Theory Plug-in for Rodin 3.x

T.S. Hoang[1], L. Voisin[2], A. Salehi[1], M. Butler[1], T. Wilkinson[1], and N. Beauger[2]

[1] ECS, University of Southamtpon, U.K.
[2] Systerel, France

Abstract. The Theory plug-in enables modellers to extend the mathematical modelling notation for Event-B, with accompanying support for reasoning about the extended language. Previous version of the Theory plug-in has been implemented based on *Rodin 2.x*. This presentation outline the main improvements to the Theory plug-in, to be compatible with Rodin *3.x*, in terms of both reliability and usability. We will also present the changes that were needed in the Rodin core to accommodate the Theory plug-in. Finally, we identify future enhancements and research directions for the Theory plug-in.

1 Introduction

The Theory plug-in *v3.0* has been implemented based on *Rodin 2.x*. The Theory plug-in enables modellers to extend the mathematical modelling notation for Event-B, with accompanying support for reasoning about the extended language. As a result for this extension, different formulae within an Event-B model can be built from different formula factories. As an example, when dealing with a proof obligation, in particular, checking if an existing proof is reusable, we have to deal with two formula factories: (1) the factory associated with the proof obligation (coming from the model), and (2) the factory associated with the existing proof. In the case where the model is altered, these formula factories might be incompatible, rendering the existing proof unusable. In Rodin 3.0, there are major changes (compared to Rodin 2.8) within the Rodin Core (`http://wiki.event-b.org/index.php/Rodin_Platform_3.0_Release_Notes`).

- **Stronger AST Library**: The API of the AST library has been strengthened to mitigate risks of unsoundness when mixing several formula factories. Now, every AST node carries the formula factory with which it was built, and the AST operations that combine several formulas check that formula factories are compatible.
- **Stronger sequent prover**: In order to improve the reliability of the proof status when working with mathematical extensions, the reasoners can be declared as context-dependent. The proofs that use a context dependent reasoner will not be trusted merely based on their dependencies, but instead they will be replayed in order to update their status. This applies in particular to Theory Plug-in reasoners, that depend on the mathematical language and proof rules defined in theories, which change over time.

The above changes directly effect the Theory plug-in, in particular to avoid building formulae from sub-formulae with different formula factories.

2 Mathematical Extensions via the Theory Plug-in

The Theory plug-in [1] enables developers to define new (polymorphic) data types and operators upon those data types. These additional modelling concepts might be defined axiomatically or directly (including inductive definitions). We have made use of the Theory plug-in capability to define domain-specific concepts and provide proof rules for reasoning about them. We focus on the following features of the Theory plug-in that are relevant for our report.

2.1 Modelling extensions

The following modelling extensions are offered by the Theory plug-in: *Polymorphic Inductive Datatypes*, *Axiomatic Datatypes*, *Operators (using Direct Definition, Inductive Definition or Axiomatic definition)*

Polymorphic Inductive Datatypes A datatype with type parameters (polymorphic) is defined using several constructors. Each constructor can have zero or more destructors. As an example, the common *List* inductive datatype is defined as in Fig. 1. A List is either an empty list (nil) or a concatenation of an element (head) and a list (tail).

```
theory ListType(T)
types
    List(T) =
        nil |
        cons(head : "T", tail : "List(T)")
end
```

Fig. 1: `List` Datatype

Axiomatic Datatypes A datatype without any definition is axiomatically defined. By convention, an axiomatic datatype satisfies the *non-emptiness* and *maximality* properties, i.e., for an axiomatic type S,

$$S \neq \varnothing \qquad \text{(non-emptiness)}$$

$$\forall x \cdot x \in S \ . \qquad \text{(maximality)}$$

An example of a axiomatic type for Real number (without any additional axioms) is in Fig. 2.

```
theory Real
types
    Real
end
```

Fig. 2: `Real` Datatype

Operators Operator can be defined *directly*, *inductively* or *axiomatically*. Fig. 3 shows the definitions of two operators `list_isEmpty` (defined directly) and `list_length` (defined inductively). An operator defined without any definition

```
operators
    list_isEmpty(l : "List(T)") = "l ≠ nil"

    list_length(l : "List(T)") =
        case l
            "nil" => "0"
            "cons(t, xs)" => "1 + list_length(l)"

end
```

Fig. 3: Some operators for `List` Datatype

will be defined axiomatically. Operator notation can be either `PREFIX` (default) or `INFIX` (for operator with two or more arguments). Further properties can be declared for operators include *associativity* and *commutativity*. Fig. 4 shows the declaration for three operators: `sum`, `zero`, and (unary) `minus`. In particular, `sum` is declared to be an infix operator which is associative and commutative. The axioms are the assumption about these operators that can be used to defined proof-rules.

