Event-B Formalization of Event-B Contexts

Jean-Paul Bodeveix, M Filali

Event-B Formalization of Event-B Contexts

Jean-Paul Bodeveix[1] and Mamoun Filali[2(✉)]

[1] IRIT-UPS, 118 Route de Narbonne, 31062 Toulouse, France
jean-paul.bodeveix@irit.fr
[2] IRIT-CNRS, 118 Route de Narbonne, 31062 Toulouse, France
mamoun.filali@irit.fr

Abstract. This paper presents an Event-B meta-modelisation of an Event-B project restricted to its context hierarchy which introduces the functional part of a development through sets, constants, axioms and theorems. We study the proposal of a new mechanism for Event-B. It consists in allowing to instantiate in a new context an already proved theorem in a given context. We investigate the validation of the instantiation mechanism in order to prove the validity of imported theorems. We also compare the proposal with similar mechanisms available within some existing theorem provers.

Keywords: Formal methods · Event-B · Meta modelisation

1 Introduction

Event-B [1] is a formal method for the rigorous development of systems. One of its salient features is the Rodin tool [2] which offers an integrated environment for developing and proving. The aim of the EBRP (Enhancing Event-B and Rodin Plus) project[1] is to enhance the framework offered by Rodin in order to better support reuse in Event-B developments thanks to the introduction of generic theories and data types. This enhancement follows the initial work of [4,7]. As a first step of the project, a light extension of the Event-B language and tool has been proposed. In this paper, we investigate an Event-B meta-level description of this extension. An Event-B model consists in a functional model made of an acyclic graph of contexts and a dynamic model using the functional part which consists of successive event-based machine refinements. We focus here on the new reusability mechanism currently studied by the EBRP project for the functional model. It consists in reusing (importing) instances of theorems and axioms considered to be parameterized by the sets and constants declared in their context. The aim of this paper is to validate this importation mechanism: more precisely, we wish to establish the *validity* of an instance of an imported theorem. For this purpose, we propose a meta-level study of Event-B context structure

[1] The project EBRP is supported by the French research agency: ANR.

1. https://doi.org/10.1007/978-3-030-77543-8_5

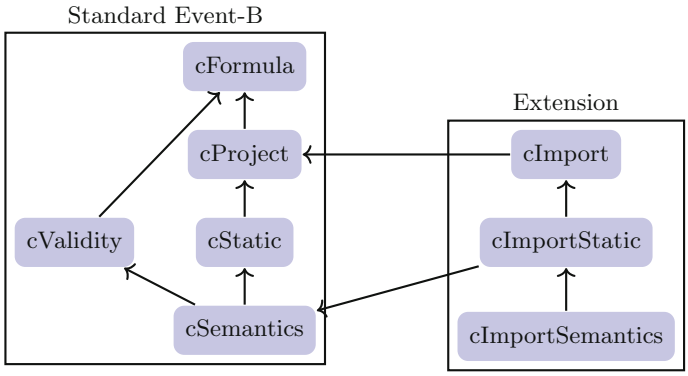

Fig. 1. Metamodel architecture

extended with importation clauses. Figure 1 shows the meta modelisation of a standard Event-B context and its extension with importation/instanciation clauses.

The rest of the paper is structured as follows: Section 2 presents an Event-B meta-model of an Event-B project made of contexts. Section 3 extends this meta-model by the proposed importation mechanism. Section 4 presents similar mechanisms that can be found in other proof environments. Section 5 concludes this paper.

2 Event-B Contexts

In this section, we give a meta-level description of a project made of a hierarchy of contexts. Starting with a high level description of formulas (context `cFormula`), we introduce the subset of valid formulas (context `cValidity`) and describe the project structure as a set of contexts (context `cProject`). Semantics constraints are introduced through the contexts `cStatic` and `cSemantics`. The new importation feature is introduced in the `cImport` context with semantic constraints in `cImportStatic` and `cImportSemantics`. To summarize, the standard representation is structured as a set of contexts corresponding to the left hand side of Fig. 1 and the proposed extension in its right hand side. Moreover, we illustrate the architecture of these contexts through UML-like diagrams[2].

