Definition of a Mathematical Language Together with its Proof System in Event-B

Jean-Raymond Abrial

Marseille
jrabrial@neuf.fr

1 Introduction

Our application domain with the B Method and Event-B is the modeling and development of industrial (embedded) systems. We have been working in this area for more than 20 years. More is described in [1] where the evolution from Z to B and Event-B is described with full details.The starting application of this approach was the driverless metro system for the metro line 14 in Paris: the B Method was used in the development of the safety critical part of the corresponding controller. In the B Method and in Event-B, the *prover part* is *absolutely central* together with the notion of *refinement*. In fact, at the beginning in the nineties, the Atelier B tool was developed together with the line 14 system itself. Later, Event-B [2] and the corresponding Rodin Platform [3] was developed with some heavy fundings from the European Commission.

In order to explain how all this emerged technically, this paper first contains an important definition of the *Mathematical Language* we used in Event-B. This done in section 2. The reason for this important section is that we want to explain how we can interface our platform [3] with many different provers. The idea is quite simple: all the provers we use are first order predicate calculus with equality provers. The idea is then to translate set theoretic statements into predicate calculus statement as explained in sub-section 2.6.

Section 2 is made of six sub-sections. The first one contains a preliminary definition of sequents, inference rules, and proofs. Then we have the presentation of our Mathematical Language. It is defined as follows: the Propositional Language (section 2.3), the Predicate Language (section 2.4), the Equality Language (section 2.5), and the Set-theoretic Language (section 2.6). Each of these languages will be presented as an *extension* of the previous one.

In section 3, we develop some of the technologies used in the Rodin Platform [3] prover: the connection to some external automatic provers in section 3.1, the idea of reasoners and tactics in section 3.2, the notion of tactic profiles in section 3.3, the interactive prover in section 3.4, and finally the "Theory" plug-in in section 3.5.

Various sections (4 to 8) give then more information and comments on our approach with the proving system of Event-B. "Some Results" in section 4, "Using Proofs" in section 5, "Comparison of Proofs" in section 6, and finally "Trends and Open Problems" in section 7. We conclude in section 8.

2 A Mathematical Language Formal Construction

This section is essentially a reprint of chapter 9 of [2]

2.1 Sequent Calculus

Definitions In this section, we give some definitions which will be helpful to present the Sequent Calculus.
(1) A *sequent* is a generic name for "something we want to prove". For the moment, this is just an informally defined notion, which we shall refine later in section 2.1. The important thing to note at this point is that we can associate a *proof* with a sequent. For the moment, we do not know what a proof is however. It will only be defined at the end of this section.
(2) An *inference rule* is a device used to construct proofs of sequents. It is made of two parts: the *antecedent* part and the *consequent* part. The antecedent denotes a finite set of sequents while the consequent denotes a single sequent. An inference rule, named say **R1**, with antecedent A and consequent C is usually written as follows:

$$\frac{\mathsf{A}}{C}\ \textbf{R1}$$

It is to be read:

> Inference Rule **R1** yields a proof of sequent C as soon as we have proofs of each sequent of A.

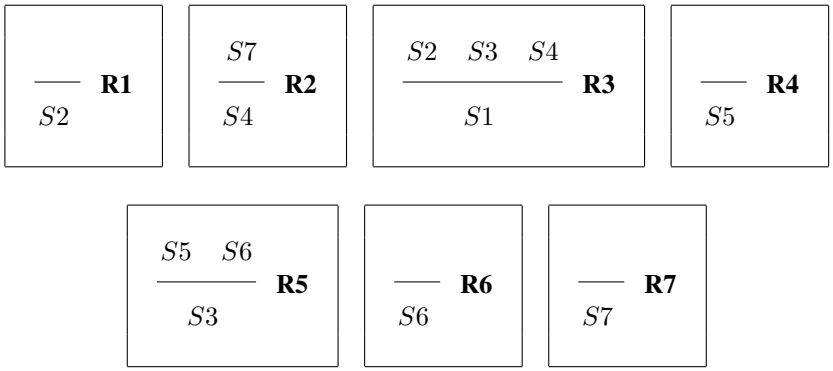
The antecedent A might be empty. In this case, the inference rule, named say **R2**, is written as follows:

$$\frac{}{C}\ \textbf{R2}$$

It is to be read:

> Inference Rule **R2** yields a proof of sequent C.

(3) A *theory* is a set of inference rules.
(4) The *proof of a sequent* within a theory is simply a finite tree with certain constraints. The nodes of such a tree have two components: a sequent s and a rule r of the theory. Here are the constraints for each node of the form (s, r): the consequent of the rule r is s, and the children of this node are nodes whose sequents are exactly all the sequents of the antecedent of rule r. As a consequence, the leaves of the tree contain rules with no antecedent. Moreover, the root node of the tree contains the sequent to be proved. As an example, let be given the following theory involving sequents $S1$ to $S7$ and rules **R1** to **R7**:



On figure 1 you can see a proof of sequent $S1$:

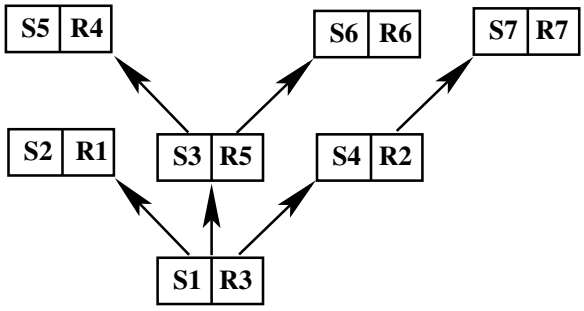


Fig. 1. A Proof

As can be seen, the root of the tree contains sequent $S1$, which is the one we want to prove. And it is easy to check that each node, say node $(S3, \textbf{R5})$, is indeed such that the consequent of its rule is the sequent of the node. More precisely, $S3$ in this case, is the consequent of rule **R5**. Moreover, we can check that the sequents of the child nodes of node $(S3, \textbf{R5})$, namely, $S5$ and $S6$, are exactly the sequents forming the antecedents of rule **R5**.

This tree can be interpreted as follows: In order to prove $S1$, we prove $S2$, $S3$, and $S4$, according to rule **R3**. In order to prove $S2$ we prove nothing more, according to rule **R1**. In order to prove $S3$ we prove $S5$ and $S6$, according to **R5**. And so on. This tree can be represented horizontally: this is indicated on figure 2. We shall now adopt this representation.

