BRILLANT :
An Open Source and XML-based platform for Rigourous Software Development

Samuel Colin, Dorian Petit, Jérôme Rocheteau, Rafaël Marcano, Georges Mariano, Vincent Poirriez

Soumis le 01/04/2005
SEFM 2005 (Software Engineering and Formal Methods)
BRILLANT: An Open Source and XML-based platform for Rigourous Software Development

Samuel Colin*, Dorian Petit†, Jérome Rocheteau†, Rafaël Marcano†, Georges Mariano† and Vincent Poirriez*

LAMIH/ROI, UMR CNRS 5830
Le Mont Houy
F-59313 Valenciennes Cedex 9, France
Email: Firstname.Lastname@univ-valenciennes.fr

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

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 on 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, and can be used to develop safe systems, from their specifications to their validation and the generation of safe code.

Keywords: B method, tool support, UML modelling, XML, proof tools, code generation

I. INTRODUCTION

During the last decade of the previous millennium, theoretical research produced the B method, which is based on the same fundamentals as Z [1] and VDM [2].

This method reconciles the pragmatic constraints of the industrial development of critical software with the strict theoretical requirements that are inherent to mathematical formalism. The B method is one of the rare successful formal methods used in industry, and supports multiple paradigms: Substitutions as a means for describing dynamic behaviour naturally; Formulas in a simple yet efficient logical framework (set theory); Composition mechanisms that simplify development; Refinements providing a safe and efficient way to obtain secure computer code from abstract specifications.

The B method was used in the METEOR project [3], as well as in less well-known development projects [4], [5] and even non-critical development projects [6].

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) were available. Unlike most software tools, these resulted from prototypes developed by industry rather than by the academic community. In fact, from 1993 to 1999, the Atelier B development 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 Bkernel, that are not well documented and specified and that are not particularly well-suited to B’s high level of abstraction. However, these apparent paradoxes can be explained by the industrial origins of the method. Still, 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 present the BRILLANT[7] framework, showing the feasibility of a safe software development system that ranges from semi-formal specification (UML) to contract-equipped code generation. This framework provides a central core into which different components can be plugged, including a central component, a UML plug-in, a proof plug-out, and a code generator plug-out.

Section II presents the central component, which allows the components of a B project to be manipulated. The parsing of B machines is examined in section II-A.1, highlighting the central role of XML as an exchange format. Abstract syntax tree manipulations, such as specification flattening, are examined in section II-B, and proof obligation generation, in section II-C. Section III describes the UML plug-in used to translate UML projects into B so as to validate them, as described in Marcano & Levy [8] and Laleau & Polack [9]. Section IV presents the proof plug-out used to validate B proof obligations, and section V describes a code generator plug-out capable of embedding contracts into the code and supporting several target languages. An example of a railway system design project now being studied [10] is used to illustrate these components. Section VI presents our conclusion based on the results of our experiments with the implementation and use of the BRILLANT platform.

Ⅰ. INTRODUCTION

During the last decade of the previous millennium, theoretical research produced the B method, which is based on the same fundamentals as Z [1] and VDM [2].

This method reconciles the pragmatic constraints of the industrial development of critical software with the strict theoretical requirements that are inherent to mathematical formalism. The B method is one of the rare successful formal methods used in industry, and supports multiple paradigms: Substitutions as a means for describing dynamic behaviour naturally; Formulas in a simple yet efficient logical framework (set theory); Composition mechanisms that simplify development; Refinements providing a safe and efficient way to obtain secure computer code from abstract specifications.

The B method was used in the METEOR project [3], as well as in less well-known development projects [4], [5] and even non-critical development projects [6].

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) were available. Unlike most software tools, these resulted from prototypes developed by industry rather than by the academic community. In fact, from 1993 to 1999, the Atelier B development 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 Bkernel, that are not well documented and specified and that are not particularly well-suited to B’s high level of abstraction. However, these apparent paradoxes can be explained by the industrial origins of the method. Still, 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 present the BRILLANT[7] framework, showing the feasibility of a safe software development system that ranges from semi-formal specification (UML) to contract-equipped code generation. This framework provides a central core into which different components can be plugged, including a central component, a UML plug-in, a proof plug-out, and a code generator plug-out.

