


Theory Plug-in User Manual

Issam Maamria, Asieh Salehi Fathabadi
University of Southampton

April 30, 2014

The Theory plug-in is a contribution to the Rodin platform that facilitates the **specification, validation, deployment** and **use** of language and proof extensions for the Event-B methodology. Language extensions are additions to the Event-B mathematical language in the form of 1) datatypes, 2) operators and 3) axiomatic definitions. Proof extensions are additions to the Event-B proving infrastructure in the form of rewrite rules, inference rules and polymorphic theorems. The specification of extensions is achieved by means of *theories* \mathbb{T} . The validation of extensions is achieved by means of proof obligations whenever appropriate. This user manual provides a comprehensive overview of the plug-in's functionality and capabilities.

For a quick start guide, the user can skip to Section 3.

 The Event-B **mathematical language** refers to the language used to write axioms, invariants, guards etc. in Event-B models.

 Event-B theories are Rodin file just like contexts and machines.

1 Motivation

Work on the Theory plug-in started as an effort to create a *Rule-based Prover* for Event-B much like the ML prover in Atelier-B. The Rule-based Prover, as it was known then, supported the definition, validation and use of rewrite rules. The Theory plug-in is the successor of the Rule-based Prover, and offers much more functionality.

Prior to Rodin v2.0, the mathematical language used in Event-B was fixed. As such, it was not possible to define reusable polymorphic operators. A workaround was to define any required operators as set constructs in contexts. Originally, contexts were supposed to provide a parametrization of machines. The aforementioned limitations of the Event-B language lead to users to use contexts for purposes for which they were not intentionally devised. Examples of operators that can be useful to users include the sequence operator (which was present in classical B mathematical language) and the bag operator.

In Rodin v2.0, a dynamic parser has been implemented for the Event-B AST. The Theory plug-in was a natural candidate for defining and using mathematical

extensions. To provide a comprehensive platform, cover for a wider range of proof rules was also needed.

 ML is a rule-based prover as opposed to the semi-decision procedure PP.

2 Capabilities


The Theory plug-in has the following capabilities:

1. **Theory Definition:**

- (a) Definition of **operators**: operators can be defined as predicate or expression operators. An expression operator is an operator that "returns" an expression, an example existing operator is *card*. A predicate operator is one that "returns" a predicate, an example existing predicate operator is *finite*.
- (b) Definition of **datatypes**: datatypes are defined by supplying the types on which they are polymorphic, a set of constructors one of which has to be a base constructor. Each constructor may or may not have destructors.
- (c) Definition of **axiomatic definitions**: axiomatic definitions are defined by supplying the types, a set of operators, and a set of axioms.
- (d) Definition of **rewrite rules**: rewrite rules are one-directional equalities that can be applied from left to right. The Theory plug-in can be used to define rewrite rules.
- (e) Definition of **inference rules**: inference rules can be used to infer new hypotheses, split a goal into sub-goals or discharge sequents.
- (f) Definition of **polymorphic theorems**: theorems can be defined and validated once, and can then be imported into sequents of proof obligations if a suitable type instantiation is available.
- (g) **Validation of extensions**: where appropriate, proof obligations are generated to ensure soundness of extensions. This includes, proof obligations for validity of inference and rewrite rules, as well as proof obligations to validate operator properties such as associativity and commutativity.

2. **Theory Deployment**: this step signifies that a theory is ready for use. Theories can be deployed after they have been optionally validated by the user. It is strongly advisable to discharge all proof obligations before deployment.

Once a theory has been deployed and introduced to its designated project, all its extensions (mathematical and proof extensions) can be used in models. In later sections, we show the scopes of theory availability.

 In the Event-B mathematical language, predicates (known as formulae in most literature) and expressions (known as terms) are two separate syntactic categories. Expressions have a type. Predicate do not.

3 Quick Start

In this section, we step through a simple tutorial on how to define and use a simple theory. Click on the links above to navigate through this tutorial.

