Practical Theory Extension in Event-B

Michael Butler[1], Issam Maamria[2]

[1] Electronics and Computer Science, University of Southampton, UK
[2] UBS, UK

Abstract. The Rodin tool for Event-B supports formal modelling and proof using a mathematical language that is based on predicate logic and set theory. Although Rodin has in-built support for a rich set of operators and proof rules, for some application areas there may be a need to extend the set of operators and proof rules supported by the tool. This paper outlines a new feature of the Rodin tool, the theory component, that allows users to extend the mathematical language supported by the tool. Using theories, Rodin users may define new data types and polymorphic operators in a systematic and practical way. Theories also allow users to extend the proof capabilities of Rodin by defining new proof rules that get incorporated into the proof mechanisms. Soundness of new definitions and rules is provided through validity proof obligations.

1 Introduction

Abrial's Event-B is a formalism for refinement-based development of discrete event systems [1]. Its deployment is supported by the Rodin toolset which includes proof obligation generation and verification through a collection of mechanical provers [2]. An Event-B machine consists of a collection of variables, invariants on those variables and a collection of guarded events that may update the machine variables. An Event-B development consists of a collection of machines linked by refinement and refinement is verified through proof obligations for preservation of gluing invariants between abstract and concrete variables. Abrial's book [1] contains a range of refinement case studies in Event-B and many other Event-B case studies have been undertaken by academic researchers (e.g., wiki.event-b.org/index.php/Event-B_Examples) and by industry (e.g.[17]).

In Event-B, types, axioms, invariants, guards and actions may be defined using a set-theoretic mathematical language. While the mathematical language supported by the Rodin tool is rich (including operators on integers, sets, relations and functions), there is always a need to extend the mathematical language to broaden further the expressivity of the modelling notation. Because proof plays such a central role in the Event-B approach, hand-in-hand with any new mathematical operator definitions, there is a need to support proofs involving those operators. The need for an extensible mathematical language and theories in Event-B was envisaged by Abrial [3] where the need for a generic extension mechanisms, as found in languages such as PVS [15] and Isabelle [14], was identified. We refer to the process of defining new mathematical types and operators, together with associated proof rules, as *theory extension.*

As well as supporting a rich mathematical language, the Rodin tool provides a range of automatic and interactive mechanical provers for proving obligations expressed in that language. In the earlier releases of Rodin, there were no mechanisms available for users to define new operators nor to extend the mechanical provers with new proof rules. Such extensions could only be undertaken by modifying the tool itself. This paper outlines recent work that overcomes this limitation, making theory extension part of the modelling process that can be undertaken in a systematic way without having to modify the Rodin tool. This has been achieved by adding a major new construct, the *Theory component*, to Event-B. This feature dis available in Rodin as a plug-in[3].

This paper outlines the theory component, how it enables the process of theory extension in the Event-B language and how it is supported in the Rodin tool. By allowing commonly-occurring structures to be captured as generic types, operators and proof rules, we allow these structures to be reused, thus reducing modelling and proof effort in the longer term. Genericity is achieved by supporting the definition of polymorphic operators and proof rules. These polymorphic operators and rules are instantiated with more specific types in modelling and proof, e.g., an operator with an argument of type $\mathbb{P}(\alpha)$ is instantiated with an argument of type $\mathbb{P}(\mathbb{Z})$.

In the earlier releases of Rodin, types were defined using set theory (power set and cartesian product) and there was no support for inductive data types (such as lists or trees). The new theory component supports the definition of inductive data types, along with recursive operator definitions and proof by induction.

It is important that any theory extensions are sound. Verifying soundness of theories is achieved through the definition of soundness proof obligations. When a modeller defines a new theory, soundness proof obligations are generated and then proved with the existing Rodin framework. This follows the standard Event-B approach where consistency of models and correctness of refinement between models is verified by discharging standard proof obligations.

This paper is structured around the main elements that may be contained in a theory, namely, operator definitions, datatype definitions, rewrite rules and inference rules. After presenting the theory component, we address important related work on proof in Event-B and mechanised theorem proving in general (Section 10). We start with a brief overview of the core Event-B mathematical language.

