Proceedings of the 9th Rodin User and Developer Workshop, 2021

Virtual
June 8th, 2021

Editors:

Asieh Salehi Fathabadi	University of Southampton, UK
Yamine Aït Ameur	Toulouse National Polytechnique Institute, France
Thai Son Hoang	University of Southampton, UK
Colin Snook	University of Southampton, UK

UNIVERSITY OF Southampton

Part I

Summary

Executive Summary

Event-B is a formal method for system-level modelling and analysis. The Rodin platform is an Eclipse-based toolset for Event-B that provides effective support for modelling and automated proof. The platform is open source and is further extendable with plug-ins. A range of plug-ins has already been developed including ones that support animation, model checking, UML-B and text editor. While much of the development and usage of Rodin takes place within past and present EU/UK-funded projects: RODIN, DEPLOY, Advance, EBRP, HiClass, HD-Sec, there is a growing group of users and plug-in developers outside these projects.

The purpose of the 9th Rodin User and Developer Workshop was to bring together existing and potential users and developers of the Rodin toolset and to foster a broader community of Rodin users and developers. For Rodin users, the workshop provided an opportunity to share tool experiences and to gain an understanding of ongoing tool developments. For plugin developers, the workshop provided an opportunity to showcase their tools and to achieve better coordination of tool development effort.

The one-day programme consisted of presentations on tool development and tool usage. The presentations are delivered by participants from academia and industry. This volume contains the abstracts of the presentations at the Rodin workshop on June 8th, 2021. The presentations are also available online at https://wiki.event-b.org/index.php/Rodin_Workshop_2021.

The workshop was co-located with the ABZ 2021 conference and held virtually. The Rodin Workshop was supported by the University of Southampton and Toulouse National Polytechnique Institute.

Finally, we would like to thank the contributors and participants, the most important part of our successful workshop. Our special thanks to Jonathan Hammond for giving the keynote speech "Safety and Security Case Study Experiences with Event-B and Rodin", and to Alexander Raschke for the organisational support of the virtual event.

Organisers

Asieh Salehi Fathabad, University of Southampton
Yamine Aït Ameur, Toulouse National Polytechnique Institute
Thai Son Hoang, University of Southampton
Colin Snook, University of Southampton

Contents

Programme of the Rodin Workshop 2021

09:00–10:30

- Domain knowledge as Ontology-based Event-B Theories - ***Ismail Mendil****, Yamine Aït-Ameur, Neeraj Kumar Singh, Dominique Méry, and Philippe Palanque*
- OntoEventB: A Generator of Event-B contexts from Ontologies - ***Idir Ait-Sadoune***
- EVBT - an Event-B tool for code generation and documentation - ***Fredrik Öhrström***
- Scenario Checker: An Event-B tool for validating abstract models - ***Colin Snook****, Thai Son Hoang, Asieh Salehi Fathabadi, Dana Dghaym, Michael Butler*

10:30–11:00 Break

11:00–12:30

- Context instantiation plug-in: a new approach to genericity in Rodin - ***Guillaume Verdier****, Laurent Voisin*
- Examples of using the Instantiation Plug-in - ***Dominique Cansell****, Jean-Raymond Abrial*
- Data-types definitions: Use of Theory and Context instantiations Plugins - ***Peter Riviere****, Yamine Aït-Ameur, and Neeraj Kumar Singh*
- Towards CamilleX 3.0 - ***Thai Son Hoang****, Colin Snook, Asieh Salehi Fathabadi, Dana Dghaym, Michael Butler*

12:30–13:30 Lunch

13:30–15:00

- Keynote: Safety and Security Case Study Experiences with Event-B and Rodin - ***Jonathan Hammond, Capgemini Engineering***
- Large Scale Biological Models in Rodin - ***Usman Sanwal****, Thai Son Hoang, Luigia Petre, and Ion Petre*
- Formal Verification of EULYNX Models Using Event-B and RODIN - *Abdul Rasheeq,* ***Shubhangi Salunkhe***

Part II

Contributions

Domain knowledge as Ontology-based Event-B Theories

I. Mendil[1], Y. Aït-Ameur[1], N. K. Singh[1], D. Méry[2], and P. Palanque[3]

[1]INPT-ENSEEIHT/IRIT, University of Toulouse, France
[2]Telecom Nancy, LORIA, Université de Lorraine, France
[3]IRIT, Université de Toulouse, France
{ismail.mendil,yamine,nsingh}@enseeiht.fr, dominique.mery@loria.fr, palanque@irit.fr

1 Context of the study

In general system engineering approaches, particularly formal methods, do not offer specific constructs allowing the designer to define formal models of domain knowledge, nor mechanisms allowing to import such existing models. However, there exist formal modelling languages and/or meta-models sometimes standardised [6] that support the formalisation of such domain knowledge.

In this paper, we show how Event-B theories [1,3,5] can be defined to formalise such domain knowledge and the Rodin Platform [2] is used to carry out the formal development and the verification process. We first give a generic theory defining an ontology modelling language and then show its instantiation in the case of the ARINC 661 standard describing interactive cockpits as an example of critical interactive systems.

This work has been achived in the context of the French national research agency (ANR) project FORMEDICIS [7][1] FORmal MEthods for the Development and the engIneering of Critical Interactive Systems

2 A theory for ontologies

Since we are interested in formalising the domain knowledge associated to critical interactive interfaces and use the domain properties in our Event-B models, we need a framework to express such knowledge.

