SoSyM manuscript No.
(will be inserted by the editor)

Design and Implementation of a Cooperative Framework for B based Software Development: BRILLANT

Dorian Petit[1], Samuel Colin[3], Georges Mariano[2], Vincent Poirriez[1], Jérome Rocheteau[2], Rafaël Marcano[2] ⋆

1 LAMIH/ROI, UMR CNRS 8530
Université de Valenciennes et du Hainaut Cambrésis
Le Mont Houy F-59313 Valenciennes Cedex 9, France
e-mail: `Firstname.Lastname@univ-valenciennes.fr`

2 INRETS-ESTAS National Institute for Transport and Safety Research
20 rue Elisée Reclus - B.P. 317 F-59666 Villeneuve d'Ascq, France
e-mail: `Firstname.Lastname@inrets.fr`

3 DEDALE Team
LORIA - Campus Scientifique
BP 239 - 54506 Vandoeuvre-lès-Nancy Cedex
e-mail: `Firstname.Lastname@loria.fr`

Received: date / Revised version: date

Abstract The need for the B method first appeared in industry, and several commercial tools have been developed to support this formalism. However, few of these tools allow reasoning related to the formalism itself or on its possible extensions. This article presents an open-source platform, with a focus on the platform's core component, the BCaml project. The tools presented here are used to show how very different approaches can be brought together around a central design to form a consistent toolbox that can be used to generate safe code, and to develop safe systems, from their specifications to their validation.

Key words B method, tool support, UML modelling, XML, proof tools, code generation

⋆ This research has been partially funded by Science and Technology for Safety in Transportation, the European Community, the Regional Delegation for Research and Technology, the Delegate Minister for Higher Education and Research, the Nord/Pas-de-Calais Region and the National Center for Scientific Research : the authors gratefully acknowledge the support of these institutions

1 Introduction

During the last decade of the previous millennium, theoretical research produced the B method, which is based on the same fundamentals as Z [11] and VDM [39]. This method reconciles the pragmatic constraints of industrial development of critical software with the strict theoretical requirements inherent to mathematical formalism. The B method is one of the rare successful formal methods used in industry, and it supports multiple paradigms: Substitutions as means for describing dynamic behaviour naturally; *Formulas* in a simple yet efficient logical framework (set theory); *Composition mechanisms* that simplify development; and *Refinements* that provide a safe and efficient way to obtain secure computer code from abstract specifications.

The B method was used in the METEOR project [5], as well as in less well-known development projects [13, 18] and even non-critical development projects [52]. The software industry adopted B largely because of the availability of software tools supporting all phases of the B development process (semantics verification, refinement, proving, automatic code generation). Unlike most software tools, B support tools resulted from prototypes developed by industry rather than by the academic community. In fact, from 1993 to 1999, the “Atelier B” development project was funded by the “Convention B”, which was a collaborative effort of the RATP (Parisian Autonomous Transportation Company), SNCF (the French National Railway Society), INRETS (the National Institute for Transport and Safety Research) and Matra Transport (now Siemens Transportation Systems), among others.

A computer scientist in the field of formal methods will perceive certain paradoxes in the implementation of the B method. For instance, the B method uses programming languages, such as B kernel, that are not well documented or specified and that are not particularly well-suited to B’s high level of abstraction. In addition, some have observed that the B module language is rather complex and hard to understand. G. N. Watson [54] goes as far as to call the proliferation of constructs in the B module system, and the rules associated with them, somewhat *daunting*. Certainly, the semantics of the B module system has been studied by several research teams. Bert *et al.* [7] have developed a component algebra to express the INCLUDES and USES clauses. Potet *et al.* [45] have studied the IMPORTS and SEES links, demonstrating that, in B, it is possible to build an erroneous project that appears to be correct; they have also proposed criteria derived from the dependency graph, which must be verified to eliminate erroneous architectures. Dimitrakos *et al.* [19] have studied the various composition clauses, focusing on the conditions that must be verified to preserve the various component properties. Of course, it can be argued that B modularity is complex because there are various kinds of restrictions to ensure correctness. In any case, the inherent complexity of the B method makes it important to provide a synthetic way to understand these modular constructs (i.e., a way with just a few rules rather than *in extenso* as done by Abrial [2]).

Though these apparent paradoxes can be explained by the industrial origins of the method, the fact remains that the industrial tools currently available for B are often inappropriate for scientific research, and thus do not provide effective support for efforts to extend the use of the B method.

To remedy this problem, we have proposed the *BRILLANT* [12] framework, showing the feasibility of a safe software development system that ranges from semi-formal specification (UML) to contract-equipped code generation(i.e., equipped with assertions from the abstract model). This framework provides a central core into which various components can be plugged, including a central component, a UML plug-in, a proof plug-out, and a code generator plug-out.

This paper is a revised and enhanced version of the article presented in [15]. Section 2 gives an overview of the BRILLANT effort and the BCaml part of the platform. Section 3 briefly introduces the case study used to illustrate our approach, and section 4 presents the central component of *BRILLANT*, which allows the components of a B project to be manipulated. The parsing of B machines is examined in section 4.1.1, highlighting the central role of XML as an exchange format. Section 5 presents the proof plug-out used to validate B proof obligations, and section 6 describes a code generator plug-out that supports several target languages and is able to embed contracts into the code. Section 7 describes the UML plug-in used to translate UML projects into B so as to validate them, as described by Marcano & Levy [33] and Laleau & Polack [28]. Finally, section 8 presents our conclusions based on the results of our experiments with the implementation and use of the *BRILLANT* platform.

2 Overview of the BRILLANT platform

Figure 1 shows the BRILLANT platform's current organization. This platform is the result of several years of academic research and development by a few Masters degree and PhD students. The original impetus for the research was the lack of open tools allowing to manipulate and experiment with the B method. Thus, our first project was to implement a parser for the B language (described in section 4.1). We immediately realized that academia, or at least a part of it, was just waiting for such an open tool since soon after its first release, two studies [8,27] implemented our tool. Encouraged by this interest, we decided to continue by addressing the generation code process. Our first task was to find a way to take the modularity of the B language into account, which we accomplished in two directions by defining the flattening(see section 4.2.1) and BHLL (section section 4.2.2) components.

Since the B method is, at its center, proof related, a proof obligations generator (the component POG described in 4.3) was also needed. At this point in time, the only meta-language used in our platform development was Objective Caml [29]. However, it was obvious to us that a cooperative development platform could not remain language dependent, which led us to define and use an XML description for B in order to facilitate connections with the tools developped by others. This has already been done for connections with the proof assistant Phox (see section 5), and the XML description for ABTools [9,10] (on the left of figure 1) created by the lab, Heudyasic, is under development in order to provide a double chain to certify code.

Some internal components were also developed to perform several B translations (from or to Tex, html, ocaml, eiffel, ...). Although we were initially interested in generating code from B specifications, other researchers were working on how to pass from semi-formal UML specifications to formal B specifications. With these researchers, we were able to develop a plug-in designed to connect their tool to the BRILLANT B abstract syntax tree (see section 7). BRILLANT now provides a skeleton for a full development chain, from B specifications, even from UML ones, to code. Some extensions can be simply plugged in and explored. In the rest of this article, we will highlight some features of the different platform components and try to share some of the lessons learned while developing BRILLANT. The next section introduces the case study that will be used as an example throughout the article.

3 Case study

The traffic control system considered in this paper has been described in detail by Jansen and Schnieder [24] (see figure 2), who use knowledge about the railway domain as a basis for the formal specifications. The problem is to specify a radio-based Railway Level-Crossing (RLC) application developed for the German Railways [22] and to adapt it to the French rail system.

This application is distributed over three sub-systems: a train-borne control (TC) system (i.e., an on-board system), a level-crossing control (LCC) system, and an operations center (OC) system. The central system is the level-crossing control system, the two others being cooperating actors that make use of the LCC.

The level crossing is situated on a single-track railway line at a point where the line crosses a road on the same level. The intersection of the road and the railway line is considered the danger zone, since train traffic and road traffic must not enter it at the same time. Note that this is the main safety constraint that must be taken into account when describing the system.

The traffic lights and barriers at the level crossing are controlled by the LCC system. This system must be activated when a train approaches the level crossing. In the activated mode, the LCC performs a sequence of timed actions in order to safely close the crossing and to ensure that the danger zone is free of road traffic. First, the yellow traffic lights are switched on, and after 3 seconds, they are switched to red. After another 9 seconds, the barriers begin to lower. If the barriers have been completely lowered within a maximum time of 6 seconds, the LCC system signals the safe state of the level crossing, thus allowing the train to pass through the intersection.

The level crossing is opened to road traffic again once the train has passed completely through the crossing area, and the LCC system switches back into the deactivated mode.