3.1 Install Theory Plug-in

Considering figure 1, the installation or update for the Theory plug-in is available under the main Rodin Update site (<http://rodin-b-sharp.sourceforge.net/updates>) under the category "Modelling Extensions". Like always, after the installation, restarting Rodin is recommended. For more details, see http://wiki.event-b.org/index.php/Theory_News_and_Support.

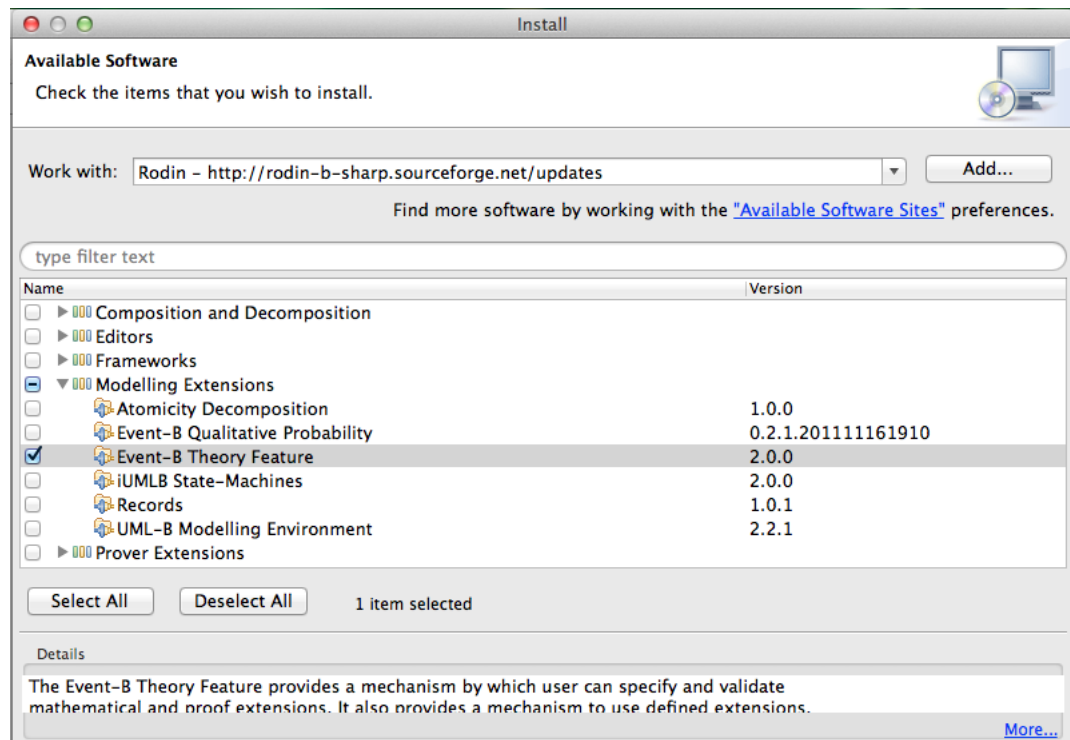



Figure 1: Install Theory Plug-in

Once the Theory plug-in is successfully installed, menu entries will be added in certain places, see figure 2. In particular, the Event-B Explorer will have an additional button  that initiates the wizard to create new theory files.

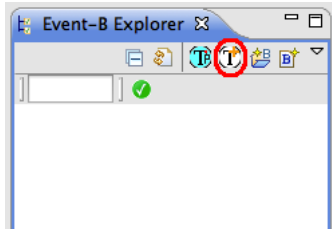




Figure 2: New Theory Button in Event-B Explorer

3.2 Create A New Theory

An additional button  (red-circled in figure 2) should appear in the Event-B Explorer. By clicking the button , a wizard that enables the creation of a new theory is initiated. Figure 3 shows the wizard in action.