2.1 Formulas

Formulas (see Fig. 2) are modelled at an abstract level. Its free variables are either sets (acting as base types) or constants. A formula denotes either an expression or a predicate. Some expressions (left unspecified here) denote types. The text of a formula is not considered.

[2] We could have used the UML-B plugin [9].

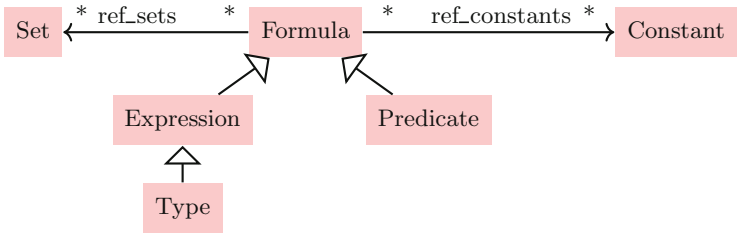

Fig. 2. cFormula context

However, the relations `ref_sets` and `ref_csts` associate respectively the referenced sets and constants. Formulas are partitioned into `Expression` and `Predicate`. Note that we have not modeled the type of an expression and typing constraints. Note also that we could have declared the two relations `ref_sets` and `ref_csts` as functions to power sets of sets or constants. However, it would make more difficult to use the relational composition in order to navigate through the metamodel. These notions are declared through the following labelled (the '@' tag) Event-B axioms[3]:

$$\begin{array}{l} \texttt{@Formula\_part}\ \ \textbf{partition}(\mathsf{Formula}, \mathsf{Expression}, \mathsf{Predicate}) \\ \quad \texttt{@ref\_sets\_ty}\ \ \mathsf{ref\_sets} \in \mathsf{Formula} \leftrightarrow \mathsf{Set} \\ \quad \texttt{@ref\_csts\_ty}\ \ \mathsf{ref\_csts} \in \mathsf{Formula} \leftrightarrow \mathsf{Constant} \end{array}$$

A `Type` is seen as a special expression which only refers to sets (not constants)[4].

$$\begin{array}{l} \texttt{@Type\_ty}\ \ \mathsf{Type} \subseteq \mathsf{Expression} \quad // \textit{ type expressions such as } \mathbb{P}(A) \textit{ and } A \times B \\ \texttt{@ref\_type}\ \ \mathsf{Type} \lhd \mathsf{ref\_csts} = \emptyset \quad // \textit{ A type expression doesn't reference a constant} \end{array}$$

An important operation on formulas is substitution (`subst`): a set can be replaced by a `Type` and a constant by an `Expression`. The main static property of a substitution: `subst_ref` is that unreferenced sets and constants can be removed from the substitution domain. This expressed through a domain restriction ($\lhd$). We use this property to show that an imported instance of a theorem remains a theorem[5] [6] [7].

$$\begin{array}{l} \texttt{@subst\_ty}\ \ \mathsf{subst} \in (\mathsf{Set} \nrightarrow \mathsf{Type}) \times (\mathsf{Constant} \nrightarrow \mathsf{Expression}) \times \mathsf{Formula} \rightarrow \mathsf{Formula} \\ \texttt{@subst\_ref}\ \ \forall \mathsf{s}, \mathsf{c}, \mathsf{f} \cdot \mathsf{s} \in \mathsf{Set} \nrightarrow \mathsf{Type} \land \mathsf{c} \in \mathsf{Constant} \nrightarrow \mathsf{Expression} \land \mathsf{f} \in \mathsf{Formula} \Rightarrow \\ \quad \mathsf{subst}(\mathsf{s} \mapsto \mathsf{c} \mapsto \mathsf{f}) = \mathsf{subst}(\ \mathsf{ref\_sets}\ [\{\mathsf{f}\}] \lhd \mathsf{s} \mapsto\ \mathsf{ref\_csts}\ [\{\mathsf{f}\}] \lhd \mathsf{c} \mapsto \mathsf{f}) \end{array}$$

[3] $A \leftrightarrow B$ denotes the set of relations from A to B: $A \leftrightarrow B \triangleq \mathcal{P}(A \times B)$.
[4] $\lhd$ denotes domain restriction: $s \lhd r \triangleq r \cap (s \times \mathbf{ran}(r))$.
[5] $x \mapsto y$ denotes the ordered pair (x, y).
[6] $s \nrightarrow t$ denotes a partial function.
[7] $r[s]$ denotes the relational image by r of the set s: $r[s] \triangleq \mathbf{ran}(s \lhd r)$.