2 Event-B Mathematical Language

In the Event-B mathematical language, *expressions* and *predicates* are separate syntactic categories. Expressions are defined using literals (e.g., 1), constants, variables and polymorphic operators (e.g., set union). Expression operators can have expressions as arguments – such an operator *op* with arguments x_1 to x_n is written in the form $op(x_1, \ldots, x_n)$. Operators can also have predicates as

[3] wiki.event-b.org/index.php/Theory_Plug-in

arguments, e.g., $(\lambda x \cdot P(x) \mid E(x))$ where $P(x)$ is a predicate and $E(x)$ is a expression.

Predicates, on the other hand, are built from predicate operators (e.g., $\in$, $\subseteq$), logical connectives and quantifiers. Predicate operators take expressions as arguments e.g., $x \in S$ has x and S as arguments.

Expressions have a *type* and we use α to denote types. Types are constructed as follows:

1. a basic set such as $\mathbb{Z}$ or a carrier set defined in an Event-B context,
2. a power set of a type, written $\mathbb{P}(\alpha)$,
3. a cartesian product of two types, written $\alpha_1 \times \alpha_2$.

The type of a expression operator *op* with arguments $x_1 \ldots x_n$ is defined using typing rules of the form:

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

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

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

Note that types can be used as set expressions within the Event-B mathematical language, e.g., $\mathbb{Z}$ is both a type and a set expression. Furthermore, the type operators ($\mathbb{P}$ and $\times$) have a second role as operators on sets. For example, suppose T is a basic type and S is a subset of T, then the expression $\mathbb{P}(S)$ is a valid expression. For set expressions S and R, we have

$$\begin{aligned} R \in \mathbb{P}(S) &\quad\Leftrightarrow\quad R \subseteq S \\ o_1 \mapsto o_2 \in S \times R &\quad\Leftrightarrow\quad o_1 \in S \;\wedge\; o_2 \in R \end{aligned}$$

Alongside typing rules, expression operators have *well-definedness* conditions. $\mathbf{WD}(E)$ 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:

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

Functions versus operators It is instructive to consider the relationship between operators and function application in Event-B. An Event-B function $f \in A \rightarrow\!\!\!\!+ B$ is a special case of a set of pairs so the type of f is $\mathbb{P}(\mathbf{type}(A) \times \mathbf{type}(B))$. The functionality of f is an additional property defined by a predicate specifying a uniqueness condition:

$$\forall x, y, y' \cdot x \mapsto y \in f \;\wedge\; x \mapsto y' \in f \;\Rightarrow\; y = y'$$

The domain of f, written $dom(f)$, is the set $\{ x \mid \exists y \cdot x \mapsto y \in f \}$. Application of f to x is written $f(x)$ which is well-defined provided $x \in dom(f)$.

It is important to note that f is not itself an operator, it is simply a set expression. The operator involved here is implicit – it is the *function application* operator that takes two arguments, f and x. To make the operator explicit, function application could have been written as $apply(f, x)$, where $apply$ is the operator and f and x are the arguments. However, in the Rodin tool, the shorthand $f(x)$ must be used.

Variables in the mathematical language are typed by set expressions. This means, for example, that a variable may represent a function since a function is a special case of a set (of pairs). Variables may not represent expression operators or predicates in the mathematical language. This means that, while we can quantify over sets (including functions), we cannot quantify over operators or predicates in Event-B.

3 Theory component

Models in Event-B are specified by means of *contexts* (static properties of a model) and *machines* (dynamic properties of a model). A theory is a new kind of Event-B component for defining theories that may be independent of any particular models. A theory is the means by which the mathematical language and mechanical provers may be extended.