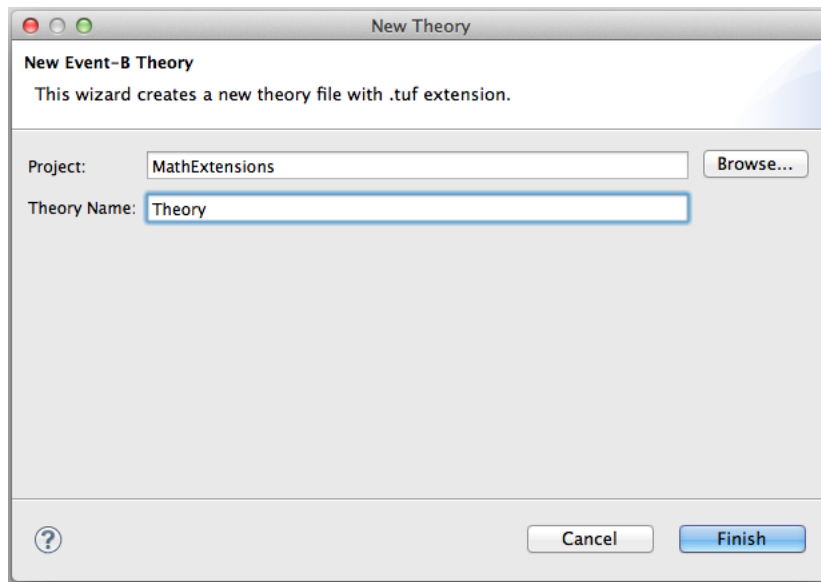


Figure 3: New Theory File Wizard

In the wizard, specify the parent project of the theory and a theory name. The project can be selected using the button on the right hand side of project name text field (akin to selecting a project when creating a new Event-B component). Click the Finish button to create the theory. If there are no name clashes between the name of the new theory and any existing resources, you should get a theory editor opened up as depicted in figure 4.

The theory editor has two pages: an Edit page and an HTML (i.e., pretty print) page. The edit page corresponds to the structured editor. The HTML

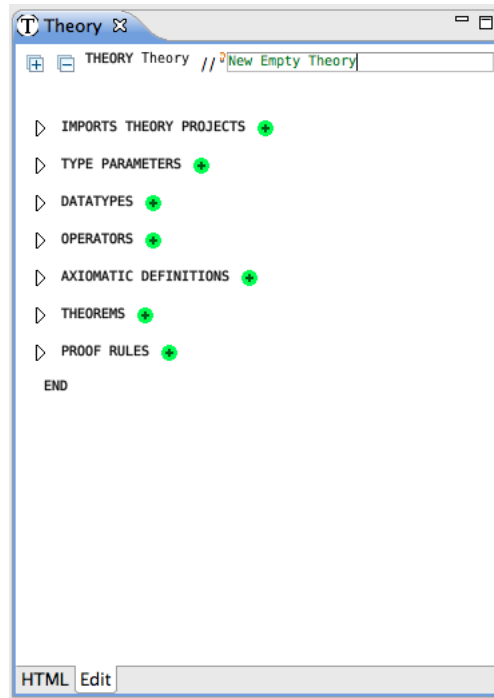


Figure 4: The Theory Editor

page is a pretty print view of the theory.

3.3 Add a Type Parameter

Type parameters in a theory specify the types on which new definitions and rules may be polymorphic. For instance, a theory of sequences can be polymorphic on one type and that is the type of elements it may hold.



Type parameters are similar in nature to carrier sets in contexts. To create a new type parameter, click on button  under the Type Parameters section of the structured editor, and specify the name of the type parameter (see figure 5).



Figure 5: Type Parameters

 Type parameters are expected to be a legal Event-B identifier. For example, *2ident* is not a legal identifier.

 As a convention, type parameters are specified as upper case strings

(same as context's carrier sets).

3.4 Create an Operator Definition

Event-B mathematical language has many useful operators. Examples include cardinality operator *card*, the finiteness predicate operator *finite* and the function application *.*(*.*). Other useful operators can be defined using the Theory plug-in. Figure 6 shows a definition of a sequence operator.