Section II presents the central component, which allows the components of a B project to be manipulated. The parsing of B machines is examined in section II-A.1, highlighting the central role of XML as an exchange format. Abstract syntax tree manipulations, such as specification flattening, are examined in section II-B, and proof obligation generation, in section II-C. Section III describes the UML plug-in used to translate UML projects into B so as to validate them, as described in Marcano & Levy [8] and Laleau & Polack [9]. Section IV presents the proof plug-out used to validate B proof obligations, and section V describes a code generator plug-out capable of embedding contracts into the code and supporting several target languages. An example of a railway system design project now being studied [10] is used to illustrate these components. Section VI presents our conclusion based on the results of our experiments with the implementation and use of the BRILLANT platform.
II. 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.

A. 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 II-A.1) that defines the B language and the abstract syntax (section II-A.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 [11].

1) A concrete grammar for B: Several B grammars have been introduced over the years, coming from a variety of sources: Cletsy (prev. Steria), which corresponds to the grammar used in AtelierB; B-Core, which corresponds to the grammar used in B-Toolkit; and Mariano’s PhD dissertation [12], 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 remarks made about 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 (AtelierB, B-Toolkit) mentioned above,
• they had to comply with the standard Lex and Yacc tools that allow LALR grammars to be defined, and
• they had to cause few conflicts as possible.

We chose OCaml as a supporting tool for our B developments for several reasons. First, it allows symbolic notations to be handled easily. In addition, it has an efficient implementation, and comes bundled with tools that allow the parsing of LALR grammars. Using this tool, we defined the B language in LALR. Some of the problems we encountered in doing so are described below.

Peculiarities in the parsing of B machines: Certain B language features made defining an adequate grammar for the B method difficult, including:

• The records, whose ; separator causes a conflict with the currently used separator for defining the sequence of two substitutions. We decided to give priority to the correct parsing of substitution sequences, to the detriment of a correct parsing of records.
• The definitions, designed to lighten repetitive, cumbersome notations, which cause a partial conflict with the LALR grammars. We decided to not expand the definitions and to keep them in the abstract syntax trees used in the original text, restricting the definitions to a subset that is expressive enough to allow them to be implemented with the LALR parser without losing too much of their usefulness.
• Several so-called reduce/reduce and shift/reduce conflicts, that represent ambiguous notations. We decided to remove the reduce/reduce conflicts, but to defer our decision about the shift/reduce conflicts.

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 \texttt{[substitution]predicate} and \texttt{[variable\_instanciation\_substitution]} constructions appear in this abstract syntax, which means that the structure can be manipulated to bring it closer to the matching mathematical definitions given in the B-Book [13].

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 due to the XSL style sheets that formulate simple recursive treatments of the XML structure, mostly transformations into other structured formats (LATEX, HTML, or PhoX, as mentioned in section II-C.3).

B. Abstract syntax tree manipulation

1) The flattening algorithm:

a) Overview: Flattening B specifications consist of eliminating the refinement and the composition links. The flattening algorithm uses one set of B components to build a single B component equivalent to the original set of B components. All the information for the different specifications are then grouped in one formal text.

This notion of flattening exists implicitly in the BBook [13]. Potet and Rouzaud used the term "flattening" in their work [14], but it was S. Behnia [13, PhD dissertation, in french] who specified the algorithm entirely, and it is this 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.

This notion of flattening exists implicitly in the BBook [13]. Potet and Rouzaud used the term "flattening" in their work [14], but it was S. Behnia [13, PhD dissertation, in french] who specified the algorithm entirely, and it is this 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.

b) The enrichment mechanism: Our flattening tool uses an enrichment mechanism that combines two specifications in one specification. This enrichment mechanism is described briefly below. (More details can be found in Petit [16]).
**Refinement:** In **refines** links, flattening consists of combining some clauses (**sets, constants properties**) and using the more concrete parts of the specifications for other clauses. For example, let us consider an abstract machine M and its refinement R. Let \( \text{VarM} \) (respectively \( \text{VarR} \)) be the variables of the machine (the refinement) and InvM (InvR), the invariants of this machine (this refinement). The variables of the flattened component are the variables of the refinement, but these variables are renamed if the same variable name exists in the more abstract component. In this case, a gluing invariant is added, and the new variable name is propagated. Let \( \text{VarR} \_l \) be this new variable clause, and InvR\_l, the invariant in which the variables are renamed. Thus, the invariant of the flattened component is \( \exists \text{VarM}.(\text{InvM} \land \text{InvR} \_l) \). Every specification property follows the same schema, in which the abstract variables are existentially quantified because they disappear in the flattened component.

