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

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

LAMIH/ROI, UMR CNRS 8530
Université de Valenciennes et du Hainaut Cambrésis
* 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*

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 [8] and VDM [29].

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 [4], as well as in less well-known development projects [10, 13] and even non-critical development projects [40].

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, 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 or 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* [9] 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 2 presents the central component, which allows the components of a B project to be manipulated. The parsing of B machines is examined in section 2.1.1, highlighting the central role of XML as an exchange format. Abstract syntax tree manipulations, such as specification flattening, are examined in section 2.2, and proof obligation generation, in section 2.3 . Section 3 describes the UML plug-in used to translate UML projects into B so as to validate them, as described in Marcano & Levy [24] and Laleau & Polack [20]. Section 4 presents the proof plug-out used to validate B proof obligations, and section 5 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 [25] is used to illustrate these components. Section 6 presents our conclusion based on the results of our experiments with the implementation and use of the *BRILLANT* platform.

2. 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.

2.1. 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 2.1.1) that defines the B language and the abstract syntax (section 2.1.2) 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 [6].

2.1.1. 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 *AtelierB*; *B-Core*, which corresponds to the grammar used in *B-Toolkit*; and Mariano's PhD dissertation [26], 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 as 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.

2.1.2. 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, which means that the structure can be manipulated to bring it closer to the matching mathematical definitions given in the B-Book [1].

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 2.3.3).

2.2. Abstract syntax tree manipulation

2.2.1. The flattening algorithm:

Overview : Flattening B specifications consists 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 [1]. Potet and Rouzaud used the term "flattening" in their work [36], but it was S. Behnia [5, 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.

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 [30])

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 *VarM* (respectively *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 *VarR_1* be this new variable clause, and *InvR_1*, the invariant in which the variables are renamed. Thus, the invariant of the flattened component is $\boxed{\exists VarM.(InvM \wedge InvR_1)}$. 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.

Implementation: The flattening tool was the first tool implemented after designing the BCaml kernel (section 2). 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.

2.2.2. The B-HLL module system:

Overview: The Harper-Lillibridge-Leroy module system (HLL) presented in Leroy [21] 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.

Instantiation: A more complete description of our work on B-HLL can be found in one of our previous articles [33]. 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 languageand 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 [6] . We added some type-checking rules to express the visibility rules described in the B-Book [1], 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 other 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.

2.3. 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.

2.3.1. 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 [1] 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 [1, B.3], with some notable exceptions:

• The assignment is defined as a multiple substitution; it serves as a basic construct 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 [1, E.7].

• The instantiation of a substitution variable ([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 [1, appendix E]. The corresponding BCaml code was written with readability in mind, making it easily matched to the rule it is derived from.

2.3.2. Proof Obligation Generation: The main steps for generating proof obligations from a project can be divided into precise steps which are described in more detail below:

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 [1, appendix F] , resulting in proof obligations with the following shape:

$[Instanciation]Hypothesis \Rightarrow [Substitution]Goal.$

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 [7]shows for instance how substitutions can be defined in Coq and PVS. Figure 1 shows an example of such an uncalculated proof obligation, derived from the B project presented in section 3.

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:

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

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.

```
(LCC ∈ ℙ1(ℕINT)
    ∧ STATE ∈ ℙ1(ℕINT)
    ∧ 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) )
```

Figure 1. Uncalculated proof obligation for the timeOut_1_showRlight operation

2.3.3. 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 LaTeX files (figure 1 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 5 in section 4.2 presents an example of an XSL stylesheet application for the proof obligation shown in figure 1. First, the header of the file is inserted, which provokes the loading of the appropriate PhoX library and finetunes the power of the prover (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 4).