We describe the overall structure of Event-B theories. A theory component has a name, a list of global type parameters (global to the theory), and an arbitrary number of definitions and rules:

theory *name*
type parameters $T_1, \ldots, T_n$

$\{\ \langle$ *Predicate Operator Definition* $\rangle$
$\mid\ \langle$ *Expression Operator Definition* $\rangle$
$\mid\ \langle$ *Data Type Definition* $\rangle$
$\mid\ \langle$ *Rewrite Rule* $\rangle$
$\mid\ \langle$ *Inference Rule* $\rangle\ \}$

An Event-B theory has a name which identifies it. A theory can have an arbitrary number of type parameters which are pair-wise distinct, in which case the theory is said to be polymorphic on its type parameters. In the following it is important to recall that the mathematical language has two syntactic categories, *predicates* and *expressions*. We look at each form of definition and rule in turn in the following sections.

4 Defining new predicate operators

A new Event-B polymorphic operator can be defined by providing the following information:

1. *Parser Information*: this includes the syntax, the notation (infix or prefix), and the syntactic class (expression or predicate).
2. *Type Checker Information*: this includes the types of the child arguments, and the resultant type if the operator is a expression operator.
3. *Prover Information*: this includes the well-definedness of the operator as well as its definition which may be used to reason about it.

A predicate operator defines a property on one or more expressions. For example, the predicate x *divides* y holds when x is an integer divisor of y. This predicate is defined in the following way:

predicate *divides*
  infix
  args $x : \mathbb{Z},\ y : \mathbb{Z}$
  condition $x \in \mathbb{N} \ \wedge\ y \in \mathbb{N}$
  definition $\exists a \cdot y = a \times x$

This declares a new operator *divides*. It is declared as infix with two arguments x and y both of type $\mathbb{Z}$. This declaration makes the predicate E *divides* F syntactically valid for integer expressions E and F. The condition specifies a well-definedness condition – in this case that x and y must be naturals ($\mathbb{N} \subseteq \mathbb{Z}$). The final clause provides the definition of x *divides* y. That is, we have

$$x \textit{ divides } y \quad \Leftrightarrow \quad \exists a \cdot y = a \times x$$

A new predicate operator may be infix or prefix. For example, if *divides* had been declared as prefix, then $\textit{divides}(E, F)$ would become syntactically valid. An infix predicate must have exactly two arguments.

Though in the above case the arguments are typed with the predefined type $\mathbb{Z}$, in general arguments may be typed using some of the type parameters defined for the theory which makes the predicate polymorphic on those type parameters.

The general structure of a basic predicate definition is as follows:

predicate *Identifier*
  (**prefix** | **infix**)
  args $x_1 : \alpha_1,\ \ldots\ , x_n : \alpha_n$
  condition $P(x_1, \ldots, x_n)$
  definition $Q(x_1, \ldots, x_n)$

5 Defining new expression operators

While a predicate operator forms a predicate from a number of expressions, an operator forms an expression from a number of expressions. We consider an example involving the representation of sequences as functions whose domains are contiguous ranges of naturals starting at 1, i.e., functions from $(1..n) \to T$. The *seq* operator takes a set s and yields all sequences whose members are in s:

operator seq
  prefix
  args $s : \mathbb{P}(T)$
  definition $\{\ f, n\ \cdot\ f \in (1..n) \to s\ \mid\ f\ \}$

Here seq is declared to be a prefix operator with a single argument represented by s of type $\mathbb{P}(T)$. Since T is a type parameter, this means that seq is polymorphic on type T. The final clause defines the expression $\mathit{seq}(T)$ in terms of the existing expression language. The definition means we have that:

$$\mathit{seq}(s)\ =\ \{\ f, n\ \cdot\ f \in (1..n) \to s\ \mid\ f\ \}$$

Note that the result type of an operator is inferred from the definition. In this case, the type of $\mathit{seq}(s)$ is $\mathbb{P}(\mathbb{Z} \leftrightarrow T)$, that is, a set of relations[4] between integers and the polymorphic type T. The following is an example of another prefix operator size that yields the size of a sequence:

operator size
  prefix
  args $m : \mathbb{Z} \leftrightarrow T$
  condition $m \in \mathit{seq}(T)$
  definition $\mathit{card}(m)$

Here, the well-definedness condition is stronger than the type declaration on m, requiring that m is an element of $\mathit{seq}(T)$.