Trains approaching the level crossing are detected via a process of continuous self-localisation on the part of the train and radio-based communication between the train and the LCC system. The vehicle sensor situated on the far side of the crossing triggers the re-opening of the barriers and tells the traffic lights to switch

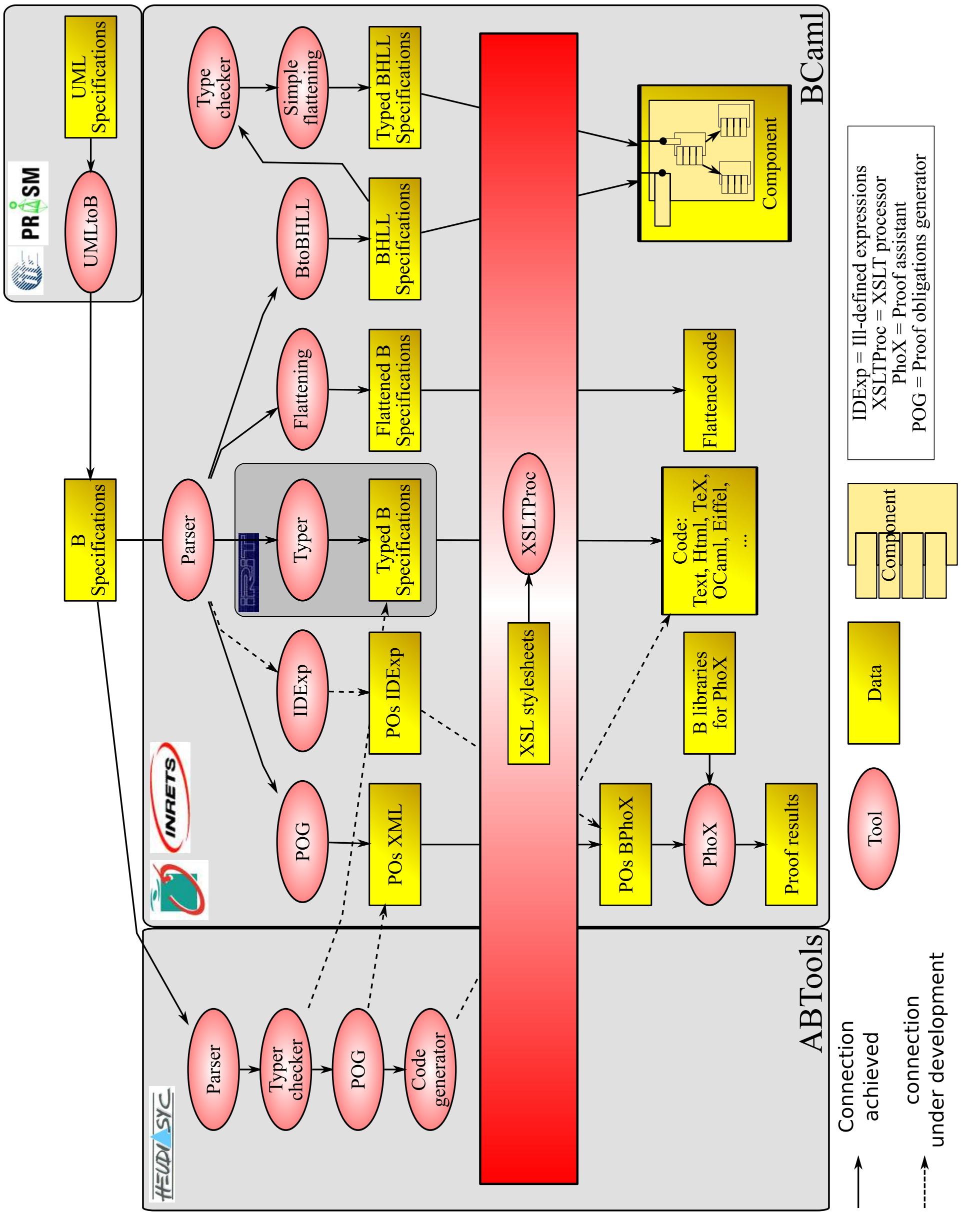

Fig. 1 The overall organization of *BRILLANT*

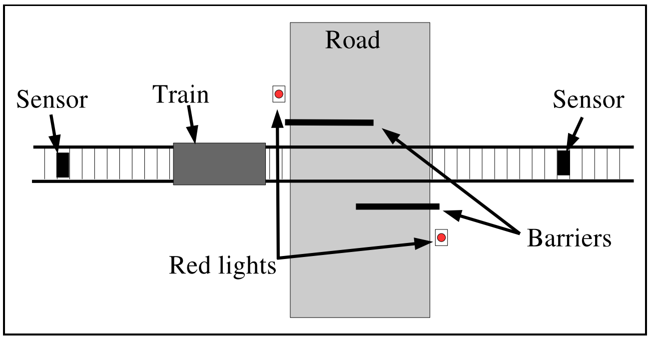

Fig. 2 The level crossing problem

off. During the activated mode, the LCC system can be in one of the following substates (Fig.3): yellow light showing; barrier closing; barrier closed; or barrier opening. Note that the time expirations occurring after the LCC is activated are denoted by the events `timeOut_1` (3 seconds later) and `timeOut_2` (9 seconds later).

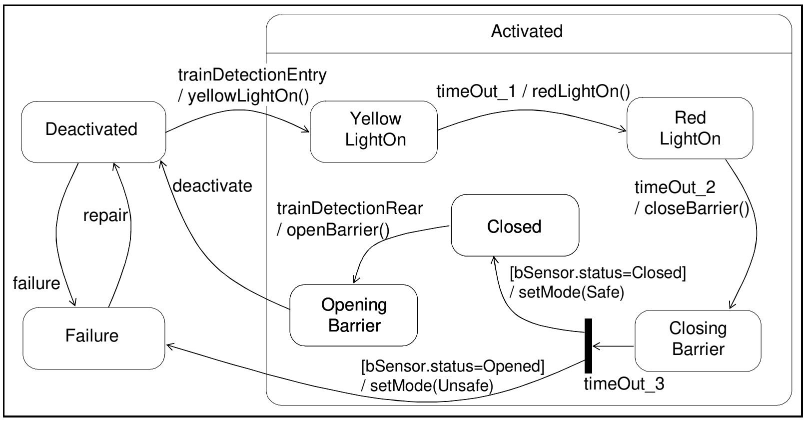

Fig. 3 State diagram of the LCC system

We will use this case study to illustrate our approach throughout the paper, except in section 6 in which we will use a bounded stack example that will be more convenient for code generation.

4 The BCaml Kernel

In this section, we describe the three component parts of *BCaml*: a parser (with an XML output library to connect *BCaml* with the outside world), libraries to handle the modularity of B projects, and a proof obligation generator.

4.1 BCaml input and output

The kernel of the *BCaml* platform is made up of the first bricks that were developed when the platform was created. These bricks specify the concrete grammar (section 4.1.1) that defines the B language and the abstract syntax (section 4.1.2) that defines the type used to manipulate the specifications. This may seem obvious, but it is nonetheless important because it makes collaboration with other developers possible. One of these collaborative projects led to the development of another brick in the *BCaml* kernel, called the *Btyper*. This brick is not described in this paper, but more details can be found in Bodeveix & Filali [8].

4.1.1 A concrete grammar for B

Several B grammars have been introduced over the years, coming from a variety of sources: *Clearsy* (prev. *Steria*), which corresponds to the grammar used in *Atelier B*; *B-Core*, which corresponds to the grammar used in *B-Toolkit*; and Mariano's PhD dissertation [36], based on the *B-Core grammar*, which was introduced to do metrics on B specifications.

In order to build a tool that would be as useful as possible, we needed to define a grammar that would take into account the criticisms of the grammars presented above. Our *BCaml* choices had to respect the following constraints:

- They had to be as compatible as possible with the machines that can be correctly parsed by the commercial tools mentioned above (i.e., *Atelier B*, *B-Toolkit*),
- they had to comply with the standard *Lex* and *Yacc* tools that allow *LALR* grammars to be defined, and
- they had to cause as few conflicts as possible.

We chose OCaml [29] as a support tool for our B development for several reasons. Not only does it allow symbolic notations to be handled easily, but in addition, it also implements efficiently and comes bundled with tools that allow the parsing of *LALR* grammars. Using these tools, we defined the B language in *LALR*.

4.1.2 Abstract syntax and XML syntax—two isomorphic formats

We used our definition of abstract syntax to directly infer an XML representation for B formal specifications. (Due to lack of space, this abstract syntax is not described here.) This XML encoding is called "B/XML" and is stored in an XML DTD file. Such abstract syntax is, as could be expected, more tolerant than concrete syntax, and contains elements that facilitate the handling of the syntax structure. For instance, the [*substitution*]*predicate* and [*variable_instanciation*]*substitution* constructions appear in this abstract syntax. The first construction corresponds to the form of a formula prior to a calculation of its weakest precondition. The second construction appears, for instance, when the parameters and operations have to be instanciated. The existence of these constructions in the abstract syntax mean that the structure can be manipulated to bring it closer to the matching mathematical definitions given in the B-Book [2].

We chose XML as our pivot format because of its flexibility and its ease-of-use with third-party tools. Using it makes our tools as independent of one another as possible, allowing a researcher to use our parser, but someone else's proof tool, for example. This flexibility is insured by the following aspects:

– XSL style sheets can be used to formulate simple recursive treatments of the XML structure, mostly transformations into other structured formats (e.g., LaTeX, HTML, or PhoX, as mentioned in section 4.3.3); and
– other programming languages can be easily used for more complex manipulations because most of the time these other languages are able to read XML data. This means that researchers can use their preferred programming language, as long as it has libraries for reading an XML format.

4.1.3 How BCaml *exploits the Abstract Syntax Tree (AST)*

BCaml takes advantage of both of the features described above. The XML approach extends the platform's plug-in/plug-out ability greatly, while the use of a well-defined, efficient meta-language as the core implementation language leads to a formal standard definition of the B grammar and allows the provision of more efficient components, easily understandable by others. The main drawback of this choice of formats is the difficulty of making both formats evolve together. When designing a component interface to be plugged in, one natural guideline is to use the core implementation language if high efficiency is required, in order to avoid a translation step (which is the choice made for the *BTyper*, the *POG* and the AST manipulation tools, for example), and to rely on the XML exchange format in all other cases (which is the alternative chosen for B/PhoX).

4.2 B modularity related AST processing

Two examples of the complex AST manipulations available in the *BRILLANT* platform are presented in the following sub-sections:

Flattening, which can be seen as a way of expanding the abstract mathematical code of a machine into a concrete code through its refinements and included machines;

Modularisation, which involves using a well-grounded modular system that produces a modular language from a flat language and a description of the desired modularity.