2.2 Validity

We first introduce a sequent as a pair formed by a set of hypothesis predicates and a conclusion predicate. Valid sequents are introduced as a subset.

```
@Sequent_def Sequent = ℙ(Predicate) × Predicate
@Valid_ty  Valid  ⊆ Sequent
```

With respect to our concerns, we consider only two axioms about sequents: monotony and substitution of free identifiers which are sets and constants. The first one states that if the hypotheses of a valid sequent are enriched, the sequent remains valid.

```
@Valid_mono ∀H1,H2,G· H1⊆H2 ∧ H1↦G ∈ Valid ⇒ H2↦G ∈ Valid
@subst_V ∀H,G,s,c· H↦G ∈ Valid ∧ s ∈ Set ⇸ Type ∧ c ∈ Constant ⇸ Expression ⇒
  {h·h ∈ H | subst(s↦c↦h)} ↦ subst(s↦c↦G)  ∈  Valid
```

2.3 Project

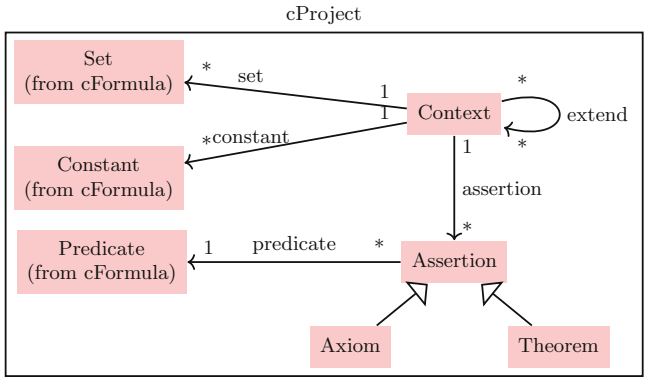

Fig. 3. cProject context

A project contains contexts denoted by the set `Context` linked by the extend relation `extend`. `extend` is declared irreflexive and transitive (Fig. 3).

```
@extend_ty extend  ∈  Context ↔ Context // c1 extends c2
@extend_irr  id ∩ extend = ∅
@extend_trans extend;extend ⊆ extend
```

Sets and constants are defined within contexts. A set or a context is defined once[8]. Note that a set or a constant can be present in unrelated contexts (through the `extend` relation).

[8] $r_1; r_2$ denotes relation composition, usually denoted $r_2 \circ r_1$. It is used to navigate in the metamodel along a chain of links.

```
@set_ty set ∈ Context ↔ Set
@constant_ty constant ∈ Context ↔ Constant
// a set is defined once in a hierarchy of contexts
@set_uniq_pred (extend;set) ∩ set = ∅
// a constant is defined once in a hierarchy of contexts
@cst_uniq_pred (extend;constant) ∩ constant = ∅
```

In a context, assertions are stated. An assertion is characterized by a predicate. `assertion` defines the relation between contexts and assertions. Axioms and theorems define distinct assertions.

```
@assert_ty predicate ∈ Assertion → Predicate
@ass_ty assertion ∈ Context ↔ Assertion
@ass_ctx assertion⁻¹ ∈ Assertion → Context
@Assert_fin finite (Assertion)
@Axiom_ty Axiom ⊆ Assertion
@Theorem_ty Theorem ⊆ Assertion
@AxThm Axiom ∩ Theorem = ∅
```

`axiom` and `thm` are respectively the restriction of `assertion` to axioms and theorems.

```
@ax_ty axiom = assertion ▷ Axiom
@thm_ty thm = assertion ▷ Theorem
```

