

Proposal for an extensible rule-based prover for Event-B

Issam Maamria, Michael J Butler, Andrew Edmunds, Abdolbaghi Rezazadeh
e-mail:im06r, mjb, ae2, ra3@ecs.soton.ac.uk

Dependable Systems and Software Engineering
Department of Electronics and Computer Science
University of Southampton

July 3, 2009

Abstract

Event-B [6] is a formalism for discrete system modelling. Key features of Event-B include the use of set theory as a modelling notation, the use of refinement to model systems at different levels of abstraction, and the use of mathematical proof to verify consistency between refinement levels. Event-B provides two constructs to model systems; contexts describe static properties (constants and carrier sets) whereas machines model dynamic behaviour (variables and events). In this document we propose a mechanism by which the mathematical language of Event-B as well as the proof capabilities of its tool [7] can be extended by users whilst maintaining the soundness of the prover. Initially, we focus on extending the prover with a subset of proof rules: *the rewrite rules*. We envisage the new mechanism to evolve to cater for the different mathematical extensions proposed in [4].

Keywords: *Event-B, rule-based prover, rewrite rules, inference rules, mathematical extensions*

1 Introduction

Modelling of discrete systems in Event-B is carried out by specifying static properties using contexts, and dynamic aspects using machines. It is evident that extending the modelling language and the proof rules of Event-B should not be done within those two constructs, for we must maintain a clear separation between that activity and modelling. Therefore, we introduce a new construct for the purpose of extending the Event-B language and its proof capabilities.

Outline. Sections 2 and 3 provide an overview of Event-B and its tool. We put particular emphasis on the proving infrastructure. Section 4 provides the general structure of the new construct. It describes the different elements that can be specified. Section 5 describes the relationship between the proposed construct, contexts and machines. Section 6 provides an insight into rewrite rules and the general issues related to them. We conclude by outlining our immediate objectives.

2 Structure of Event-B Mathematical Language

In the Event-B mathematical language [10], predicates and expressions are separate syntactic categories. Expressions are defined in terms of constants (e.g., 1), variables and operators (e.g., \cup). Expression operators can have expressions as arguments. They can also have predicates as arguments e.g., $(\lambda x \cdot P(x) \mid E(x))$ where $P(x)$ is a predicate and $E(x)$ is an expression.

Predicates, on the other hand, are built from basic predicates e.g., $x \in S$, logical connectives and quantifiers. Basic predicates take expressions as arguments e.g., $x \in S$ has x and S as arguments.

Expressions have a type which can be one of the following:

1. a basic set such as \mathbb{Z} or a carrier set supplied by the modeller in contexts;
2. a power set of another type;
3. a cartesian product of two types.

Expression operators have typing rules of the form:

$$\frac{\mathbf{type}(x_1) = \alpha_1 \dots \mathbf{type}(x_n) = \alpha_n}{\mathbf{type}(op(x_1, \dots, x_n)) = \alpha}.$$

Arguments of a basic predicate must satisfy its typing rule e.g., the typing rule for the basic predicate $finite(R)$ is:

$$\mathbf{type}(R) = \mathbb{P}(\alpha).$$

Alongside typing rules, expression operators have well-definedness predicates. $\mathbf{WD}(E)$ is used to denote the well-definedness predicate of expression E . Proof obligations are generated (if necessary) to establish the well-definedness of expressions appearing in models. To illustrate, we consider the expression $card(E)$ for which we have:

$$\mathbf{WD}(card(E)) \Leftrightarrow \mathbf{WD}(E) \wedge finite(E).$$

We limited our discussion here to the most relevant details that will be useful in later sections. Next, we present an overview of Event-B proving infrastructure. This is important as the implementation of the ideas to follow will be based on the existing Rodin architecture. For a general discussion of the overall architecture, we refer to [3].

3 The Proving Infrastructure

In this section, we outline the architecture of the Proof Manager. We also present an overview of the external provers integrated within the toolset.

The Proof Manager keeps track of the proofs associated with each proof obligation. Its internal architecture can be viewed as an integration of a *static* part and an *extensible* part. The extensible part generates the proof rules which can then be used to construct proofs. The Proof Manager can be extended by implementing more proof rules. Constructing and maintaining proofs (proof housekeeping) is the task of the static part of the Proof Manager. In what follows, we briefly describe the general ideas of the implementation [9] without delving into too much technical details.