**Importation:** In the **imports** links (as in the **includes** links), flattening consists of merging some clauses (**sets, constants properties, invariant, initialisation**), instantiating the parameters in the imported machine, and then expanding the operation of the machine called in the implementation phase.

**c) Implementation:** The flattening tool was the first tool implemented after designing the BCaml kernel (section II). One of the aims of this implementation was to "evaluate" the kernel’s usability and to add to the platform those tools/libraries that would be useful for manipulating B specifications.

First, the specification dependency graph had to be made manipulatable, since it is necessary to navigate through the specifications in order to build the successive flattened components. Therefore, we developed a library called BGraph that implements the dependency graph type and the functions needed to manipulate that graph.

Second, it was necessary to verify all the conditions that allow a set of B components to be flattened. The total implementation of the flattening tool (condition verification + algorithm implementation) requires about 3000 lines of OCaml code.

2) The B-HLL module system:

a) **Overview:** The Harper-Lillibridge-Leroy module system (HLL) presented in Leroy [17] formalises 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 was 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 or (sub-)modules) and functors (functions from modules to modules) in the obtained modular language.

b) **Instantiation:** A more complete description of our work on B-HLL can be found in one of our previous articles [18]. Our efforts to instantiate the HLL module system were divided into two parts. The first part involved defining the abstract language of the B-base language under study, based on the abstract syntax defined during the development of the BCaml Kernel. From this abstract syntax, we removed the part dedicated to the modularity language and then we developed a mapping function from the BCaml kernel abstract syntax to our new abstract syntax.

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 J.P. Bodeveix and M. Filali [11]. We added some type-checking rules to express the visibility rules described in the B-Book [13], 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.

To translate the visibility rules, we had to divide the classes of things that can be defined in a B specification into several sub-classes. For example, the substitution constructions were divided into B0 substitution and non-B0 substitution. This sub-division allowed us to express certain rules, such as "an abstract variable can not be used in a B0 substitution, but can be used in the others substitutions".

The B_to_BHLL tool that translates the specification from the kernel’s abstract syntax into our B-HLL syntax also generates four interfaces for each B component. Each of these interfaces is used to simulate the four composition links under study: **includes, imports, sees** and **uses**. By generating these interfaces, we can translate the visibility rules that make up the B module language.

C. Generating proof obligations

In this section, we first describe the method used to implement the actual calculus for the weakest precondition. Then we show how it can be used to generate a B project’s proof obligations. We then present the different options available for exporting these proof obligations to other formats and other tools.

1) **Generalised Substitution Language (GSL):** In order to generate proof obligations for B machines, we must be able to calculate the weakest preconditions of the substitutions. We chose to use the approach defined in the B-Book [13] by reducing B substitutions to their smallest syntactic and semantic set (i.e. generalised substitutions). In the following paragraphs, we will use GSL to designate both the syntactic set and the substitutions that occupy it. We define the GSL in BCaml as an abstract data type, as is described in the B-Book [13, B.3], with some notable exceptions:

- The affectation is defined as a multiple substitution; it serves as a basic construction once the parallel substitutions have been reduced.
- The repetition substitution "\"\" does not appear; we chose instead to use the while substitution, since it does not exist in the loop proof rules [13, E.7].
- The instantiation of a substitution variable \((\text{variable:=expression}\) substitution) is reduced before transforming the substitution.

With the help of the abstract data type, proof obligations can be generated according to the rules described in the B-Book [13, appendix E]. The corresponding BCaml code was
written with readability in mind, making it easily matched to the rule it is derived from.

2) Proof Obligation Generation: The main steps for generating proof obligations from a project, shown in figure 1, can be divided into precise steps which are described in more detail below:

![Diagram](image)

Fig. 1. From B machines to proof obligations

Thus, the generation of a machine’s proof obligations is divided into precise steps we describe more in detail:

**Parsing:** First, the machine and all the machines it depends on are parsed. This parsing phase is followed by a scoping phase in which all unique identifiers that represent the same variable name, machine name or operation name are made equal. In fact, prior to the parsing phase, each identifier is associated with a unique stamp; however, when the parsing is finished, all the identifiers have different stamps. The scoping phase acts to make the stamps for those identifiers representing the same variable, machine or operation name equal with respect to visibility.