2.4 Static Correcness

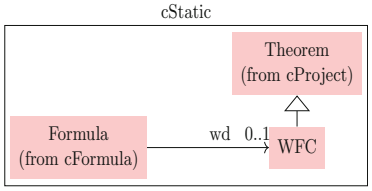

Fig. 4. cStatic context

The static correctness of a project is introduced in the context cStatic. First, sets and constants used by an assertion should be visible from the current context (Fig. 4).

```
// sets referenced by an assertion are declared
// in the context of the axiom or its ancestry
@sets_ext assertion ; predicate ; ref_sets ; set⁻¹ ⊆ id ∪ extend
// sets referenced by an assertion are declared
// in the context of the axiom or its ancestry
@csts_ext assertion ; predicate ; ref_csts ; constant⁻¹ ⊆ id ∪ extend
```

Second, well-formedness conditions are introduced through the subset `WFC` of `Theorem`. Static correctness is defined by associating through the `wd` partial function a wellformedness condition to a formula. It is an assertion to be proven, i.e. a theorem.

```
@WFF_ty WFC ⊆ Theorem
@WD_ty wd ∈ Formula ⇸ WFC
```

Furthermore, a `WFC` is well formed by construction and thus does not appear in the domain of `wd`.

```
@WD_WD predicate[WFC] ∩ dom(wd) = ∅
```

The well-definedness condition of a formula does not reference new sets or constants with respect to the initial formula.

```
@sets_wd wd;predicate; ref_sets ⊆ ref_sets
@csts_wd wd;predicate; ref_csts ⊆ ref_csts
```

2.5 Semantics

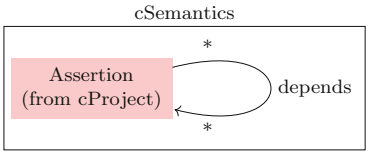

Fig. 5. cSemantics context

The soundness of a context is established through the proof of the validity of its theorems. The proof is abstracted by specifying axioms it uses. We introduce them through the `depends` relation (Fig. 5).

```
@depends_ty depends ∈ Assertion ↔ Assertion
```

An assertion can only depend on assertions that are visible from the current context. We also state that an assertion depends on its well-definedness condition.

```
@depends_extends assertion ;depends ⊆ (id ∪ extend); assertion
@depends_WD predicate;wd ⊆ depends
```

Moreover, the `depends` relation is supposed to be irreflexive and transitive.

```
@depends_irr id ∩ depends = ∅
@depends_trans depends;depends ⊆ depends
```

A sequent is built from theorems. Its hypotheses are all the assertions on which the theorem depends. Its conclusion is the predicate associated to the theorem itself. The semantics of the `theorem` annotation is thus defined by stating that this sequent is valid.

@THM_V ∀t· t ∈ Theorem ⇒ (depends;predicate)[{t}] ↦ predicate(t) ∈ Valid

3 Instantiation of Assertions

This section presents a metamodelisation and the validation of an instanciation mechanism proposed by the EBRP project. It is structured as a set of contexts as shown by the right hand side of Fig. 1.

3.1 Informal Presentation

Let us consider a simple generic example with an axiom used to prove a theorem:

context gen
sets T
axioms
@axm1 ∀x,y·x ∈ T∧y ∈ T ⇒ x=y
theorem @th1 T≠∅ ⇒ (∃x·T={x})
end

context instance1
axioms *// import th1 with T mapped to* ℤ
@@th ℤ≠∅ ⇒ (∃x·ℤ={x}) *// gen|T:=ℤ|th1*
end

Within a tool developed by the EBRP project, the proposed syntax to achieve the instanciation of `th1` in context `instance1` is given as a comment. It contains three fields: the context to be imported, instanciation parameters and the name of the target assertion. The instanciated formula can then be automatically generated.

Theorem `th1` can be proved in context gen using axiom `atm1`. `th` is a (considered correct) instance of `th1`. However, while it is expected that importing a theorem should give a theorem, actually `th` is not a theorem. For the assertion "imported theorems are valid" to be valid, a sufficient condition can be that all previous axioms should be imported before as theorems (to be proved), and with the same instance parameters. This is illustrated by the context `instance2` (see Fig. 6).

context instance2
axioms *// import axm1 and th1*
theorem @@PO ∀x,y·x ∈ ℤ∧y ∈ ℤ ⇒ x=y *// gen|T:=ℤ|axm1*
@@th ℤ≠∅ ⇒ (∃x·ℤ={x}) *// gen|T:=ℤ|th1*
end

So imported axioms appearing before imported theorems should become *proof obligations* (thus marked as theorems). Imported theorems should not be proved again and thus appear as axioms.