Reasoning extensions To support reasoning about the user-defined datatypes and operators, the Theory plug-in offers 5 different *proof tactics*:

- Manual inferencing
- Manual rewriting (including expanding definitions)
- Automatic inferencing
- Automatic rewriting
- Automatic definition expanding

These proof tactics are configured using the proof rules as a part of the user-defined theories. The manual tactics corresponding to application of a single proof rule or expanding a single definition. For rewriting tactics, they often

```
operators

    sum(x : "Real", y : "Real") : "Real" infix associative commutative

    zero : "Real"

    minus(x : "Real") : "Real"
axioms
    @sum_associative "∀x, y · x sum y = y sum x"
    @sum_zero "∀x · x sum zero = x"
    @sum_minus "∀x · x sum minus(x) = zero"
end
```

Fig. 4: Some operators for `Real` Datatype

work at a particular location of the formula. Automatic applications of proof rules mean to repeatedly apply one or more rules until no progress can be made. There are two type of proof rules supported by the Theory plug-in, namely, rewrite rules and inference rules.

Rewrite rules A simple (unconditional) rewrite rule contains information on how to rewrite a formula (often to a simpler form). A rewrite rule can also be *conditional* where the results of the rewriting depends on the circumstances. In Fig. 5, Rules `isEmpty_nil_rewrite` and `isEmpty_cons_rewrite` are unconditional where Rule `isEmpty_rewrite` is conditional.

Inference rules An inference rule contains a list of *given conditions* (possibly empty) and the inferred clause. In Fig. 5, Rule `isEmpty_nil_inference` is an example of inference rules. An inference rule can be applied backward or forward (by default, it is applicable in both direction).

3 Changes to the Rodin Core

3.1 Compatibility of Formula Factories

The first main part of the work is to ensure that the Theory plug-in verifies the construction of the AST formula to guarantee the compatibility of the formula factories. Since the formula factories are basically constructed from the core (standard) factory with mathematical extensions (datatypes, operators), compatibility between different formula factories is reduced to equalities between datatypes and operators.

– **(Inductive) Datatype**: Two inductive datatypes are the same if they have the same signature, i.e., name, type parameters, constructors, destructors.

```
rules
    @isEmpty_nil_rewrite "list_isEmpty(nil)" == "⊤"
    @isEmpty_cons_rewrite "list_isEmpty(cons(t,xs))" == "⊥"

    @isEmpty_rewrite "list_isEmpty(xs)" ==
       "xs = nil" => "⊤" |
       "xs ≠ nil" => "⊥"

    @isEmpty_nil_inference
      "xs = nil"
    |-
      "list_isEmpty(xs)"

end
```

Fig. 5: Example of proof rules for `List` Datatype

- **Axiomatic Datatype**: Two axiomatic datatypes are the same if they have the same signature, i.e., name.
- **Operators (axiomatically defined or with direct/inductive definitions)**: Two operators are the same if they have the same signature, i.e., name, and arguments (including argument types). Note that this comparison does NOT take into account the actual definitions/properties of the operators.

With the above definition, the Theory plug-in correctly compares the formula factory in constructing proofs and checking reusability of existing proofs.

3.2 Supporting Infix Predicate Operator

This particular additional functionality is mainly for improving the usability of the Theory plug-in. We have implemented support for introducing *Infix predicate operator*. For example, consider the operator *smr* (smaller than) for a datatype *Real* numbers. With prefix operator, for expressing x smaller than y, we write

$$smr(x, y)$$

which is unnatural. With the support for infix operator, we can write the formula as

$$x\ smr\ y$$

which improves the readability of the formal text. This support requires modification of both the Rodin platform core and the Theory plug-in core.

3.3 Supporting Type Specialisation

The Theory plug-in has to instantiate several generic datatypes and operators. Such an instantiation means that both types and variables have to be replaced

at the same time. For instance, the classical list operator is defined as $List(T)$, but can be used as $List(1 \mathinner{..} 3)$.

The same issue also needs to be addressed by other plug-ins, such as generic instantiation plug-ins that allow to make a generic development and later instantiate it into another development.

Therefore, all the machinery for instantiating at the same time types and variables has been developed withing the Rodin core in Rodin 3.x under the name of specialisation (the *instantiation* name being already used for another purpose). This specialisation mechanism can be applied to any type, type environment or formula and guarantees type safety (if the input is well-typed, then the output is guaranteed to also be well-typed).

4 Changes to the Theory Plug-in

4.1 Improvement on Pattern Matching

The pattern matching facility of the Theory plug-in has been upgraded to use directly the support from the Event-B core for specialising (instantiating) formulae. This ensures that the information for specialising formulae is type-consistent. The following examples illustrate some consistent specialisations (which are unsupported in the previous version).

$Patterns$		$Formulae$
S	$\longrightarrow$	$\mathbb{P}(S)$
S	$\longrightarrow$	$S \times T$

Another important improvement on pattern matching is the implementation for matching associative operators (which is only implemented for some special cases before). The matching for associative formulae is done based on a simple greedy algorithm. The following examples illustrate the result of pattern matching for an associative operator, namely forward composition ;.

Patterns	Formulae	Result
$f; \{x \mapsto c\}$	$g; h; \{y \mapsto c\}$	$f \leftarrow g; h$
		$x \leftarrow y$
		$c \leftarrow c$
$e; f$	$g; h; \{y \mapsto c\}$	$e \leftarrow g$
		$f \leftarrow h; \{y \mapsto c\}$

Note that in the second example, another possible result for pattern matching would be $[e \leftarrow g; h, f \leftarrow \{y \mapsto c\}]$. Our algorithm only returns a single matching result that it found first in the case where there more than one possible matching.