4.2.1 The flattening algorithm

B specifications are flattened by eliminating the refinement and composition links. The flattening algorithm aims to build a single B component, starting with a set of B components (a B "model") and grouping all the (selected) information extracted from the various specifications grouped into one formal text.

The resulting component is the "equivalent" of the initial model from the point of view of automatic code generation.

This notion of flattening exists implicitly in the B-Book [2]. Though Potet and Rouzaud used the term "flattening" in their work [46], it was Behnia [6] who specified the algorithm entirely, and it's her specification that we used in our tool. The principle of the algorithm is to connect the specification from the leaves (where only the `IMPORTS` and `REFINES` links are taken into account) to the root machine of the project.

Implementation: The flattening tool was the first tool implemented after the *BCaml* kernel was designed (section 4). This implementation was intented to "evaluate" the kernel's usability and to add those tools/libraries that would be useful for manipulating B specifications to the platform. In order to implement the tool, two things had to be done. First, the specification dependency graph had to be equipped to navigate through the specifications in order to build the successive flattened components. To accomplish this, we developed a library called BGraph, which implements the dependency graph type and the functions needed to manipulate that graph. Second, all the conditions that allow a set of B components to be flattened had to be verified.

4.2.2 The B-HLL module system

Overview The Harper-Lillibridge-Leroy module system (HLL) presented in Leroy [30] formalizes the Standard ML-like modules. The HLL system provides a means for adding a module language to a module-less core language. This system also permits a formal semantic to be given to an existing module language, as is the case for the ML modules. Moreover, this powerful semantic is able to implement the module language with relative simplicity.

Once the HLL module system has been instantiated, it is possible to define structures (i.e., list of values, types, modules or sub-modules) and functors (module-to-module functions) in the obtained modular language. A more complete description of our work on B-HLL can be found in our article published in 2004 [43].

Instantiation Figure 4 sums up our use of the HLL system. As shown in the middle level in figure, the HLL module system includes three functors. These functors are given modules defining the abstract grammar and the type-checking rules (upper level of the figure) and in turn produce several modules (lower level of the figure) that deal with the modularity of the language.

Our efforts to instantiate the HLL module system were divided into two parts, which are shown on the upper level of Figure 4. The first part involved defining the abstract language of the B core language under study, based on the abstract syntax defined during the development of the *BCaml* kernel. From this abstract syntax, we removed the part of the syntax dedicated to the modularity language, and then we developed a mapping function from the *BCaml* kernel abstract syntax to our new abstract syntax.

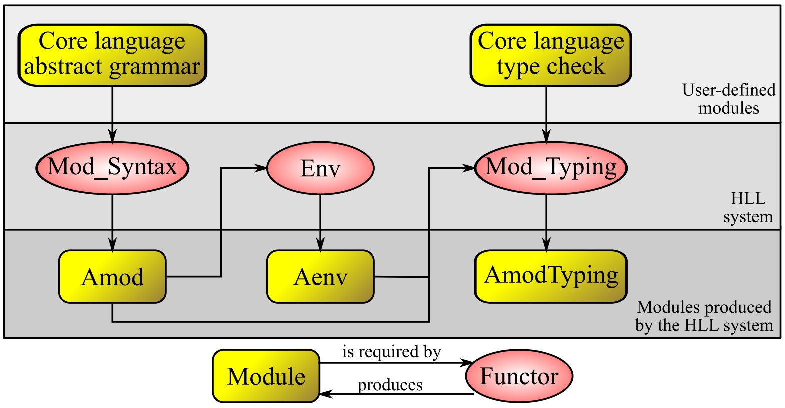

Fig. 4 Using the HLL module system

The second part of the instantiation involved defining the type checker. The types and the type-checking algorithm we used were adapted from the work of Bodeveix and Filali [8]. We added some type-checking rules to express the visibility rules described in the B-Book [2], and we also defined type-checking rules that take into account B language particularities, such as the semi-hiding principle and the prohibition of calling a given operation in the component where that operation is defined. (More details about this can be found in [43,40]).

The modules produced by the HLL system are:

Amod: the module for modularity functions, indicating what kinds of modules are available;
Aenv: the module for environment handling, indicating how information can be retrieved from referenced modules;
AModTyping: the module for modular typechecking, indicating how types can be inferred in a modular context.

By using the HLL system, it is possible to build a modular language for B almost automatically, without knowing much about the internals of the HLL system.

4.2.3 Code generation feedback

Although implementing the flattening algorithm validated the already developed bricks and constituted our first complete code production chain, using a published formally-defined system like the HLL module system as the core of the code generation process had an unexpected benefit in that it clarified the notion of modularity in the B language. Though the B developers do not need to know the details of how modularity is implemented, they do need to know that the generation tool complies with all the visibility rules specified in the B-Book. This tool allows structured code to be generated in components equipped with contracts (see section 6).

4.3 Generating proof obligations

In this section, we first describe the method used to implement the existing calculus for the weakest precondition. Then, we show how this calculus can be used to generate the proof obligations of a B project. Last, we present the various options available for exporting these proof obligations to other formats and other tools.

4.3.1 Generalized Substitution Language (GSL)

In order to generate proof obligations for B machines, we must be able to calculate the weakest preconditions of the substitutions. In a nutshell, the weakest precondition calculus allows the minimal state for a given program, or *substitution*, to be calculated in order to verify a given predicate, or *postcondition*. Hence, when we use “calculated” or “uncalculated” here, we refer to the state of a weakest precondition calculation. A proof obligation (PO) is calculated if all the weakest precondition calculations and all the variables instanciations have taken place. Thus, a calculated PO looks like a regular predicate, while an uncalculated PO still contains the substitutions expressed in a B component.

We chose to follow the approach defined in the B-Book [2]: reducing B substitutions to their smallest syntactic and semantic set (i.e., generalized substitutions). In the following paragraphs, we use *GSL* to denote both the syntactic set and the substitutions that compose it. Following the B-Book [2, B.3], we define the *GSL* in *BCaml* as an abstract data type, with the following notable exceptions:

- The assignment is defined as a multiple substitution; it serves as a basic construct once the parallel substitutions have been reduced, and thus can be viewed as an optimisation.
- The repetition substitution "$\frown$" is used to reason about the semantics of the *while substitution* and to propose a sufficient predicate [2, E.7] that could be used instead of the necessary and sufficient predicate.
- The instanciation ($[variable := expression]substitution$) of a substitution variable (the parameters of an operation, for instance) is reduced before transforming the substitution. (This idea is not documented precisely by Abrial [2], and thus it corresponds to an extrapolation on our part.)

The first and the third exception are rather straightforward, but the second requires a more detailed explanation. The real weakest precondition can be only obtained from a fixpoint application over this *repetition substitution*. This fixpoint can not be calculated in general by programming it. Thus, although the repetition GSL appears in the B-Book, it never appears in the actual calculations of the proof obligations, so we chose not to include it. Nonetheless, should the need arise, it could be introduced, because its semantics have been defined precisely.

With the help of the abstract data type, proof obligations can be generated according to the rules described in the B-Book [2, appendix E]. The corresponding

BCaml code was written with readability in mind, making it easy to match the code with the rule from which it is derived.

4.3.2 Proof Obligation Generation

The main steps for generating proof obligations from a project can be divided into precise steps. These steps are described in more detail below:

Parsing First, the machine and all the machines it depends on are parsed. To improve efficiency, we decided to directly use the *BCaml* kernel libraries directly for parsing rather than reading the XML files produced by the parser.

Generation of formulas The formula generation step is based on the B-Book [2, appendix F], resulting in proof obligations with the following form: $[Instanciation]Hypothesis \Rightarrow [substitution]Goal$.

This generation method allows more handling flexibility later on, for instance when debugging the proof obligation generator, or when showing students how proof obligations are generated, or when the proof tool applies the substitution to the goal. Figure 5 shows an example of an uncalculated proof obligation, derived from the B project presented in section 7.

```
  (LCC ∈ ℙ1 (ℤ)
  ∧ STATE ∈ ℙ1 (ℤ)
  ∧ STATE = {Deactivated,ShowingYlight,ShowingRlight,
              ClosingB,OpeningB,ClosedB,Failure}
  ∧ ...
  ∧ Yellow.lState(yellowLight(obj)) = On)
⇒
  [  state(obj) := ShowingRlight
  || Yellow.switchOff(yellowLight(obj))
  || Red.switchOn(redLight(obj))
  ]
   ( lcc ⊆ LCC
   ∧ lcc_barrier ∈ lcc → barrier
   ∧ ...
   ∧ ∀obj.
      ( obj ∈ lcc
        ∧ bStatus(lcc_sensor(obj)) = Opened
        ∧ bState(lcc_barrier(obj)) = Closed
      ⇒
         mode(obj) = Unsafe
      )
   )
```

Fig. 5 Uncalculated proof obligation for the **timeOut_1_showRlight** operation

Optimizations Several additional optimisations, or processing procedures, can be applied to the generated formulas. For example, formulas can be calculated, resulting in predicates that contain no substitutions. It is also possible to split the goal, by splitting the formula into as many formulas as there are members of the conjunction in the goal:

$$(H \Rightarrow G_1 \wedge \ldots \wedge G_n) \rightsquigarrow (H \Rightarrow G_1), \ldots, (H \Rightarrow G_n)$$

Other possible optimizations that were not implemented include removing formulas when the goal is trivially true or appears in the hypotheses, or changing the form of the formula to adapt it to a precise theorem prover. Certainly, it is sometimes easier to apply such transformations to the abstract syntax tree than to XML files using stylesheets. We did not implement these optimisations because we do not believe that semantic interpretation is the province of the PO generator, but rather of the prover, despite the fact that this semantic interpretation can be easily related to the abstract syntax (e.g., like a goal appearing in the hypotheses).