In our case, domain knowledge is formalised using ontologies. Therefore, as a first step, we have developed a generic theory allowing to describe ontologies. An extract of this theory is given in Listing 1. Classes `C`, properties `P` and instances `I` are defined as type parameters and a set of other relevant operators is provided.

`OntologiesTheory` entails several useful theorems thanks to the definition of the operators. `thm1` is an example, of a theorem establishing the transitivity of the `isA` operator. Another example is `thm2` which is trivial but has a great benefit for discharging poof-obligations in Event-B models.

[1] `https://anr.fr/Projet-ANR-16-CE25-0007`

```
THEORY OntologiesTheory
TYPE PARAMETERS C,P,I
DATA TYPES Ontology(C,P,I)
CONSTRUCTORS consOntology(classes:ℙ(C),properties:ℙ(P),instances:ℙ(I),
classProperties:ℙ(C×P),classInstances:ℙ(C×I),classAssociations:ℙ(C×P×C)
    , instanceAssociations:ℙ(I×P×I))
OPERATORS
isWDInstancesAssociations <predicate> (o: Ontology(C, P, I) ...
getInstanceAssociations <expression> (o: Ontology(C, P, I))
well−definedness isWDInstancesAssociations(o) ...
isWDOntology <predicate> (o: Ontology(C, P, I))
direct definition isWDClassProperites(o) ∧ isWDClassInstances(o) ∧
    isWDClassAssociations(o) ∧ isWDInstancesAssociations(o)
isA <predicate>(o: Ontology(C, P, I),c1: C,c2: C)
well−definedness isWDOntology(o),ontologyContainsClasses(o, {c1, c2})
direct definition getInstancesOfaClass(o,c1)⊆getInstancesOfaClass(o,c2)
addInstancesToAClass <expression> (o: Ontology(C, P, I),c: C,ii: ℙ(I))
well−definedness isWDOntology(o),ontologyContainsClasses(o, {c}),
    ontologyContainsInstances(o, ii),¬ classContainsInstances(o, c, ii)
direct definition consOntology(getClasses(o), getProperties(o),
getInstances(o),getClassProperties(o),getClassInstances(o) ∪ ({c} × ii),
    getClassAssociations(o), getInstanceAssociations(o))
isVariableOfOntology <predicate> (o:Ontology(C,P,I),ipvs:ℙ(I×P×I))
well−definedness isWDOntology(o)
direct definition ipvs ⊆ { i1 ↦ p ↦ i2 | i1 ∈ I ∧ p ∈ P ∧ i2 ∈ I ∧
i1 ↦ p ↦ i2 ∈ instances(o) × properties(o) × instances(o) ∧
    (∃c1, c2 · c1 ∈ C ∧ c2 ∈ C ∧ {c1, c2} ⊆ getClasses(o) ⇒
     (c1↦p↦c2∈getClassAssociations(o)∧p∈getClassProperties(o)[{c1}] ∧
       i1∈getClassInstances(o)[{c1}]∧i2∈getClassInstances(o)[{c2}]))}
THEOREMS
thm1: ∀o, c1, c2, c3· o ∈ Ontology(C, P, I) ∧ isWDOntology(o) ∧ c1 ∈ C ∧
    c2 ∈ C ∧ c3 ∈ C ∧ ontologyContainsClasses(o, {c1, c2, c3})
        ⇒ (isA(o, c1, c2) ∧ isA(o, c2, c3) ⇒ isA(o, c1, c3))
thm2: ∀o, cs1, cs2 · o ∈ Ontology(C, P, I) ∧ isWDOntology(o) ∧ cs1 ⊆ C ∧
    cs2 ⊆ C ∧ cs1 ≠ ∅ ∧ cs2 ≠ ∅ ∧ ontologyContainsClasses(o, cs1) ∧
    ontologyContainsClasses(o,cs2)
        ⇒(ontologyContainsClasses(o,cs1∪cs2))
...
END
```

Listing 1: Ontology Modelling Language Data Type

3 The case of Arinc 661

ARINC 661 [4] defines a standard Cockpit Display System (CDS) interface intended for all types of aircraft installations. The primary objective is to minimize the cost to the airlines, directly or indirectly. It normalises the definition of cockpit display system (CDS) interface and the communication protocol with user applications. In particular, its objective is to

- minimize the cost of acquiring new avionic systems to the extent it is driven by the cost of CDS development;
- minimize the cost of adding new display function to the cockpit during the life of an aircraft;
- minimize the cost of managing hardware obsolescence in an area of rapidly evolving technology;
- introduce interactivity to the cockpit, thus providing a basis for airframe manufacturers to standardize the Human Machine Interface (HMI) in the cockpit.

he standard defines two external interfaces between the CDS and the aircraft systems. The first is the interface between the avionics equipment (user systems) and the display system graphics generators. The second is a means by which symbology and its related behavior are defined.