Proof obligations are generated to verify the well-definedness of definitions. The validity proof obligation for operator definitions ensures that expressions involving that operator are well-defined. The proof obligation specifies that, assuming the arguments are well-defined and the explicit well-definedness condition for the operator holds, then the definition itself is well-defined. An important aspect of defining an operator is the well-definedness condition to be used. A simple strategy may use the well-definedness of the operator's direct definition. An advantage of a user-supplied condition is the possibility of strengthening well-definedness conditions to simplify proofs. In order to ensure that a supplied condition is in fact stronger than the default (i.e., the one inferred from the direct definition), proof obligations are generated.

For example, the definition of size leads to a proof obligation requiring that $\mathit{card}(m)$ is well-defined whenever $m \in \mathit{seq}(T)$. This is provable from the condition that m is a sequence since any element of $\mathit{seq}(T)$ has a finite domain and $\mathit{card}(m)$ is well-defined when m is finite (in Section 8 this is expressed as an inference rule).

Operators may be infix in which case they may be declared to be associative and commutative. For example, the concatenation operator on sequences, declared as follows, is associative:

[4] $\alpha_1 \leftrightarrow \alpha_2$ is shorthand for $\mathbb{P}(\alpha_1 \times \alpha_2)$

operator $\frown$
  infix assoc
  args $m : \mathbb{Z} \leftrightarrow T,\ \ n : \mathbb{Z} \leftrightarrow T$
  condition $m \in seq(T)\ \wedge\ n \in seq(T)$
  definition $m\ \cup\ \{\ i, x\ \cdot\ i \mapsto x \in n \mid size(m) + i \mapsto x\ \}$

The general form of an operator definition is as follows:

operator *Identifier*
  (**prefix** | **infix**) [**assoc**] [**commut**]
  args $x_1 : \alpha_1,\ \ \ldots,\ x_n : \alpha_n$
  condition $P(x_1, \ldots, x_n)$
  definition $E(x_1, \ldots, x_n)$

A conditional expression of the form, $COND(p, e1, e2)$, may be used to define operators (p is a predicate while $e1$ and $e2$ are expressions). For example, the *max* operator, that yields the maximum of two integers, is defined using a conditional expression as follows:

operator *max*
  infix assoc commut   *// declare max to be associative and commutative*
  type parameters T
  args $x : \mathbb{Z},\ \ y : \mathbb{Z}$
  definition $COND(\ x \geq y\ ,\ x\ ,\ y\ )$

Declaring an operator to be associative and commutative gives rise to proof obligations to verify these properties. Since the Rodin provers automatically make use of commutativity and associativity properties of operators, to avoid circular proofs, the proof obligations must be specified in terms of the operator definition rather than the operator itself, e.g., the above declaration of *max* will give rise to the following commutativity proof obligation:

$$COND(\ x \geq y\ ,\ x\ ,\ y\ )\quad =\quad COND(\ y \geq x\ ,\ y\ ,\ x\ )$$

6 Defining new datatypes

A new datatype declaration defines a new type constructor together with constructor and destructor functions for elements of the new type. For example the usual inductive list type constructor is defined as follows:

datatype *List*
  type args T
  constructors
    nil
    $cons(\ head : T,\ tail : List(T)\ )$

This defines

- A new type constructor *List*. $List(T)$ becomes a type for any type T.
- A set operator *List*. $List(s)$ is a set expression – the set of lists whose members are in set s
- Two constructors *nil* and *cons*
- Two destructors *head* and *tail*
- An induction principle on *List*

The general form of an inductive data definition is as follows:

datatype *Ident*
  type args $T_1 \dots T_n$
  constructors
    $c_1(\ d_1^1 : \alpha_1^1,\ \cdots,\ d_1^j : \alpha_1^j\ )$
    $\vdots$
    $c_m(\ d_m^1 : \alpha_m^1,\ \cdots,\ d_m^k : \alpha_m^k\ )$