Here, the theorem `PO` cannot be proved. It follows that unsoundness of the context `instance2` is clearly pointed out, which is the expected behavior. To sum up, `instance1` should be rejected because an axiom preceding `th1` has not be imported as theorem; `instance2` is accepted by the static type checker but cannot be validated by the user. `instance3` is an example satisfying the static rules and for which the proof obligation for `atm1` instance can be discharged.

```
context instance3 // a correct instance of gen
sets Unit
constants void
axioms
  @part partition(Unit, {void})
  theorem @@atm1 ∀x,y·x ∈ Unit∧y ∈ Unit ⇒ x=y // gen|T:=Unit|axm1
  @@th Unit≠∅ ⇒ (∃x·Unit={x}) // gen|T:=Unit|th1
end
```

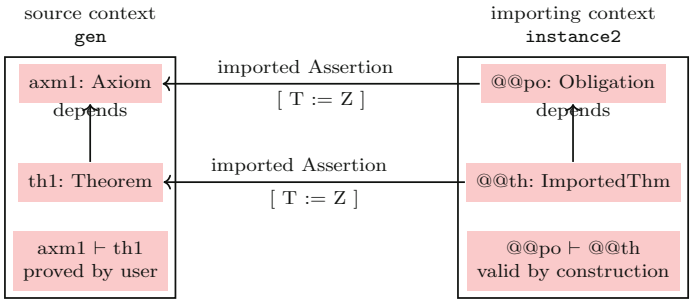

Fig. 6. Imported theorem

3.2 Importation of External Assertions

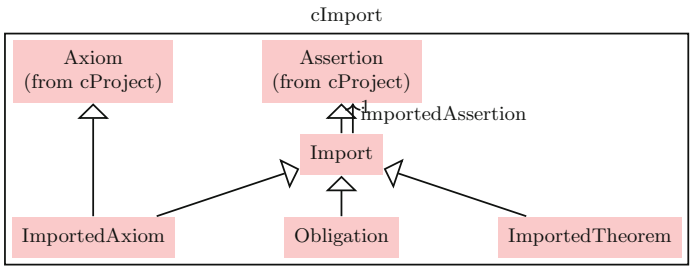

Fig. 7. cImport context

Importation points are added to standard Event-B assertions inside contexts. An importation point is a reference to an assertion of a remote context. Three kind of importations are distinguished: `imported axioms`, `imported theorems` and `obligations` (Fig. 7).

axioms
@Import_ty Import ⊆ Assertion
@imports_ty imports = assertion ▷ Import
@obl_ty Obligation ⊆ Import
@ImportedThm_ty ImportedTheorem ⊆ Import
@ImportedAxm_ty ImportedAxiom ⊆ Axiom

Obligations should be proved by the importing context and are thus declared as a subset of `Theorem`. An `ImportedTheorem` is valid by construction. An `ImportedAxiom` is an instance of an axiom that remains axiomatic in the importing context.

Note that imported theorems should not be proved again. Either their correctness are guaranteed by a meta-level argument (Transformation verification approach), or a proof can be automatically generated and checked (translation validation approach).

@Import_part **partition** (Import, ImportedAxiom, ImportedTheorem, Obligation)
@importedAssertion_ty importedAssertion ∈ Import → Assertion
@importedContext_def importedContext = importedAssertion;assertion^{-1}
@isObligation importedAssertion[Obligation] ⊆ Axiom
@isTheorem importedAssertion[ImportedTheorem] ⊆ Theorem ∪ ImportedTheorem

Last, a substitution of remote sets and constants is associated to each importation point. Sets are substituted by type expressions and thus only refer to sets. Constants are substituted by any expression of compatible type and may refer to sets or constants[9].