3.1 ARINC 661 Concepts Declaration

We have considered the ARINC 661 specification document aiming to describe specific case studies —weather radar system. We have identified a set of relevant concepts, after a thorough analysis of the specification document. The formalisation of the ARINC 661 proceeds by instantiating `OntologiesTheory`, it yields the Event-B theory `ARINC661Theory` in Listings 2, 3 and 4. Retained concepts are often ARINC 661 widgets like `Label`, `CheckButton`, etc. However other elements are introduced for organisation purposes where the widgets may be used like `wellBuiltClassProperties` and `wellBuiltTtypesElements`.

```
THEORY ARINC661Theory
IMPORT THEORY PROJECTS OntologiesTheory
AXIOMATIC DEFINITIONS ARINC661Axiomatisation:
TYPES ARINC661Classes, ARINC66Properties, ARINC661Instances
OPERATORS
ARINC661_BOOL <expression> () : ARINC661Classes
A661_TRUE <expression> () : ARINC661Instances
A661_FALSE <expression> () : ARINC661Instances
CheckButtonStateClass <expression> () : ARINC661Classes
Label <expression> () : ARINC661Classes
A661_EDIT_BOX_NUMERIC_VALUES_CLASS <expression> () :
    ARINC661Classes
RadioBox <expression> () : ARINC661Classes
CheckButton <expression> () : ARINC661Classes
EditBoxNumeric <expression> () : ARINC661Classes
hasChildrenForRadioBox <expression> () : ARINC66Properties
hasCheckButtonState <expression> () : ARINC66Properties
hasValue <expression> () : ARINC66Properties
SELECTED <expression> () : ARINC661Instances
UNSELECTED <expression> () : ARINC661Instances
wellBuiltClassProperties<expression>():ℙ(ARINC661Classes×ARINC66Properties)
wellBuiltClassAssociations <expression> () : ℙ(ARINC661Classes ×
    ARINC66Properties × ARINC661Classes)
wellbuiltTypesElements<expression>():ℙ(ARINC661Classes×ARINC661Instances)
isWDRadioBox <predicate> (o: Ontology(ARINC661Classes,
    ARINC66Properties, ARINC661Instances) ) :
well-definedness isWDOntology(o)
isWDARINC661Ontology <predicate> (o: Ontology(ARINC661Classes,
    ARINC66Properties, ARINC661Instances) ) :
consARINC661Ontology <expression> (ii: ℙ(ARINC661Instances),
cii: ℙ(ARINC661Classes×ARINC661Instances),
ipvs:ℙ(ARINC661Instances ×ARINC66Properties×ARINC661Instances)):
Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)
well-definedness isWDARINC661Ontology(consOntology(ARINC661Classes,
ARINC66Properties,ii,wellBuiltClassProperties,wellbuiltTypesElements∪cii
,wellBuiltClassAssociations, ipvs))
isVariableOfARINC661Ontology <predicate> (o: Ontology(ARINC661Classes,
ARINC66Properties, ARINC661Instances),
ui: ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances)) :
well-definedness isWDOntology(o)
...
```

Listing 2: ARINC 661 theory declarations

3.2 ARINC 661 Concepts Definition

Since ontology standards do not define explicitly operators (they rely on ad'hoc APIs on their XML representation) to manipulate the concepts they describe, we have defined a set of operators allowing to manipulate the concepts of the `ARINC661Theory`. Moreover, the definition of the concepts are given in the shape of axioms. In particular, the effective type parameters for ontology instantiation are defined in *ARINC661ClassesDef*, *ARINC661PropertiesDef* and *ARINC661InstancesDef*. Also, `consARINC661Ontology` is provided for a valid construction of operator under the condition —formalised as well-definedness condition—that the arguments are valid. In addition, `isWDARINC661Ontology` allows the checking of the validity of a given ARINC 661 ontology.

```
AXIOMS
ARINC661ClassesDef: partition(ARINC661Classes, {ARINC661_BOOL},
    {ARINC661_STRING_CLASS}, {Label},{RadioBox}, {CheckButton}, ...)

ARINC66PropertiesDef: partition(ARINC66Properties, {hasVisible},
    {hasEnable},{hasAnonymous},{hasChildrenForRadioBox}, ...)
ARINC661InstancesDef: partition(ARINC661Instances, {A661_TRUE},
    {A661_FALSE},{A661_TRUE_WITH_VALIDATION}, LabelInstances, ...)
wellBuiltClassProperties:
wellBuiltClassProperties = ({Label} × {hasVisible,...})    ∪
    ({RadioBox} × {hasWidgetType, hasParentIdent, hasVisible, ...}) ∪
    ({CheckButton}×{hasWidgetType,hasVisible,hasEnable, ...})∪ ...)
consARINC661Ontology: ∀ii, cii, ipvs · ii ∈ ℙ(ARINC661Instances) ∧
    cii ∈ ℙ(ARINC661Classes × ARINC661Instances) ∧
    ipvs ∈ ℙ(ARINC661Instances×ARINC66Properties × ARINC661Instances) ∧
    wellbuiltTypesElements ∩ cii = ∅ ∧ ii ⊆ WidgetsInstances
    ⇒ consARINC661Ontology(ii, cii, ipvs) =
            consOntology(ARINC661Classes, ARINC66Properties, ii,
                wellBuiltClassProperties, wellbuiltTypesElements ∪ cii,
                wellBuiltClassAssociations, ipvs))
isWDEditBoxNumeric:
    ∀o·o∈ Ontology(ARINC661Classes,ARINC66Properties,ARINC661Instances)⇒
    (isWDRadioBox(o)⇔((∀ed,v·ed↦hasValue↦v∈getInstanceAssociations(o)
    ⇒v ∈ A661_EDIT_BOX_NUMERIC_ADMISSIBLE_VALUES)    )
isWDARINC661Ontology:
∀o· o ∈ Ontology(ARINC661Classes, ARINC66Properties, ARINC661Instances)
    ⇒(isWDOntology(o)∧isWDRadioBox(o)∧isWDEditBoxNumeric(o)⇒
        isWDARINC661Ontology(o))
...
```