**Generation of formulas:** This step is based on the B-Book [13, appendix F], resulting in proof obligations with the following shape:

\[
\text{Instanciation} \Rightarrow \text{Hypothesis} \Rightarrow \text{Substitution} \Rightarrow \text{Goal}.
\]

This generation method allows more handling flexibility later on, for instance during debugging, or when showing students how proof obligations are generated, or when the proof tool applies the substitution to the goal. Bodeveix [19] shows for instance how substitutions can be defined in Coq and PVS. Figure 2 shows an example of such an uncalculated proof obligation, derived from the B project presented in section III.

**Optimisations:** Several additional optimisations, or treatments, 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:

\[
(H \Rightarrow G_1 \land \ldots \land G_n) \Rightarrow (H \Rightarrow G_1), \ldots, (H \Rightarrow G_n)
\]

Other possible, but not implemented, optimisations, include removing formulas when the goal is trivially true or appears in the hypotheses, or changing the shape 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.

**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 it is, and the machine from which it is generated. 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 containing all the free variables of the formula because some theorem provers requires all variables be bound. This tag helps the stylesheet generate a file for such a theorem prover more easily.

3) Exporting to other tools: Once the proof obligations in the XML format are available, the XSL style sheets allow them to be exported to other tools. For instance, the proof obligations can be transformed into BPhoX files (figure 2 is an example of the results obtained), into text files, into HTML files which improve the readability of the formulas, or into BPhox files which allow the proof obligations to be verified.

Figure 8 in section IV-B presents an example of an XSL stylesheet application for the proof obligation shown in figure 2. First, the header of the file is inserted, which provokes the loading of the appropriate PhoX library and finetunes the power of the proper (the flag commands). Then, the free variables of the formula (the identifiers after the \quantifier) are quantified. The formula itself is inserted in the BPhoX syntax, by replacing the conjunctions of the hypotheses by implications, in order to facilitate the work of PhoX. Finally, the command that is given to the prover is added to start the proof (Try intros; auto.). In the next step, the theorem prover is fed the generated proof obligations file (see section IV).

All of these steps (including the replacing of the conjunctions in the hypotheses with implications) are done by the XSL stylesheet, demonstrating the ad hoc quality of this technology designed for simple treatments involving recursivity.

Now that we have presented the BCaml core, we can present the following:

\[
(LCC \in \mathbb{P}_I (I \mathbb{N})) \\
\land \text{STATE} \in \mathbb{P}_I (I \mathbb{N}) \\
\land \text{STATE} \in \{\text{Deactivated} , \text{ShowingYlight}, \text{ShowingRlight}, \text{ClosingB}, \text{OpeningB}, \text{ClosedB}, \text{Failure}\} \\
\land \ldots \\
\land \text{Yellow} \cdot \text{IState(yellowLight(obj))} = \text{On} \Rightarrow [\text{state(obj)} := \text{ShowingRlight} \\
|| \text{Yellow} \cdot \text{switchOff(yellowLight(obj))} \\
|| \text{Red} \cdot \text{switchOn(redLight(obj))} ] \\
( lcc \subseteq \text{LCC} \\
\land \text{lcc} \cdot \text{barrier} \in \text{lcc} \Rightarrow \text{barrier} \\
\land \ldots \\
\land \text{\forall obj} \cdot (\text{obj} \in \text{lcc} \\
\land \text{bStatus(lcc} \cdot \text{sensor(obj)))} = \text{Opened} \\
\land \text{bState(lcc} \cdot \text{barrier(obj))} = \text{Closed} \Rightarrow \text{mode(obj)} = \text{Unsafe} )
\]

Fig. 2. Uncalculated proof obligation for the timeOut_\_showRlight operation
the different plugins/plugouts revolving around it, starting with a translator from UML/OCL specifications to B.