The screenshot shows the 'OPERATORS' section with a dropdown menu for 'seq' (labeled 1). The 'Syntactic Class' is set to 'expression' (labeled 2). The 'Notation' is set to 'PREFIX' (labeled 3). The 'Associativity' is set to 'not associative' (labeled 4). The 'Commutativity' is set to 'not commutative' (labeled 5). The 'arguments' section shows 'a' (labeled 7) and 'P(A)' (labeled 8). The 'direct definition' section is active, showing a formula: $\{n, f \cdot n \in \mathbb{N} \wedge f \in 1..n \rightarrow a \mid f\}$ (labeled 9). The formula is highlighted with a red box. The 'recursive definition' section is inactive.

Figure 6: Sequence Operator Definition

Note the numbering in figure 6. The following explains each part of the definition:

1. **Syntax Symbol:** this specifies the syntax token that will be reserved for the new operator (sequence in our example). It should not clash with any previously defined symbol in the mathematical language available to the theory.
2. **Syntactic Class:** this specifies whether the new operator is an expression operator or a predicate operator. For example, the cardinality operator *card* is an expression operator of integer type, and the finiteness operator *finite* is a predicate operator. In our case, the sequence operator is an expression operator. The button can be toggled off if a predicate operator is required instead.
3. **Notation:** this specifies whether the symbols is a prefix or an infix operator. In the existing mathematical language, *+* is an infix operator whereas *partition* is a prefix predicate operator. In our example, the sequence operator is specified as prefix.
4. **Associativity:** this specifies whether the operator is associative. Note that this has semantic implications, and as such a proof obligation is generated to check the associativity property.
5. **Commutativity:** this specifies whether the operator is commutative. Note that this has semantic implications, and as such a proof obligation is generated to check the commutativity property.

6. **Operator Arguments:** an operator may have a number of arguments (all of which may be expressions).
7. **Argument Identifier:** this specifies the name of the argument of the operator. It has to be a legal Event-B identifier (similar to carrier sets, constant, variables etc.).
8. **Argument Type:** this specifies the type of the argument. In our case, the sequence operator takes a set of type A . Since A is a type parameter, the sequence is polymorphic.
9. **Direct Definition:** this provides the direct definition of the operator. In our case (see the red-boxed field), it asserts that sequences are total functions from a contiguous integer range starting at 1 to the set a the argument of the operator seq .



- Only operators that take two arguments of the same type can be tagged as commutative. Of course, then, one has to prove the mathematical property.
- An operator can be tagged as being associative if it satisfies the three conditions: (1) it is an expression operator, (2) it takes two (or more) arguments of the same type, (3) the type of the operator is the same as that of its arguments. Of course, then, one has to prove the mathematical property.
- Operators that are tagged associative have to be tagged as infix as well.
- The argument type can be a type or a set expression. If the argument type is a set expression, then the type of the argument is inferred. Furthermore, the additional restriction (i.e., that the argument belongs to a set expression) is added as a well-definedness condition for the operator.



- As a convention, names of operators should be lower case.
- As a convention, names of operator arguments should be lower case.

The operator in figure 7 defines size for sequences.

This definition asserts that the operator $seqSize$ takes one argument of type $\mathbb{Z} \leftrightarrow A$. This definition, also, triggers a proof obligation to prove the strength of the well-definedness condition provided. Namely, one has to prove that $\forall s \cdot s \in seq(A) \implies finite(s)$. We leave this as an exercise to the reader.

Figure 8 is a definition of a predicate operator.

The definition of $seqIsEmpty$ does not trigger any proof obligation for well-definedness strength. This is due to the fact that the corresponding condition is a trivial predicate, namely: $\forall s \cdot s \in seq(A) \implies s \in seq(A)$.

The head operator on sequences can be defined as in figure 9.