Listing 3: ARINC 661 theory definitions

3.3 ARINC 661 theory Theorems

Last, the most important part concerns the properties embedded in the theory in the form of theorems. They are particularly useful to formalise standard requirements. Moreover, the validation of the fact that the ontology has the right structure is done through the theorems `thm1` and `thm2`. There are two important properties ensuring that the structure of the ontology is valid: the classes are related to properties already defined and similarly that the class associations component encompasses only the provided classes and properties. The theorem proofs are discharged thanks to the definition of `wellBuiltClassProperties`, `wellBuiltClassAssociations` and `wellBuiltTypesElements`

```
THEOREMS
thm1: ∀ii , cii , ipvs ·
ii ∈ ℙ(ARINC661Instances) ∧ cii ∈ ℙ(ARINC661Classes×ARINC661Instances)∧
ipvs ∈ ℙ(ARINC661Instances × ARINC66Properties × ARINC661Instances) ∧
wellbuiltTypesElements ∩ cii = ∅ ∧ ii ⊆ WidgetsInstances
    ⇒ isWDClassProperites(consARINC661Ontology(ii , cii , ipvs))
thm2: ∀ii , cii , ipvs ·
ii ∈ ℙ(ARINC661Instances) ∧ cii∈ ℙ(ARINC661Classes×ARINC661Instances) ∧
ipvs ∈ ℙ(ARINC661Instances× ARINC66Properties×ARINC661Instances) ∧
wellbuiltTypesElements ∩ cii = ∅ ∧ ii ⊆ WidgetsInstances
    ⇒ isWDClassAssociations(consARINC661Ontology(ii , cii , ipvs))
END
```

Listing 4: ARINC 661 theory theorems

4 Conclusion

This approach shows that axiomatising domain knowledge as ontologies expressed in Event-B theories is a suitable solution to handle standard requirements in system design. The defined theory for ARINC 661 standard specification has been used to develop Event-B models for several case studies like WXR user interface and TCAS application. We have used the defined data types to type state variables. Axioms and theorems have been used to prove specific properties on these case studies. The ontology description theory is presented as playing a the role of scaffolding for producing a domain-specific theories thanks to the Event-B theories featuring type genericity.

Due to the complexity of the theories developed for the aforementioned objective, we reported a serious bug to the development team of Plug-in Theory which was fixed and integrated in a future release. All Event-B developments are available and interested reader may contact the first author for a copy.

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 **12**(6) (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., Voisin, L.: Proposals for mathematical extensions for Event-B. Tech. Rep. (2009)
4. ARINC: Arinc 661 specification: Cockpit display system interfaces to user systems, prepared by aeec, published by sae, 16701 melford blvd., suite 120, bowie, maryland 20715 usa (June 2019)
5. Butler, M., Maamria, I.: Mathematical extension in Event-B through the rodin theory component (2010)
6. Calegari, D., Mossakowski, T., Szasz, N.: Heterogeneous verification in the context of model driven engineering. Science of Computer Programming, Elsevier Journal. **126**, 3–30 (2016)
7. Formedicis, `https://anr.fr/Projet-ANR-16-CE25-0007`

OntoEventB: A Generator of Event-B contexts from Ontologies

Idir Ait-Sadoune

LMF, CentraleSupelec, Paris-Saclay University
Plateau de Saclay, Gif-Sur-Yvette, France
idir.aitsadoune@centralesupelec.fr

1 Introduction

When designing hardware or software system, the integration of domain constraints becomes a determining factor to ensure a great match with the system requirements. This domain knowledge is most often modelled using ontologies that allow the expression of the domain properties. In the IMPEX project[1], we propose an approach to integrate domain ontologies into a system development process based on Event-B. It consists to annotate Event-B models using the ontology concepts, this assumes a formalization of the domain ontology in the Event-B method. Therefore, we propose an extensible generic transformation approach that develops an Event-B specification based on an ontology described in an ontological language. The integration of the domain ontology allows to constrain the system under design with the domain ontology and to validate domain properties.

In this paper, we present a generic approach to integrate domain description formalized by ontologies (OWL, OntoML, ...) into an Event-B formal development process. The proposed approach is conducted by transformation rules that define each ontological concept, the corresponding Event-B formalisation leading to build Event-B contexts expressing ontology concepts. This approach is implemented by the OntoEventB plug-in that has been developed to automatically support the formalisation of ontologies using the Event-B method.

2 Domain constraints integration approach

In order to integrate domain ontologies in the Event-B development process, we propose to formalize the ontology as a system data model within a context component. Thus, the machine variables take their values in ontology concepts and inherit domain constraints. The proposed integration approach is operated in a three steps process:

[1] This work was supported by a grant from the French national research agency ANR ANR-13-INSE-0001 (IMPEX Project http://impex.loria.fr).

1. *Formalization step.* The first step in the development process consists to formalize the system in the Event-B method. This leads to developing the machine component modelling the system behaviour using variables and events.
2. *Transformation step.* During this step, the domain ontology is translated into Event-B formalism. An ontology is translated into an Event-B context using sets, constants and axioms.
3. *Annotation step.* Once the context describing the ontology obtained, the integration of domain constraints is carried out by annotating machine variables by ontology context entities.