III. FROM UML/OCL MODELS TO B SPECIFICATIONS

The different plugins/plugouts mentioned in the introduction revolve around the BCaml core described above, starting with a translator that changes UML/OCL specifications to B specifications. Our work continues the work begun by Marcano & Levy [20] on combining UML and B for consistency checking, while also taking OCL annotations into account [21]. Adding OCL constraints is a useful way to capture the key safety properties of the system being constructed. The main purpose of our work is to facilitate the construction of a B formal specification, using automated tool support. Our process, which is shown in figure 3, breaks down into the three steps described below:

1) From UML to XML. The Poseidon [22] modelling tool was chosen for drawing the UML model and generating its associated XMI file (model.xmi). A transformation file (xmi2xml.ml) is written in the XSLT language to translate the XMI file into a XML file (model.xml) that represents the original UML file (hence, the name xmi2xml rather than xmi2xml.ml).

2) From XML to UML-parsed models. The IOXML processor parses XML models elements of the XML file into OCaml-compliant data types according to the UML abstract syntax tree definition (uml.ml). Therefore the resulting file (model.ml) can be used to generate the B specification.

3) From UML models to B specifications. The uml2b module translates UML classes, state diagrams and OCL constraints into B specifications. The translation rules are implemented in OCaml as mappings of the UML abstract syntax (uml.ml) into the B abstract syntax (b.ml).

We chose to connect the tool directly to the abstract syntax tree of BCaml rather than producing B concrete specifications in order to obtain a smoother integration for both tools. Though the same programming language is employed (OCaml), it is still possible to produce B concrete specifications by using the XML output plus a stylesheet to generate B ASCII files.

A. UML-based modelling

In this section, the construction of a B specification from a UML/OCL model is illustrated using the example of a radio-based Railway Level Crossing (RLC) [23]. (A complete description of the traffic control system considered here can be found in Jansen & Schnieder [24]).

1) Class and state diagrams: The following objects interact with the Level Crossing Control system (LCC), as shown in figure 4: the lights, the barriers, the vehicle sensors, the train-borne control system and the operations centre. Only one side of the railway line in the level crossing is considered here in order to make the system specifications more readable. 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, and then LCC performs a series of actions with specific timing constraints. First, the traffic lights are switched to the yellow light, and after 3 seconds, they are switched to red. Nine seconds later, the barriers begin to drop. The LCC system signals the safe state of the level crossing if the barriers reach their lowest level within a maximum of 6 seconds, thus allowing the train to pass the level crossing.

In the activated mode, the LCC system may be in one of the following substates (figure 5): showing the yellow light; closing the barrier; maintaining the barrier closed; or opening the barrier. The time expirations following the LCC’s activation are denoted by the following events: timeOut_1 (3 seconds later), timeOut_2 (9 seconds after timeOut_1) and timeOut_3 (6 seconds after timeOut_2).

2) adding OCL constraints: Safety properties are included in the invariants of the system, in order to insure that they are not lost between the abstract specification and the implementation phases. In the case study, the notion of “train passing the crossing area” is connected to the activation of the railway level crossing. To accomplish this activation task, the front end of the train and the rear end of the train must somehow be detected. We assume that the train can be detected directly using abstract vehicle sensors. Similarly, the barrier state is also detected by sensor, this time a barrier sensor. The following is an example of OCL constraints on the LCC system class (figure 4):

- 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:
B. Generating the B specification

The B specification resulting from the steps described above is composed of abstract machines representing each class. A root abstract machine specifies the whole system’s structure and introduces all the associations between classes.

1) Formalisation of class and state diagrams: An abstract machine formalising a class describes the deferred set of all the possible instances of the class (i.e. BARRIER), as well as the subset of its existing instances (i.e. barrier). Each attribute is formalised by a variable defined as a total function between the set of instances and the attribute type.

Since associations between classes represent couples of instances, they are expressed in B as binary relations between the existing class instances (figure 6).Associations can be expressed more precisely using the values for the multiple roles, constraining the binary relation (→) as a function (→), partial function (→), injection (→) or bijection (→) with additional properties on its domain or range.

Each transition between two states is formalised by a B operation, whose name is the name of the incoming event concatenated with the name of the action. 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 (figure 5). The transition from the showingRlight state to the closingB state activated by the event timeOut is formalised as shown in figure 6. Note that some of the information obtained from the OCL definition of the operation closeBarrier is included here, since this operation is activated by the event timeOut.

When the same event could activate two different transitions depending on the guard condition, then both transitions are obtained from the OCL definition of the operation closeBarrier as shown in figure 6. Note that some of the information is included here, since this operation is activated by the event timeOut.