Sequents for a Mathematical Language We now refine our notion of sequent in order to define the way we shall make proofs with our Mathematical Language. Such a language contains constructs called *Predicates*. For the moment, this is all what we know about our Mathematical Language. Within this framework, a sequent S, as defined

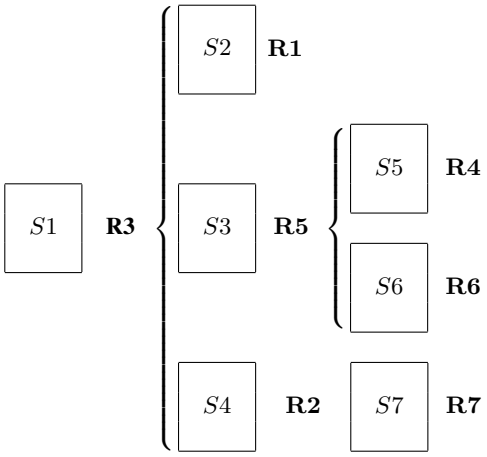


Fig. 2. Another Representation of the Proof Tree

in the previous section, now becomes a more complex object. It is made of two parts: the *hypotheses* part and the *goal* part. The hypothesis part denotes a finite set of predicates while the goal part denotes a single predicate. A sequent with hypotheses H and goal G is written as follows:

$$\mathsf{H} \vdash G$$

This sequent is to be read as follows:

Goal G holds under the set of hypotheses H

This is the sort of sequents we want to prove. It is also the sort of sequents we shall have in the theories associated with our Mathematical Language. Note that the set of hypotheses of a sequent might be empty and that the order and repetition of hypotheses in the set H is meaningless.

Initial Theory We now have enough elements at our disposal to define the first rules of our proving theory. Note again that we still don't know what a predicate is. We just know that predicates are constructs we shall be able to define within our future Mathematical Language. We start with three basic rules which we first state informally and then define more rigorously. They are called **HYP**, **MON**, and **CUT**. Here are their definitions:

– **HYP**: If the goal P of a sequent belongs to the set of hypotheses of this sequent, then it is proved.

$$\frac{}{\mathsf{H},\ P\ \vdash\ P}\ \textbf{HYP}$$

– **MON**: In order to prove a sequent, it is sufficient to prove another sequent with the same goal but with less hypotheses.

$$\frac{\mathsf{H}\ \vdash\ Q}{\mathsf{H},\ P\ \vdash\ Q}\ \textbf{MON}$$

– **CUT**: If you succeed in proving a predicate P under a set of hypotheses H, then P can be added to the set of hypotheses H for proving a goal Q.

$$\frac{\mathsf{H}\ \vdash\ P \qquad \mathsf{H},\ P\ \vdash\ Q}{\mathsf{H}\ \vdash\ Q}\ \textbf{CUT}$$

2.2 Rule Schema

Note that in the rules defined in the previous section, the letter H, P and Q are, so-called, *meta-variables*. The letter H is a meta-variable standing for a finite set of predicates, whereas the letter P and Q are meta-variables standing for predicates. Clearly then each of the previous "rules" stands for more than just one rule: it is better to call them *rule schemas*. This will always be the case in what follows.

2.3 The Propositional Language

In this section we present a first simple version of our Mathematical Language, it is called the Propositional Language. It will be later refined to more complete versions: Predicate Language (section 2.4), Equality Language (section 2.5), Set-theoretic Language (section 2.6).

Syntax Our first version is built around five constructs called *falsity*, *negation*, *conjunction*, *disjunction*, and *implication*. Given two predicates P and Q, we can construct their conjunction $P \wedge Q$, their disjunction $P \vee Q$, and their implication $P \Rightarrow Q$. And given a predicate P, we can construct its negation $\neg\, P$. This can be formalized by means of the following syntax:

$$
\begin{array}{rl}
\mathit{predicate} ::= & \bot \\
& \neg\, \mathit{predicate} \\
& \mathit{predicate} \wedge \mathit{predicate} \\
& \mathit{predicate} \vee \mathit{predicate} \\
& \mathit{predicate} \Rightarrow \mathit{predicate}
\end{array}
$$

This syntax is clearly ambiguous, but we do not care about it at this stage. Only note that conjunction and disjunction operators have stronger syntactic priorities than the implication operator. Moreover, conjunction and disjunction have the same syntactic priorities so that parentheses will always be necessary when several such distinct operators are following each other. Also note that this syntax does not contain any "base" predicate (except $\bot$): such predicates will come later in sections 2.5 and 2.6.

Enlarging the Initial Theory The initial theory of section 2.1 is enlarged with the following inference rules:

$$\frac{}{\mathsf{H},\ \bot\ \vdash\ P}\ \textbf{FALSE\_L} \qquad \frac{\mathsf{H}\ \vdash\ P \qquad \mathsf{H}\ \vdash\ \neg P}{\mathsf{H}\ \vdash\ \bot}\ \textbf{FALSE\_R}$$

$$\frac{\mathsf{H},\ \neg Q\ \vdash\ P}{\mathsf{H},\ \neg P\ \vdash\ Q}\ \textbf{NOT\_L} \qquad \frac{\mathsf{H},\ P\ \vdash\ \bot}{\mathsf{H}\ \vdash\ \neg P}\ \textbf{NOT\_R}$$

$$\frac{\mathsf{H},\ P,\ Q\ \vdash\ R}{\mathsf{H},\ P \wedge Q\ \vdash\ R}\ \textbf{AND\_L} \qquad \frac{\mathsf{H}\ \vdash\ P \qquad \mathsf{H}\ \vdash\ Q}{\mathsf{H}\ \vdash\ P \wedge Q}\ \textbf{AND\_R}$$

$$\frac{\mathsf{H},\ P\ \vdash\ R \qquad \mathsf{H},\ Q\ \vdash\ R}{\mathsf{H},\ P \vee Q\ \vdash\ R}\ \textbf{OR\_L} \qquad \frac{\mathsf{H},\ \neg P\ \vdash\ Q}{\mathsf{H}\ \vdash\ P \vee Q}\ \textbf{OR\_R}$$

$$\frac{\mathsf{H},\ P,\ Q\ \vdash\ R}{\mathsf{H},\ P,\ P \Rightarrow Q\ \vdash\ R}\ \textbf{IMP\_L} \qquad \frac{\mathsf{H},\ P\ \vdash\ Q}{\mathsf{H}\ \vdash\ P \Rightarrow Q}\ \textbf{IMP\_R}$$

As can be seen, each kind of predicates, namely falsity, negation, conjunction, disjunction, and implication, is given two rules: a left rule, labelled with **_L**, and a right rule, labelled with **_R**. This corresponds to the predicate appearing either in the hypothesis part (left) or in the goal part (right) of the consequent of the rule.