Constructor and destructor names must be distinct. Types in Event-B are assumed to be non-empty, and this must hold for datatypes. As such, each newly defined datatype must have a base constructor, i.e., a constructor that does not refer to the datatype being defined. Here each α_j^i is a type that may include occurrences of the type being defined $Ident(T_1 \dots T_n)$. If α_j^i does include occurrences of $Ident(T_1 \dots T_n)$, then α_j^i must be *admissible*, i.e., α_j^i is $Ident(T_1 \dots T_n)$ or is formed from a cartesian product or an existing inductive data type. Without the admissibility check, the datatype cannot be constructed. In the context of Event-B, the admissibility check rules out the following datatype definition

$$t(\alpha) \quad ::= \quad C_1 \quad | \quad C_2(\mathbb{P}(t))$$

since there is no injective function of type $\mathbb{P}(t) \rightarrow t$ by Cantor's theorem.

Proof by induction is supported in Rodin though a special reasoner that generates an induction scheme for any particular hypothesis or goal of a proof.

6.1 Pattern matching with datatypes

When defining basic predicates and operators on inductive types, the usual pattern matching may be used. For example the *size* function on inductive lists is defined as follows:

operator *size*
  prefix
  args $a : List(T)$
  definition

match a	
nil	0
$cons(x, b)$	$1 + size(b)$

Since a is of type $List(T)$ the argument a may be matched against each of the constructors for $List$.

predicate *member*
  prefix
  args $x : T,\ a : List(T)$
  definition

match a	
nil	$false$
$cons(y, b)$	$x \neq y \Rightarrow member(x, b)$

Pattern matching and conditional expressions can be used together. Here is an example of an operator definition that removes duplicates in a list and uses a conditional expression:

operator *remdup*
  prefix
  args a
  condition $a \in List(T)$
  definition

match a	
nil	nil
$cons(x, b)$	$COND(\ member(x, b)\ ,\ remdup(b)\ ,\ cons(x, remdup(b))\ )$

Type constructors as set operators In Section 2 we stated that type constructors ($\mathbb{P}$ and $\times$) also serve as set expression operators. Data type constructors can also be used as set operators. For example, suppose S is a set expression of type $\mathbb{P}(T)$, then $List(S)$ is a set expression specifying the set of inductive lists whose elements all come from S. $List(S)$ satisfies the following properties:

$$nil \in List(S)$$
$$cons(x, t) \in List(S) \quad \Leftrightarrow \quad x \in S\ \wedge\ t \in List(S)$$

7 Rewrite Rules

A rewrite rule is used in automatic or interactive proof to rewrite an expression or predicate in order to faciliate proof. A rewrite involves a left hand pattern and one or more right hands. Each right hand may be guarded by some condition. For example, the following rewrite rule defines two ways of rewriting the expression $card(i..j)$ depending on a condition on i and j ($i..j$ is the set of integers between i and j):

rewrite *CardIntegerRange*
  auto manual complete
  vars $i : \mathbb{Z},\quad j : \mathbb{Z}$
  lhs $card(i..j)$

rhs

$i \leq j$	$j - i + 1$
$i > j$	0

This rule states that $card(i..j)$ may be rewritten to $j-i+1$ if $i \leq j$ and rewritten to 0 if $i > j$. The above declaration means that the rewrite rule can be used in automatic and interactive proof modes. The 'complete' declaration means that the disjunction of the guards must be true. The variables of the rule (i and j) serve as meta variables that can be matched with any expression of the appropriate type.