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 recur-

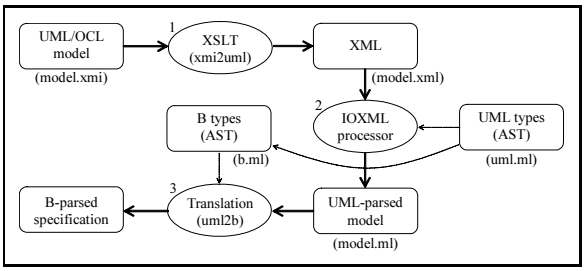

Figure 2. The proposed process

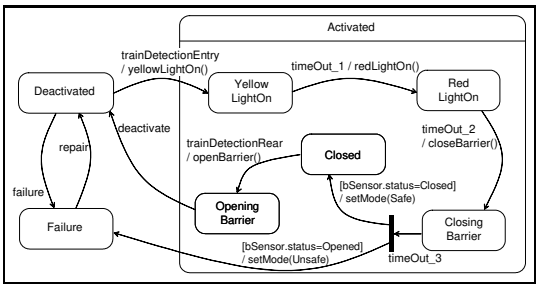

Figure 3. State diagram of the LCC system

sivity.

Now that we have presented the BCaml core, we can present the different plugins/plugouts revolving around it, starting with a translator from UML/OCL specifications to B.

3. 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 [24] on combining UML and B for consistency checking, while also taking OCL annotations into account [27]. 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 2, breaks down into the three steps described below:

From UML to XML: The Poseidon [35] modelling tool was chosen for drawing the UML model and generating its associated XMI file (model.xmi). A transformation file (xmi2uml) 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 xmi2uml rather than xmi2xml).

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

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.

3.1. 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) [16]. (A complete description of the traffic control system considered here can be found in Jansen & Schnieder [18]).

3.1.1. Class and state diagrams: The Level Crossing Control system (LCC) controls the traffic lights and barriers of a level crossing. It interacts with the vehicle sensors, the train-borne control system and the operations centre. When the system is activated at the approach of a train, it must perform a series of actions, as illustrated by the state diagram in figure 3. Several actions have specific timing constraints: for instance, switching from yellow lights to red lights shall happen after 3 seconds. In figure 3, time expirations following the LCC's activation are denoted by the events prefixed `timeOut`. A full class diagram of this system can be found in [25].

3.1.2. adding OCL constraints: Without going into detail, the OCL constraints helps specifying safety properties and ensuring these properties are not lost between the abstract specification and the implementation phases. These constraints are naturally found later in the invariant of the generated B machines. Additional OCL constraints also help adding supplementary information that can not be found in the state diagrams. For instance, the closeBarrier operation, raised by the event timeOut_2, 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
```

3.2. 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.

3.2.1. 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

```
MACHINE
   LCC_System
INCLUDES
   Barrier, BarrierSensor, Yellow.Light,
   Red.light, TrainborneCS
VARIABLES
   barrier, bState
INVARIANT
   lcc_barrier ∈ lcc →barrier ∧...

OPERATIONS

timeOut_1_redLightOn(obj) =
      PRE
         obj ∈ lcc ∧
          state(obj) =ShowingYlight ∧
          bStatus(lcc_sensor(obj)) =Opened ∧
          bState(lcc_barrier(obj)) =Opened ∧
          Red.lState(redLight(obj)) =Off ∧
          Yellow.lState(yellowLight(obj)) =On
      THEN
          state(obj) :=ShowingRlight
       || Yellow.switchOff(yellowLight(obj))
       || Red.switchOn(redLight(obj))
      END;
```

Figure 4. Formalisation of state diagrams

attribute is formalised by a variable defined as a total function between the set of instances and the attribute type. Associations between classes are expressed in B as binary relations between the existing class instances (figure 4). These relations can be expressed precisely by using the wide spectrum of relation definitions in B, and by stating additional properties on their domains or ranges.

Each transition is formalised by a B operation whose name is the name of the incoming event concatenated with the name of the action. The precondition of the operation is deduced from the transition guard and the substitution describes the transition to the new state.

3.2.2. Formalisation of OCL constraints: As depicted in section 3.1.2, two kinds of OCL constraints make their way into the resulting B project :

• The OCL constraints that specify class invariants are combined with the invariant of the related B machines

• The OCL constraints that complete the information of state diagrams are translated into B preconditions, for the precondition part (see section 3.1.2) and B substitutions, such as the predicate statement, for the postcondition part. For instance, the call to red and yellow light operations from the `timeOut_1_redLightOn` operation has been obtained by translation of the following OCL constraint's postcondition:

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

The closing of the barrier does not appear in this operation, but in another `timeOut` operation in order to conform with the translation of the state diagram. In the next section, we introduce B/PhoX, a proof plugout for the *BRILLANT* platform.

4. From B proof obligations to correctness

The BCaml Kernel provides the first two important types of B tools, presented in Abrial's *B#* [2, 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 [34] because it can be extended to the B mathematical foundations; its GPL licence permits distribution with BCaml; its developers were willing to 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, the B extension of PhoX, the translation from B to the PhoX extension and the B/PhoX GNU Make script that binds those tools together.

4.1. 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.[38].

4.2. 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 5 in section 2.3.3 shows a PhoX proof obligation generated from the proof obligation shown in figure 1, after it has been calculated and saved into 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 merely defined by an inductive commutation. In this way, our translation is sound using the PhoX system of simple types. Moreover, every non-freeness