3 Ontology transformation : The *OntoEventB* plugin

The development of a transformation approach emerges as a natural choice for the expression of an ontology description in the Event-B language. This approach allows the transformation of an ontology described into an ontology language into an Event-B specification. It takes as inputs the constructs used to describe an ontology in the different ontology languages and as outputs Event-B language constructions. The transformation approach is based on correspondences between ontology languages and the Event-B language semantics.

The proposed ontology transformation approach in Event-B, detailed in [2,1], is fully supported by the *OntoEventB* RODIN plug-in[2] that automatically produces the Event-B formalization related to an ontology (OWL or Plib). The *OntoEventB* plugin takes as input an ontology description file and generates the corresponding Event-B Context.

4 Conclusion

Our results show that it is possible to handle formally domain knowledge in formal system developments with Event-B and the Rodin platform. Ontologies have been formalized within Event-B as contexts and a Rodin plug-in has been developed for this purpose. The proposed approach consists of defining models allowing to handle formal verification techniques and make it possible to handle explicit domain knowledge in such formal models.

References

1. Aït-Sadoune, I., Mohand-Oussaïd, L.: Building formal semantic domain model: An event-b based approach. In: Model and Data Engineering - 9th International Conference, MEDI 2019, Toulouse, France, October 28-31, 2019, Proceedings. pp. 140–155 (2019)
2. Mohand-Oussaïd, L., Aït-Sadoune, I.: Formal modelling of domain constraints in Event-B. In: Model and Data Engineering - 7th International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings. pp. 153–166 (2017)

[2] OntoEventB : https://wdi.supelec.fr/software/OntoEventB/

EVBT — an Event-B tool for code generation and documentation

Fredrik Öhrström
May 13, 2021
fredrik.ohrstrom@viklauverk.com

Evbt is a command line tool for generating code and documentation from Event-B models created using the Rodin toolset. It contains its own independent implementation of an Event-B formula parser and a typing system. The tool expects already proven Event-B models as input. Even though the tool is work in progress, it can already generate documentation and code for several existing models, for example the model for traffic lights controlling access to a bridge.[1]

To generate documentation for a Rodin workspace, you run:
`evbt docgen tex workspace/BridgeTrafficLights` to create `BridgeTrafficLights.tex`.

To generate code, you run: `evbt codegen c++ workspace/BridgeTrafficLights/m3.bum` to create a `BridgeTrafficLights.h` and a `.cc` file. For this model you also need to add `#define d 17` to the cc file since that constant is undefined in the model. You also need to add suitable set/get events to read and modify the state from the outside.

The evbt code generation creates a state machine hidden behind an API in a header file. The API will consist of events which in the final refinement still have parameters. Such events are translated into API functions with parameters mirroring the event parameters.

EVENT addLoan
Loan a book to a borrower, the book must not be on loan already.

ANY

borr
book

WHERE

grd1:	$borr \in borrowers$	Valid borrower.
grd2:	$book \in books$	Valid book.
grd3:	$book \mapsto borr \notin loans$	Not a necessary test, but used for this example anyway.
grd4:	$book \notin \mathrm{dom}(loans)$	The book is not loaned out already.

THEN

act1:	$loans(book) := borr$	Add a new loan in the storage.

END

```
bool LibraryImplementation::addLoan(uint64_t borr,uint64_t book)
{
    // Valid borrower.
    bool grd1 = borrowers.count(borr);
    if (!grd1) return false;
    // Valid book.
    bool grd2 = books.count(book);
    if (!grd2) return false;
    // Not a necessary test, but used for this example anyway.
    bool grd3 = loans.count(book)==0 || loans[book] != borr;
    if (!grd3) return false;
    // The book is not loaned out already.
    bool grd4 = loans.count(book)==0;
    if (!grd4) return false;
    // Add a new loan in the storage.
    loans[book] = borr;
    traceEvent("addLoan");
    return true;
}
```

[1] Jean-Raymond Abrial (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

Values are returned from the API function through parameters prefixed with "out_".

EVENT whoBorrowsBook
Return who is borrowing a book.

ANY
book
out_borrower

WHERE

grd1:	$book \in books$	Querying a valid book?
grd2:	$book \in \mathrm{dom}(loans)$	That is on loan?
grd3:	$out\_borrower = loans(book)$	Return the result through out.

END

```
bool LibraryImplementation::whoBorrowsBook(uint64_t book,uint64_t *out_borrower)
{
    // Querying a valid book?
    bool grd1 = books.count(book);
    if (!grd1) return false;
    // That is on loan?
    bool grd2 = loans.count(book);
    if (!grd2) return false;
    // Return the result through out.
    *out_borrower = loans[book];
    traceEvent("whoBorrowsBook");
    return true;
}
```

The evbt tool also provides a console for exploration of Event-B models and formulas. You start the console with: `evbt console` and you can now type for example: `add defaults` to fill the symbol table with a few default symbols for predicates and variables. You can now type `show formula "(P & x:BOOL) => Q"` and it will parse and print this formula: $(\mathtt{P} \wedge x \in \mathrm{BOOL}) \Rightarrow \mathtt{Q}$

If you start the console with a Rodin workspace as an argument:
`evbt console workspace/Library` then you can also print parts of the model with the command:
`show part "Library/events/whoBorrowsBook/guards"` or an invariant:
`show part "Library/invariants/inv3"`

You can embed evbt console commands inside a tex document:
`evbt docmod tex source.tex dest.tex workspace/Library`
You can now write `EVBT(...console command...)` in the `source.tex` file. These commands will be picked up by evbt, executed, and the result inserted into `dest.tex`. For example the event whoBorrowsBook described earlier was inserted into this document with the command `EVBT(show part "Library/events/whoBorrowsBook")`

The workshop covering the evbt tool will discuss the current features in more detail and how they are implemented in evbt as well as future directions for code generation. The evbt source can be downloaded here:
`https://github.com/viklauverk/EventBTool` and is available under the AGPLv3 license.