The transition from the showingRlight state to the closingB state activated by the event timeOut is formalised as shown in figure 6. Note that some of the information obtained from the OCL definition of the operation closeBarrier is included here, since this operation is activated by the event timeOut. (The OCL translation is described below.) When the same event could activate two different transitions depending on the guard condition, then both transitions are formalised by the same operation of the B machine. The SELECT close is used to describe each transition, as illustrated in figure 6 for the formalisation of the event timeOut.

Fig. 4. Class diagram of the RLC system

Fig. 6. Formalisation of state diagrams
2) Formalisation of OCL constraints: Two types of OCL constraints are taken into account in our method. The first type of constraint specifies a class invariant. The second type of constraint specifies a precondition and/or a postcondition for an operation. In the first case, the translation of the OCL constraint consists of combining in a conjunction a new predicate with the invariant of the related B machine, whereas in the second case, it consists of completing an operation of that machine. The formalisation of the OCL invariant of the LCC system is shown in figure 7.

![Fig. 7. Formalisation of the OCL invariants](image)

OCL pre- and postconditions are used to complete the B machine operations. In figure 6, the precondition of the operation timeOut_1 not only requires the LCC system to be in the yellowLight state (which is generated from the state diagram), but also requires that the red light be switched on and the barrier be closed, producing the following translation of the OCL predicate:

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

The postcondition of the operation initially includes only the substitution "state(obj) := ClosingB", thus setting the new state of the LCC instance (obj). It is completed by translating the OCL postcondition (which generates the parallel substitutions in figure 6) into B:

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

IV. FROM B PROOF OBLIGATIONS TO CORRECTNESS

The BCaml Kernel provides the first two important types of B tools, presented in Abrial’s B# [25, section 4]. The first includes the lexer, parser and typer and 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 within BCaml for a pragmatic reason: building a B prover takes much more time than developing dedicated libraries in an already existing prover for B according to our specifications. Instead, we built an add-on that can be replaced. We included the PhoX proof checker [26] because it can be work closely with us; and its highly intuitive syntax minimises the libraries’ development time.

Our contributions to a PhoX-based B prover include a process killer used to control the proof time, a B extension of PhoX, the translation from B to the PhoX extension and the B/PhoX GNU Make script that binds those tools together.

A. The bphox GNU Make script

A B prover must verify whether each proof obligation is a theorem or not. In the BCaml context, every B/XML proof obligation has to be translated into the PhoX syntax and has to be proved. PhoX produces a po.pho file from a po.phx translated proof obligation when the proof is successful. The B/PhoX proof session that follows involves a two-step transformation, depending on the file extension. This process corresponds exactly to the GNU Make transformation using suffix schemata, and can be copied and configured to link BCaml with other theorem provers. The principal property that must be preserved throughout this process is that every sentence is a B theorem, if and only if its translation is a B/PhoX theorem, which has been proven by Rocheteau & al.[27].

B. The bgop2phox XSL style sheet

The translation step consists of applying our XSL bgop2phox style sheet to the B/XML proof obligations using a XSLT processor. The XSL transformation schema allows recursive mappings. Our translation is also recursively defined. A first order language à la B is composed of different symbols for functions, relations, connectors and quantifiers. Figure 8 in section II-C.3 shows a PhoX proof obligation generated from the proof obligation shown in figure 2, after it has been calculated and saved into an XML file.

```
add_path "/usr/share/brillant/bphox/".
Import BLib.
flag auto_lvl 2.
flag auto_type true.
theor 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,ShowingRight) app (obj))
      in Activated)) &
      ((bStatus app ((lcc_sensor app (obj)))) = Opened))
   ) ->
   ((mode app (obj)) = Unsafe))) })
   }
Try intros ;; auto.
save.
```

Fig. 8. One of the exploded, converted to B/PhoX, proof obligations for timeOut_1_showRlight
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 $S^*$, such that its extension to the first-order terms formulae is merely defined by an inductive commutation. In this way, our translation is sound using the PhoX system of simple types. Moreover, every non-freeness rule and every substitution rule is easily obtained by the $\lambda$ binder properties.