@isAxiom importedAssertion[ImportedAxiom] ⊆ Axiom
@importedContext_present importedContext^{-1};importedAssertion ⊆ axiom
@set_subst_ty set_subst ∈ Import × Set ⇸ Type
@cst_subst_ty cst_subst ∈ Import × Constant ⇸ Expression
// formal set parameters are declared in the source context
@setp_decl dom(set_subst) ⊆ ran(imports ⊗ set)
// formal constant parameters are declared in the source context
@cstp_decl dom(cst_subst) ⊆ ran(imports ⊗ constant)

Referred sets and constants should be visible by the importing context. All remotely accessed sets and constants should be substituted.

// constants of actual parameters for constants are visible by the importing context
@cs_rc (imports⊗(Context×Constant));cst_subst; ref_csts ⊆ (id∪extend);constant
// sets of actual parameters for constants are visible by the importing context
@cs_rs (imports⊗(Context×Constant));cst_subst; ref_sets ⊆ (id∪extend);set
// sets of actual parameters for sets are visible by the importing context
@ss_rs (imports⊗(Context×Set));set_subst; ref_sets ⊆ (id∪extend);set
// all sets used by the imported axiom are substituted
@sr_subst importedAssertion; predicate; ref_sets ⊆ dom(set_subst)

[9] $r \otimes s \triangleq \{x \mapsto (y \mapsto z) \mid x \mapsto y \in r \wedge x \mapsto z \in s\}$.

Note that the model could be refined to introduce typing conditions and constrain expressions to be used by substitutions.

3.3 Static Verification of Importations

In order to guarantee the validity of imported theorems, we link the dependency relation and importation clauses: all dependent assertions on which the imported assertion depends should also be imported. The importing clause should depend on these imports. Derived imports should use compatible substitutions, i.e. common sets or constants should be substituted by the same expressions.

```
@import_depends ∀ctx,imp·
  ctx↦imp ∈ imports ∧ importedAssertion(imp) ∈ Theorem ∪ ImportedTheorem ⇒
    (∀ax· importedAssertion(imp) ↦ ax ∈ depends ⇒
      (∃impa· ctx↦impa ∈ imports ∧
      imp ↦ impa ∈ depends ∧
      ax = importedAssertion(impa) ∧
      (∀s· imp↦s ∈ dom(set_subst) ∧ impa↦s ∈ dom(set_subst) ⇒
        set_subst(imp↦s) = set_subst(impa↦s)) ∧
      (∀c· imp↦c ∈ dom(cst_subst) ∧ impa↦c ∈ dom(cst_subst) ⇒
        cst_subst(imp↦c) = cst_subst(impa↦c))
      ))
```

3.4 Correctness of Theorem Instantiation

We first define the semantics of an imported assertion as the assertion obtained by applying the substitution declared in the importation clause to the imported assertion:

```
@importPredicate ∀imp· imp ∈ Import ⇒
  predicate(imp) = subst({s↦t | (imp↦s)↦t ∈ set_subst} ↦
        {c↦e | (imp↦c)↦e ∈ cst_subst} ↦ predicate(importedAssertion(imp)))
```

The correctness theorem states that the sequent formed by assertions on which the import depends and imported statement is valid. The imported statement can be itself an instance of another distant statement. We thus suppose that the union of the dependency and importation graphs is acyclic. The base case of the result is then given by the following theorem:

```
theorem @ImportValid ∀ctx,imp·ctx↦imp ∈ imports ∧
    importedAssertion(imp) ∈ Theorem ⇒
      (depends;predicate)[{imp}] ↦ predicate(imp) ∈ Valid
```

Given a context `ctx` and an importation point `imp`,

– let `th=importedAssertion(imp)` and suppose it is a theorem. By axiom `THM_V` of `cSemantics` (Sect. 2.5) the following sequent is valid:

```
(depends;predicate)[{th}] ↦ predicate(th)
```

– Using the set and constant substitutions (S, C) declared in importation point `imp`, we have `predicate(imp) = subst(S,C,predicate(th))` through axiom `importPredicate` of Sect. 3.4.
– Using axiom `subst_V` of Sect. 2.2, we can instanciate the sequent to get a new valid sequent Sq_2: `subst(S,C)[(depends;predicate)[{th}]]` $\mapsto$ `th`.
– Thanks to axiom `import_depends` of Sect. 3.3, antecedents of the imported theorem have been imported before with compatible substitutions, i.e. `imp` depends on these importation points.
– Thanks to axiom `subst_ref` of Sect. 2.1, applying substitutions (S, C) gives the same result.
– Thus, thanks to the monotonicity of validity (axiom `Valid_mono` of Sect. 2.2), the sequent concluding on `th` and containing its dependencies contains enough hypotheses to be valid.

