifications or the code (or both) accordingly. Once code that complies with all the class invariants is written, no error is issued by the JML common tools. Checking that written code complies with JML-specified invariants is automatically done by JML. It eliminates students' responsibility of keeping track of how properties a program must respect are affected by changes in the program code.

4.2 Experience with the Course

We want to emphasise here the importance of thinking of invariant properties when developing software. Thinking about invariants prior to writing code is a practice to which students do not easily adhere. Having a previous formal specification of the application and systematically using a tool, *i.e.*, the JML common tools, for checking the correctness of the code as it is written forces students to think about how the written code affects the consistency and the correctness of the whole program.

It is our experience that invariants are the key notion in formal software development that makes a difference with respect to traditional (non formal methods based) software engineering courses; whereas students attending traditional software engineering courses are not used to think of invariants properties leading their software developments. In general, students feel intimidated by the idea of coming up with an invariant. Often, they design code that can make their programs be in an inconsistent state. We endeavour to help students to surmount this fear so as to write their programs in a more mathematically correct way. We strongly believe JML already helps students in this sense, from furnishing a friendly Java-like syntax, to making it possible to use first-order logic predicates in JML specifications naturally.

5 Model-driven Software Engineering in B

This course illustrates the advantages of using formal techniques for modelling systems and developing software. The course puts forward the idea of using these techniques as guiding principles and methodologies for thinking a system model from scratch in a disciplined way. The event B approach is developed in the first part of the course and the B method in the second (see Sections 2.4 and 2.3). Both are based on the analysis of a collection of case studies.

The course follows what Abrial calls "the parachute strategy", in which systems are first considered from a very abstract and simple point of view, with

broad fundamental observations. This view usually comprises few simple invariant properties that students can easily grasp, *e.g.*, defining what can reasonably be expected from the operation of such a system. Liveness properties are also introduced at this level. Once the system is completely understood at this level, students are required to develop small variants and to prove all obligations. The second step is to consider a refinement by adding a viewpoint observation in such a way that the new model keeps a palpable behavioural relation with its abstraction. Students are then encouraged to find by themselves precise relationships between abstract and concrete variables that guarantee coherent behaviour between both models.

As more experience is gained in the modelling framework, development of more complete systems comprising several refinements is attempted. In these refinements, the way how invariant properties can be used as a tool for tuning events is stressed. Systems that are very familiar to students, such as a soccer match, are used at this stage for them to be able to develop an intuitive, yet precise, idea of the use of refinement conditions. In the final stage of the course, a model of a more real-life system is constructed, alternating designs discussed in class with refinements supplied by students in their homework.

Although a broad review of first order logic and set theory is given at the beginning of the course, our strategy is to introduce event B modelling language constructs, such as relations, functions, sets and their operations as they are needed to express some specific property of the system at hand. Even the review of logic is done in a very "instrumental" way, stressing its use in modelling. Students are never required to do by-hand formal proofs of any logic formula, but they are constantly encouraged to argue about its validity. The idea behind this is to help students build a more "friendly" attitude towards mathematical formalisms, by showing them how a machine can do the tedious job of proving a formula in their place.

Our course uses the Rodin platform intensively (`http://wiki.event-b.org/index.php/Rodin_Platform`). This platform comprises an intuitive graphical interface based on Eclipse (`http://www.eclipse.org`) in which users can edit, prove and animate an event B model. In the beginning of the course, students are required to edit and type-check models in Rodin, though, instead of proving them, the animator plug-in is used to understand the model behaviour. In any case, initial models are chosen so that Rodin can automatically prove them. As familiarity with the tool increases, the models considered have a few proof obligations that Rodin cannot prove automatically. At this point, the "Mathematical Language" chapter of the Rodin manual [2] is presented. This provides a intuitive, yet rigorous, introduction to propositional and predicate logic from the vantage point of carrying out automatic proofs in those systems. Students can then understand Rodin proof strategies, and also relate this knowledge to how they can better guide Rodin's interactive prover. From this point on, students are required to fully prove their developments with Rodin.

A second part of the course uses event B to construct programs. This is done in event B by considering an abstract system that just specifies preconditions

(context part and invariants) having an event computing in one step the result of the program (i.e. specifying its post-condition). Refinements of this initial model proceed just as for system models. The last refinement is such that each event could easily be coded in some imperative programming language. Rules to automatically translate this last refinement into code can be easily devised but, unfortunately, as of this writing they have not been integrated into Rodin so that students must do this by hand.