Reasoners are objects that can generate concrete proof rules from a given sequent (and potentially other input e.g., a predicate in the case of the *cut rule*). These can be viewed as schemas for rules. The reasoner is successful if its rule schema is applicable to the given sequent. Reasoners need to be *logically valid* and *re-playable* (deterministic).

Proof Trees are objects based on Proof Tree Nodes. Each node is made of a sequent, a concrete proof rule (generated by a reasoner or could be *null*) and a list of children nodes (could be *null* if the proof rule is *null*). A node can be pending (its proof rule is *null*, and consequently the node has no children nodes) or non-pending otherwise. For each proof obligation, a proof tree is constructed where the sequent of its root node is the same as the obligation. If the proof rule is not *null*, it must be applicable to the sequent, and the children list corresponds to the result of the application.

Tactics are used to manipulate proof trees. There is a distinction between *basic tactics* and *tactical tactics*. Basic tactics act on a single node and include: pruning (successful for non-pending nodes) and reasoner application (successful if the reasoner is applicable). Tactical tactics include applying a tactic on all pending nodes, repeating tactic application and sequentially applying a list of tactics.

The Proof Manager can be *extended* with new reasoners and tactics. There is also an interface for encapsulating calls to external provers. The idea is to encapsulate prover calls as reasoner applications that are considered successful if the prover succeeds in proving the sequent. One drawback of this is that information about how the external prover managed the proof (e.g., the list of used hypotheses) is not always available to the Proof Manager. Here, we describe two provers that have been integrated into the proving infrastructure.

1. *The Predicate Prover (PP)*: this prover is built around a hierarchy of provers. It contains a decision procedure for propositional logic and a semi-decision procedure for first order logic. Another major component is the translator from set theory to first order logic. It is built in accordance with the set-theoretic construction outlined in the B-Book [2]. Statements involving complex set-theoretic operators are reduced to statements involving set membership only. The resulting statements can be considered as predicate logic statements with the set membership being left un-interpreted [5].
2. *The ML Prover (ML)*: is a rule-based prover used in the Logic Solver. The Logic Solver is the compiler-interpreter used for B. PP was originally developed to validate the many proof rules of ML. ML and PP are part of Atelier-B which provides the proving infrastructure for B.

3.1 Proving Modes

The Proof Manager can work in two modes: automatic mode and interactive mode. The user can interact with the Proof Manager by specifying what rules to apply to the current sequent. The Proof Manager uses *auto-tactics* and *post-tactics* when in automatic mode. Auto-tactics are tactics applied when a new proof obligation is generated in order to discharge, simplify or split it. They can also be invoked by the user. If the obligation is not yet discharged, the user can then interact with the Proof Manager. Post-tactics are tactics applied after each saved interaction in an attempt to discharge, simplify or split the resulting sequents. Finally, the Proof Manager can be extended with new auto-tactics and post-tactics.

4 The Theory Construct

In order to facilitate extending the mathematical language and the proof rules in a systematic and sound fashion, we propose a new construct, we may refer to as *Theory*. Theories will provide a mechanism by which the user can specify new *datatypes* (introducing new type constructors) and *definitions* to extend the mathematical language. It will also make it easy to extend the proof capabilities of the Event-B tool by specifying *rewrite* and *inference rules*. Proof obligations will be generated, and these need to be discharged to ensure the soundness of the theory with respect to some well-defined criteria. This new construct will be a step forward in terms of realising the many extensions proposed in [4].

Figure 1 describes the potential elements of the theory construct. In what follows, we discuss each of these. Our objective here is to explain what is required of the new construct. As our initial concern is *rewrite rules*, we glance over the elements and present examples when appropriate.