Final files and trace information Once the formulas have been generated, some trace information is embedded into the resulting file. Trace information can be found in the absolute name of the file, which reflects the kind of proof obligation that is in the file, and the machine from which it is generated. This trace information can be used later to find problematic parts of a B project if the corresponding proof cannot be achieved. The XML information in the file contains not only the predicate itself, but also a root tag named (for obvious reasons) `ProofObligation`. In addition, the file contains a tag that includes all the free variables of the formula because some theorem provers require that all variables be bound. This tag helps the stylesheet to generate a file for such theorem provers more easily.

4.3.3 Exporting to other tools

Once the proof obligations in the XML format are available, the XSL stylesheets allow them to be exported to other tools. For instance, the proof obligations can be converted into LaTeXfiles (figure 5 is an example of the results obtained); into text files, which are easily read by humans; into HTML files, which improve the readability of the formulas; or into a format suitable for a prover, in order to verify the proof obligations.

Figure 6 in section 5.1 presents the result of an XSL stylesheet application to the proof obligation shown in figure 5. In the next step, the theorem prover is fed the generated proof obligations file (see section 5). All of these steps (including replacing the conjunctions in the hypotheses with implications) are done via the XSL stylesheet, demonstrating the *ad hoc* suitability of this technology designed for simple treatments involving recursivity.

More complex transformations might be doable with stylesheets, but it would run the risk of becoming uselessly wordy and, more importantly, less maintainable. For this reason, we advise using stylesheets only for translations that preserve the overall structure.

5 From B proof obligations to correctness

BCaml provides the first two important types of B tools, presented in Abrial's *B#* [3, section 4]. The first includes the lexer, parser and typer; the second, the proof obligation generator. The third and last important B tool is the automatic, interactive prover. We chose not to develop such a tool with *BCaml* for a pragmatic reason: building a B prover according to our specifications takes much more time than developing dedicated libraries for an already existing prover. Instead, we built a replaceable add-on. We included the PhoX proof checker [44] because 1) it can be extended to the B mathematical foundations; 2) its GPL licence permits distribution along with *BCaml*; 3) its developers were willing to work closely with us; and 4) its intuitive syntax minimises library development time.

Our contributions consist of libraries dedicated to set theory for the PhoX proof assistant and the necessary extensions to make the proving task automatable. These extensions include:

– **A time-out process controller** The PhoX proof assistant is interactive but can be used on the command-line to "compile" (i.e., replay) proof scripts. We decided to add time-out limits in order to identify difficult proofs more quickly so they can be handled by a human.
– **XSL stylesheets** These stylesheets are used for translating B proof obligations, saved as XML files, to B/PhoX proof obligations.
– **A GNU Make script** These scripts re used to handle the whole chain from B proof obligations to proved formulas.

The result is a plugout for B projects, defined on top of the PhoX proof assistant, for automatically and interactively proving B proof obligations.

5.1 The bgop2phox *XSL style sheet*

```
add_path "/usr/share/brillant/bphox/".
Import Blib.
flag auto_lvl 2.
flag auto_type true.
theorem op
/\Activated,BARRIER,Closed,ClosedB,
  Closing,ClosingB,Deactivated,DownSpeed,
   ...,
  Yellow.lState,Yellow.light (
(LCC in (part1 Z)) ->
(STATE in (part1 Z)) ->
...
((Yellow.lState app (yellowLight app (obj))) = On)
  -> (
 /\obj ((
   ((obj in lcc) &
    ((state <+ \o (o = obj,ShowingRlight) app (obj))
       in Activated)) &
    ((bStatus app (lcc_sensor app (obj))) = Opened))
   ->
  ((mode app (obj)) = Unsafe)) ) )
 .
Try intros ;; auto.
save.
```

Fig. 6 One of the exploded proof obligations for **timeOut_1_showRlight**, converted to B/Phox

During the translation step, our XSL *bgop2phox* stylesheet is applied to the B/XML proof obligations using a XSLT processor. The XSL transformation schema allows recursive mapping. Thus, our translation is also defined recursively. A first-order language *à la B* is composed of various symbols for functions, relations, connectors and quantifiers. Figure 6 is the output of the XSL stylesheet applied to the formula in figure 5, after it was calculated and saved in an XML file.

A high-order language *à la PhoX* is a simply-typed lambda calculus with some typed constants. Our translation is based on associating every first-order B symbol S with a B/PhoX expression $S^{\dagger}$, such that its extension to the first-order terms formulae is simply defined by an inductive commutation. We showed [50] that under reasonable assumptions (basic constants and functions are similar up to the † translation) proofs made with B/PhoX are equivalent to B proofs. As a consequence, using the PhoX system of simple types makes our translation sound. Moreover, every non-freeness rule and every substitution rule can be easily obtained through the λ binder properties.

5.2 The blib *PhoX library*

The PhoX library for B reflects the first three chapters of the B-Book. The content of the library is outlined briefly here because the process for embedding B into PhoX is based on it. (More details are available in Rocheteau *et al.* [50]). In fact, the library is a collection of successive libraries for predicate calculus with equality, the boolean domain, cartesian products, set operators, binary relations, functions, arithmetic theory and finite sequencing.

5.3 The B/PhoX proof process

A successful proof for a B project is built using a fixed-point process involving the time-out controller. The whole set of generated proof obligations is run through a first session of the prover, with proofs that take longer than an arbitrary value being forcibly stopped via the time-out controller. A second session is then run, but only for those proof obligations that could not be proved, with a greater time-out value. All subsequent sessions follow the same logic, each time increasing the time-out value, until all proof obligations have been proved or until the engineer decides to stop everything and to prove the problematic formulas interactively. There is one drawback to this method: because PhoX is used as a "black box", the state of the ongoing proof is not saved at that moment it is stopped; thus the next session will have to replay the proof from the beginning.

Table 1 shows the result of a proof session for the famous "Boiler" B project. This table shows that all the reasonably easy proofs are finished within a small time-out value. The results are similar to the results for the same B project using the Atelier B tool, which demonstrates the viability of our approach using an interactive prover for automated proving.

Time-out	1 s.	5 s.	60 s.
Generated proof obligations		2295	
Successful	1823	1955	1971
Failed	0	0	0
Stopped	462	340	324
Proof Rate	79%	85%	85%

Table 1 Proof results for the Boiler

6 From B specifications to code

The generation process for producing flat code is illustrated in Figure 7 and the process for producing component-oriented code is illustrated in Figure 8. (More details on our approach to generating code can be found in references [41] and [42].) To generate flat code, the specifications have to be parsed, annotated with

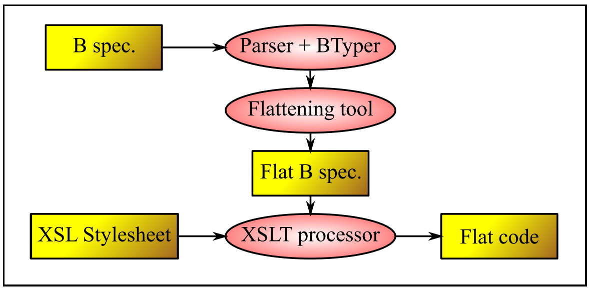

Fig. 7 The code generation process to obtain flat code

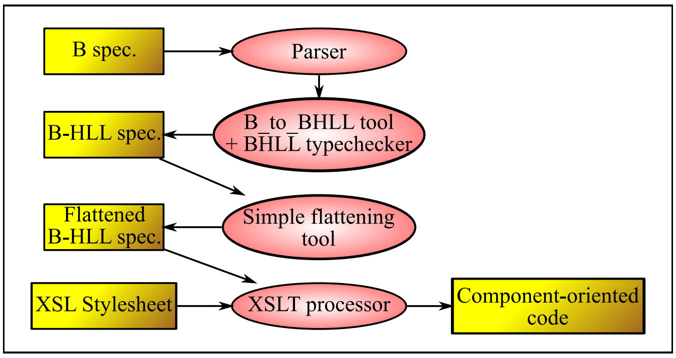

Fig. 8 The code generation process to obtain component-oriented code

their types, and then flattened. Code can be easily generated from the flat B specifications by using an XSLT processor and the appropriate stylesheet. To generate component-oriented code, the specifications must be parsed, and then translated into *BHLL* specifications; these, in turn, are annotated with their type. Once the specifications have been typed, they are run through the part of the flattening algorithm dedicated to eliminating refinement links in order to produce *BHLL* components. A stylesheet is then applied to the components thus obtained in order to generate the code. Since the structure of the specifications is maintained, we call this type of code generation, component-oriented code generation.

Figure 9 presents a B specification of a short and well-known example: a bounded stack. The code presented in figure 10 is generated from this specification. The package specifications use the generic Ada construction to translate the parameters that specify the size of the stack. Our approach to code generation allows the properties that are expressed in the specifications to be put into the code.