C. The bli2 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 & al.[27]). It contains successive libraries for predicate calculus with equality, the boolean domain, cartesian products, set operators, binary relations, functions, arithmetic theory and finite sequencing.

D. The chronos process killer

The proof step consists of producing a po.pho file from a po.phx one. The existence of the po.pho file means that the required obligation holds. The absence of this file can mean that the proof obligation does not hold. It can also mean that the proof can not be completed due to lack of time or space, causing the proof session to loop endlessly. In order to deal with this problem, every PhoX call is controlled by a process killer named chronos.

The behaviour of the chronos produces the following proof obligation classification: those with a successful proof, those with a failed proof and those with a killed proof. The following two tables give the results for two different time-out proof sessions for the famous boiler B project.

<table>
<thead>
<tr>
<th>Time-out</th>
<th>1 s.</th>
<th>5 s.</th>
<th>60 s.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated proof obligations</td>
<td>2295</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successful</td>
<td>1823</td>
<td>1955</td>
<td>1971</td>
</tr>
<tr>
<td>Failed</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Killed</td>
<td>462</td>
<td>340</td>
<td>324</td>
</tr>
<tr>
<td>Proof Rate</td>
<td>79%</td>
<td>85%</td>
<td>85%</td>
</tr>
</tbody>
</table>

The successful proof obligation set is built using a fixed-point application. Assuming an increasing function $f$ on natural numbers, which means that the $n + 1$th time-out is greater than the $n$th one, the first session runs over the whole set of generated proof obligations and the $n + 1$th session runs only over the killed proof obligations of the $n$th session. However, since using PhoX as a “black box” does not allow us to save the proof state at its kill moment, the next session replays the unkilled proof obligations from the beginning.

V. FROM B SPECIFICATIONS TO CODE

The code generation process is summarised in Figures 9 and 10. The first figure illustrates the generation process for producing flat code, and the second figure illustrates the process for producing a component oriented code (more details on our approach to generate code can be found in [28] and in [29]).

To generate flat code, the specifications have to be parsed and annotated with types, and then flattened. From the flat B specification, the code can be generated simply by using a XSLT processor and the appropriate style-sheet. To generate component-oriented code, the specifications must be parsed, and then translated into B-HLL specifications, which are then annotated with types. Then, the part of the flattening algorithm dedicated to eliminating refinement links is run. A style sheet is applied to the B-HLL components thus obtained in order to generate the code.

Figure 11 presents a B specification of a bounded stack. The code presented in figures 12 and 13 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 us putting the properties that are expressed in the specifications into the code. (Please note that the code generation step did not use the example introduced in section III, because there is currently no refinement or implementation for this example).
VI. DISCUSSION, CONCLUSION AND PERSPECTIVES

A. Conclusion

BRILLANT is advantageous in that it can be used to test and/or validate B-related experiments, and in fact, we have been the first users of many of the prototypes presently available for the platform (bparser, bgop, btyper, bphox,...).

The BRILLANT platform design has two principal orientations: the use of open and standardised formats and the open availability of the source codes for the tools (OCaml and/or Java so far). We have been working to finetune the open availability of the source codes for the tools (OCaml, ...). We have been happy to provide our assistance to those who would like to try to use the tools within the context of their own research.

The information presented in this article would appear to demonstrate that we have reached our goal: open and standardised formats (XML) have been used throughout the whole platform, and this platform has become the testbed for several other fundamental research projects (UML/OCL/B coupling in section III, proof in section IV, code generation in section V).

B. Perspectives

The next evolutions of BRILLANT will be based on integrating technologies that endorse the use of open formats. The following evolutions are planned: the use of XML schemas [36] instead of DTDs for the validation of XML files; increased traceability between UML models, B machines, proof obligations and other derived models (generated code, test cases,...) thanks to the flexibility of XML; the representation of B models as projects databases using XPath [37] and XML-Query [38]; a distributed platform architecture using XML-RPC [39], that will allow the parser and prover to be represented as servers to which B projects can be sent for parsing or validation. Lastly, an ergonomic interaction mode for the different platform tools will be defined, 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 [20], temporal extensions for B [40], and safe software components generation [29].

Much work remains to be done, and the platform developers are happy to provide their assistance to those who would like to try to use the tools within the context of their own research. All the necessary resources for building BRILLANT tools are available on a web site dedicated to collaborative free software development [7].
REFERENCES