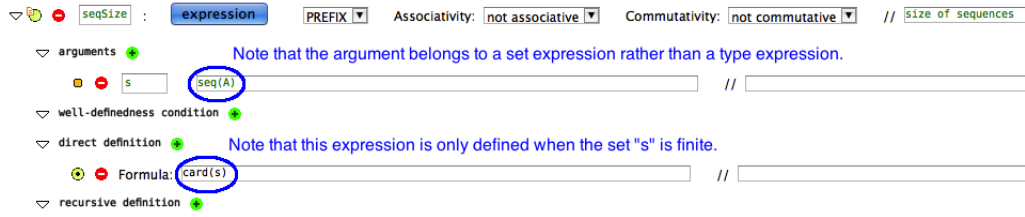


Figure 7: Sequence Size Operator Definition



Figure 8: A Predicate Operator Definition

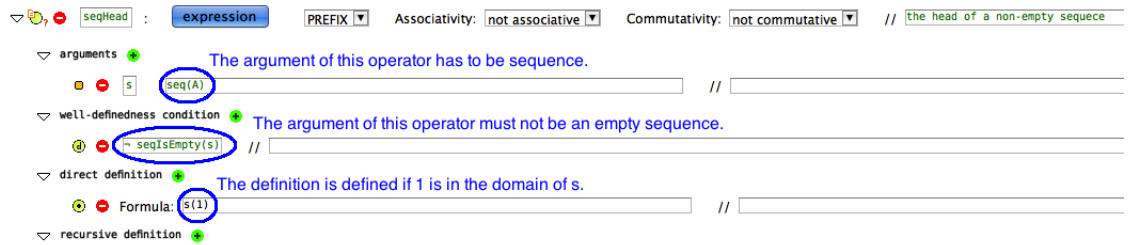


Figure 9: Sequence Head Operator Definition

Figure 10 shows the well-definedness strength proof obligation corresponding to the previous definition of *seqHead*.

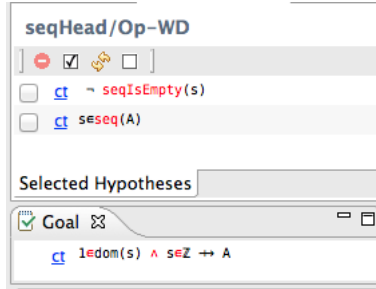


Figure 10: Sequence Head WD Strength PO

As a summary, have a look at figure 11 which is taken from the HTML view of our theory.

```

OPERATORS
*seq : seq(a : P(A))  EXPRESSION  PREFIX      // The sequece operator
direct definition
seq(a : P(A)) ≐ {n, f · n ∈ N ∧ f ∈ 1..n → a | f} // a set of finite toal functions
*seqSize : seqSize(s : seq(A))  EXPRESSION  PREFIX      // size of sequences
direct definition
seqSize(s : seq(A)) ≐ card(s)
*seqIsEmpty : seqIsEmpty(s : seq(A))  PREDICATE  PREFIX      // predicate operator that checks
// whether a given sequence is empty
direct definition
seqIsEmpty(s : seq(A)) ≐ seqSize(s)=0
*seqHead : seqHead(s : seq(A))  EXPRESSION  PREFIX      // the head of a non-empty sequece
well-definedness condition
¬ seqIsEmpty(s)
direct definition
seqHead(s : seq(A)) ≐ s(1)
*seqTail : seqTail(s : seq(A))  EXPRESSION  PREFIX      // the tail of a non-empty sequence
well-definedness condition
¬ seqIsEmpty(s)
direct definition
seqTail(s : seq(A)) ≐ λi·i∈1..seqSize(s)-1|s(i+1)
*seqPrepend : seqPrepend(s : seq(A), e : A)  EXPRESSION  PREFIX      // prepends an element to a sequence
direct definition
seqPrepend(s : seq(A), e : A) ≐ {1↦e} ∪ {λi·i∈2..seqSize(s)+1|s(i-1)}
*seqAppend : seqAppend(s : seq(A), e : A)  EXPRESSION  PREFIX      // appends an element to a sequence
direct definition
seqAppend(s : seq(A), e : A) ≐ s ∪ {(seqSize(s)+1)↦e}
END

```

Figure 11: Various Sequence Operators

The proof obligations associated with an operator definition are the following:

1. **./Op-WD** operator well-definedness strength if a well-definedness condition is explicitly specified.
2. **./Op-COMMUT** the commutativity proof obligation, generated if the operator is tagged as commutative.
3. **./Op-ASSOC** the associativity proof obligation, generated if the operator is tagged as associative.