4.2 Supporting Unicode Typesetting for Real Number Operators

Another usability improvement that we have implemented is the support for typesetting *Real* number operators in Unicode. Instead of ASCII text, we use the following Unicode symbols for the common operators that we used in the modelling.

Real operators	ASCII Typeset	Unicode
The set of *Real* number	`REAL`	$\mathbb{R}$
Plus	`+r`	$\oplus$
Minus	`-r`	$\ominus$
Multiply	`*r`	$\otimes$
Divide	`/r`	$\oslash$
Less than	`<r`	$<$
Less than or equal	`<=r`	$\leqslant$
Greater than	`>r`	$>$
Greater than or equal	`>=r`	$\geqslant$

4.3 Usability Improvement/Reimplementation Rule-based Prover

As mentioned before, the rule-based prover of the Theory plug-in offers 5 different tactics:

1. Manual inferencing
2. Manual rewriting
3. Automatic inferencing
4. Automatic rewriting
5. Automatic definition expanding

Each tactic is a wrapper around one or more reasoners. Roughly speaking, each reasoner manages an application of a single inference/rewrite rule. Table 1 shows the relationship between the tactics and the corresponding reasoner (for the pervious, i.e., *v3.0*, and the current implementation, i.e., *v4.0*). The decision

Tactic	v3.0	v4.0
Manual inferencing	Single reasoner	Single reasoner
Manual rewriting	Single reasoner	Single reasoner
Automatic inferencing	Single reasoner	*Multiple reasoners*
Automatic rewriting	Single reasoner	*Multiple reasoners*
Automatic definition expanding	Single reasoner	*Multiple reasoners*

Table 1: Tactics and Reasoners

to implement the automatic tactics combining multiple reasoners is for *usability*. Instead bundling the effect of apply several rules into a single proof node, we separately apply the rules one-by-one. The result is as follows

- The proofs are easy to understand.
- The proofs are less prone to changes (e.g., changes in the model).
- The proofs are easy to adapt, e.g., one can keep a part of the proofs produced automatically (by pruning) and proceed further manually.

Reasoner Input In the previous version, the context of the proof obligation is passed as an input to the reasoner. In this version, the context is retrieved from the origin of the proof obligation. This ensure that the same context, e.g., the formula factory, is used for the obligation and the proof. Further more, due to the changes to the automatic reasoners described previously, the reasoner inputs for these reasoners are adapted accordingly. Table 2 summarises the differences between v3.0 and v4.0 of the tool.

Reasoner	v3.0	v4.0
Manual Inferencing	– PO Context – Rule Meta-data – Application hypothesis (if forward inference) or `null` (if backward inference)	– Rule Meta-data – Application hypothesis (if forward inference) or `null` (if backward inference)
Manual Rewrite	– PO Context – Rule Meta-data – Application hypothesis (if rewriting a hypothesis) or `null` (if rewriting the goal) – Rewrite position	– Rule Meta-data – Application hypothesis (if rewriting a hypothesis) or `null` (if rewriting the goal) – Rewrite position
Automatic Inferencing	– PO Context	– Rule Meta-data – Forward or backward
Automatic Rewriting	– PO Context	– Rule Meta-data

Table 2: Reasoner Input

Well-definedness When instantiating a proof rule, each instantiation expression required to be *well-defined*, to ensure that the resulting sequents are well-defined. In *v3.0*, the well-definedness sub-goals are generated one for each instantiation expression and are mixed with the other sub-goals, and in some cases are not generated at all. We have consolidate the generation of the WD sub-goals (in *v4.0*) by combing all these sub-goals into a single sub-goal (using conjunction)

and add this new WD sub-goal as (always) the first sub-goal when applying a proof rule. As a result, the proof are much less prone to changes.

5 Future Work

In this paper, we have highlighted the major changes to the Theory plug-in and the Rodin Core, focusing on bringing the Theory plug-in to the latest version of the Rodin platform. In particular, the update requires some update to the core of the Rodin Platform itself, hence will be publicly available after the next release (*v3.3*) of the Rodin Platform. At the same time, we also improved the usability of Theory plug-in, especially focusing on features that required changes from the Rodin plug-in core.

In the next release, we will focus our attention to the usability of the plug-in, by gathering feedbacks from its users. Some of the ideas for improvement are:

- Matching facility for *associative and commutative operators* (currently ignoring commutativity).
- Support for user-defined tactics
- Support for predicate variables in theories
- Theory instantiation (different abstraction-level of theories).

Acknowledgements

Laurent Voisin and Nicolas Beauger have been partially funded by the French Research Agency under grant ANR-13-INSE-0001 (IMPEX project). Thai Son Hoang, Michael Butler and Toby Wilkinson are supported by the ASUR Programme project 1014 C6 PH1 104.

References

1. Michael J. Butler and Issam Maamria. Practical theory extension in Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013.