The general form of a rewrite rule for expressions is as follows (where the lhs and rhs are expressions):

rewrite *Name*
  [**auto**] [**manual**] [**complete**]
  vars $x_1 : \alpha_1, \ \ldots, \ x_n : \alpha_n$
  condition $P(x_1, \ldots, x_n)$
  lhs $E(x_1, \ldots, x_n)$
  rhs

$Q_1(x_1, \ldots, x_n)$	$E_1(x_1, \ldots, x_n)$
$\vdots$	$\vdots$
$Q_m(x_1, \ldots, x_n)$	$E_m(x_1, \ldots, x_n)$

A number of validity obligations are required to ensure the soundness of a rewrite rule:

- The conditions must be well-defined: $P \wedge WD(E) \ \Rightarrow WD(Q_i)$
- Each rhs must be well-defined: $P \wedge WD(E) \wedge Q_i \ \Rightarrow WD(E_i)$
- Each rhs must equal the lhs: $P \wedge Q_i \ \Rightarrow E = E_i$

In addition, if the rule is declared to be case complete, then a completeness condition is required ($P \ \Rightarrow \ Q_1 \vee \cdots \vee Q_m$).

The general form of a rewite for predicates is similar (with the lhs and rhs being predicates). The validity obligations are similar to those for expression rewrites.

8 Inference Rules

An inference rule has a list of hypothesis and a consequent. It is parameterised by one or more variables. For example, the following inference rule has two hypotheses and a consequent that may be inferred from the hypotheses:

rule *FiniteSeq*
  vars $s : \mathbb{P}(T), \ m : \mathbb{Z} \leftrightarrow T$
  given
    $m \in seq(s)$
  infer
    $finite(m)$

Here is another inference rule showing that sequence concatenation is closed for elements of $seq(s)$:

```
rule  Concat1
    vars   $s, m, n$
    given  $s \subseteq T$
           $m \in seq(s)$
           $n \in seq(s)$
    infer  $m \frown n \in seq(s)$
```

The general form of an inference rule is as follows:

```
rule  Name
    vars   $x_1, \ldots, x_n$
    given
           $P_1(x_1, \ldots, x_n),\ \ldots,\ P_m(x_1, \ldots, x_n)$
    infer
           $Q(x_1, \ldots, x_n)$
```

A number of validity obligations are required to ensure the soundness of an inference rule:

- The rule must be well-defined: $WD(P_1 \wedge \cdots \wedge P_m) \Rightarrow WD(Q)$
- The rule must be provable: $P_1 \wedge \cdots \wedge P_m \Rightarrow Q$

Using Inference Rules Inference rules can be used in a backward style as well forward style. If used in backward style, the prover discharges or splits the goal. If applied in a forward style, more hypotheses get generated.

9 Axiomatic Definitions

The constructs we have outlined so far in this paper allow for direct definitions of new operators, inductive data types and recursive definitions of new operators over inductive types. For some types and operators this is not enough. For example, theories of integers and reals are typically defined axiomatically. That is, the types are not inductive data types, rather they are assumed to exist axiomatically and are supplied with a set of basic operators whose properties are defined axiomatically. In the case of integers and reals, these operators are arithmetic operators that satisfy algebraic properties such as commutativity, associativity, distribution and simplification properties. We are currently adding support for axiomatic types and operators to the theory extension mechanism in Rodin[5]. One difference with the direct and recursive definitions is that we do not define full soundness obligations for the axiomatic definitions. For now, we assume that the theory modeller has ensured the soundness of a collection of axioms externally. We can define basic well-definedness obligations on the axioms however.

[5] http://wiki.event-b.org/index.php/Theory_Plug-in

Event-B already supports lambda expressions of the form $(\lambda x \cdot P(x) \mid E(x))$ where P is a constraint on x and the lambda function yields $E(x)$ for x satisfying P. Axiomatic definitions in theories also allow us to mimic other binder operators by defining operators on lambda expressions. For example, consider summation over a collection of integers that sums each expression $E(x)$ for every x satisfying predicate P:

$$\Sigma x \cdot P(x) \mid E(x)$$

This can be represented by defining a *SIGMA* operator on functions with the form

$$\mathit{SIGMA}(\lambda x \cdot P(x) \mid E(x))$$

SIGMA is defined as a new operator on functions satisfying the following axioms:

$$\begin{array}{rcll} \mathit{SIGMA}(\varnothing) & = & 0 & \\ \mathit{SIGMA}(\{x \mapsto y\}) & = & y & \\ \mathit{SIGMA}(s \cup t) & = & \mathit{SIGMA}(s) + \mathit{SIGMA}(t) & \quad \mathit{provided}\ s \cap t = \varnothing \end{array}$$