Provided that the target language has semantics similar to that of “B implementable code” (i.e., the substitution of an `IMPLEMENTATION`), then it is theoretically possible to use this target language for code generation. However, as indicated beforehand generating code using XSL stylesheets is only manageable for translations that maintain the original structure. For this reason, OCaml and Ada generations can be done with stylesheets, but it would be too difficult to do in the

```
MACHINE stack(stack_size)
CONSTRAINTS
  stack_size ∈ ℕ
∧ stack_size ≥ 1
∧ stack_size ≤ MAXINT
VARIABLES
  the_stack, stack_top
INVARIANT
  the_stack ∈ (1..stack_size) → ℕ
∧ stack_top ∈ ℕ
∧ stack_top ≥ 0
∧ stack_top ≤ stack_size
INITIALISATION
   the_stack :∈ (1..stack_size) → ℕ
|| stack_top := 0
OPERATIONS

push(addval) ≙
 PRE
   stack_top < stack_size
 ∧ addval ∈ ℕ
 THEN
   stack_top := stack_top + 1
 || the_stack(stack_top + 1) := addval
 END;
 ...
END
```

```
IMPLEMENTATION stack_1(stack_size)
REFINES stack
INITIALISATION
  the_stack := (1..stack_size) × {0};
  stack_top := 0
OPERATIONS

push(addval) ≙
  BEGIN
    stack_top := stack_top + 1;
    the_stack(stack_top) := addval
  END;
 ...
END
```

Fig. 9 A B specification of a bounded stack

```
generic
  stack_size : natural ;

package stack is
  function is_empty return boolean ;
  procedure push ( addval : in natural ) ;
  procedure pop ;
  function top return natural ;
  function initialised return boolean ;
end stack ;
```

```
package body stack is
--# invariant stack_top >= 0
--# and stack_top <= stack_size

  the_stack : array (1..Stack_size) of natural ;
  stack_top : 0..Stack_size ;

  procedure push ( addval : in natural ) is
  begin
     --# pre stack_top < stack_size
     stack_top := stack_top + 1;
     the_stack ( stack_top ) := addval ;
  end push ;

  ...

begin --initialisation
  stack_top := 0
end stack ;
```

Fig. 10 The specification and the body of the bounded stack Ada package

assembly language, for instance. In that case, more thorough structural manipulation would be easier to handle directly in the OCaml code of BCaml.

7 From UML/OCL models to B specifications

This section introduces an UML plug-in for BCaml that helps to verify the consistency of an UML model by translating it into a B specification and using the verification tools validate the B specification, which in turn ensures the consistency of the UML model. Sections 7.1,7.2 and 7.3 presents the steps involved in specifying a UML model and the associated safety constraints. The different steps are illustrated using the example of the railway level-crossing example presented in section 3. Sections 7.4, 7.5, and 7.6, respectively, describe how to translate UML diagrams into B, how to enrich the obtained B specification with the UML model's OCL constraints, and how to prove the consistency of the UML model by verifying the enriched B specification.

7.1 From requirements to UML models

The "elicitation problem" (i.e., the creation of a valid initial description of the required system properties) is essential to ensure the correctness of the future system. The consistency of the system depends on the developer's ability to understand and incorporate key safety properties. Therefore, knowledge of the context in which the system operates plays an important role in eliciting system requirements. Our approach to requirement analysis is based on the approach taken by Marcano *et al.* [34], in which both the static and dynamic properties of the system are taken into account through various UML diagrams.

Describing the entities involved and their invariants facilitates comprehension of the static properties. Describing the way that system actors will interact with each other and the system leads to a full comprehension of the dynamic properties of the future system. The Object Constraint Language (OCL) is used to express all the properties that cannot be expressed through diagrammatic notation alone (i.e., hypotheses and facts related to subsystems, classes, attributes and associations). OCL is also used to describe the system safety conditions. Thus, system modeling can be divided into two steps:

- First, UML sequence diagrams must be defined in order to describe both correct operating scenarios and failure scenarios
- Second, the expected behaviour of the system must be described completely as a set of OCL pre- and post-operational conditions

UML state diagrams will be defined for the combined operations of the entire system in order to describe the overall interaction of the system with each actor in its environment.

7.2 UML-based-modelling

In the railway example described in section 3, the central system is the level-crossing control (LCC) system, the two others-a train-borne control (TC) system and an operations center (OC) system-being cooperating actors that make use of the LCC. To create a UML model, first, it is necessary to identify the main entities that must be modelled to determine possible LCC system failure conditions. A primary cause of such failure conditions could be malfunctioning sensors or actuators. Defects leading to failures may be detected in the main physical structures or in the control systems themselves. In this case study, only a limited number of failures are considered: failures of the yellow or red traffic lights (which are considered separately), the barriers, and the vehicle sensor, and the delay or loss of messages sent through the radio network. For any of these failures, the following objects that interact with the LCC system (Fig.11) are examined: the lights, the barriers, the vehicle sensors, the train-borne control system, and the operations center.

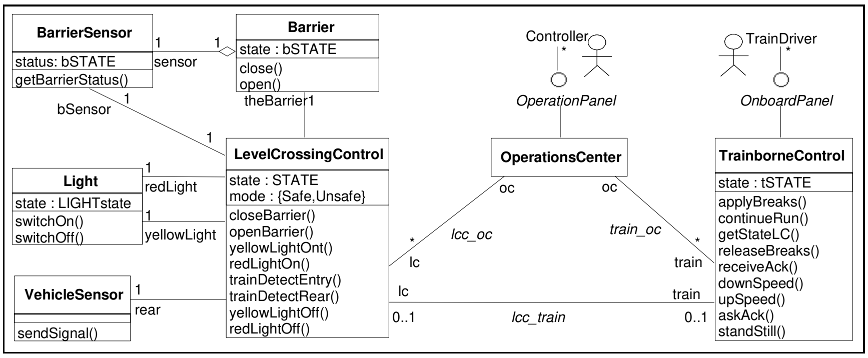

Fig. 11 RLC system : Class diagram

7.3 Adding OCL constraints

Using a standard formal language for constraint specification is an important step towards formalizing complex models, particularly in the context of critical safety systems. The purpose of OCL is to allow the constraints related to system objects to be formally specified, preserving the comprehensibility and readability of the UML models. OCL facilitates the statement of the properties and the invariants of the objects, as well as that of the pre/post-conditions for the operations. OCL also provides a navigation mechanism that allows attributes, operations and associations to be referenced in the context of a class or an object (a class variable), and query operators that permit a set of elements to be selected and/or modified. Each OCL expression has a specific type and belongs to a specific context. The context of an OCL expression determines its scope. Only the visible elements in the context of the expression can be referenced by means of navigation expressions.

Safety properties are included in the system invariants in order to propagate them from the abstract specification phase to the implementation phase. The main property of the LCC system is to prevent both road and rail traffic from entering the danger zone at the same time; to do so, the control specifications for the crossing area and its barrier, as well as any trains that may pass through the level crossing at any time, must be modelled at a high level of abstraction. For these reasons, the following OCL invariants are specified for these classes, as shown in Fig.12:

1. **Req.**_If there is a train crossing the danger zone then the barrier is closed_
   context CrossingArea inv:
   not(self.train->isEmpty()) implies self.barrier.state=Closed
2. **Req.**_If the barrier of the crossing area is open, then no train is approaching the danger zone_ context Barrier inv:
   self.state=Opened implies self.guards.train->isEmpty()
3. **Req.**_If the barrier of the crossing area is closed, then a train is crossing the intersection_ context PhysicalTrain inv:
   self.crosses.barrier=Closed

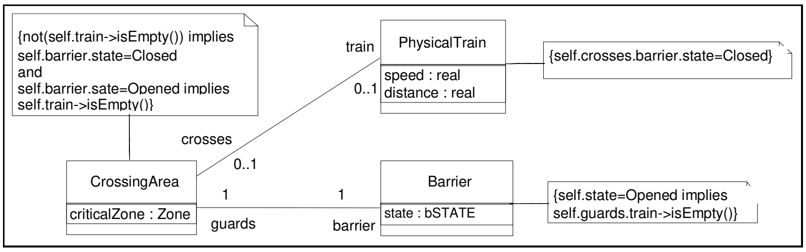

Fig. 12 Constraints related to the danger zone

These constraints must hold true for a more detailed design once decisions have been made about the actual type of hardware to be used in an implementation. In this case study, the notion of "train passing through the intersection" is connected to the activation of the railway level crossing. In order to accomplish this task, the front and the rear of the train must somehow be detected. We assume that the train can be detected directly through use of abstract vehicle sensors. The barrier state is detected by introducing a barrier sensor. In light of these assumptions, the previous OCL invariants can be refined by adding the following LCC system constraints for the class shown in Fig.11:

1. **Req.**_The red light is switched on whenever the barrier is closed, and the yellow light is switched on when the barrier is closing. If both the yellow and the red lights are switched off, then the barrier is open._
   context LCC_System inv:
   self.theBarrier.state=Closed implies self.redLight.state=On
   and self.theBarrier.state=Closing implies self.yellowLight.state=On
   and self.yellowLight.state=Off and self.redLight.state=Off
   implies self.theBarrier.state=Opened
2. **Req.**_If a train is in the danger zone, the level crossing is in an activated state composed of four substates (WaitingAck, Closing, Closed, Opening)._
   context LCC_System inv:
   not(self.train->isEmpty()) implies self.state=Activated and

Set(Activated)=Set(WaitingAck->Union(Closing)->Union(Closed)->Union(Opening))

3. **Req.** *If the LCC system is in the activated state while the barrier is open, then the level crossing is in an unsafe mode.*

   context LCC_System inv:

   self.state=Activated and self.bSensor.state=Opened implies self.mode=Unsafe

4. **Req.***If the registered state of the barrier is closed and the trigger sensor indicates that it is open, then the level crossing is in an unsafe mode. This is the case when the barrier is in the closing state; the LCC remains unsafe until the barrier is completely closed.*

   context LCC_System inv:

   self.bSensor.state=Opened and self.theBarrier.state=Closed) implies self.mode=Unsafe

The operations of the LCC class are specified with OCL pre- and post-conditions. OCL is also used in sequence diagrams to complete the preconditions and invariants related to operations. Although state diagrams are used to derive the initial specification of each operation (i.e., the description of a state transition), OCL constraints are needed to add supplementary information that can not be retrieved from the state diagrams.