The final part of the course deals with software systems. These, in principle, could be modelled in event B just as any other system. In our view, however, the lack of constructs to relate machines in different ways makes its use problematic in a pedagogical sense. The point is that students are, in other courses and in their initial job experiences, used to build software out of software pieces already available. In some versions of the course, we decided to use the B model explicit module integration constructs (e.g. using available machines within a system being developed). B machine constructs are introduced as needed for the model at hand. Specification of systems such as (a simplification of) the project management plug-in (see 4.1) are typical projects students consider for the *Software Processes* course.

5.1 Experience with the Course

As said before, the course is scheduled in the sixth semester, when students have already taken courses in programming, object oriented design and database models. They thus come with a background about how software is built that is very different from what they find in the event B course. This causes a sceptical attitude from the beginning that makes it difficult to keep alive the necessary motivation. We use two strategies to tackle this problem. First, systems considered at the beginning of the course pertain to situations that are familiar to students, yet not trivial. Second, in the software development part, students are required to constantly assess advantages and disadvantages of the same problem modelled with B and with the traditional methodologies used in other courses. Nevertheless, we have found that for this purpose a transition going from traditional ways of thinking systems to JML and then to event B greatly diminishes the initial scepticism and, at the same time, helps students to place each methodology in the right context of application.

A second issue is the use of mathematical formalisms. Students are evermore demanding to see clear relationships between their future professional activity and the mathematics they are being taught. In the first offerings of this course, mathematical formalisms were given a central role from the outset and thus all logic and mathematics needed to express event B proof obligations and constructs was presented at the beginning of the course. This only aggravated the motivational problem. Current versions of the course, as stated above, develops mathematical constructs along the course, always with a clear view of what precisely they are intended for in the problem at hand. Moreover, students are required to express in their own words those fundamental properties of a system, prior to their formalisation using the mathematical language of event B.

Using two slightly different models and their corresponding tools in the course poses problems. In principle, the whole course could be based on the B method, using the Atelier B (`http://www.atelierb.eu/index-en.php`) tool. However, students have many problems with the interactive prover of Atelier B because of its black-box feel, with a command line approach that makes it hard to have a glimpse of exactly what hypotheses are being tried, what are available and therefore what strategies could better guide the proof. Work is underway by Atelier B maintainers to devise a more intuitive interactive prover, so this might change in the future.

5.2 Formal Software Development of MIO in B

The MIO mass transportation system was recently inaugurated in Cali. The system comprises a series of articulated buses following the main corridor routes of Cali, complemented with feeding buses connecting Cali with its outskirts. Besides serving as a palliative of the chaotic current transport service of Cali, the MIO system has renewed local people's sense of belonging to Cali. Our students thus feel excited about formally modelling the MIO and somehow contributing to the progress of Cali.

The MIO system is partially modelled in class by us; the rest is left to students as home-work. The MIO transport system is composed of articulated buses travelling along dedicated lanes of city avenues. Bus stations are constructions where users enter by validating a magnetic card in a reader. A turnstile unblocks when the card is validated so that the card owner can pass through. When a bus arrives, sliding doors in the station open in synchronisation with the bus doors, and passengers can enter or exit. Sensors at station doors identify when a bus door is opening. The station doors remain open during some fixed number of seconds. At any time, only one bus can be parked in a station. Users top-up their magnetic cards at machines in the stations.

There are two further motivations for introducing and modelling the MIO system in our course. Firstly, the system is already part of students daily lives so that they know very well how it works, and, secondly, a convincing complete simplified model can be constructed that illustrates most of the features of the event B modelling technique. The whole model comprises an abstract machine, five contexts and seven refinement machines. A summary of all these is shown in table 5.

To give a broad idea of the development path of the system we discuss next the second refinement in table 5. The abstract model and the first refinement define the observations in the left of table 3. The second refinement includes those in the right. Refinement 2 is rather "orthogonal" with respect to refinement 1. The idea is to show students how new observation viewpoints lead to new events and, possibly, to stronger conditions in old events. The refinement models by set inclusion the fact that only authorised card holders can be in a station at a given time. It also gives the opportunity to discuss possible future enhancements to the current operation of the MIO system by keeping track, for example, of bus occupancy.

Students are encouraged to discuss what fundamental aspects of the operation of the system they experience each day should be present at this level and what could be deferred for further refinements. This discussion is always directed towards ways to express their contributions as part of the invariant, rather than thinking about possible new events. Property $dom(perst) \subseteq authp$, for instance, expressing the fact that only authorised card holders can be inside a station, was proposed by the majority of the students as pertaining to this level.