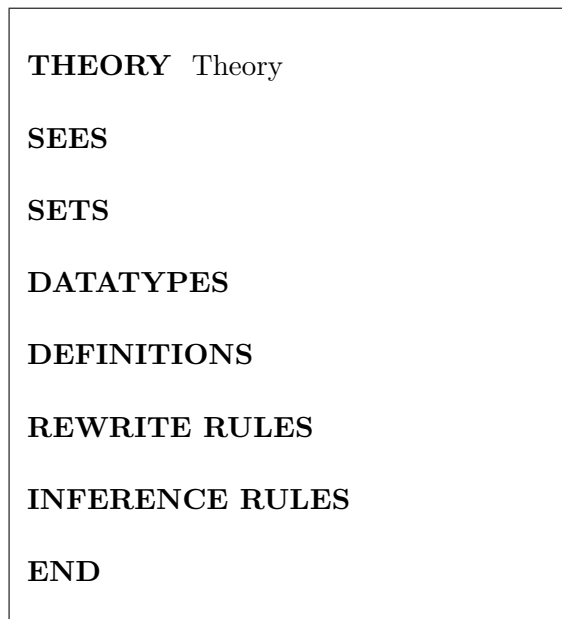


FIGURE 1: The Theory Construct

4.1 SEES

If a theory **T1** *sees* another theory **T0**, this has the effect of making datatypes, definitions, rewrite rules as well as inference rules specified in **T0** accessible to **T1** as if they were defined in **T1**. A theory can see multiple theories. Finally, theories should be organised according to a taxonomy based on structures, e.g., theories about sets, relations, integers, etc.

4.2 SETS

The *sets* provide the types (analogous to carrier sets in contexts) on which the theory is parameterised. All the datatypes, definitions and rules will be parameterised on those types. This follows a similar technique employed in PVS [8].

4.3 DATATYPES

The set theory of Event-B can be extended with a new *algebraic type* by defining a new *type construction operator* such as \times , and one or more *element construction operator* such as \mapsto . This is discussed further in [4].

4.4 DEFINITIONS

Currently, *basic predicates* (e.g., $E \in S$) and *operators* (e.g., \cup and \cap) are directly coded into the Rodin platform. The ultimate objective is to be able to specify new basic predicates and operators without writing Java code. The capability of specifying infix notation will be important. This element of the theory construct can be used for this very purpose. For more on this, we refer the reader to [4].

4.5 REWRITE RULES

Rewrite rules can be used to transform expressions and predicates to equivalent forms. This element of the theory construct will be used to specify two types of rewrite rules: *conditional* and *unconditional* rules. This is discussed in more depth in §6.

4.6 INFERENCE RULES

Inference rules are used to construct proofs of sequents. An inference rule has a list of *antecedent* sequents and one *consequent* sequent. It has the following form:

$$\frac{A_1, \dots, A_n}{C} \mathbf{r}$$

where A_i are the antecedents and C is the consequent. This element of the theory construct can be used to add such rules. Proof obligations will be generated in order to validate them.

An example inference rule (implemented in Rodin) is the following:

$$\frac{\vec{H}, P \vdash R \quad \vec{H}, Q \vdash R}{\vec{H}, P \vee Q \vdash R} \mathbf{case}$$

where \vec{H} is a list of predicates. This type of rules cannot be expressed in the mathematical language of Event-B as it requires the use of predicate variables. It is easy to extend the language syntax with predicate variables (should only be used within theories, and are not available for modelling). However, the proof obligations generated to validate these inference rules may require the use of higher-order provers such as Isabelle/HOL [1] or PVS [8].

5 Theories, Contexts and Machines

A theory **T1** can see another theory **T0**; this makes all that is specified in **T0** accessible to **T1**. All the theories seen by **T0** will also be available to **T1**. On the subject of the relationship between the two modelling constructs (contexts and machines) and theories, *project-wide* and *tool-wide* accessibility of theories will be investigated.

6 Rewrite Rules

Rewrite rules are used in the same way as inference rules to construct proofs of sequents. To illustrate this point, the following rewrite rule:

$$\{x \cdot x \in S \mid x\} \hat{=} S \quad (R_\Delta)$$

with x not free in S , can be used in the following way:

$$\frac{\vec{H}, P(S) \vdash R}{\vec{H}, P(\{x \cdot x \in S \mid x\}) \vdash R} \quad (I_{\Delta_h})$$

and

$$\frac{\vec{H} \vdash R(S)}{\vec{H} \vdash R(\{x \cdot x \in S \mid x\})} \quad (I_{\Delta_g})$$

where \vec{H} is a list of predicates, P and R are predicates and x is not free in S .