Scenario Checker: An Event-B tool for validating abstract models

Colin Snook, Thai Son Hoang, Asieh Salehi Fathabadi, Dana Dghaym, and Michael Butler

ECS, University of Southampton, Southampton, U.K.
{a.salehi-fathabadi, cfs, t.s.hoang, d.dghaym, m.j.butler}@soton.ac.uk

The Scenario Checker is a plugin tool for the Rodin platform for Event-B. It allows scenarios to be animated on Event-B models for validation purposes. Scenarios can be recorded, re-played and extended. The model can be annotated to distinguish and prioritise internal events and designate private variables. During *recording*, events that are designated as internal are automatically executed when enabled so that a form of 'run to completion' (or *big step*) is provided to represent the systems responses to changes in its environment. This allows the user to focus on developing the scenario in the environment while efficiently executing the response of the system. If necessary the user can take control of the response by executing internal events singly. This may be useful when the model still contains non-deterministic behaviour. The recorded sequence of external events and the values of non-private variables at each step can be saved in a scenario file. During *playback*, the sequence of external events to be executed is taken from the recorded scenario file while the internal events are again fired automatically. Hence the same scenario can be replayed after the model has been changed in order to test that scenario is still correctly executed in the new version of the model. At the end of each big step, critical (i.e. non-private) variables are compared with previous recorded values in order to highlight deviations. While replaying a scenario, the tool can be changed to record mode at any point so that an alternative ending to the scenario can be explored. This allows alternative scenarios to be efficiently developed from a common preamble.

The Scenario Checker is based on the ProB model checker and can be run in parallel with state visualisation tools such as BmotionStudio (which is included within ProB) and UML-B State-machine animation. The tool consists of the following views:

- Scenario Checker Control Panel view consisting of buttons to change the mode of the tool and to restart or save the recording. It also allows external events to be selected in record mode or shows the next external event to be executed in replay mode.
- Scenario Checker State view showing the state of non-private variables and, during replay mode, highlighting any differences.
- Scenario Checker Console view showing the execution of big step runs and other significant events.

The Scenario Checker is designed to allow abstract models to be validated. This implies that correspondingly abstract scenarios will be developed and refined (or

abstracted) in line with the model refinements. We have proposed a method for scenario based modelling (using the Scenario Checker) in [1]. Figure 1 shows the scenario checker in record mode (bottom 3 views) with BMotionStudio (top left) and Statemachine animation (top right). Figure 2 shows the scenario checker in playback mode, replaying the scenario recorded in Figure 1.

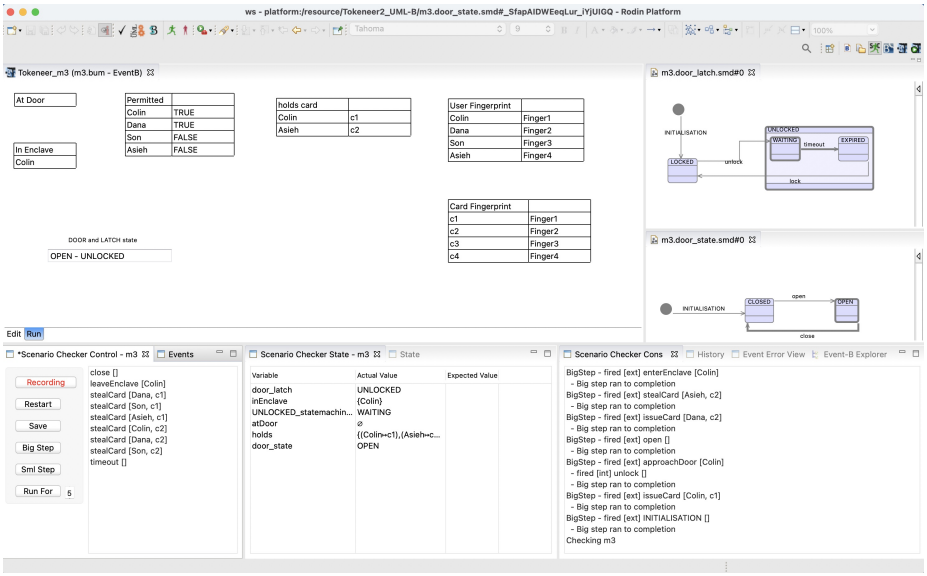


Fig. 1: Scenario Checker in recording mode

Fig. 2: Scenario Checker in playback mode

Acknowledgement: This work is supported by the following projects: - Hi-Class project (113213), which is part of the ATI Programme, a joint Government

and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture.

References

1. Colin Snook, Thai Son Hoang, Dana Dghaym, Asieh Salehi Fathabadi, and Michael Butler. Domain-specific scenarios for refinement-based methods. *Journal of Systems Architecture*, 112:101833, 2021.