It is important to notice that we do not "define" the various propositional calculus operators with any kind of "truth table". We rather say how sequents involving such operators can be proved.

Derived Rules Besides the previous rules the following *derived* rule (among many others) is quite useful. It says that for proving a goal P it is sufficient to prove it first under hypothesis Q and then under hypothesis $\neg\, Q$.

$$\frac{\mathsf{H},\ Q\ \vdash\ P \qquad \mathsf{H},\ \neg\, Q\ \vdash\ P}{\mathsf{H}\ \vdash\ P}\ \textbf{CASE}$$

For proving a derived rule, we assume its antecedents (if any) and prove its consequent. With this in mind, here is the proof of derived rule **CASE**:

$$\frac{\mathsf{H}\ \vdash\ P}{\mathsf{H}\ \vdash\ P}\ \textbf{CUT}\left\{\begin{array}{l}\dfrac{\mathsf{H}\ \vdash\ Q \vee \neg\, Q}{\mathsf{H}\ \vdash\ Q \vee \neg\, Q}\ \textbf{OR\_R} \qquad \dfrac{\mathsf{H}, \neg\, Q\ \vdash\ \neg\, Q}{\mathsf{H}, \neg\, Q\ \vdash\ \neg\, Q}\ \textbf{HYP} \\[2ex] \dfrac{\mathsf{H}, Q \vee \neg\, Q\ \vdash\ P}{\mathsf{H}, Q \vee \neg\, Q\ \vdash\ P}\ \textbf{OR\_L}\left\{\begin{array}{l}\dfrac{\mathsf{H}, Q\ \vdash\ P}{\mathsf{H}, Q\ \vdash\ P}\ \textbf{assumed antecedent} \\[2ex] \dfrac{\mathsf{H}, \neg\, Q\ \vdash\ P}{\mathsf{H}, \neg\, Q\ \vdash\ P}\ \textbf{assumed antecedent}\end{array}\right.\end{array}\right.$$

Methodology The method we are going to use to build our Mathematical Language must start to be clearer: it will be very systematic. It is made of two steps: first we augment our syntax. Then either the extension corresponds to a simple facility. In that case, we give simply the definition of the new construct in terms of previous ones. Or the new construct is not related to any previous constructs. In that case, we augment our current theory.

Extending the Proposition Language The Proposition Language is now extended by adding one more construct called *equivalence*. Given two predicates P and Q, we can construct their equivalence $P \Leftrightarrow Q$. We also add one predicate: $\top$. As a consequence, our syntax is now the following:

$$
\begin{array}{rl}
predicate ::= & \bot \\
& \top \\
& \neg\, predicate \\
& predicate \wedge predicate \\
& predicate \vee predicate \\
& predicate \Rightarrow predicate \\
& predicate \Leftrightarrow predicate
\end{array}
$$

Note that implication and equivalence operators have the same syntactic priorities so that parentheses will be necessary when several such distinct operators are following each other. Such extensions are defined in terms of previous ones by mere rewriting rules:

Predicate	Rewritten
$\top$	$\neg\ \bot$
$P \Leftrightarrow Q$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$

The following derived rules can be proved easily:

$$\frac{\mathsf{H} \vdash P}{\mathsf{H},\ \top \vdash P}\ \textbf{TRUE\_L} \qquad \frac{}{\mathsf{H} \vdash \top}\ \textbf{TRUE\_R}$$

Note that rule **TRUE_L** can be proved using rule **MON** but the reverse rule (exchanging antecedent and consequent), which holds as well, cannot. We leave it as an exercise to the reader to prove these rules.

2.4 The Predicate Language

Syntax In this section, we introduce the Predicate Language. The syntax is extended with a number of new kinds of predicates and also with the introduction of two new syntactic categories called *expression* and *variable*. A *variable* is a simple identifier. Given a non-empty list of variables x made of pairwise distinct identifiers and a predicate P, the construct $\forall x \cdot P$ is called a *universally quantified predicate*. Likewise, given a non-empty list of variables x made of pairwise distinct identifiers and a predicate P, the construct $\exists x \cdot P$ is called an *existentially quantified predicate*. An *expression* is either a variable or else a *paired expression* $E \mapsto F$, where E and F are two expressions. Here is this new syntax:

$$
\begin{array}{lll}
predicate & ::= & \bot \\
 & & \top \\
 & & \neg\, predicate \\
 & & predicate \ \wedge \ predicate \\
 & & predicate \ \vee \ predicate \\
 & & predicate \ \Rightarrow \ predicate \\
 & & predicate \ \Leftrightarrow \ predicate \\
 & & \forall var\_list \cdot predicate \\
 & & \exists var\_list \cdot predicate \\
 & & \\
expression & ::= & variable \\
 & & expression \mapsto expression \\
 & & \\
var\_list & ::= & variable \\
 & & variable,\ var\_list
\end{array}
$$

This syntax is also ambiguous. Note however that the scope of the universal or existential quantifiers extends to the right as much as they can, the limitation being expressed either by the end of the formula or by means of enclosing parentheses.

Predicates and Expressions It might be useful at this point to clarify the difference between a predicate and an expression. A predicate P is a piece of formal text which can be *proved* when embedded within a sequent as in:

$$\mathrm{H} \vdash P$$

A predicate does not denote anything. This is not the case of an expression which always denotes an *object*. An expression cannot be "proved". Hence predicates and expressions are incompatible. Note that for the moment the possible expressions we can define are quite limited. This will be considerably extended in the Set-theoretic Language defined in Section 2.6.