Note. For the time being, we use the following understanding of (I_{Δ_g}) and (I_{Δ_h}) . $P(\{x \cdot x \in S \mid x\})$ (under the line) denotes a predicate P with a sub-term $\{x \cdot x \in S \mid x\}$. Given $P(\{x \cdot x \in S \mid x\})$, $P(S)$ (above the line) denotes P with a specific occurrence of $\{x \cdot x \in S \mid x\}$ replaced by S .

The rewrite rule representation is simpler and more compact than that of inference rules. In this section, we analyse what makes a valid rewrite rule. Further research will be carried out on issues related to rules *applicability*. Here, we use the symbol $\hat{=}$ to denote *rewriting*.

We recall that the Event-B mathematical language distinguishes between predicates and expressions as two separate syntactic categories. In the next two sub-sections, we present two types of rewrite rules. The last sub-section deals with the general issues related to the applicability of rewrite rules.

6.1 Unconditional Rewrite Rules

Definition 6.1. An unconditional rewrite rule is of the form

$$lhs \hat{=} rhs$$

where lhs and rhs are formulas of the same syntactic class in the Event-B mathematical language, and rhs only contains free variables from lhs . If lhs is an expression, then

$$\mathbf{type}(lhs) = \mathbf{type}(rhs).$$

Definition 6.2. A type environment of a rewrite rule r , denoted $env(r)$, provides typing conditions for all the free variables occurring in the rewrite rule.

Definition 6.3. An unconditional rewrite rule r of the form $lhs \hat{=} rhs$ with a type environment $env(r)$, is *valid* provided the following sequents are valid:

1. $[hyp, \mathbf{WD}(lhs) \vdash \mathbf{WD}(rhs)]$ where hyp includes typing conditions according to $env(r)$.
2. $[hyp, \mathbf{WD}(lhs) \vdash lhs = rhs]$ if lhs is an expression, or;

3. $[hyp, \mathbf{WD}(lhs) \vdash lhs \Leftrightarrow rhs]$ if lhs is a predicate.

Condition (1) simply states that the right hand side of a rewrite rule should not be less well-defined than its left hand side. That is, a valid rewrite rule preserves well-definedness. The type environment env is important especially with rules that involve supertypes.

6.2 Conditional Rewrite Rules

Rewrite rules can be conditional. As an example, consider

$$\begin{aligned} card(i..j) &\hat{=} \begin{array}{l} i \leq j : j - i + 1 \\ i > j : 0 \end{array} \quad (CR_{\Delta}) \end{aligned}$$

where $i \in \mathbb{N}$ and $j \in \mathbb{N}$. This rule simply states that $card(i..j)$ can be rewritten to $j - i + 1$ if $i \leq j$ and 0 otherwise. We have the following definitions:

Definition 6.4. A conditional rewrite rule is of the form

$$\begin{aligned} lhs &\hat{=} C_1 : rhs_1 \\ &\dots \\ &C_n : rhs_n \end{aligned}$$

where:

1. lhs and rhs_i (for all $i \in 1..n$) are formulas of the same syntactic class in the Event-B mathematical language.
2. C_i (for all $i \in 1..n$) are predicates.
3. $\mathbf{type}(lhs) = \mathbf{type}(rhs_i)$ (for all $i \in 1..n$) if lhs is an expression.
4. C_i (for all $i \in 1..n$) only contain free variables from lhs .
5. rhs_i (for all $i \in 1..n$) only contain free variables from lhs .

Definition 6.5. A conditional rewrite rule r of the form

$$\begin{aligned} lhs &\hat{=} C_1 : rhs_1 \\ &\dots \\ &C_n : rhs_n \end{aligned}$$

with a type environment $env(r)$, is *valid* provided the following sequents are valid:

1. $[hyp, \mathbf{WD}(lhs) \vdash \mathbf{WD}(C_i)]$ for all $i \in 1..n$ and hyp includes typing conditions according to $env(r)$.
2. $[hyp, \mathbf{WD}(lhs), C_i \vdash \mathbf{WD}(rhs_i)]$ for all $i \in 1..n$.
3. $[hyp, \mathbf{WD}(lhs), C_i \vdash lhs = rhs_i]$ for all $i \in 1..n$ if lhs is an expression, or;
4. $[hyp, \mathbf{WD}(lhs), C_i \vdash lhs \Leftrightarrow rhs_i]$ for all $i \in 1..n$ if lhs is a predicate.