Table 3. Static part abstraction &refinement 1(left), refinement 2 (right)

constants	
std	door in station
axioms	
	$std \in DOOR \rightarrowtail\!\!\!\!\!\rightarrow ST$
variables	
$binst$	bus in station
opd	open station doors
$authb$	buses authorised to depart
invariant	
	$binst \in BUS \rightarrowtail ST$
	$opd \subseteq dom(std)$
	$std[opd] \subseteq ran(binst)$
	$authb \subseteq dom(binst)$
	$binst[authb] \cap std[opd] = \varnothing$

variables	
$authp$	persons authorised to enter
$perst$	persons inside a station
$perb$	persons in a bus
invariant	
	$perst \in PERSON \nrightarrow ST$
	$perb \in PERSON \nrightarrow BUS$
	$dom(perst) \cap dom(perb) = \varnothing$
	$authp \subseteq PERSON$
	$dom(perst) \subseteq authp$
	$dom(perb) \subseteq authp$

Some events of refinement 2 are shown in table 4. Events are rather simple. The message that is constantly given to students is that the construction of refinements should always be guided by the invariant. Students must always express the need of an event guard they propose in terms of an invariant property that must be maintained, such as, for example, the first guard of the "enter" event. Skill in using event B language constructs like the relational inverse is motivated by giving simple data structure-like intuitions for it. Students easily come up with, for example, the guard in the third line of the "open_door" event stating that, in order to open a station door, the bus currently parked at the station where the given door is, cannot have already been authorised to leave. These types of event guards, aiming at avoiding "ping-pong" loops between two events are discovered by the students in a "critics game". Students are divided in groups of designers and critics. Credit is given to the former for coming up with correct events and to the latter for finding loopholes, particularly with respect to invariant properties. Later in the design such synchronisation patterns, in particular when material equipment is involved, are analysed in light of the "design patterns" proposed by Abrial in one of his event B examples.

Table 4. refinement 2: some events

depart =
any b
where
$b \in authb$
then
$binst := binst \setminus \{b \mapsto binst(b)\}$
$authb := authb \setminus \{b\}$
end

open_door =
any d
where
$d \in DOOR \wedge\ d \notin opd$
$std(d) \in ran(binst)$
$\wedge\ binst^{-1}(std(d)) \notin authb$
then
$opd := opd \cup \{d\}$
end

close_door =
any d
where
$d \in opd$
then
$authb := authb \cup \{binst^{-1}(std(d))\}$
$opd := opd \setminus \{d\}$
end

enter =
any p, s
where
$p \in authp \wedge\ p \notin dom(perst)$
$\wedge\ p \notin dom(perb) \wedge s \in ST$
$\wedge\ s \notin ran(perst)$
then
$perst := perst \cup \{p \mapsto s\}$
end

authorise =
any p
where
$p \in PERSON \wedge p \notin authp$
then
$authp := authp \cup \{p\}$
end

get_in_bus =
any p
where
$p \in dom(perst)$
$std^{-1}(perst(p)) \in opd$
then
$perbus := perbus \cup \{p \mapsto binst^{-1}(perst(p))\}$
$perst := perst \setminus \{p \mapsto perst(p)\}$
end

exit_from_bus =
any p
where
$p \in dom(perb)$
$perb(p) \in dom(binest)$
$std^{-1}(perst(p)) \in opd$
then
$perbus := perbus \setminus \{p \mapsto perb(p)\}$
$perst := perst \cup \{p \mapsto binest(perb(p))\}$
end

Table 5. Components of the MIO system

Component	Observations	Feature learnt
Abstraction	what bus is parked in what station	Modelling with partial functions
Refinement 1	station doors, set of parked buses allowed to leave	Linking refinement variables to abstract ones
Refinement 2	card holders in a station, card holders authorised to enter a station	Maintaining invariant on two linked functional variables, dealing with relational inverse
Refinement 3	Entrance control (abstract): card reader light, turnstile	Modelling equipment, tune events from invariants, model with relational range/domain restriction
Refinement 4	Equipment controller: card reader, controller, message channels	Interplay between physical and controller events, when/how to introduce software
refinement 5	passing through physical turnstile	Modelling message synchronisation
Refinement 6	Controller software: Modelling a message channel	Representing data structures as refinements
Refinement 7	sensors at station doors	Modelling weak and strong synchronisation patterns

6 Conclusion and Future Work

We believe formal methods tools have attained a level of maturity today that we should move into their use in practical settings, *e.g.*, making the word "formal" in "formal software development" the rule and not the exception. People in academia have a say on this. We can contribute to making this happen by helping students to build skills in formal software development, initiating students in the use of formal methods tools, and guiding them in the process of discovering the close embracing relation between software models and mathematical formalisms. That is, we can contribute to this by helping students to build skills to be put in practice in the "unconquered territory". This is an ambitious, yet feasible purpose.