10 Related Work

Event-B theories are similar in principle to Isabelle [13] and PVS [15], though Isabelle and PVS theories are wider in scope. Theories in Isabelle and PVS can be used to carry significant modelling and reasoning activities. We argue that combining modelling and theory development in Event-B provides a comparable level of sophistication to that of Isabelle and PVS theories. Event-B modelling uses set theory which can provide powerful expressive power that is close to higher order logic [4]. The addition of the theory component ensures that polymorphism can be exploited to enhance the expressive power of the Event-B mathematical language.

The architecture of proof tools continues to stir up much heated debate. One of the main talking points is how to strike a reasonable balance between three important attributes of the prover: efficiency, extensibility and soundness. In [8], Harrison outlines three options to achieve prover extensibility:

1. If a new rule is considered to be useful, simply extend the basic primitives of the prover to include it.
2. Use a full programming language to specify new rules using the basic primitives. The new rules ultimately decompose to these primitives.
3. Incorporate the *reflection* principle, so that the user can add and verify new rules within the existing infrastructure.

Many theorem provers including Isabelle [13] and HOL [6] employ the LCF (Logic of Computable Functions) approach of Milner [11]. The functional language ML [12] is used to implement these systems, and acts as their metalanguage. The approach taken by such systems is to use ML to define data types corresponding to logical entities such as expressions and theorems. A number of ML functions are provided that can generate theorems; these functions implement the basic inference rules of the logic. The ML type system ensures that theorems are only constructed by the aforementioned functions. Therefore, the LCF approach offers both "reliability" and "controllability" of a low level proof checker combined with the power and flexibility of a sophisticated prover [8]. On the flip side, however, a major drawback for this approach is that each newly developed proof procedure must decompose into the basic inference rules. There are cases where this may not be possible or indeed an efficient solution, e.g., the truth table method for propositional logic [5].

The PVS [15] system follows a similar approach to LCF with more liberal support for adding external provers. This liberality comes at a risk of introducing soundness bugs. It, however, presents the user with several choices of automated provers which may ease the proving experience. A comparison between Isabelle/HOL and PVS from a user's point of view is presented in [7]. Interestingly, it mentions that "soundness bugs are hardly ever unintentionally explored" during proof, and that "most mistakes in a system to be verified are detected in the process of making a formal specification". A similar experience is reported when using the Rodin platform [10].

Schmalz [18] defines the Event-B logic using a shallow embedding in Isabelle/HOL [14]. Schmalz provides a comprehensive specification of the logic of Event-B. He gives semantics, devises soundness preserving extension methods, develops a proof calculus similar to [9], and proves its soundness. He presents a formal language for expressing rules (including non-freeness conditions) and shows how to reason about the soundness of Event-B rules. The Event-B type operators such as $\mathbb{P}$ and $\times$ are defined by means of their Isabelle/HOL counterparts. Type substitutions are central to a logic that supports polymorphism, and are also introduced. Binders, expressions and predicates are introduced and are assigned Isabelle/HOL semantics by means of a number of higher-order logic constructs. Note that Schmalz considers predicates to be HOL terms with a boolean type $\mathcal{B}$. Ways of conservatively extending the Event-B logic are outlined and the proof system of Event-B is shown to be sound [18].

11 Concluding

Polymorphic structures such as sequences, bags and stacks are very useful and common modelling elements, but they are absent from the core syntax of Event-B. Prior to our work, functions could be used to mimic operators through axiomatic definitions in Event-B contexts (e.g. see [16]) – but these functions are not polymorphic. Furthermore, from our experience of using the Rodin tool, if a new proof rule is required, a bureaucratic process has to be initiated where resources have to be allocated depending on the urgency of the request. We argue that the practical contributions of the work outlined here are that it:

– complements the Event-B methodology and make it a more rounded formalism,

– provides an appealing platform to end users because it has facilities for meta-reasoning to complement reasoning and modelling in Event-B,
– reduces the dependency on the Java programming language and specialised knowledge of Rodin architecture in order to extend the language and proof mechanisms.