```
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.
```

Figure 5. One of the exploded, converted to B/Phox, proof obligations for timeOut_1_showRlight

rule and every substitution rule is easily obtained by the λ binder properties.

4.3. 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 & al.[38]). It contains successive libraries for predicate calculus with equality, the boolean domain, cartesian products, set operators, binary relations, functions, arithmetic theory and finite sequencing.

4.4. 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 table gives the results for several different time-out proof sessions for the famous boiler B project.

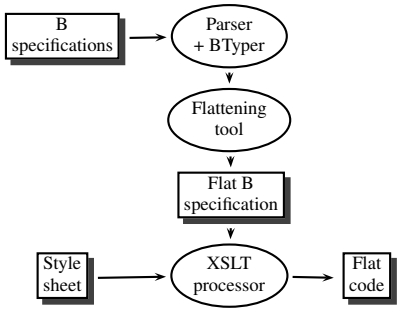

Figure 6. The code generation process to obtain flat code

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

The successful proof obligation set is built using a fixed-point application. Assuming an increasing function f on natural numbers, which means that the $n+1^{\text{th}}$ time-out is greater than the n^{th} one, the first session runs over the whole set of generated proof obligations and the $n+1^{\text{th}}$ 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.

5. From B specifications to code

The code generation process is summarised in Figures 6 and 7. The first figure illustrates the generation process for producing flat code, and the second figure illustrates the process for producing component-oriented code (more details on our approach to generate code can be found in [31] and in [32]).

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 8 presents a B specification of a bounded stack. The code presented in figure 9 is generated from this specification. The package specifications use the generic Ada construction to translate the parameters that specify the size of

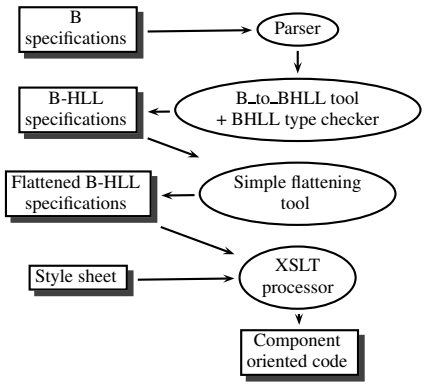

Figure 7. The code generation process to obtain component oriented code