Inference Rules for Universally Quantified Predicates The universally and existentially quantified predicates require introducing corresponding rules of inference. As for propositional calculus, in both cases we need two rules: one for quantified assumptions (left rule) and one for a quantified goal (right rule). Here are these rules for universally quantified predicates:

$$\frac{\text{H},\ \forall x \cdot P,\ [x := E]P\ \vdash\ Q}{\text{H},\ \forall x \cdot P\ \vdash\ Q}\ \textbf{ALL\_L}$$

$$\frac{\text{H}\ \vdash\ P}{\text{H}\ \vdash\ \forall x \cdot P}\ \textbf{ALL\_R}\ (x \text{ not free in H})$$

The first rule (ALL_L) allows us to add another assumption when we have a universally quantified one. This new assumption is obtained by instantiating the quantified variable x by any expression E in the predicate P: this is denoted by $[x := E]P$. The second rule (ALL_R) allows us to remove the "$\forall$" quantifier appearing in the goal. This can be done however only if the quantified variable (here x) *does not appear free* in the the set of assumptions H: this requirement is called a *side condition*. In the sequel we shall write x $\underline{\text{nfin}}$ P to mean that variable x is not free in predicate P. The same notation is used with an expression E. We omit in this presentation to develop the syntactic rules allowing us to compute non-freeness as well as substitutions. We have similar rules for existentially quantified predicates:

$$\frac{\text{H},\ P\ \vdash\ Q}{\text{H},\ \exists x \cdot P\ \vdash\ Q}\ \textbf{XST\_L}\ (x \text{ not free in H and } Q)$$

$$\frac{\text{H}\ \vdash\ [x := E]P}{\text{H}\ \vdash\ \exists x \cdot P}\ \textbf{XST\_R}$$

As an example, we prove now the following sequent:

$$\forall x \cdot (\exists y \cdot P_{x,y})\ \Rightarrow\ Q_x\quad \vdash\quad \forall x \cdot (\forall y \cdot P_{x,y}\ \Rightarrow\ Q_x)$$

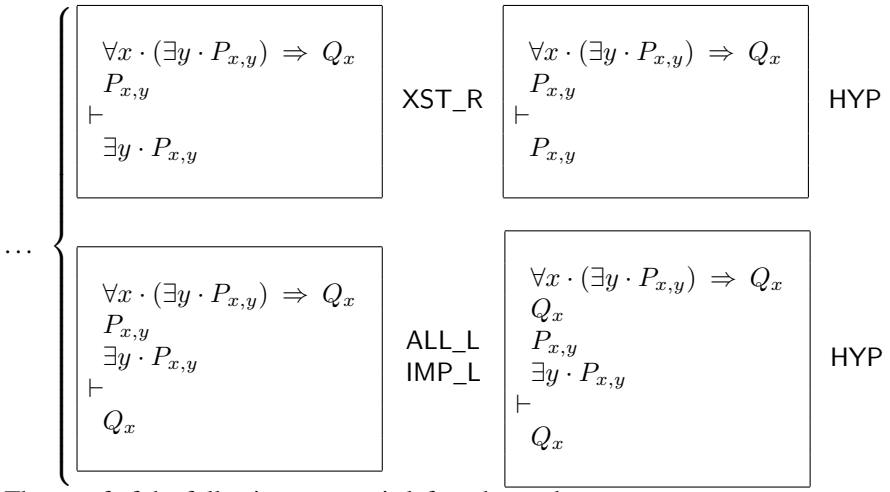
where $P_{x,y}$ stands for a predicate containing variables x and y only as free variables, and Q_x stands for a predicate containing variable x only as a free variable.

$$\forall x \cdot (\exists y \cdot P_{x,y})\ \Rightarrow\ Q_x \quad\vdash\quad \forall x \cdot (\forall y \cdot P_{x,y}\ \Rightarrow\ Q_x)$$

ALL_R
ALL_R
IMP_R

$$\forall x \cdot (\exists y \cdot P_{x,y})\ \Rightarrow\ Q_x,\ P_{x,y} \quad\vdash\quad Q_x$$

CUT ...



The proof of the following sequent is left to the reader:

$$\forall x \cdot (\forall y \cdot P_{x,y} \Rightarrow Q_x) \quad \vdash \quad \forall x \cdot (\exists y \cdot P_{x,y}) \Rightarrow Q_x$$

2.5 Introducing Equality

The Predicate Language is once again extended by adding a new predicate, the *equality predicate*. Given two expressions E and F, we define their equality by means of the construct $E = F$. Here is the extension of our syntax:

$$\begin{array}{lcl}
predicate & ::= & \bot \\
 & & \top \\
 & & \neg\, predicate \\
 & & predicate \wedge predicate \\
 & & predicate \vee predicate \\
 & & predicate \Rightarrow predicate \\
 & & predicate \Leftrightarrow predicate \\
 & & \forall var_list \cdot predicate \\
 & & \exists var_list \cdot predicate \\
 & & expression = expression \\
 & & \\
expression & ::= & variable \\
 & & expression \mapsto expression
\end{array}$$

Note that we shall use the operator $\neq$ in the sequel to mean, as is usual, the negation of equality. The inference rules for equality are the following:

$$\frac{[x := F]\mathrm{H},\ E = F\ \vdash\ [x := F]P}{[x := E]\mathrm{H},\ E = F\ \vdash\ [x := E]P}$$ **EQ_LR**

$$\frac{[x := E]\mathrm{H},\ E = F\ \vdash\ [x := E]P}{[x := F]\mathrm{H},\ E = F\ \vdash\ [x := F]P}$$ **EQ_RL**

It allows us to *apply* an equality assumption in the remaining assumptions and in the goal. This can be made by using the equality from left to right or from right to left. Subsequent rules correspond to the reflexivity of equality and to the equality of pairs. They are both defined by some rewriting rules as follows:

Operator	Predicate	Rewritten
Equality	$E = E$	$\top$
Equality of pairs	$E \mapsto F \ = \ G \mapsto H$	$E = G \ \wedge \ F = H$

The following rewriting rules, within which x is supposed to be not free in E, are easy to prove. They are called the *one point rules*:

Predicate	Rewritten
$\forall x \cdot x = E \ \Rightarrow \ P$	$[x := E]P$
$\exists x \cdot x = E \ \wedge \ P$	$[x := E]P$

2.6 The Set-theoretic Language

Our next language, the Set-theoretic Language, is now presented as an extension to the previous Predicate Language.

Syntax In this extension, we introduce some special kind of expressions called *sets*. Note that not all expressions are set: for instance a pair is not a set. However, in the coming syntax we shall not make any distinction between expressions which are sets and expressions which are not.

We introduce another predicate the *membership predicate*. Given an expression E and a set S, the construct $E \in S$ is a membership predicate which says that expression E is a *member* of set S.

We also introduce the basic set constructs. Given two sets S and T, the construct $S \times T$ is a set called the *Cartesian product* of S and T. Given a set S, the construct $\mathbb{P}(S)$ is a set called the *power set of* S. Finally, given a list of variables x with pairwise distinct identifiers, a predicate P, and an expression E, the construct $\{x \cdot P \mid E\}$ is called a *set defined in comprehension*. Here is our new syntax:

$$
\begin{array}{lll}
predicate & ::= & \ldots \\
 & & expression \in expression \\
 & & \\
expression & ::= & variable \\
 & & expression \mapsto expression \\
 & & expression \times expression \\
 & & \mathbb{P}(expression) \\
 & & \{\, var\_list \cdot predicate \mid expression \,\}
\end{array}
$$

Note that we shall use the operator $\notin$ in the sequel to mean, as is usual, the negation of set membership.

