Data-types definitions: Use of Theory and Context instantiations Plugins

Peter Riviere, Neeraj Kumar Singh, Yamine Aït-Ameur

Data-types definitions: Use of Theory and Context instantiations Plugins

Peter Riviere, Yamine Ait-Ameur, and Neeraj Kumar Singh

IRIT/INPT-ENSEEIHT
2 rue Charles Camichel 31071 Toulouse cedex 7. France
{peter.riviere,yamine,neeraj.singh}@toulouse-inp.fr

1 Introduction

In the context of the French national research agency (ANR) EBRP-EventB-Rodin-Plus [4][1] (Enhancing Event-B and Rodin) project, an extension of the Rodin platform [2] supporting the design of Event-B [1] models has been designed in the form of a plugin [6], namely the *context Instantiation plugin*. It allows the definition of generic contexts and their instantiation to define generic and reusable theories. Instantiable Sets and Constants with their axioms and theorems are defined in a context has been designed. A mechanism for instantiating such generic contexts by importing useful axioms and theorems in another context. A language for describing such instantiations has been defined. It is parsed in order to generate instantiated contexts.

In the work of [3,5], a mathematical extension of Event-B allowing the definition of theories was proposed and implemented in the so-called *Theory Plugin*.

In this paper, we investigate the correspondence between theories formalised in the theory plugin and those formalised in the context instantiation plugin. We present transformation rules that allow us to describe theories through contexts and their instantiation. The goal of these transformations is to provide an additional way to model theories in the core Event-B modelling language.

These correspondences for possible data-type definitions are described further below.

2 Direct definitions of data-types

The correspondence between data-types (non-inductive) and operators defined as *direct definitions* in the Theory Plugin is presented in this section.

2.1 Data-type transformation

Figures 1a and 1b show the correspondence between a data-type defined in a theory with type parameters and constructors (parameterised or not) and a context. Only two type parameters and two constructors have been defined for the sake of clarity in the presentation.

[1] https://www.irit.fr/EBRP/

```
THEORY Data_Type_Schema
TYPE PARAMETERS T1, T2
DATATYPES
    Struct :
        cons1 //base case 1
        cons2(el1:T1, el2:T2) //base
            case 2
```

(a) Type parameters and constructor definitions

```
CONTEXT Data_Type_Schema
SETS T1, T2, Struct
CONSTANTS cons1, cons2, cons2Type, el1
    , el2
AXIOMS
axm1: Partition(Struct, {cons1}, cons2Type)
axm2: cons2 ∈ T1 × T2 ⤖ cons2Type
axm3: el1 ∈ cons2Type → T1
axm4: el2 ∈ cons2Type → T2
```

(b) A corresponding context with sets and constants

Fig. 1: Data-type correspondence

2.2 Direct definitions of Operators: expressions

As shown on Figures 2a and 2b, theory based direct definitions of operators correspond to partial functions (defined using a lambda expression) where typing and the Well-definedness conditions are used to define the domain of these functions.

```
THEORY Direct_Expr_Schema
TYPE PARAMETERS T1, T2
OPERATORS
    op <expression>
        (arg1 : T1, arg2 : T2)
      well−definedness WD1, WD2
      direct definition
        Exp(arg1, arg2...)
```

(a) Theory based Direct definitions of operators: expression

```
CONTEXT Direct_Expr_Schema
SET T1, T2
CONSTANTS op
AXIOMS
axmn: op = (λargs1 ↦ args2·
              args1 ∈ T1∧
              args2 ∈ T2∧
              WD1 ∧ WD2 |
              Exp(args1, args2))
```

(b) A context for operators definitions: expression

Fig. 2: Direct definitions of operators: expression

2.3 Direct definitions of Operators: predicates

The same principle applies to operators defining predicates as it does to operators defining expressions. The correspondence for operators defined as predicates is shown in Figures 3a and 3b.

```
THEORY Direct_Predicate_Schema
TYPE PARAMETERS T1, T2
OPERATORS
    op <predicate>
        (arg1 : T1, arg2 : T2)
      well−definedness WD1, WD2
      direct definition
        P(arg1, arg2...)
```

(a) Theory based Direct definitions of operators: predicate

```
CONTEXT Direct_Predicate_Schema
SET T1, T2
CONSTANTS op
AXIOMS
axmn: op = (λargs1 ↦ args2·
              args1 ∈ T1∧
              args2 ∈ T2∧
              WD1 ∧ WD2 |
              bool(P(args1, args2)))
```

(b) A context for operators definitions: predicate

Fig. 3: Direct definitions of operators: predicates

3 Axiomatic data-types definitions

Axiomatic definitions on Figures 4a and 4b correspond to direct definitions in Section 2, except that the expression or predicate is not given in the function definition. In axiom $axmDefOp$, an axiomatically defined operator is formalised as a total function on the domain restricted by the well-definedness conditions and a resulting type (Res_Type). Then, in the theory, each axiom that characterises this operator is translated as an axiom in the context.