4 Related Concepts

In this section, we review modularity constructs that can be found in various theorem provers. We reuse the same example to illustrate their features and compare them with respect to some key features.

4.1 Section Mechanism in Coq

Variables or hypotheses can be declared in a Coq [10] section and used freely in the rest of the section.

```
Section Gen.
  Variable T: Type.
  Hypothesis axm1: forall (x y: T), x=y.
  Theorem th1: (exists x:T, True) → exists x:T, (forall y:T, x=y).
    intros. destruct H as [x _]. exists x; auto.
  Qed.
End Gen.
```

When the section is closed, variables or hypotheses used by definitions or theorems are made parameters. Here `th1` is now seen as a function parameterized by a type `T`, a proof of `axm1` property, and a proof that `T` is inhabited. An instance of `th1` can be obtained through a partial call of `th1` with a type and a proof, leading to `th` definition.

```
Section Instance.
  Inductive unit: Type := One.
  Lemma unit_eq: forall (x y: unit), x=y.
    intros; destruct x; destruct y; auto.
  Qed.
  Definition th := th1 unit unit_eq.
End Instance.
```

Note that it is not necessary to introduce the lemma `unit_eq` before instantiating the theorem `th1`: a proof obligation could be generated through the use of `Program Definition`.

4.2 Module Mechanism in Coq

The theorem `th1` is now proved inside a parameterized module (or functor). Its parameter is typed by the module type `tGen` declaring `T` and `axm1`.

```
Module Type tGen.
  Parameter T: Type.
  Parameter axm1: forall (x y: T), x=y.
End tGen.
Module Gen(U: tGen).
  Theorem th1: (exists x:U.T, True) → exists x:U.T, (forall y, x=y).
    intros. destruct H as [x _]. exists x. apply U.axm1.
  Qed.
End Gen.
```

In order to use the contents of `Gen`, it must be instanciated by passing a module compliant with `tGen`. We introduce the module `U` defining a one-element type and proving the required property. Then `Gen` can be instanciated, which leads to the module instance `I`.

```
Module Instance.
  Module U <: tGen.
    Inductive unit: Type := One.  Definition T := unit.
    Lemma axm1: forall (x y: unit), x=y.
      intros; destruct x; destruct y; auto.
    Qed.
  End U.
  Module I := Gen U.
  Definition th := I.th1.
End Instance.
```

4.3 Locales in Isabelle/HOL

Locales [3] introduce a module system in the theorem prover Isabelle [11]. In the following, the locale `gen` is parameterized by the variable `T` typed as a set over the polymorphic type `'a` and states the assumption `atm1` over the variables of the set `T`. Thanks to this assumption, the theorem `th1` is then proved.

```
locale gen =
  fixes T :: "'a set"
  assumes atm1: "∀ x ∈ T. ∀ y ∈ T. x=y"
begin
  theorem th1: shows "T ≠ ∅ → (∃x ∈ T. T={x})"
  proof using atm1 by blast
  qed
end
```

The constant `S` is then *defined* as the singleton $\{1\}$. The latter set is used to give an *intepretation* to the locale `gen`. Then, this intepretation requires to

discharge the assumptions of the locale considered as proof obligations. After unfolding the definition of `S` and thanks to the powerful tactic `auto` these obligations is automatic.

```
definition "S = {1}"
interpretation i: gen "S" unfolding S_def by unfold_locales auto
```

4.4 Clones of Why3

In `why3` [5], a theory can declare abstract types and axioms which are used to prove theorems:

```
theory Gen
  type t
  axiom axm1: ∀ x y:t. x=y
  goal th1: (∃ x:t. ⊤ ) → ∃ x:t. ∀ y:t. x=y
end
```

The theory can be instanciated by given values to abstract types. Then axioms automatically become proof obligations. Proof attempts are then performed by the tool, and the contents of the instantiated theory become available to check remaining declarations of the `Instance` theory. The mechanism is less heavy but similar to Coq modules.

```
theory Instance
  type u = Unit
  clone Gen as G with type t = u (* axm1 instance is a proof obligation *)
end
```