Axioms of Set Theory The axioms of the set-theoretic Language are given under the form of equivalences to various set memberships. They are all defined in terms of rewriting rules. Note that the last of these rules defines equality for sets. It is called the *Extensionality Axiom*.

Operator	Predicate	Rewritten	Side Cond.
Cartesian product	$E \mapsto F \in S \times T$	$E \in S \ \wedge \ F \in T$	
Power set	$E \in \mathbb{P}(S)$	$\forall x \cdot x \in E \ \Rightarrow \ x \in S$	x <u>nfin</u> E x <u>nfin</u> S

Operator	Predicate	Rewritten	Side Cond.
Comprehension	$E \in \{ x \cdot P \mid F \}$	$\exists x \cdot P \ \wedge \ E = F$	x nfin E
Equality	$S = T$	$S \in \mathbb{P}(T) \ \wedge \ T \in \mathbb{P}(S)$	

As a special case, set comprehension can sometimes be written $\{ F \mid P \}$, which can be read as follows: "the set of objects with shape F when P holds". However, as you can see, the list of variables x has now disappeared. In fact, these variables are then *implicitly determined* as being all the free variables in F. When we want that x represent only *some*, but not all, of these free variables we cannot use this shorthand.

A more special case is one where the expression F is exactly a single variable x, that is $\{ x \cdot P \mid x \}$. As a shorthand, this can be written $\{ x \mid P \}$, which is very common in informally written mathematics. And then $E \in \{ x \mid P \}$ becomes $[x := E]P$ according to the second "one point rule" of section 2.5.

Elementary Set Operators In this section, we introduce the classical set operators: inclusion, union, intersection, difference, extension, and the empty set.

$$
\begin{array}{lll}
predicate & ::= & \ldots \\
 & & expression \subseteq expression \\
 & & \\
expression & ::= & \ldots \\
 & & expression \cup expression \\
 & & expression \cap expression \\
 & & expression \setminus expression \\
 & & \{expression\_list\} \\
 & & \varnothing \\
 & & \\
expression\_list & ::= & expression \\
 & & expression,\ expression\_list
\end{array}
$$

Notice that the expressions in an $expression\_list$ are not necessarily distinct.

Operator	Predicate	Rewritten
Inclusion	$S \subseteq T$	$S \in \mathbb{P}(T)$
Union	$E \in S \cup T$	$E \in S \ \lor \ E \in T$
Intersection	$E \in S \cap T$	$E \in S \ \land \ E \in T$
Difference	$E \in S \setminus T$	$E \in S \ \land \ \neg (E \in T)$
Set extension	$E \in \{a, \ldots, b\}$	$E = a \ \lor \ \ldots \ \lor \ E = b$
Empty set	$E \in \varnothing$	$\bot$

Generalization of Elementary Set Operators The next series of operators consists in generalizing union and intersection to sets of sets. This takes the forms either of an operator acting on a set or of a quantifier.

$$
\begin{array}{rcl}
\ldots & & \\
& & \\
expression & ::= & \ldots \\
& & \mathrm{union}(expression) \\
& & \bigcup var\_list \cdot predicate \mid expression \\
& & \mathrm{inter}(expression) \\
& & \bigcap var\_list \cdot predicate \mid expression
\end{array}
$$

Operator	Predicate	Rewritten	Side Cnd.
Generalized union	$E \in \mathsf{union}\,(S)$	$\exists s \cdot s \in S \ \land \ E \in s$	$s \ \underline{\mathsf{nfin}} \ S$ $s \ \underline{\mathsf{nfin}} \ E$
Quantified union	$E \in \bigcup x \cdot P \mid T$	$\exists x \cdot P \ \land \ E \in T$	$x \ \underline{\mathsf{nfin}} \ E$

Operator	Predicate	Rewritten	Side Cnd.
Generalized intersection	$E \in \mathrm{inter}(S)$	$\forall s \cdot s \in S \Rightarrow E \in s$	s $\underline{\mathsf{nfin}}$ S s $\underline{\mathsf{nfin}}$ E
Quantified intersection	$E \in \bigcap x \cdot P \mid T$	$\forall x \cdot P \Rightarrow E \in T$	x $\underline{\mathsf{nfin}}$ E

The last two rewriting rules require that the set $\mathrm{inter}(S)$ and $\bigcap x \cdot P \mid T$ be *well defined*. This is presented in the following table:

Set construction	Well-definedness condition
$\mathrm{inter}(S)$	$S \neq \varnothing$
$\bigcap x \cdot P \mid T$	$\exists x \cdot P$

Binary Relation Operators We now define a first series of binary relation operators: the set of binary relations built on two sets, the domain and range of a binary relation, and then various sets of binary relations.

$$
\begin{array}{lll}
\ldots & & \\
& & \\
expression & ::= & \ldots \\
& & expression \leftrightarrow expression \\
& & \mathrm{dom}(expression) \\
& & \mathrm{ran}(expression) \\
& & expression \nleftrightarrow expression \\
& & expression \leftrightarrow\!\!\!\!\!\rightarrow expression \\
& & expression \nleftrightarrow\!\!\!\!\!\rightarrow expression
\end{array}
$$

Operator	Predicate	Rewritten	Side Cnd.
Binary relations	$r \in S \leftrightarrow T$	$r \subseteq S \times T$	
Domain	$E \in \mathrm{dom}\,(r)$	$\exists y \cdot E \mapsto y \ \in \ r$	y nfin E y nfin r
Range	$F \in \mathrm{ran}\,(r)$	$\exists x \cdot x \mapsto F \ \in \ r$	x nfin F x nfin r
Total relations	$r \in S \nleftrightarrow T$	$r \in S \leftrightarrow T \ \wedge \ \mathrm{dom}\,(r) = S$	
Surjective relations	$r \in S \leftrightarrow\!\!\!\!\rightarrow T$	$r \in S \leftrightarrow T \ \wedge \ \mathrm{ran}\,(r) = T$	

The next series of binary relation operators define the converse of a relation, various relation restrictions and the image of a set under a relation.

$$
\begin{array}{rl}
expression & ::= \ \ldots \\
 & \quad expression^{-1} \\
 & \quad expression \lhd expression \\
 & \quad expression \rhd expression \\
 & \quad expression \hookleftarrow expression \\
 & \quad expression \hookrightarrow expression \\
 & \quad expression[expression]
\end{array}
$$