```
MACHINE
  stack(stack_size)
CONSTRAINTS
  stack_size : ℕ∧stack_size ≥1
∧ stack_size ≤MAXINT
VISIBLE_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
```

Figure 8. A B specification of a bounded stack

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 3, because there is currently no refinement or implementation for this example).

6. Discussion, conclusion and perspectives

6.1. Comparison with other works

Several other formal methods also benefit from the same kind of tools, developed with a similar design: using open-source high-level languages and formats, or both. Many of them can be found at [15], along with links to the formal methods they implement. In addition, freely available (but not open-source) tools exist for the B method as well: B4Free[3] (distributed by Clearsy) and ProB [37]. The latter is an animator and model-checker programmed in Prolog, and uses the XML files produced by the jBTools [19] as an input.

We can also compare *BRILLANT* with projects of similar nature, ambitions and/or design: *Rodin* [39] (for B#), *Overture* [28] (for VDM++) and the Comprehensive Z Tools [12](*CZT*). These and the *BRILLANT* project share several common points: they use an XML-based interchange format, and are driven by research needs (*BRILLANT*, *CZT*), industrial interests (*Rodin*) or both (*Overture*). Moreover, all these projects have a similar architecture, where the core tools (parsing, typechecking, testing/validating plus other possible plugins/plugouts) are clearly separated. The difference lies mainly in the underlying implementation and its facilities: on the one hand, *Rodin* and *Overture* are based on the Eclipse IDE [14] and thus provide a very consistent development environment. Writing mandatory parts of such a framework (parsing, compiling, graphical interface, test suites,...) is therefore made, if not easier, at least more scalable and reusable. On the other hand, *CZT* and *BRILLANT* are rather viewed as a more or less loosely connected set of tools: the *CZT* are developed with Java (edition of Z specifications is even supported by jEdit), and most tools of *BRILLANT* are developed in OCaml. We can therefore tighten the comparison between *CZT* and *BRILLANT*:

• The most proponent tool of *BRILLANT*, BCaml, uses an immutable type (using the terminology of the *CZT* documentation [12]), thus avoiding the problems described for the development of *CZT*

• The future addition of B-HLL to BCaml required the unicity of identifiers. The chosen approach for BCaml was to make a scoping phase follow the parsing phase in order to associate each identifier with a unique identity. Later on, the treatments involving fresh variables is made easier thanks not only to that unicity, but also to the recursive nature of syntax trees, which then allows the retrieving of all free variables. Indeed, several libraries providing functions for high-level data structures (lists, sets, hash tables,...) exist

• As in *CZT*, the tools of BCaml can use the abstract syntax tree directly instead of the XML exchange files in order to speed up processing.

We can also note that the same kind of problems are described in the publications describing *Overture* [28], although the problem here was more linked to the chosen compiler generator than to the compilation technique. As a conclusion for this quick comparison, we can also note several points:

• Our choice of a programming language (OCaml) proved to be worthy: indeed, the most part of the development was (and still is) made by intern students (thus not full-time engineers) and PhD students