4.5 Modules of TLA$^+$

In TLA$^+$ [8], modules can state assumptions and use them for proofs.

```
1 ──────────── MODULE gen ────────────
2 CONSTANTS T
3 ASSUME atm1 ≜ ∀ x, y ∈ T : x = y
5 THEOREM th1 ≜ T ≠ {} ⟹ (∃ x ∈ T : T = {x})
6 ⟨1⟩ QED BY atm1
8 ──────────────────────────────────────
```

In order to instantiate a module, one has to provide the constants (and the variables) that are used in this module. It is also possible to *inherit* the theorems of this instantiated module. However, each such theorem has an additional hypothesis consisting of the assumptions of the instantiated module. Otherwise stated, the reuse of a theorem is possible once the assumptions of its module have been discharged.

```
1  ┌──────────────────── MODULE instance ────────────────────
3  CONSTANT Z
5  THEOREM PO ≜ ∀ x, y : x ∈ Z ∧ y ∈ Z ⟹ x = y
7  i ≜ INSTANCE gen WITH T ← Z
9  THEOREM th ≜ Z ≠ {} ⟹ (∃ x ∈ Z : Z = {x})
10 ⟨1⟩ QED BY i!th1, PO
12 └─────────────────────────────────────────────────────────
```

4.6 Summary

In this section, we summarize some features offered by the module systems of the different proof environments. We have considered three criteria:

- The instantiation syntax: is it possible to extract a single theorem from a module? How formal parameters are given? Do they take implicit values (effective parameter with the same name as the formal parameter)?
- Interaction with provers: is it possible to prove obligations before instantiating the module, during module instantiation, or does the parameter property become a hypothesis of the extracted theorem?
- Extensibility of generic modules: is it possible to extend the generic theory by adding new parameters or new theorems?

	Coq Sections	Coq Modules	Isabelle	Why3	TLA	Rodin
Theorem import	✓					✓
Named parameters				✓	✓	
Implicit parameters					✓	✓
Type param. synthesis	✓	✓	✓	✓	✓	✓
Anticipated proof	✓	✓	✓	✓		✓
Proof obligation	✓		✓	✓		✓
Axioms as assumptions	✓				✓	
Extensibility		✓	✓	✓	✓	✓

As said in the introduction, the current integration of these features is investigated as a light extension of the Rodin tool. Ticks in the Rodin column indicate in-development features. Extensibility is by nature present through Rodin context extension. The other features presented in the table have no impact on the meta-level analysis and will be reflected by the choices done in the final IDE of the tool under construction.

5 Conclusion

In this paper, we have tried to put forward the meta level description of an extension currently studied within the EBRP project. The aim of this meta-modelisation is to validate the expected properties of theorem instantiation. It

has been done using current Event-B contexts. This axiomatic formalization could be considered as the formal specification of a static semantic checker of (extended) Event-B models. Also, an interesting evolution of this work could be to take into account the enhancements that are currently being implemented within the EBRP project, as well as some features already available within the plugin theory [7] (inductive types, definition by cases, ...). It would be an interesting validation of these enhancements. A more ambitious aim would be to standardize the syntax and semantics of Event-B [6] through Event-B. It would need to take into account machines, refinements, types, ...

Acknowledgement. We thank the anonymous reviewers for their helpful comments.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. Ballarin, C.: Locales: a module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014)
4. Butler, M., Maamria, I.: Practical theory extension in Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 67–81. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_5
5. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
6. Hallerstede, S.: On the purpose of Event-B proof obligations. Formal Aspects Comput. **23**(1), 133–150 (2011)
7. Hoang, T.S., Voisin, L., Salehi, A., Butler, M.J, Wilkinson, T., Beauger, N.: Theory Plug-in for Rodin 3.x. CoRR, abs/1701.08625 (2017)
8. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
9. Snook, C., Butler, M.: UML-B: a plug-in for the Event-B tool set. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, p. 344. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_32
10. The Coq Development Team. The Coq Proof Assistant, January 2021
11. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_7