Consider the closing of the barrier raised by the event `timeOut_1`. The precondition of the operation `closeBarrier` ensures that the yellow light is switched on before sending the closeBarrier order, in addition to ensuring that the barrier has not yet been closed. The postcondition ensures that the state of the yellow light is off, the state of the red light is on, and the state of the barrier is closed. The operation is specified as follows:

context LCC_System::closeBarrier
pre: self.yellowLight.state=On and self.theBarrier.state=Opened
post: self.yellowLight.state=Off and self.redLight.state=On and
self.theBarrier.state=Closed

7.4 Formalization of classes and state diagrams

Our main goal is to extract an initial B specification (called the "abstract" specification) from the UML diagrams and to use it to check for inconsistencies. To do so, an abstract machine is associated to each class. Subsequently, the B method is used to provide details about each component with regard to the behavior of class operations and the global invariants. At this point, the system developer must make several important decisions concerning unspecified properties and then introduce these properties into the UML diagrams using OCL constraints. The UML diagrams are then translated into B, and the resulting B specification is used to check the consistency of all the UML diagrams and OCL constraints.

Consequently, the B specification is not a good illustration of code generation because the overall process was designed with consistency checking in mind rather than code generation. However, the existence of a tool allows us to explore in what

```
MACHINE Barrier
USES BarrierSensor
SETS BARRIER
VARIABLES barrier, bState
INVARIANT
   barrier ⊆ BARRIER
∧ bState ∈ barrier → bSTATE
INITIALISATION
   barrier ,bState := ∅,∅
OPERATIONS

openBarrier(obj) ≘
  PRE
    obj ∈ barrier
  ∧ bState(obj) = Closed
  THEN
    bState(obj) := Opened
  END;

closeBarrier(obj) ≘
  PRE
    obj ∈ barrier
  ∧ bState(obj) = Opened
  THEN
    bState(obj) := Closed
  END;

setBState(obj, newBState) ≘
  PRE
    obj ∈ barrier
  ∧ newBState ∈ bSTATE
  THEN
    bState := bState <+ {obj ↦ newBState}
  END;
```

```
res ⟵ getBState(obj) ≘
  PRE
    obj ∈ barrier
  THEN
    res := bState(obj)
  END;

obj ⟵ createBarrier ≘
  PRE
    barrier ≠ BARRIER
  THEN
    ANY new
    WHERE new ∈ BARRIER − barrier
    THEN
        barrier := barrier ∪ {new}
    || bState := bState ∪ {new ↦ Opened}
    || obj := new
    END
  END;

supBarrier(obj) ≘
  PRE
    obj ⊆ barrier
  THEN
     barrier := barrier − obj
  || bState := obj ⩤ bState
  END

END
```

Fig. 13 Formalization of classes (machine Barrier)

measure code generation is feasible, and what adaptations to the process need to be made in order to be able to generate code from UML specifications.

7.4.1 Classes

Consider the class *Barrier* and its first B specification, presented in Fig. 13. Since a class includes both the static and dynamic properties of a set of objects, it seems natural to model it using one abstract machine. The resulting abstract machine Barrier describes the deferred set BARRIER of all the possible instances of the class *Barrier* . The set of existing instances is modelled using a variable barrier constrained to be a subset of BARRIER.

Each attribute (i.e., *bState*) is represented by a variable (i.e., bState) defined in the INVARIANT clause as a total function between the set barrier and its associated type (i.e., bSTATE). Each operation of the machine has at least one parameter obj representing the object on which the operation is called. It may have a list of typed arguments args, which will be completed in the later translation of the state diagrams and OCL constraints.

7.4.2 Formalization of state diagrams

State diagrams are used to introduce the behavioral (i.e., dynamic) properties of the system into the B specification. The set of all possible states of a class is formalized using an abstract set that is defined in the corresponding B machine. An abstract variable is used to reference the current state of the class objects. This variable is defined as a total function, whose domain is the set of instances and whose range is the set of possible states. Each transition between two states is formalized by a B operation, whose name is that of the incoming event. Whereas the precondition of the operation is deduced from the transition guard, the postcondition describes the transition to the new state. Let us consider the state diagram of the `LCC_System` class shown in figure 3: the transition from the `showingYlight` state to the `closingB` state activated by the event `timeOut_1` is formalized as shown in Fig.14. Note that we have included some information obtained from the OCL definition of the operation closeBarrier, since this operation is activated by the event `timeOut_1`. (The OCL translation is described below.)

When the same event can activate two different transitions depending on the guard condition, then both transitions are formalized by the same operation of the B machine. The non-deterministic construction `SELECT` is used to describe each transition, as illustrated in Fig.14 for the formalization of the event `timeOut_2`. The time constraints (not shown in the example, see reference [35] for details) are handled in the precondition by checking the value of a clock variable defined in a abstract clock machine. The progression of time is added to the body of the operations, by calling the relevant operation of the clock machine with a value of time progress. This value can be made indeterministic by using the "unbounded choice" construct of B.

```
MACHINE LCC_System
...
OPERATIONS
...
timeOut_1(obj) ≙
  PRE
    obj ∈ lcc
  ∧ state(obj) = YellowLightOn
  ∧ bStatus(lcc_sensor(obj)) = Opened
  ∧ bState(lcc_barrier(obj)) = Opened
  ∧ Red.lState(redLight(obj)) = Off
  ∧ Yellow.lState(yellowLight(obj)) = On
  THEN
      state(obj) := ClosingB
   || closeBarrier(lcc_barrier(obj))
   || Yellow.switchOff(yellowLight(obj))
   || Red.switchOn(redLight(obj))
  END;
```

```
timeOut_2(obj) ≙
  PRE
    obj ∈ lcc
  ∧ state(obj) = ClosingB
  ∧ bState(lcc_barrier(obj)) = Closed
  ∧ Red.lState(redLight(obj)) = On
  ∧ Yellow.lState(yellowLight(obj)) = Off
  THEN
    SELECT bStatus(lcc sensor(obj))=Closed
    THEN state(obj):=ClosedB
       || mode(obj):=Safe
    WHEN bStatus(lcc_sensor(obj))=Opened
    THEN state(obj):=Failure
    ELSE skip
    END
  END;
...
END
```

Fig. 14 Formalization of state diagrams

Once the classes and state diagrams have been translated and integrated into the initial specification, OCL constraints are used to complete the B machine invariants and operations.

7.5 Formalization of OCL constraints

In this section, we explain how OCL expressions are translated into B expressions, using the rules defined for the OCL meta-model in Marcano's PhD dissertation [26].

The Object Constraint Language has thus far been defined semi-formally using textual descriptions, a grammar that specifies the concrete syntax, and examples that illustrate the semantics of the expressions. Such a presentation style is adequate for illustrating OCL concepts, but it is not sufficient for providing a rigorous semantic. This semi-formal nature of the OCL definition, which often leads users to interpret the UML models ambiguously, restricts its use in critical safety applications. This difficulty is increased by the lack of tools supporting the analysis of OCL expressions and the "proof" of complete UML models.

Recent work proposing a precise semantic for OCL has been carried out by Richters and Gogolla [48]. In addition, both these authors [49] and Jackson *et al.* [23] have published research related to tools for verifying UML designs. The first publication proposes an approach for validating UML models using simulation, and the second proposes an object model analyzer that uses Alloy, which is based on Z. Two of the authors of the present paper have also previously worked to formalize OCL with B, using a system of translation rules between the abstract syntaxes of both languages [32].

In the *BRILLANT* plug-in, two types of OCL constraints are taken into account. The first type of constraint specifies an invariant of a class, and the second type specifies a precondition and/or a postcondition of an operation. In the first case, translating the OCL constraint consists of combining a new predicate with the invariant of the related B machine, whereas in the second case, it requires completing an operation of the machine. The formalization of the LCC system's OCL invariant is shown in Fig.15.

```
MACHINE LCC System
CONSTANTS Activated
PROPERTIES Activated ∈ STATE ∧ Activated={YellowLightOn,ClosingB,OpeningB,ClosedB}
INVARIANT
 ...
   ∀obj.(obj ∈ lcc ∧ bState(lcc_barrier(obj))=Closed ⇒ Red.lState(redLight(obj))=On)
 ∧ ∀obj.(obj ∈ lcc ∧ bState(lcc_barrier(obj))=Closing ⇒ Red.lState(yellowLight(obj))=On)
 ∧ ∀obj.(obj ∈ lcc ∧ Yellow.lState(yellowLight(obj))=Off ∧ Red.lState(redLight(obj))=Off
          ⇒ bState(lcc_barrier(obj))=Opened)
 ∧ ∀obj.(obj ∈ lcc ∧ objdom(lcc_train) ⇒ state(obj) ∈ Activated)
 ∧ ∀obj.(obj ∈ lcc ∧ state(obj)=Activated ∧ bStatus(lcc_sensor(obj))=Opened ⇒ mode(obj)=Unsafe)
 ∧ ∀obj.(obj ∈ lcc ∧ bStatus(lcc_sensor(obj))=Opened ∧ bState(lcc_barrier(obj))=Closed ⇒ mode(obj)=Unsafe)
 ...
END
```

Fig. 15 Formalization of OCL invariants

In the same way that OCL predicates enrich the UML model, OCL pre- and post-conditions are used to enrich B machine operations. In Fig.14, the pre-condition of the operation `timeOut_1` not only requires that the LCC system be in the `yellow-Light` state (which is generated from the state diagram) but also that the red light be switched on and the barrier be closed. These three contraints together consitute the translation of the OCL predicate:

self.yellowLight.state=On and self.theBarrier.state=Opened.

The post-condition of the operation initially includes only the substitution `state(obj):=ClosingB` that sets the new state of the LCC instance (obj). This post-condition is completed by translating the OCL post-condition

self.yellowLight.state=Off and self.redLight.state=On and self.theBarrier.state=Closed

into B, which generates the following parallel substitutions:

```
   closeBarrier (lcc_barrier(obj))
|| Yellow.switchOff (yellowLight(obj))
|| Red.switchOn (redLight(obj))
```

Please note that although OCL constraints can be sufficient for expressing the behavior of the model, it can be very difficult to extract this behavior in order to express it in terms of B operations. Thus, state diagrams are necessary to provide the skeleton for the behavior, upon which additional information can be imported from the OCL pre- and post-conditions.