3.5 Specify a Polymorphic Theorem

A polymorphic theorem is no different, in principle, from theorems defined in contexts and machines. The Theory plug-in, however, provides facilities to instantiate and use these theorems in proofs. See the example in figure 12.

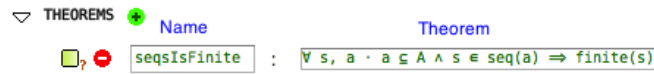


Figure 12: Sequence Finiteness Theorem

The previous theorem articulates the fact the sequences as specified in our example are finite. As with theorems in contexts and machines, you have to prove validity and well-definedness of the theorem. The proof obligations associated with a theorem are the following:

1. **./S-THM** the validity proof obligation.
2. **./WD-THM** the well-definedness proof obligation.

Figure 13 shows other theorems that can be defined in relation to our theory of sequences so far:

```

THEOREMS
seqsIsFinite      : forall s, a. a subset A and s in seq(a) implies finite(s)
seqsMonotone     : forall s, a, b. (a subset A and s in seq(a) and a subset b) implies s in seq(b)
tailSeqIsSeq     : forall s, a. a subset A and s in seq(a) and not seqIsEmpty(s) implies seqTail(s) in seq(a)
seqPrependIsSeq  : forall s, a, e. a subset A and s in seq(a) implies (seqPrepend(s, e) in seq(a union {e}))
seqAppendIsSeq   : forall s, a, e. a subset A and s in seq(a) implies (seqAppend(s, e) in seq(a union {e}))

```

Figure 13: Sequence Theorems

✓ A theorem can be instantiated (e.g., in the previous example, the type parameter A can be substituted with a type expression that is acceptable in the context of the sequent under consideration). We will later show how this is achieved.

3.6 Specify an Inference Rule

Inference rules are proof rules that can be used to: (1) infer new hypotheses in a proof, or (2) split the goal into sub-goals, or (3) discharge a proof obligation. The general structure of an inference rule is as follows:

$$\begin{array}{l} \textit{Given} \\ G_0, \dots, G_n \\ \textit{Infer} \\ I \end{array}$$

where each of G_0, \dots, G_n and I is an Event-B predicate. The above inference rule can be read in the following two ways: "given conditions G_0, \dots, G_n one can infer I ", and "one can prove I , if one can prove each of G_0, \dots, G_n ".


Inference rules can be defined as part of a block of "Proof Rules". Each proof rules block may contain a number of metavariables, rewrite rules and inference rules, see figure 14. To create a rules block, under the heading "PROOF RULES" in the structured editor, press .



Figure 14: Proof Rules Block

Metavariables define the variables used to specify proof rules. Considering figure 15, each metavariable has a name and a type. For our example, we need one metavariable s .

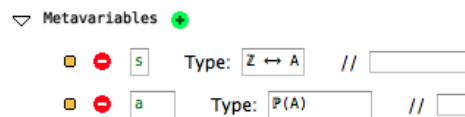


Figure 15: Defining a Metavariable

The example in figure 16 shows an inference rule concerning finiteness of sequences:

The applicability of a proof rule indicates whether the rule should be applied automatically or interactively. The description provides a human-readable view of the rule. The description provided will be the tool tip for the rule in the

▾ Inference Rules +
 ▾ seqIsFiniteInf : Applicability: interactive Description: sequences are finite //
 ▾ Given +
 s ∈ seq(a) not in hypothesis //
 ▾ Infer +
 finite(s) //

Figure 16: Sequence Finiteness Inference Rule

proof UI. The inference rule in figure 17 is an automatic rule that states that the tail of a non-empty sequence is a sequence.


▾ tailSeqIsSeqInf : Applicability: automatic Description: tail is a sequence //
 ▾ Given +
 s ∈ seq(a) not in hypothesis //
 ¬ seqIsEmpty(s) not in hypothesis //
 ▾ Infer +
 seqTail(s) ∈ seq(a) //

Figure 17: Sequence Tail Inference Rule

4 Scoping, Colour coding and Using of Theories

A *theorypath* is a means to introduce the deployed theories in a project scope.