Operator	Predicate	Rewritten	Side Cnd.
Converse	$E \mapsto F \in r^{-1}$	$F \mapsto E \in r$	
Domain restriction	$E \mapsto F \ \in \ S \lhd r$	$E \in S \ \ \wedge \ \ E \mapsto F \in r$	
Range restriction	$E \mapsto F \ \in \ r \rhd T$	$E \mapsto F \in r \ \ \wedge \ \ F \in T$	

Operator	Predicate	Rewritten	Side Cnd.
Domain subtraction	$E \mapsto F \ \in \ S \lhd r$	$\neg\, E \in S \ \ \wedge \ \ E \mapsto F \in r$	
Range subtraction	$E \mapsto F \ \in \ r \rhd T$	$E \mapsto F \in r \ \ \wedge \ \ \neg\, F \in T$	
Relational Image	$F \in r[U]$	$\exists x \cdot x \in U \ \wedge \ x \mapsto F \in r$	x nfin F x nfin r x nfin U

Let us illustrate the relational image. Given a binary relation r from a set S to a set T, the image of a subset U of S under the relation r is a subset of T. The image of U under r is denoted by $r[U]$. Here is its definition:

$$r[U] \ = \ \{\, y \mid \exists x \cdot x \in U \ \wedge \ x \mapsto y \ \in \ r \,\}$$

This is illustrated on figure 3. As can be seen on this figure, the image of the set $\{a, b\}$ under relation r is the set $\{m, n, p\}$.

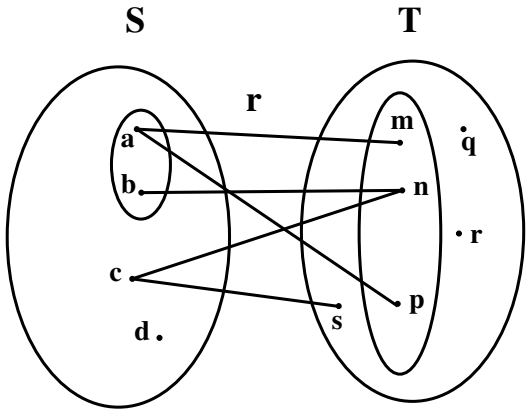


Fig. 3. Image of a Set under a Relation

Our next series of operators defines the composition of two binary relations, the overriding of a relation by another one, and the direct and parallel products of two relations.

$$
\begin{array}{lll}
expression & ::= & \ldots \\
 & & expression \,;\, expression \\
 & & expression \circ expression \\
 & & expression \lhd\!\!+ expression \\
 & & expression \otimes expression \\
 & & expression \parallel expression
\end{array}
$$

Operator	Predicate	Rewritten	Side Cnd.
Forward composition	$E \mapsto F \in f \,;\, g$	$\exists x \cdot E \mapsto x \in f \;\; \wedge \;\; x \mapsto F \in g$	x nfin E x nfin F x nfin f x nfin g
Backward composition	$E \mapsto F \in g \circ f$	$E \mapsto F \in f \,;\, g$	

Given a relation f from S to T and a relation g from T to U, the forward relational composition of f and g is a relation from S to U. It is denoted by the construct $f \,;\, g$. Sometimes it is denoted the other way around as $g \circ f$, in which case is is said to be the backward composition.

Operator	Predicate	Rewritten
Overriding	$E \mapsto F \in f \lhd\!\!+ g$	$E \mapsto F \in (\mathrm{dom}\,(g) \lhd f) \;\cup\; g$
Direct product	$E \mapsto (F \mapsto G) \in f \otimes g$	$E \mapsto F \in f \;\; \wedge \;\; E \mapsto G \in g$
Parallel product	$(E \mapsto F) \mapsto (G \mapsto H) \in f \parallel g$	$E \mapsto G \in f \;\; \wedge \;\; F \mapsto H \in g$

The overriding operator is applicable in general to a relation f from, say, a set S to a set T, and a relation g also from S to T. When f is a function and g is the singleton function $\{x \mapsto E\}$, then $f \lhd\!\!+ \{x \mapsto E\}$ replaces in f the pair $x \mapsto f(x)$ by the pair $x \mapsto E$. Notice that in the case where x is not in the domain of f, then $f \lhd\!\!+ \{x \mapsto E\}$ simply adds the pair $x \mapsto E$ to the function f. In this case, it is thus equal to $f \cup \{x \mapsto E\}$.

Functional Operators In this section we define various function operators: the sets of all partial and total functions, partial and total injections, partial and total surjections, and bijections. We also introduce the two projection functions as well as the identity function.

$$
\begin{array}{lll}
expression & ::= & \ldots \\
& & \mathrm{id} \\
& & expression \pfun expression \\
& & expression \fun expression \\
& & expression \pinj expression \\
& & expression \inj expression \\
& & expression \psur expression \\
& & expression \surj expression \\
& & expression \bij expression \\
& & \mathrm{prj}_1 \\
& & \mathrm{prj}_2
\end{array}
$$

Operator	Predicate	Rewritten
Identity	$E \mapsto F \in \mathrm{id}$	$E = F$
Partial functions	$f \in S \pfun T$	$f \in S \rel T \ \land \ (f^{-1} ; f) \subseteq \mathrm{id}$
Total functions	$f \in S \fun T$	$f \in S \pfun T \ \land \ S = \mathrm{dom}\,(f)$
Partial injections	$f \in S \pinj T$	$f \in S \pfun T \ \land \ f^{-1} \in T \pfun S$
Total injections	$f \in S \inj T$	$f \in S \fun T \ \land \ f^{-1} \in T \pfun S$
Partial surjections	$f \in S \psur T$	$f \in S \pfun T \ \land \ T = \mathrm{ran}\,(f)$
Total surjections	$f \in S \surj T$	$f \in S \fun T \ \land \ T = \mathrm{ran}\,(f)$

Operator	Predicate	Rewritten
Bijections	$f \in S \rightarrowtail\!\!\!\!\!\rightarrow T$	$f \in S \rightarrowtail T \quad \wedge \quad f \in S \twoheadrightarrow T$
First projection	$(E \mapsto F) \mapsto G \in \mathrm{prj}_1$	$G = E$
Second projection	$(E \mapsto F) \mapsto G \in \mathrm{prj}_2$	$G = F$