Context instantiation plug-in: a new approach to genericity in Rodin*

Guillaume Verdier[1], Laurent Voisin[2]

[1] LMF/CentraleSupélec, Université Paris-Saclay
guillaume.verdier@irit.fr
[2] Systerel
laurent.voisin@systerel.fr

1 Introduction

The Rodin platform [1] offers an integrated development environment for designing software with Event-B [2]. One of the limitations of its core language is the lack of genericity: it is not possible to define generic data structures or to prove abstract theorems that hold for any type. When such generic constructs are needed, users end up copying and pasting large parts of contexts just to change a type. This approach is obviously cumbersome and error-prone. Alternatively, some plug-ins offer facilities for type parametricity. In particular, the Theory plug-in [4] provides many extensions to the mathematical language that can be used in Rodin, one of them enabling type parametricity. However, these extensions are introduced through a new type of files, *theories*, requiring users to learn how to write these theories and to go through a process of deployment to make them available in Rodin contexts.

As part of the EBRP project, a new, lighter approach has been proposed by Jean-Raymond Abrial. It is based on existing contexts and simply adds an option to instantiate theorems of a context in another one. Intuitively, it is similar to the copy-and-paste method, but done in a safe way by a plug-in rather than manually.

2 Plug-in usage

Defining a generic context can be done using just core Rodin tools, without the plug-in itself. Any context can be instantiated; its carrier sets and constants can be used as generic parameters and substituted with a concrete type or value during instantiation.

Instantiating theorems of a generic context is done by creating an axiom and specifying what it instantiates in the comment box. This specification identifies the instantiated theorem (its name, the name of the context in which it is defined and optionally the name of the project) and may provide some type or value substitutions. It can be manually written or generated through a graphical wizard.

In its most basic operating mode, the plug-in checks that the predicate of the axiom matches the predicate of the instantiated theorem, after applying the substitutions if necessary. Therefore,

*This work is supported by the French ANR project Event-B Rodin Plus (EBRP, ANR-19-CE25-0010).

Rodin users who used to manually copy-and-paste generic theorems can just add comments describing the instantiation and the plug-in will check that no errors were made.

For new developments, users can leave the predicate empty and merely provide the specification of the instantiation: the plug-in can generate the predicate of the instantiated theorem.

Since the plug-in uses Rodin's contexts, users of the plug-in can freely share their work with other users who do not have the plug-in installed: they will see normal contexts with comments describing the instantiations and will be able to use them as any other context.

3 Future improvements

This new plug-in is still in an early stage of development and could use various improvements. The features highlighted in this document are therefore subject to change.

Instantiation is currently done on a theorem basis, which can be cumbersome if one wants to instantiate many theorems from a single context, although the wizard can generate all the instantiations at once. It could be more user-friendly to instantiate contexts first, similarly to the import mechanism, and then be able to use their theorems freely.

When a theorem is instantiated, an axiom is created: considering that the theorem has been proved on some abstract types and constants, there is no need to ask users to reprove it on more specific types or values. However, some issues appear if the proof of the instantiated theorem depends on axioms that may not hold on the substituted types or values [3]. Different solutions have been proposed, such as imposing some restrictions on axioms in instantiated contexts or adding these axioms as proof obligations of the instantiation.

4 Conclusion

The context instantiation plug-in offers a new approach to genericity in the Rodin platform directly integrated in contexts. It should be easy to adopt in existing projects and require little changes to be useful. It is already being used by members of the EBRP project. In particular, Jean-Raymond Abrial and Dominique Cansell have implemented an extensive set of generic contexts ranging from generic data types (such as lists and binary trees) to mathematical theories.

The plug-in is still under active development and should be publicly released soon.

References

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. *Rodin: an open toolset for modelling and reasoning in Event-B.* International Journal on Software Tools for Technology Transfer, 12(6):447–466, Nov 2010.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] Jean-Paul Bodeveix and Mamoun Filali. *Event-B formalization of Event-B contexts.* Submitted.

[4] Thai Son Hoang, Laurent Voisin, Asieh Salehi, Michael J. Butler, Toby Wilkinson, and Nicolas Beauger. *Theory plug-in for Rodin 3.x.* CoRR, abs/1701.08625, 2017.

Examples of using the Instantiation Plug-in

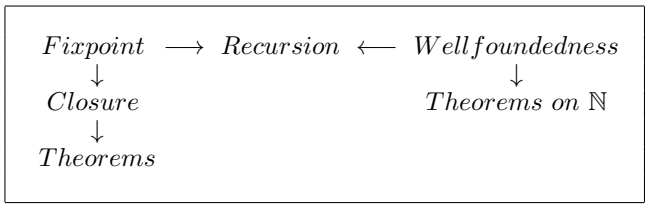
Dominique Cansell and Jean-Raymond Abrial

Lassy and Marseille, France

dominique.cansell@gmail.com jrabrial@neuf.fr

In a companion paper [1], Guillaume Verdier and Laurent Voisin presented a new approach to genericity in the Rodin toolset: this approach is made practical by means of an Instantiation Plug-in. In the present short paper[1] we propose some examples of using this new approach. Note that we constructed more examples: we only present here the most important ones. These examples are preliminary as the plug-in is still under development as stated in [1]. The key to this presentation is to show how such examples can be structured using the Instantiation Plug-in.

Two basic examples are independent: Fixpoint and Wellfoundedness. Other examples depend directly or indirectly of them. This is indicated in the following diagram.