Significant effort is required to develop sound theories. Theory hierarchies are a useful structuring mechanism to create operator taxonomies as is the practice in Isabelle [13]. The effort required to create and validate theories can be decomposed into two large phases:

1. Theory specification phase: new datatypes, operators and proof rules are specified. In this phase, particular attention should be paid to specifying any auxiliary operators that facilitate the use of the main newly introduced structures. In the case of the sequence theory, the *seq* operator is the main structure of the theory, and a number of auxiliary operators, e.g., $emptySeq$, $seqHead$ and $seqTail$, are also defined.
2. Theory validation phase: in this phase, proof obligations are considered and discharged by the user. This phase helps with uncovering errors in the specification of operators and proof rules, in the same way that interactive proof can reveal errors in models.

Therefore, theory development is an iterative process. It is a recurring observation that developing sound theories may take at least the same amount of effort as when developing consistent models. However, the major advantage of using theories is the reusability of definitions thanks to their polymorphic nature. Finally, the familiarity of our approach to users (reactive development, the use of proof obligations and the use of the existing Rodin user interface for specifying and validating theories) ensures that the theory component provides a practical way to extend the Event-B language and proof infrastructure.

Acknowledgements

This work was funded as part of the FP7 DEPLOY (www.deploy-project.eu) and FP7 ADVANCE (www.advance-ict.eu) projects. Various people contributed to the discussions about theory extension including Jean-Raymond Abrial, Nicolas Beauger, Andy Edmunds, Stefan Hallerstede, Alexei Iliasov, Cliff Jones, Michael Leuschel, Thomas Muller, Carine Pascal, Abdolbaghi Rezazadeh, Asieh Salehi, Matthias Schmalz, Renato Silva, Colin Snook and Laurent Voisin.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. J.-R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

3. Jean-Raymond Abrial. B$^{\#}$: Toward a synthesis between Z and B. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB*, volume 2651 of *Lecture Notes in Computer Science*, pages 168–177. Springer, 2003.
4. Jean-Raymond Abrial, Dominique Cansell, and Guy Laffitte. "Higher-Order" Mathematics in B. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 370–393. Springer-Verlag, 2002.
5. Istituto Trentino Di Cultura, Alessandro Armando, Alessandro Armando, Alessandro Cimatti, and Alessandro Cimatti. Building and executing proof strategies in a formal metatheory. In *Advances in Artifical Intelligence: Proceedings of the Third Congress of the Italian Association for Artificial Intelligence, IA*AI'93, Volume 728 of Lecture Notes in Computer Science*, pages 11–22. Springer-Verlag, 1993.
6. Mike Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic, 1985.
7. David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In *Theorem Proving in Higher Order Logics, number 1479 in Lect. Notes Comp. Sci*, pages 123–142. Springer, 1998.
8. John Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz`.
9. Farhad Mehta. A Practical Approach to Partiality - A Proof Based Approach. In *ICFEM*, pages 238–257, 2008.
10. Farhad Mehta. *Proofs for the Working Engineer.* PhD Thesis, ETH Zurich, 2008.
11. Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford, CA, USA, 1972.
12. Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.
13. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle Logics: HOL, 2000.
14. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
15. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-calvert. PVS Language Reference, 2001.
16. Ken Robinson. Reconciling axiomatic and model-based specifications reprised. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 223–236. Springer, 2008.
17. Alexander Romanovsky and Martyn Thomas, editors. *Industrial Deployment of System Engineering Methods.* Springer, 2013.
18. Matthias Schmalz. The Logic of Event-B. Technical Report 698, ETH Zurich, Switzerland, 2010. http://www.inf.ethz.ch/research/disstechreps/techreports.