Lambda Abstraction and Function Invocation We now define *lambda abstraction*, which is a way to construct functions, and also function invocation, which is a way to call functions. But first we have to define the notion of *pattern of variables*. A pattern of variables is either an identifier or a pair made of two patterns of variables. Moreover, all variables composing the pattern must be distinct. For example, here are three patterns of variables:

$$\mathsf{abc}$$

$$\mathsf{abc} \mapsto \mathsf{def}$$

$$\mathsf{abc} \mapsto (\mathsf{def} \mapsto \mathsf{ghi})$$

Given a pattern of variables x, a predicate P, and an expression E, the construct $\lambda\, x \cdot P \,|\, E$ is a lambda abstraction, which is a function. Given a function f and an expression E, the construct $f(E)$ is an expression denoting a function invocation. Here is our new syntax:

$$\begin{array}{lll} \mathit{expression} & ::= & \ldots \\ & & \mathit{expression}(\mathit{expression}) \\ & & \lambda\, \mathit{pattern} \cdot \mathit{predicate} \,|\, \mathit{expression} \\ \\ \mathit{pattern} & ::= & \mathit{variable} \\ & & \mathit{pattern} \mapsto \mathit{pattern} \end{array}$$

In the following table, l stands for the list of variables in the pattern L.

Operator	Predicate	Rewritten
Lambda abstraction	$F \in \lambda L \cdot P \mid E$	$F \in \{l \cdot P \mid L \mapsto E\}$
Function invocation	$F = f(E)$	$E \mapsto F \in f$

The function invocation construct $f(E)$ requires a well-definedness condition, which is the following:

Expression	Well-definedness condition
$f(E)$	$f^{-1} ; f \subseteq \mathrm{id} \quad \wedge \quad E \in \mathrm{dom}(f)$

3 Prover Technologies Used in the Rodin Platform

3.1 Connecting to Various External Provers

On the Rodin Platform [3], we use various "external" provers. As explained in the introduction, the connection to these external provers is made by mans of a translation of the set theoretic statement (defining the predicate to be proved) into a predicate calculus statement. As a very simple example, suppose we have to prove a statement like $S \subseteq T$ where S and T stand for some set expressions. It is translated as follows: $\forall x \cdot x \in S' \Rightarrow x \in T'$ where S' and T' stand for the translations of S and T respectively.

The external provers we have on the Rodin Platform are the, so called, Predicate Prover (an internally developed predicate calculus prover) and some SMT provers (Alt-Ergo, CVC3, VeriT, Z3) [7]. There is even a plug-in for translating Event-B sequents to an embedding in HOL which allows to perform proofs with the Isabelle proof assistant.

3.2 Reasonners and Tactics

The text of this section is a copy from [6].

Like the rest of the Rodin platform, the prover has been designed for openness. The main code of the prover just maintains a proof tree in Sequent Calculus and does not contain any reasoning capability. It is extensible through reasoners and tactics. A reasoner is a piece of code that, given an input sequent, either fails or succeeds. In case of success, the reasoner produces a proof rule which is applied to the current proof tree node.

Reasoners could be applied interactively. However this would be very tedious. Reasoner application can thus be automated by using tactics that take a more global view of

the proof tree and organise the running of reasoners. Tactics can also backtrack the proof tree, that is undo some reasoner applications in case the prover entered a dead-end.

The core platform contains a small set of reasoners written in Java that either implement the basic proving rules (HYP, CUT, FALSE_L, etc.), or perform some simple clean-up on sequents such as normalisation or unit propagation (generalised modus-ponens). These reasoners allow to discharge the most simple proof obligations. As already explained, they are complemented by reasoners that link the Rodin platform to external provers, such as those of Atelier B (ML and PP) and SMT solvers (Alt-Ergo, CVC3, VeriT, Z3, etc.) [7].

3.3 Tactic Profile

As explained in the previous section, the Rodin Platform is provided with some elementary tactics including calls to external provers. Such tactics can be put together in, so-called, *tactic profiles*. The user can define several such profiles and attach them interactively (in the Rodin Platform preference framework) to the prover. These has the effect of automatising proofs.

3.4 Interactive Proofs

When the automatic treatment of the prover fails, the user can perform an interactive proof. The idea is to give the possibility to the user to call some elementary tactics explicitly. In practice, this is done by clicking on some emphasised symbols either on the goal part or the hypothesis part of the sequent to be proved.

For example, if the following sequent $\mathsf{H} \vdash S \subseteq T$ is to be proved, then by clicking on the emphasised symbol $\subseteq$, the sequent to be proved is transformed to $\mathsf{H} \vdash \forall x \cdot x \in S \Rightarrow x \in T$. The user can then click on the emphasised symbol $\forall$: this has the effect of transforming the present sequent to $\mathsf{H} \vdash x \in S \Rightarrow x \in T$. Finally, by clicking on the emphasised symbol $\Rightarrow$, we obtain the following sequent $\mathsf{H}, x \in S \vdash x \in T$, and so on. As can be seen, activating interactively the sequent to be proved, has the effect of decomposing gradually this sequent into smaller ones.

3.5 Theory Plug-in

The text of this section is a copy from the paper [6]. This plug-in of the Rodin Platform, defined in [4], allows one to extend the basic mathematical operators of Event-B. These operators are polymorphic. They can be defined explicitly in terms of existing ones. As examples of these extensions, we can define the concept of well-foundedness, that of fixpoint, that of relational closure, and so on.

It is also possible to give some axiomatic definitions only. An interesting outcome of this last feature allows one to define the set of Real numbers axiomatically. Moreover, the user of this plug-in can add some corresponding theorems, and inference or rewriting rules able to extend the provers. It is also possible to define new (possibly recursive) types. This very important plug-in has been developed by Issam Maamria and Asieh Salehi in Southampton University.