7.6 Verification of the entire model

In order to automate the formalization process, we implemented a prototype tool that derives B specifications from UML/OCL models. The strength of this tool is that it does not depend on any UML modelling tool. Instead, it does the translation using XMI files (i.e., files in an XML format describing a UML architecture) as input. As a result, this tool can still be used even if UML modelling tools change their Application Programming Interface.

Once the whole B formal specification has been generated from the UML/OCL model, it has to be type-checked and then verified through a proof process. A dedicated tool is then used to automatically generate and prove the proof obligations (POs). The POs guarantee that the B machine's operations conform to its invariant. Each operation raises proof obligations related to its pre-condition and substitution parts. The non-proven POs are used to detect inconsistencies between the invariant and the preconditions, as well as to detect the incompleteness of a post-condition.

If a proof obligation cannot be proven using the theorem prover, then the developer must review the related OCL invariant or operation and make the necessary modifications to allow the obligation to be proved. Our approach is a one-way approach: it allows UML/OCL to be translated into B, but not the other way round. When the type-checker or the prover finds an error in the specification, the user must first understand the B specification and then search in the UML/OCL model to find the error. Please note that it is quite simple for the developer to find the UML element associated to a B expression because the names are roughly the same and each OCL expression is translated into a simple B expression. Thus, in order to facilitate the task, we have made it possible to create and maintain concrete links between the UML/OCL models and B specifications throughout the development process.

8 Discussion, conclusion and perspectives

8.1 Comparison with other works

Several other formal methods also employ the same kinds of tools, and were developed similarly to use either open-source high-level languages or formats, or both. Many of them can be found in the www formal methods' virtual library [21], along with links to the formal methods they implement. In addition, freely available (but not open-source) tools exist for the Bmethod: B4Free [4] (distributed by *Clearsy*) and ProB [47]. The former is a parser, type-checker, proof obligation generator and prover programmed in the B0 language. The latter is an animator and model-checker programmed in Prolog, and uses the XML files produced by the jBTools [25] as input.

BRILLANT can also be compared to projects of a similar nature, that have similar ambitions and/or designs: Rodin [51] (for *B#*), Overture [38] (for VDM++) and the Community Z Tools [17] (*CZT*). These projects share several common points. For example, they all use XML-based interchange formats, and they all have similar architectures, in which the core tools (e.g., parsers, typecheckers, testers/validators, plug-ins and/or plug-outs) are clearly separated. Some are driven by research needs (*BRILLANT*, *CZT*), some by industrial interests (Rodin), and some by both (Overture).

However, despite their similar architectures, the tools' underlying implementations are quite different. Rodin and Overture are based on the Eclipse IDE [20] and thus provide a very consistent development environment. Writing the mandatory parts of such a framework (e.g., parsing, compiling, graphic interfaces, test suites) is therefore more scalable and reusable, if not any easier. *CZT* and *BRILLANT*, on the other hand, are more or less a loosely connected set of tools: the first was developed using Java (their edition of Z specifications is even supported by jEdit), while most of the *BRILLANT* tools were developed in OCaml.

Because *CZT* and *BRILLANT* appear to have more points in common, it is appropriate to focus this comparison on them. *BRILLANT* was developed before the others, although the ideas behind *CZT* appeared around the same time[1].

The developers of *CZT* have provided information about the design decisions behind their tools. In the following paragraphs, these decisions are analyzed by comparing them with similar decisions that have been made at one time or another during the development of *BRILLANT*. (All citations come from reference [17].)

BCaml, *BRILLANT*'s most important tool, uses what in *CZT* terminology [17] is called an immutable type, which helps *BRILLANT* to avoid the problems described by the developers of *CZT*. Initially, these developers implemented a "mutable" Abstract Syntax Tree (AST) approach, then moved to a combined approach

[1] A sketch of a formal developement platform can be found in reference [36], ch-2, p-29, published in 1997. The CZT initiative was launched around September 2001

called "a 'Both' approach", but finally ended up shifting gradually towards an "immutable" approach. This means that, initially, the AST (i.e., the type representing the Z specifications) could be modified during tool execution, which allowed more efficient algorithms to be used, but made the sharing of common subtrees (i.e., common pieces of specifications) difficult. In addition, given the initial "mutability" of the AST, the programmers had to ensure that it did not change during certain specific parts of the execution, which required a comprehensive knowledge of the overall tools. The 'Both' approach tried to combine the advantages of the mutable approach and the immutable approach, but only succeeded in increasing complexity. *CZT* has now moved towards the immutable approach, both because it is less error-prone and because it makes things simpler for the developer. Thus, by using OCaml, *BRILLANT* avoided the pitfalls of optimization for the immutable type of the AST. This made sense to us for 2 reasons: 1) the literature usually advocates using a static tree to represent abstract data structures, and 2) the OCaml language actually encourages such an implementation.

When developing *BRILLANT*, we wanted to make it possible to add B-HLL to *BCaml* at some later date, which required insuring identifier unicity. The *CZT* developers also faced this problem, which arises when an abstract structure defines binders, either as universal or existential quantifiers in predicate calculus, or as local vs. global variables in programming languages. In *CZT*, three common solutions to this problem were tried: renaming bound variables when necessary, which is a traditional solution, but one that makes it "incredibly easy to make subtle errors"; using De Bruijn indices, which is "easier to get right", but leads to unreadability and complexity; or using unique names for all bound variables, which is safer, but requires unique names for all models of a Z project, and ends up making the output less readable. They finally chose the last solution for *CZT*, with the additional benefit that the unique attributes of the identifiers could be exported in the XML AST in order to use the ID/IDREF (unique identifiers/symbolic link to unique identifiers) standard attributes. For *BRILLANT*, we chose a similar solution in *BCaml*: we decided to institute a scoping phase after the parsing phase in order to associate each identifier with a unique identity. Later on, the processing of new variables was facilitated not only by that unicity, but also by the recursive nature of the syntax trees, which allows all free variables to be retrieved. Indeed, several libraries that provide functions for high-level data structures (e.g., lists, sets, hash tables) exist in OCaml: and we were able to benefit from using these libraries immediately.

Another frequent problem is slow processing times. However, both teams managed to avoid this problem: like the tools in *CZT*, the *BCaml* tools in *BRILLANT*, interact via the abstract syntax tree directly instead of XML exchange files, which speeds up the processing.

All the above problems have been described in publications dealing with *Overture* [38], although the problem seems to have been more connected to their choice of compiler generator than to the compilation technique itself. Since the Rodin project is still in the development stage, we would encourage its developers to look at the projects discussed above, among others, in order to benefit from their experience.

From our perspective, *BRILLANT* has a very relevant process for developing for formal methods. Implementing the B method with our tools helps to highlight the weaknesses of the mathematical definition of the method, such as the lack of documentation for handling variable instanciations in the weakest precondition calculus, or shortcomings of its more technical definition, such as the problem of parsing B models with standard Lex and Yacc tools. Moreover, like UNIX, *BRILLANT* adheres to a philosophy of separate but interconnected tools. Not surprisingly then, the consequence is that extensions and extra tools are easily added to the platform. From that perspective, we think that *BRILLANT* is a more than adequate tool. Since most development projects are done by PhD students and young interns who are not yet full-time engineers, the simplicity of programming language chosen for *BRILLANT* (OCaml) is a definite plus. Its powerful, well-documented, high-level libraries, and its ability to blend various programming design paradigms, already well-known in the academic scientific community, make it a most appropriate choice. In addition, the by-the-book formalism implemented in *BRILLANT* makes integrating extensions, such as the B-HLL module system, much easier than the other possibilities.

8.2 Conclusion

The *BRILLANT* platform design has two principal advantages: it uses open and standardized formats, and the source codes for its tools (OCaml and/or Java so far) are openly available. In addition, it can be used to test and/or validate B-related experiments, and in fact, we were the first users of many of the prototypes now available for the platform (e.g., bparser, bgop, btyper, bphox). We have been working to finetune the platform to help it meet the needs of other theoretical research projects, including but not limited to extending the B language, improving the current tools, providing couplings with other provers (e.g., Coq, Harvey), and offering other validation formalisms (e.g., model-checking).

From the information presented in this article, it would appear that we have reached our goal of providing an open and standardised format (XML) for a platform that has become a testbed for several other fundamental research projects (e.g., UML/OCL/Bcoupling in section 7, proofs in section 5, code generation in section 6). All the source code examples in the article (i.e., B machines, logical formulas, XML, PhoX) either come from *BRILLANT*, or derive from its use: The B machines are LaTeX files in the *BRILLANT* style, and the LaTeX files were obtained by applying an XSL stylesheet to XML abstract machines, themselves obtained by parsing ASCII B machines.

The following table provides some figures about the size of the project:

Component	Lines of code	Comments
kernel	8900	
parser	3200	
type-checker	12500	(it has some duplicate code with the kernel)
PO generator	4000	(GSL libraries included)
flattening tool	1700	
bhll	2400	
bphox	4200	(PhoX libraries and stylesheets)
uml plug-in	20300	

8.3 Perspectives

In the future, we will work to integrate technologies that endorse the use of open formats into the *BRILLANT* platform. The following modifications are planned: the use of XML schemas [53] instead of DTDs for validating XML files; the use of some recent promising results to replace XSLT processors with OCaml-Duce[37]; the use of XML flexibility to increase traceability between UML models, B machines, proof obligations and other derived models (e.g., generated code, test cases); the representation of B models as projects databases using XPath and XML-Query [16]; and a distributed platform architecture using XML-RPC [55], which will allow the parser and prover to be represented as servers to which B projects can be sent for parsing, validation or other tasks (in a kind of "B-forge").