6.3 Applying Rewrite Rules

Consider the following expression in the mathematical language

$$\{x \cdot x \in \mathbb{N} \cup \{x\} \mid x\}.$$

Applying rule (R_Δ) to the previous expression yields $\mathbb{N} \cup \{x\}$. This is evidently invalid because the *side condition* that x is not free in S , is not respected. Therefore, an *application* of a valid rule is *valid* only when its side conditions are met.

6.3.1 Applying Unconditional Rewrite Rules

Generally speaking, applying the valid rule

$$lhs \hat{=} rhs$$

to the sequents

$$\frac{\vec{H}, P(lhs) \vdash R}{\vec{H} \vdash R(lhs)}$$

yields the following inference rules

$$\frac{\frac{\vec{H}, P(rhs) \vdash R}{\vec{H}, P(lhs) \vdash R}}{\vec{H} \vdash R(rhs)}}{\vec{H} \vdash R(lhs)}$$

respectively, given that the side conditions of the rule hold for lhs .

6.3.2 Applying Conditional Rewrite Rules

Applying conditional rewrite rules is more complex given their more elaborate structure. Similarly to unconditional rules, they can be applied to both the goal and hypotheses of a sequent. In our discussion, we distinguish between hypotheses and goal application of a rule. Consider the valid rule:

$$lhs \hat{=} C_1 : rhs_1 \\ \dots \\ C_n : rhs_n$$

1. *Hypotheses Application*: Applying the rule to the sequent

$$\vec{H}, P(lhs) \vdash R$$

yields the inference rule

$$\frac{\vec{H}, C_1, P(rhs_1) \vdash R \dots \vec{H}, C_n, P(rhs_n) \vdash R}{\vec{H}, P(lhs) \vdash R}$$

given that the side conditions of the rule hold for lhs .

2. *Goal Application*: Applying the rule to the sequent

$$\vec{H} \vdash R(lhs)$$

yields the following inference rule

$$\frac{\vec{H}, C_1 \vdash R(rhs_1) \dots \vec{H}, C_n \vdash R(rhs_n)}{\vec{H} \vdash R(lhs)}$$

given that the side conditions of the rule hold for lhs .

6.3.3 Side Conditions

Consider the sequent

$$\vec{H}, \overbrace{\forall i \cdot i \in \mathbb{N} \Rightarrow (\exists j \cdot j \in \mathbb{N} \wedge \underbrace{card(i..j)}_{lhs} = 2)}^{P(lhs)} \vdash R.$$

Clearly, rule (CR_Δ) cannot be applied in the way we outlined in §6.3.2. This is due to the fact that i and j are bound within $P(lhs)$, and cannot be taken out to form the conditions $i \leq j$ and $i > j$. The same issue arises with the following sequent:

$$\vec{H} \vdash \overbrace{\forall i \cdot i \in \mathbb{N} \Rightarrow (\exists j \cdot j \in \mathbb{N} \wedge \underbrace{card(i..j)}_{lhs} = 2)}^{R(lhs)}.$$

It is tempting to think that if i and j were defined in \vec{H} , it would be possible to apply the rule. However, in the general case, introducing the new conditions C_i (for all $i \in 1..n$) poses further challenges in terms of maintaining the well-definedness of the involved sequents. One objective of this research is to clarify this type of side conditions. The possibility of automating applicability checking (i.e. relieving the theory developer from specifying side conditions) will also be investigated. Other types of side conditions include (exemplified by rules):

- $S \subseteq Ty \hat{=} \top$ with the condition that Ty is a type expression. This side condition can be enforced by making the pattern matching engine observe the typing environment of the rule.
- $S \subseteq \{x \cdot P(x) \mid x\} \hat{=} \forall y \cdot y \in S \Rightarrow P(y)$ with the condition that y is not free in S and $\{x \cdot P(x) \mid x\}$. This condition can be enforced by choosing a fresh name for y everytime the rule is applied.