Pontificia Universidad Javeriana has committed to the purpose of making formal methods and formal methods tools more popular in software development and in software industry. Firstly, Pontificia Universidad Javeriana computer science department keeps a close relation with ParqueSoft, the biggest Colombian technological centre. This relation encompasses the running and submission of R&D projects to Colciencias (the Colombian national science foundation), and projects are under their way to be submitted to international funding. Pontificia Universidad Javeriana further organises a student entrepreneurship program with ParqueSoft. Undergraduate students in the seventh semester can join a software venture formation program under the leadership of ParqueSoft. There they get hands-on experience in software development and marketing by integrating software development groups in young companies. In the last semester they propose software development ideas backed by more realistic knowledge of

market needs. ParqueSoft can then choose to launch start-ups based on some of the most innovative ideas presented.

Secondly, the teaching of formal methods is key part of the software engineering undergraduate courses in the computer science department. As an example of formal methods courses taught in the science department, we have presented in this paper a JML-based formal development course, and a model driven course in B. The JML course allows students to have a first contact with formal specification of programs. Program correctness is introduced gradually. In our course, students enjoy evolving JML specifications and Java code, and checking the code for flaws. Our explanation for this positive attitude is that students regard the whole process of specification and verification as a different sort of programming. That is, that given a specification, a correct program must be constructed that respects the specification. The essence of "assigning programs to meanings".

It has been our experience in our formal methods courses that invariant is a key central notion in formal development courses, either in the form of class invariants as in JML, or as a refinement gluing invariant in the B method for software development, that leads software developments. This is students' first face-off with invariants. We acknowledge students often win this battle.

In these experiences, we have found important to develop in students a standpoint of complementarity with regard to methodologies and techniques in software modelling and construction. We are careful not to present formal methods as "better" methodologies that should replace other strategies in all situations. On the contrary, we encourage students to think on ways the knowledge gained in formal method courses may help improving their approach in traditional software engineering methods.

The authors are currently lecturing together a "formal modelling of systems using discrete mathematics" formal methods course for a recently opened master in engineering offered by Pontificia Universidad Javeriana computer science department. This year is our first year of lecturing this course. The course is organised half about the JML part, and half part on software development in B. It has been our intention not only to provide students with two complementary approaches to software construction, but also to communicate the idea that the part in B is a continuation of the part in JML. That is, we want to find common ways to relate the teaching of formal methods in JML and formal methods in B.

In [5] F. Bouquet *et al.* present the JML2B tool that checks JML specifications for inconsistencies by porting the specifications into B machines. All the available B tools can then be used for checking the B code. The translation from JML to B machine goes straightforward. JML invariants are expressed as B invariants universally quantified over all the instances of the class. Method preconditions translation includes conditions on the types of the method parameters. JML `assignable` classes, restricting which variable may be changed by a method call, are translated into the "any" form of events (see Section 2.4 for an introduction to events). We envision defining a formal methods teaching methodology that considers going from informal software requirements to JML specifications and Java code, straight down to B machines in near future.

References

1. J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J-R. Abrial. *Rodin deliverable D7, chapter V, Event-B mathematical Language*. Information Society Technologies, 2005.
3. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inf.*, 77(1-2):1–28, 2007.
4. J.R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Informatica*, 77(1,2):1–24, 2007.
5. F. Bouquet, F. Dadeau, and J. Julien. JML2B: Checking JML specifications with B machines. In *The 7th International B Conference*, pages 285–288, 2007.
6. C. Breunesse, N. Catano, M. Huisman, and B. Jacobs. Formal methods for Smart Cards: An experience report. *Science of Computer Programming*, Issues 1–3, 55:53–80, March 2005.
7. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe 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.
8. N. Catano, F. Barraza, D.García, P. Ortega, and C. Rueda. A case study in JML-assisted software development. In *Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008)*, volume 240 of *Electronic Notes in Theoretical Computer Science*, pages 5–21, July 2009.
9. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`, 2008.
10. Johannes Link. *Unit Testing in Java*. Morgan Kaufmann, 2003.
11. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.
12. B. Meyer. *Object Oriented Software Construction*. Prentice Hall PTR, 1997.
13. S. Owre, N. Shankar, J.M. Rushby, and D.W.J Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI, Nov. 2006.
14. A. Robinson and A. Voronkov. *Handbook of Automated Reasoning*. MIT Press, 2001.
15. S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.
16. J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, 1989.
17. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall, Inc., 1996.