We also plan to finalize the integration of ABTOOLS [10,9] within *BCaml*. Now part of *BRILLANT*, both of these tools were initially developped independently. ABTOOLS, which provides an open environment based on ANTLR and JAVA, makes it easier to design and test extensions of the B language. Lastly, we hope to define an ergonomic interaction mode for the various platform tools, by proposing a graphic interface suitable for the underlying platform technologies. This interaction will, consequently, rely heavily on XML technologies.

Several other projects, these more related to the fundamental research currently under way, also offer interesting perspectives for the future, such as UML/OCL/B coupling [33], temporal extensions for B [14], and safe software component generation [42]. Much work remains to be done, and the platform developers will be happy to provide their assistance to those who would like to try to use the tools in the context of their own research. All the necessary resources for building *BRILLANT* should be available on the web site dedicated to collaborative free software development [12].

References

1. *ZB'2000 – International Conference of B and Z Users*, volume 1878 of *Lecture Notes in Computer Science (Springer-Verlag)*, Helsington, York, UK YO10 5DD, August 2000.
2. Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

3. Jean-Raymond Abrial. $B^{\#}$: Toward a Synthesis between Z and B. In *ZB'2003 - Formal Specification and Development in Z and B*, pages 168–177, 2003.
4. B4Free. `http://www.b4free.com/`.
5. Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 369–387, 1999.
6. Salimeh Behnia. *Test de modèles formels en B : cadre théorique et critères de couvertures*. Thèse de doctorat, Institut National Polytechnique de Toulouse, October 2000.
7. Didier Bert, Marie-Laure Potet, and Yves Rouzaud. A study on components and assembly primitives in B. In Henri Habrias, editor, *Proceedings of 1st Conference on the B method*, Putting into Practice methods and tools for information system design, pages 47–62, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. B1996, IRIN Institut de recherche en informatique de Nantes.
8. Jean-Paul Bodeveix and Mamoun Filali. Type synthesis in B and the translation of B to PVS. In *ZB'2002 – Formal Specification and Development in Z and B* [31], pages 350–369.
9. Jean-Louis Boulanger. Abtools : Another b tool. In Ricardo J. Machado Johan Lilius, Felice Balarin, editor, *ACSD, Third International Conference on Application of Concurrency to System Design*, pages 231 – 232, Guimaraes, Portugal, June 2003. IEEE. ABTools provides an open environnement based on ANTLR and JAVA and provides somes facilities for design and test an extension for the B language.
10. Jean-Louis Boulanger. ABTools: A Free Tools for the B Method. In *WMSCI 2005, 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando, USA, jul 2005.
11. S.M. Brien and J.E. Nicholls. Z base standard: Version 1.0. Technical Monograph PRG-107, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, UK, November 1992.
12. BRILLANT. `http://gna.org/projects/brillant`.
13. M. Carnot, C. DaSilva, B. Dehbonei, and F. Mejia. Error-free software development for critical systems using the B-methodology. *IEEE*, pages 274–281, 1992.
14. Samuel Colin, Georges Mariano, and Vincent Poirriez. Duration calculus: A real-time semantic for B. In *First International Colloquium on Theoretical Aspects of Computing*. UNU-IIST, september 2004. Guiyang, China.
15. Samuel Colin, Dorian Petit, Jérôme Rocheteau, Rafaël Marcano, Georges Mariano, and Vincent Poirriez. BRILLANT : An open source and XML-based platform for rigourous software development. In *SEFM (Software Engineering and Formal Methods)*, Koblenz, Germany, september 2005. AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press.
16. World Wide Web Consortium. XQuery: the W3C query language for XML – W3C working draft. Available at http://www.w3.org/TR/xquery/, 2001.
17. Comprehensive Z Tools. `http://czt.sourceforge.net/`.
18. Babak Dehbonei and Fernando Mejia. Formal development of safety-critical software systems in railway signalling. In M. G. Hinchey and J. P. Bowen, editors, *Applications of Formal Methods*, Series in Computer Science, pages 227–252. Prentice Hall International, 1995.
19. Theo Dimitrakos, Juan Bicarregui, Brian Matthews, Tom Maibaum, and Ken Robinson. Compositional structuring in the B method: A logical viewpoint of the static context. In *ZB'2000 – International Conference of B and Z Users* [1], pages 107–126.
20. Eclipse. `http://www.eclipse.org/`.
21. The www formal methods' virtual library. `http://vl.fmnet.info/`.

22. Betriebliches Lastenheft für FunkFahrBetrieb. Stand 1.10.1996, 1996.
23. Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the Alloy costraint analyzer. In *International Conference on Software Engineering*, Limerick, Ireland, June 2000.
24. L. Jansen and E. Schnieder. Traffic control system case study: Problem description and a note on domain-based software specification. technical report, 2000.
25. jBTools. `http://lifc.univ-fcomte.fr/~tatibouet/JBTOOLS/`.
26. Rafael Marcano Kamenoff. *Spécification formelle à objets en UML/OCL et B : Une approche transformationnelle*. Thèse de doctorat, Université de Versailles – PRiSM, December 2002.
27. Regine Laleau and Amel Mammar. A generic process to refine a B specification into a relational database implementation. In *ZB'2000 – International Conference of B and Z Users* [1], pages 22–41.
28. Régine Laleau and Fiona Polack. Coming and going from UML to B : A proposal to support traceability in rigorous is development. In *ZB'2002 – Formal Specification and Development in Z and B* [31], pages 517–534.
29. X. Leroy, D. Doligez, D. Garrigue, J.and Rémy, and J. Vouillon. The objective caml system. Technical report, INRIA, 2005. Software and documentation available on the Web `http://caml.inria.fr/`.
30. Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
31. LSR-IMAG. *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, Grenoble, France, January 2002.
32. Rafael Marcano and Nicole Levy. Transformation rules of OCL constraints into B formal expressions. In *CSDUML'2002, Workshop on critical systems development with UML. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.
33. Rafael Marcano and Nicole Levy. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.
34. Rafael Marcano, Georges Mariano, and Philippe Bon. UML-based design and formal analysis of railway traffic control systems. In Schnieder E. and Tarnai G., editors, *FORMS'2004 / FORMAT'2004*, pages 173–182, Braunschweig, Germany, December 2004.
35. Rafaël Marcano, Samuel Colin, and Georges Mariano. A formal framework for uml modelling with timed constraints : Application to railway control systems. In *SVERTS: Specification and Validation of UML models for Real Time and Embedded Systems*, Lisbon, Portugal, October 2004. (in conjunction with 7th International Conference on the Unified Modeling Language, UML 2004).
36. Georges Mariano. *Évaluation de logiciels critiques développés par la méthode B : une approche quantitative*. Thèse de doctorat, Universitée de Valenciennes et du Hainaut-Cambrésis, Dec 1997.
37. OCamlDuce website. `http://www.cduce.org/ocaml`.
38. Overture (VDM++). `http://www.overturetool.org/`.
39. P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

40. Dorian Petit. *Génération automatique de composants logiciels sûr à partir de spécifications formelles B*. PhD thesis, Université de Valenciennes et du Hainaut Cambrésis, Décembre 2003. Numéro d'ordre 03-34.
41. Dorian Petit, Georges Mariano, Vincent Poirriez, and Jean-Louis Boulanger. Automatic Annotated Code Generation from B Formal Specifications. In G. Tarnai and E. Schnieder, editors, *Symposium on Formal Methods for Railway Operation and Control Systems*, pages 37–44. L'Harmattan, May 2003. ISBN 963 9457 45 0.
42. Dorian Petit, Vincent Poirriez, and Georges Mariano. The B method and the component-based approach. *Journal of Design & Process Science: Transactions of the SDPS*, 8(1):65–76, Mars 2004. ISSN 1092-0617.
43. Dorian Petit, Vincent Poirriez, and Georges Mariano. Reuse of ML module system for the B language. In *Forum on specification and Design Languages*, September 2004.
44. PhoX website. {http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI}.
45. Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the B method. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, volume 1393 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 46–65, Montpellier, April 1998. B1998, LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag.
46. Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the B method. In *B'98 : The 2nd International B Conference*, pages 46–65, 1998.
47. ProB. http://www.ecs.soton.ac.uk/~mal/systems/prob.html.
48. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *Proceedings of the 17t International conference on Conceptual Modelling, ER'98*, volume 1507 of *LNCS*. Springer-Verlag, 1998.
49. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *Proceedings UML 2000*, 2000.
50. Jérôme Rocheteau, Samuel Colin, Georges Mariano, and Vincent Poirriez. Évaluation de l'extensibilité de PhoX : B/PhoX un assistant de preuves pour B. In *Journées Francophones pour les Langages Applicatifs*, pages 139–153, January 2004.
51. Rodin-B#. http://rodin-b-sharp.sourceforge.net/.
52. Bruno Tatibouët, Antoine Requet, Jean-Christophe Voisinet, and Ahmed Hammad. Java card code generation from B specifications. In In J.S. Dong and Eds. J. Woodcock, editors, *ICFEM*, volume 2885, pages 306–318. Formal Methods and Software Engineering, Springer-Verlag, 2003.
53. H. S. Thompson, D. Beech, M. Maloney, and Mendelsohn Mendelsohn. "XML Schema Part 1: Structures". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-1/.
54. Geoffrey Norman Watson. A comparison of modularity in B and Cogito. In S. Reeves L. Groves, editor, *Formal Methods Pacific*, pages 263–286, 1997.
55. XML-RPC. Internet remote procedure call. *http://www.xmlrpc.com/spec*, 1999.