In order to use a math extension, defined in a theory, in a project scope:

1. **Deploy the theory** A theory can be deployed by a pop-up menu , by right-clicking on the theory, figure 18. .

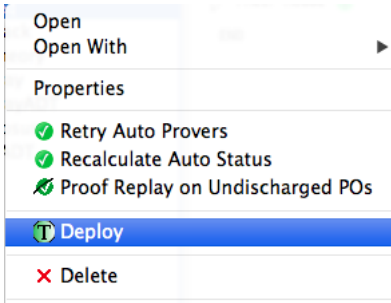



Figure 18: Deploy Theory Button in Event-B Explorer

2. **Import the deployed theory in a *theorypath***, which is defined in the project scope. By clicking the button , a wizard that enables the creation of a new *theorypath* is initiated (See figure 19).

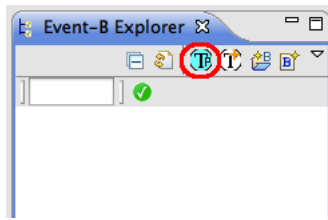








Figure 19: New Theorypath Button in Event-B Explorer

Colour Coding: A theory is created white , after deploying it turns to green . When a deployed green theory is modified, it turns to amber , indicating that the deployed version is different to the developing version.

- **White** : New theory: the theory is new and has not been deployed yet. (such a theory is not accessible to be imported either in another theory or a *theorypath*)
- **Green** : Deployed updated theory: the theory is deployed and updated.
- **Amber** : Deployed out-dated theory: the theory is modified after deployment; the deployed version of the theory is not sync with the current state of the theory.



- A machine/context accesses (local/global) theories imported directly or indirectly by a *theorypath* within the same project as the machine/context.
- A *theorypath* can imports deployed (local/global) theories.
- A theory can import deployed (local/global) theories.



- A *local* theory in a project scope is a theory defined inside a project; Whereas a *global* theory is a theory defined in a separated project.
- As illustrated in figure 20, if theory $T1$ in $Prj1$ imports theory $T2$ from $Prj2$, and theory $T1$ is imported in a *theorypath* created in the project $Prj3$; Then $T1$ is directly and $T2$ is indirectly accessible in the $Prj3$ scope.

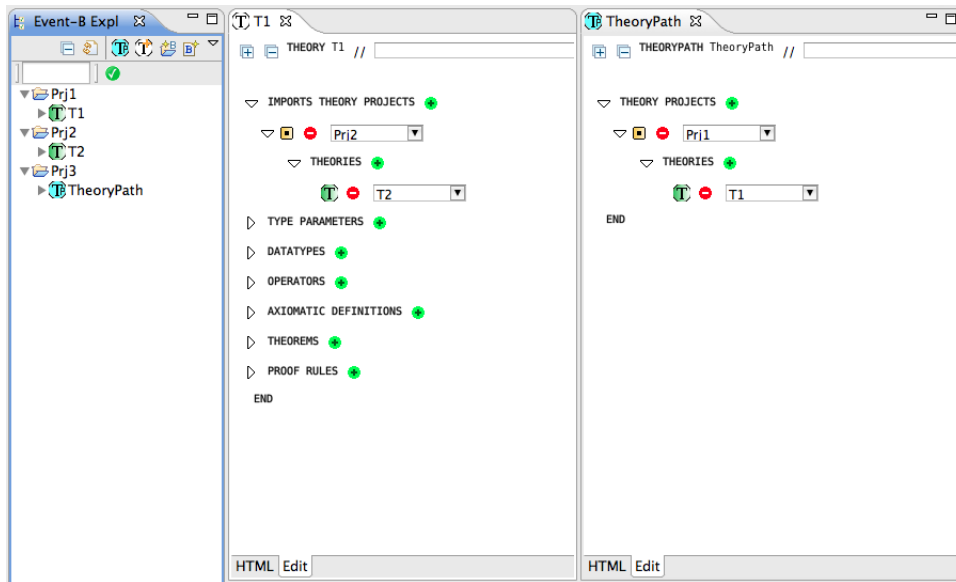


Figure 20: Accessibility of Theories