4 Some Results

The most common proof generating deduction tools are the Predicate Prover (developed internally) and also the SMT provers we mention earlier (Alt-Ergo, CVC3, VeriT, and Z3). We are very happy with these provers. In the future we will probably try other proof systems although it is not decided yet which ones. In section 3.4 we gave a small examples of an interactve proof. Here is another example, that of the proof of the following sequent:

$$r \in S \leftrightarrow T \ \vdash \ A1 \subseteq A2 \ \Rightarrow \ r[A1] \subseteq r[A2]$$

where $r[A1]$ or $r[a2]$ stand for the images of the set $A1$ or $A2$ under the relation r. The automatic proof goes as follows. Each line contains first an indication of the inference rule (or sometimes the rewriting rule) that is applied, then the goal of the sequent is shown after the ":" symbol (the hypotheses of the sequent are not shown):

$\Rightarrow$ in goal : $A1 \subseteq A2 \ \Rightarrow \ r[A1] \subseteq r[A2]$
  remove $\subseteq$ in goal : $r[A1] \subseteq r[A2]$
    remove $\in$ in goal : $\forall x \cdot x \in r[A1] \ \Rightarrow \ x \in r[A2]$
      $\forall$ goal (frees x) : $\forall x \cdot (\exists x0 \cdot x0 \in A1 \wedge x0 \mapsto x \in r) \ \Rightarrow \ \ldots$
        $\Rightarrow$ in goal : $(\exists x0 \cdot x0 \in A1 \wedge x0 \mapsto x \in r) \ \Rightarrow \ (\exists x0 \cdot x0 \in A2 \wedge x0 \mapsto x \in r)$
          $\exists$ hyp $(\exists x0 \cdot x0 \in A1 \wedge x0 \mapsto x \in r)$: $\exists x0 \cdot x0 \in A2 \wedge x0 \mapsto x \in r$
            $\exists$ goal (inst $x0$) : $\exists x0 \cdot x0 \in A2 \wedge x0 \mapsto x \in r$
              $\wedge$ goal : $x0 \in A2 \wedge x0 \mapsto x \in r$
                $\forall$ hyp p (inst $x0$) : $x0 \in A2$
                  hyp : $x0 \in A1$
                  hyp : $x0 \in A2$
                hyp : $x0 \mapsto x \in r$

Notice that the goals are optional. In this case, we would obtain the following proof:

$\Rightarrow$ in goal
  remove $\subseteq$ in goal
    remove $\in$ in goal
      $\forall$ goal (frees x)
        $\Rightarrow$ in goal
          $\exists$ hyp $(\exists x0 \cdot x0 \in A1 \wedge x0 \mapsto x \in r)$
            $\exists$ goal (inst $x0$)
              $\wedge$ goal
                $\forall$ hyp p (inst $x0$)
                  hyp
                  hyp
                hyp

As can be seen, the proof indentation reflects the tree structure of the proof. In an interactive proof, the user can easily navigate within this tree. By pointing to a node in the tree, the user can also get more details about the usage of the inference rule used in that node. It can also hide some parts of the tree or do some copy/paste of some sub-tree when a similar sub-proof is needed somewhere. The user can also "review" a node in

the tree without providing a proof for it yet. It is very convenient when applying the cut rule in order to define a local lemma (to be proven later). We are satisfied with the generated proofs: they are detailed enough. However these proofs are not intended to be read, just used at some specific moment in the proof process. The simple proof we have just shown is clearly already a bit difficult to read. It is very frequent to have far bigger proofs, which are thus totally unreadable. In the shown proof, the entire sequent is not shown, it could have been of course, but it'll make the proof even more difficult to read.

At the moment, we are very satisfied with the system at hand. In fact, the introduction of SMT solvers into our proving system was a big step forward. Notice that these provers can be mentioned explicitly in a tactic profile, resulting in many proofs, that were not done automatically before this introduction, now becoming automatic proofs.

Concerning model-checking, it is already present in the Rodin Platform. It is called PROB [5]. It has been developed in the University of Duesseldorf in Germany. We see it has a very useful complement to our proof system. It is particularly interesting when the user finds some difficulties in doing a proof. Using the model-checker can easily help finding some counter-examples showing that the user tries to prove something that cannot be proven.

5 Using Proofs

Proof are used to discharge proof obligations generated by the proof obligation generator, a tool the Rodin Platform. This tool generates many different proof obligations, among which the more important are invariant proof obligations and refinement proof proof obligations. All this is defined in great details in [2].

Witnesses are directly defined in our models, thus avoiding many existential proofs. The only important properties we extract from a proof is whether it has been done automatically or interactively, or if it not discharged or has some reviewed nodes still in it. Out of this, we determine some important statistics calculated on all the proofs of an entire development. As an example, the B development for the Paris metro line 14 generated 27,800 proofs. the system developed for the Charles de Gaule airport generated 43,600 proofs. More recently the development of an embedded system formalising an aircraft landing system with Event-B generated 2328 proofs on the Rodin Platform. All these proofs were discharged automatically.

The construction of code out of the last refinement of a model is independent of the proofs. We are filtering the last refinement of a model in order to determine whether it can be translated into code. The system for the line 14 metro system generated 86,000 lines of ADA. For the CDG shuttle, 158,000 lines of ADA were generated.

The proofs are not explicitly attached to the generated code but all proofs of a development can be consulted directly on the Rodin Platform, together with the mentioned statistics.

6 Comparison of Proofs

The question of comparing proofs is not relevant in our domain. The quality of a proof (essentially its length and the number of explicit quantified variable instantiations) is an important qualitative factor. This quality is a good indication of the overall quality of the corresponding formal model. The proportion of automatic proofs is also an important factor of the overall quality of the model.

7 Trends and Open Problems

Our current trend is to experiment with the "Theory" plug-in that has been developed recently (see section 3.5). Our idea is to incorporate more and more the usage of this plug-in in our future developments. In particular, this plug-in allows us to incorporate an axiomatisation of the Real Numbers. This very important for extending the usage of Event-B to model hybrid systems.

Within the next ten years, our vision is to develop more systems using this formal approach, thus reducing the need for programming and replace it by the need to proving.

8 Conclusions

In this paper, we describe the proving approach we developed over the years in order to model computerised systems using a formal method (Event-B) based on refinement.

References

1. J-R. Abrial. *From Z to B and then Event-B: Assigning Proofs to Meaningful Programs*. In E.B. Johnsen and L. Petre, editors, IFM, volume 7940 of Lecture Notes in Computer Science, pages 1Ð15. Springer, 2013.
2. J.R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press 2010.
3. http: //www.event-b.org *Rodin Platform*
4. M. Butler and I. Maamria. *Practical theory extension in Event-B*. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, Theories of Programming and Formal Methods, volume 8051 of Lecture Notes in Computer Science, pages 67 to 81. Springer Berlin Heidelberg, 2013.
5. M. Leuschel and M. Butler. *ProB: An Automated Analysis Toolset for the B Method*. International Journal on Software Tools for Technology Transfer 2008.
6. L. Voisin and J.R. Abrial *The Rodin Platform Has Turned Ten* ABZ 2014
7. D. Deharbe, P. Fontaine, Y. Guyot, and L. Voisin. *SMT solvers for Rodin*. In Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z, ABZ'12, pages 194 to 207, Berlin, Heidelberg, 2012. Springer-Verlag.