THEORY Axm_Schema
TYPE PARAMETERS T1, T2
AXIOMS OPERATORS
op $<expression>$
$(arg1 : T1, arg2 : T2) : res : Res\_type$
well−definedness WD1,WD2...
AXIOMS
axm1 : $Exp1(op, \ldots)$
...
axmn : $Expn(op, \ldots)$

(a) A theory based definition of axiomatic operators.

CONTEXT Axm_Schema
SET T1, T2
CONSTANTS op
AXIOMS
axmDefOp : $op \in \{args1 \mapsto args2 \cdot$
$arg1 \in T1 \land arg2 \in T2 \land WD1 \land WD2\}$
$\rightarrow Res\_type$
axm1 : $Exp1(op, \ldots)$
...
axmn : $Expn(op, \ldots)$

(b) A context based definition of axiomatic operators.

Fig. 4: Correspondence for axiomatic definitions of operators.

4 Inductive data-types definitions

In the Event-B modelling language, inductively defined data-types do not have their direct correspondence. This correspondence requires the introduction of a generic definition for inductive structures.

4.1 Generic context for inductive definitions by JR. Abrial and D. Cansell

To define Theory based inductive data-types correspondence, we use the generic definitions introduced by J.R. Abrial and D. Cansell in the EBRP project.

CONTEXT SchemaRecGen
SETS S_type, B_type
CONSTANTS wellfounded, fix, FrSB, S, B
AXIOMS
axm0: $S \subseteq S_type$
axm6: $B \subseteq B_type$
@axm1: $wellfounded = \{r \cdot r \in S \leftrightarrow S \land (\forall p \cdot p \subseteq S \land p \subseteq r[p] \Rightarrow p = \emptyset\}$
@axm2: $fix \in (\mathbb{P}(S \times B) \rightarrow \mathbb{P}(S \times B)) \rightarrow \mathbb{P}(S \times B)$
@axm3: $\forall h \cdot h \in \mathbb{P}(S \times B \rightarrow \mathbb{P}(S \times B) \land (, b \cdot a \subseteq b \land b \subseteq S \times B \Rightarrow h(a) \subseteq h(b)) \Rightarrow fix(h) = h(fix(h))$
axm4: $FrSb = \{r \mapsto g \mapsto fr \mid r \in wellfounded \land g \in S \times (S \nrightarrow B) \nrightarrow B \land$
$(\forall x, f \cdot x \in S \land f \in S \nrightarrow B \land r\tilde{[}\{x\}](f) \Rightarrow x \mapsto f \in dom(g)) \land$
$fr = fix(\lambda p \cdot p \in S \leftrightarrow B \mid$
$\{x, h \cdot x \in S \land r\tilde{[}\{x\}] \lhd h \subseteq p \land r\tilde{[}\{x\}] \lhd h \in r\tilde{[}\{x\}] \rightarrow B \mid x \mapsto g(x \mapsto r\tilde{[}\{x\}] \lhd h)\})\}$
lem1: $FrSb \in \{r \mapsto g \mid r \in wellfounded \land g \in S(S \nrightarrow B) \nrightarrow B \land (\forall x, f \cdot x \in S \land f \in S \nrightarrow B \land r\tilde{[}\{x\}] \subseteq dom(f)$
$\Rightarrow x \mapsto f \in dom(g))\} \rightarrow (S \leftrightarrow B)$
THEOREMS
thm1: $\forall r, g, fr \cdot r \mapsto g \mapsto fr \in FrSB \Rightarrow fr \in S \rightarrow B$
thm2: $\forall r, g, fr \cdot r \mapsto g \mapsto fr \in FrSB \Rightarrow (\forall x \cdot x \in S \Rightarrow fr(x) = g(x \mapsto r\tilde{[}\{x\}] \lhd fr))$

Fig. 5: A generic context with inductive sets definition operator $FrSb$.

The context `SchemaRecGen` of Figure 5 uses the definitions of well-founded relations and the fixpoint operator from contexts not shown in this paper. They are brought up for clarity.

The most important feature is the constant $FrSB$ allowing to define the semantics of operators defined on inductive types. It use the type constructors and the fixpoint operator. This function is further applied to formalise the theory based defined inductive types and operators.

4.2 Correspondence schema

Inductive definitions are given in two steps: first the data-type definition using inductive sets definitions and second the operators manipulating this data-type.