```
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;
```

Figure 9. The bounded stack Ada package specification and body

• Following the implemented formalism *by-the-book* made the integration of extensions (such as the B-HLL module system) easier
• *BRILLANT* has the anteriority, although the idea of the *CZT* seem to be contemporary
• The fact that, at the time of writing, the *Rodin* project is still at the development stage should encourage the developers to look at the other projects (presented here, for instance) to benefit from their experience

6.2. 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 platform to help it meet the needs of other theoretical research projects, including but not limited to extensions of the B language, improvements in the current tools, couplings with other provers (such as Coq, Harvey) and other validation formalisms (e.g. model-checking).

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 3, proof in section 4, code generation in section 5). As an example, all the source code examples (B machines, logical formulas, XML, PhoX) either come from the *BRILLANT* platform, or derive from its use. For instance, the B machines here are LaTeX files using the style of *BRILLANT* for B machines, the LaTeX files being obtained by the application of an XSL stylesheet to XML abstract machines, themselves obtained from the parsing of ASCII B machines.

6.3. 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 [41] 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 and XML-Query [42]; a distributed platform architecture using XML-RPC [43], 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 [24], temporal extensions for B [11], and safe software components generation [32]. 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 [9].

References

[1] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[2] J.-R. Abrial. B# : Toward a Synthesis between Z and B. In *ZB'2003 - Formal Specification and Development in Z and B*, pages 168–177, 2003.

[3] B4Free. `http://www.b4free.com/`.

[4] P. Behm, P. Benoit, A. Faivre, and J.-M. 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.

[5] S. Behnia. *Test de modèles formels en B : cadre théorique et critères de couvertures*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Oct. 2000.

[6] J.-P. Bodeveix and M. Filali. Type synthesis in B and the translation of B to PVS. In *ZB'2002 – Formal Specification and Development in Z and B* [22], pages 350–369.

[7] J.-P. Bodeveix, M. Filali, and C. Munoz. Formalisation de la méthode B en COQ et PVS. In *AFADL'2000* [23], pages 96–110.

[8] S. Brien and J. Nicholls. Z base standard: Version 1.0. Technical Monograph PRG-107, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, UK, November 1992.

[9] BRILLANT. `http://gna.org/projects/brillant`.

[10] 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.

[11] S. Colin, G. Mariano, and V. Poirriez. Duration calculus: A real-time semantic for B. In *First International Colloquium on Theoretical Aspects of Computing*. UNU-IIST, september 2004. Guiyang, China.

[12] Comprehensive Z Tools. `http://czt.sourceforge.net/`.

[13] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In Hinchey and Bowen [17], pages 227–252.

[14] Eclipse. `http://www.eclipse.org/`.

[15] The www formal methods' virtual library. `http://vl.fmnet.info/`.

[16] B. L. f. FunkFahrBetrieb. Stand 1.10.1996, 1996.

[17] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Series in Computer Science. Prentice Hall International, 1995.

[18] L. Jansen and E. Schnieder. Traffic control system case study: Problem description and a note on domain-based software specification. technical report, 2000.

[19] jBTools. `http://lifc.univ-fcomte.fr/~tatibouet/JBTOOLS/`.

[20] R. Laleau and F. 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* [22], pages 517–534.

[21] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[22] LSR-IMAG. *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, Grenoble, France, Jan. 2002.

[23] LSR/IMAG. *Approches Formelles dans l'Assistance au Développement de Logiciels*, LSR/IMAG – BP 72 38402 Saint-Martin d'Heres Cedex – Grenoble – France, Jan. 2000. LSR/IMAG.

[24] R. Marcano and N. 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.

[25] R. Marcano, G. Mariano, and P. Bon. UML modelling as the basis for formal analysis of railway traffic control systems. In *Formal Methods for Automation and Safety in Railway and Automotive Systems FORMS'2004*, page to be published, Braunschweig, Dec. 2004. Technische Universitat Braunschweig.

[26] G. 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.

[27] O. M. G. OMG. Object constraint language, version 2.0. final adopted specification, omg document ptc/2003-10-14, October 2003.

[28] Overture (VDM++). `http://www.overturetool.org/`.

[29] 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.

[30] D. Petit, G. Mariano, and V. Poirriez. Flattening B Components for Code Generation. Technical Report INRETS/RT-01-716-FR, INRETS, July 2001. http://www.univ-valenciennes.fr/LAMIH/ROI/dpetit/Biblio/Pub/RR/Flattening/RT-01-716-FR.ps.gz.

[31] D. Petit, G. Mariano, V. Poirriez, and J.-L. 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.

[32] D. Petit, V. Poirriez, and G. 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.

[33] D. Petit, V. Poirriez, and G. Mariano. Reuse of ML module system for the B language. In *Forum on specification and Design Languages*, September 2004.

[34] PhoX website. {`http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI`}.

[35] Poseidon UML modelling tool website. `http://www.gentleware.com`.

[36] M.-L. Potet and Y. Rouzaud. Composition and refinement in the B method. In *B'98 : The 2nd International B Conference*, pages 46–65, 1998.

[37] ProB. `http://www.ecs.soton.ac.uk/~mal/systems/prob.html`.

[38] J. Rocheteau, S. Colin, G. Mariano, and V. 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, Jan. 2004.

[39] Rodin-B#. `http://rodin-b-sharp.sourceforge.net/`.

[40] B. Tatibouët, A. Requet, J.-C. Voisinet, and A. Hammad. Java card code generation from B specifications. In I. J. Dong and E. J. Woodcock, editors, *ICFEM*, volume 2885, pages 306–318. Formal Methods and Software Engineering, Springer-Verlag, 2003.

[41] H. S. Thompson, D. Beech, M. Maloney, and M. Mendelsohn. "XML Schema Part 1: Structures". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-1/.

[42] World Wide Web Consortium `http://www.w3.org`.

[43] XML-RPC. Internet remote procedure call. *http://www.xmlrpc.com/spec*, 1999.