7 Important Considerations

- The theory developer's job can be made more effective by employing an automatic type inference procedure. However, this may be a little out of reach. *In-place* typing of free identifiers (using the theory as well as other built-in sets) can greatly ease the task of specifying rewrite rules. For

instance, assuming the availability of a set T , one can specify the rewrite rule $(S : \mathbb{P}(T)) \cap T \cong S$ where ":" denotes the in-place typing of S .

- Side conditions are of extreme importance to the soundness of the prover. Some side conditions can be automatically enforced, whereas others may require the theory developer to specify them as a part of defining the rule. Research will be carried out to address this particular issue.
- Another consideration of a more practical nature is the importance of distinguishing between *theory development* and *theory deployment*. Theory development refers to the activity of specifying the theory elements, the static checking, proof obligation generation as well as discharging the resulting proof obligations (for rewrite rules, these are discussed in 6.1 and 6.2). Theory deployment refers to the subsequent step of making statically checked and verified theories available to the prover. This is analogous to the process by which Java libraries are developed and deployed. The use of a similar approach to Java's **classpath** will be investigated.
- In addition to side conditions, there are other issues related to rewrite rules applicability. Some rewrite rules make formulas within sequents grow considerably. Multiplication distribution over addition, for instance, is an example of such *expansive* rules. These rules *should* not be applied automatically as they do not *simplify* formulas, they expand them. Permutative rules (e.g., commutativity law for addition) can also be dangerous to apply in an automatic fashion. These types of rules, however, should be candidate for *manual* (interactive) application.

8 Related Work

The theorem prover Isabelle [11] provides a fragment of higher order logic that can be used as a meta-logic for specifying other logics. Isabelle/HOL refers to the specialisation of Isabelle to higher order logic, whereas Isabelle/ZF is the one for ZF set theory. The syntax as well as the rules of the derived logics are specified in the meta-logic [12]. This provides an effective mechanism for extending the derived logics.

In Isabelle/HOL, the simplifier can be augmented with new simplification rules using the meta-logic directive [**simp**], e.g.,

```
theorem thm1[simp]: "rev(rev xs) = xs"
```

Note that in Isabelle/HOL theories the meta-logic and the object logic syntax co-exist. This is however undesirable for Event-B constructs since we want to maintain a clear separation between the activity of modelling and the activity of extending the logic of modelling.

9 Summary

In this report, we discussed possible extensions to the Event-B language and the Rodin tool. Our work will initially focus on adding the facility to extend the prover's rule base with rewrite rules (conditional and unconditional) within the existing architecture of Rodin. We will then elaborate the treatment and implementation of general inference rules and new operator definitions in Rodin. These will require the addition of predicate variables to the mathematical language and the ability for the user to define new mathematical operators. The proposed Theory construct will be a step forward in terms of realising the many extensions envisaged in [4]. The proof obligations generated from the new construct will ensure that the soundness of the prover is not compromised by adding new rules.

References

- [1] *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, London, UK, 2002.
- [2] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.
- [3] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *International Conference on Formal Engineering Methods (ICFEM)*, 2006.
- [4] J.-R. Abrial, M. Butler, M. Schmalz, S. Hallerstede, and L. Voisin. Proposals for Mathematical Extensions for Event-B, 2009.
- [5] Jean R. Abrial and Dominique Cansell. *Click'n Prove: Interactive Proofs within Set Theory*. 2003.
- [6] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inf.*, 77(1-2):1–28, 2007.
- [7] Michael Butler and Stefan Hallerstede. The Rodin Formal Modelling Tool. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, December 2007.
- [8] Ricky W Butler. An Elementary Tutorial on Formal Specification and Verification Using PVS. Technical report, 1993.
- [9] Farhad Mehta. *Proofs for the Working Engineer*. PhD Thesis, ETH Zurich, 2008.
- [10] Christophe Metayer and Laurent Voisin. The Event-B Mathematical Language, 2009.
- [11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelles logics: Hol.
- [12] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989.