Inductive data-type definition. A recursive definition is based on an inductive type, which is depicted in Figures 6a and 6b as a set comprehension in which the inductive properties are encoded and the constants are elements of this set. The $IndType$ theory data-type corresponds to the set $IndType$, which is defined from the carrier set $IndTypeTYPE$. The $IndTypeSET$ defines the set of n-uplets corresponding to the n constructors of the data-type. In our case, $cons1\_El \mapsto cons2\_El \mapsto consinduc1\_El \mapsto consinduc2\_El$.

```
THEORY Ind_Data_Type_Schema
TYPE PARAMETERS T
DATATYPES
  IndType :
    cons1
          //base case 1
    cons2(el:T)
          //base case 2
    consinduc1(el :IndType)
          // inductive case 1
    consinduc2(el1 : T, el2 IndType)
          // inductive case 2
```

(a) Inductive type definition

```
CONTEXT Ind_Data_Type_Schema
SETS IndTypeTYPE, T
CONSTANTS IndType, IndTypeSET, cons1, cons2,
      consinduc1, consinduc2
AXIOMS
axm1: IndTypeSET =
      {indtype_El ↦ cons1_El ↦ cons2_El ↦
                  consinduc1_El ↦ consinduc2_El |
          indtype_El ⊆ IndTypeTYPE∧
          cons1_El ∈ indtype_El∧
          cons2_El ∈ T ⤖ (indtype_El \
              (ran(consinduc1_El)∪
              ran(consinduc2_El) ∪ {cons1_El}))∧
          consinduc1_El ∈ indtype_El ⤖ (indtype_El \
              (ran(cons2_El) ∪ ran(consinduc2_El)∪
              {cons1_El}))∧
          consinduc2_El ∈ T ⤖ (indtype_El \
              (ran(consinduc1_El) ∪ ran(cons2_El)∪
              {cons1_El}))∧
        (∀tr · cons1_El ∈ tr ∧ cons2_El[T] ⊆ tr∧
                          consinduc1_El[tr] ⊆ tr∧
                          consinduc2_2[T × tr] ⊆ tr
              ⇒ indtype_El ⊆ tr)
      }

axm2: IndType ↦
        cons1 ↦ cons2 ↦
        consinduc1 ↦ consinduc2
                  ∈ IndTypeSET
```

(b) Corresponding context with sets and constants

Fig. 6: Correspondence for inductive type

Inductive data-type operator definition. The correspondence for an inductively defined operator is shown in Figures 7a and 7b. The $FrSb$ operator is used for the defined inductive data-type $IndType$ in both base (with definitions of expressions $exp1$ and $exp2$) and inductive cases (with definitions of expressions $ExpInd1$ and $ExpInd2$).

```
THEORY
TYPE PARAMETERS T1, T2
OPERATORS
 op <expression> (
    arg1 : T1, arg2 : T2
                 )
 well-definedness
    WD1, WD2...
 recursive definition
  case cons1 =
       Exp1(···)
     //base case 1
  case cons2 =
       Exp2(···)
     //base case 2
  case consinduc1 =
       ExpInd1(op, ···)
     // inductive case 1
  case consinduc2 =
       ExpInd2(op, ···)
    // inductive case 2
```

(a) A theory based on Recursive definition

```
CONTEXT
INSTANCIATES SchemaRecGen Ind_Data_Type_Schema
SETS T, T1, T2
CONSTANTS op
AXIOMS
axmn: op = FrSB(
        {
        e ↦ ind_el | e ∈ IndType∧
        ind_el ∈ IndType∧
      (∃el · el ∈ T∧
        ind_el = consinduc1(el ↦ e)∨
      (∃el · el ∈ T∧
        ind_el = consinduc2(el ↦ e))
        } ↦
        {
         e ↦ f ↦ res |
         e ∈ IndType∧
         f ∈ {ind_el | ind_el ∈ IndType ∧ WD1 ∧ WD2} → Res_Type
       // Function domain definition
         (∀el, ind_el · el ∈ T ∧ ind_el ∈ IndType∧
             (e = consinduc1(el ↦ ind_el)∨
              e = consinduc2(el ↦ ind_el))
                  ⇒ ind_el ∈ dom(f))∧
      //Definition of base case
            (
            e = cons1(...) ⇒ res = Exp1(...)∨
            e = cons2(...) ⇒ res = Exp2(...)
            )∧
      // Definition of inductive case
            ((∃el, ind_el · el ∈ T ∧ ind_el ∈ IndType∧
                e = consinduc1(el ↦ ind_el) ⇒ res = ExpInd1(f, ...))∨
            (∃el, ind_el · el ∈ T ∧ ind_el ∈ IndType∧
                e = consinduc2(el ↦ ind_el) ⇒ res = ExpInd2(f, ...)))
          }
                    )
```

(b) A context based on Recursive definition

Fig. 7: Corresponding schema of recursive definition of operators

5 Conclusion

We provided a set of correspondences that allow theory-based data types to be translated as contexts. Except for the inductive definitions, which require the use of operators defining inductive sets borrowed from a generic context, this transformation is straightforward.

When we translate theories to context, we obtain context definitions expressed in the native Event-B modeling language, but we lose the structuring and semantic information available in the theories.

References

1. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (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 (Nov 2010)
3. Butler, M., Maamria, I.: Mathematical extension in Event-B through the rodin theory component (2010)
4. Ebrp, `https://www.irit.fr/EBRP/`
5. Hoang, T.S., Voisin, L., Salehi, A., Butler, M., Wilkinson, T., Beauger, N.: Theory plug-in for rodin 3.x (2017)
6. Verdier, G., Laurent, V.: Context instantiation plug-in: a new approach to genericity in Rodin. Rodin Workshop at ABZ'2021 (2021)