1 Fixpoint

Given a set S and a set function h built on S: $h \in \mathbb{P}(S) \to \mathbb{P}(S)$, a fixpoint of h is a subset $\text{fix}(h)$ of S such that $\text{fix}(h) = h(\text{fix}(h))$. Here is a proposal:

$$\text{fix}(h) \;\widehat{=}\; \text{inter}(\{s \mid s \subseteq S \;\wedge\; h(s) \subseteq s\})$$

Assuming that the function h is monotone; we have (Tarski)

$$(\forall a, b \cdot a \subseteq b \;\Rightarrow\; h(a) \subseteq h(b)) \;\Rightarrow\; \text{fix}(h) = h(\text{fix}(h)$$

Moreover $\text{fix}(h)$ is the least fixpoint:

$$\forall t \cdot t = h(t) \;\Rightarrow\; \text{fix}(h) \subseteq t$$

2 Closure

Given a set S and a relation r built on S: $r \in S \leftrightarrow S$, the $\text{closure}(r)$ is defined to be the following fixpoint:

$$\text{closure}(r) \;=\; \text{fix}(\lambda s \cdot s \in S \leftrightarrow S \mid r \cup (s\,;r))$$

Since we use the fixpoint operator, we have to instantiate (adapt) its definition as provided within the corresponding section. This is exactly what is allowed by the Instantiation Plug-in.

[1] This work is supported by the French ANR project Event-B Rodin Plus (EBRP, ANR-19-CE25-0010)

3 Closure Theorems

We now instantiate the closure definition and prove the following theorems:

$$\text{closure}(r) \;=\; \text{fix}(\lambda s \cdot s \in S \leftrightarrow S \mid r \cup (s \,;\, r))$$

$$\text{closure}(r^{-1}) \;= (\text{closure}(r))^{-1}$$

4 Wellfoundedness

We are given a set S and a binary relation r built on S: $r \in S \leftrightarrow S$. If for any x belonging to the range of r, we follow r^{-1} and reach a point which is not in the range of r after a *finite* travel, the relation r is said to be well-founded: $\text{wf}(r)$. It can be given the following formal definition:

$$\text{wf}(r) \;\;\widehat{=}\;\; \forall p \cdot p \subseteq S \;\wedge\; p \subseteq r[p] \;\Rightarrow\; p = \varnothing$$

It can be proved that we have an induction rule for a set with a well-founded relation. Here is the corresponding formal definition:

$$\forall q \cdot q \subseteq S \;\wedge\; (\forall x \cdot x \in S \;\wedge\; r^{-1}[\{x\}] \subseteq q \;\Rightarrow\; x \in q) \;\Rightarrow\; S \subseteq q$$

5 Theorems on Natural Numbers

We now instantiate the set S of previous section to the set $\mathbb{N}$ of natural numbers. It can be proved that the relation "$+1$" on $\mathbb{N}$ is well-founded. As a consequence, we can deduce the classical induction rule for $\mathbb{N}$

6 Recursion

We are given a set S and a well-founded relation r built on S. We are given another set B and a function g defined as follows: $g \in (S \times (S \pfun B)) \pfun B$. Then there exists a unique total function f from S to B with the following property:

$$\forall x \cdot x \in S \;\Rightarrow\; f(x) = g(x \mapsto r^{-1}[\{x\}] \domsub f)$$

This function is defined by means of a fixpoint. We have thus to instantiate the definition of wellfoundedness and that of fixpoints defined in earlier sections.

7 Conclusion

All proofs alluded in this paper were successfully performed using the Instantiation Plug-in and the Rodin Tool.

References

1. Guillaume Verdier and Laurent Voisin: *Context Instantiation Plug-in: a new approach to genericity in Rodin*

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.

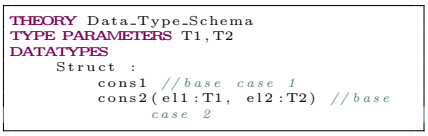
2 Direct definitions of data-types

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

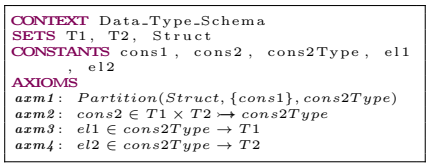
2.1 Data-type transformation

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/



(a) Type parameters and constructor definitions

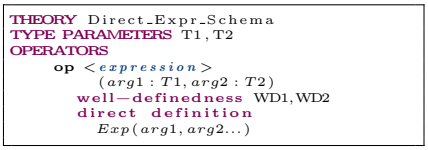


(b) A corresponding context with sets and constants

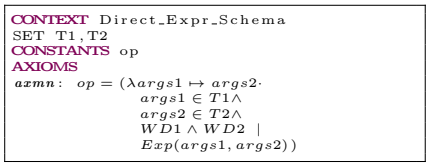
Fig. 1: Data-type correspondence

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.



(a) Theory based Direct definitions of operators: expression

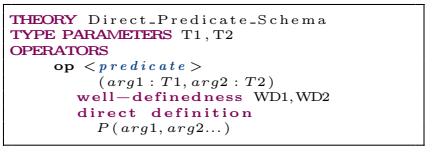


(b) A context for operators definitions: expression

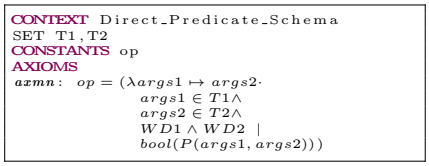
Fig. 2: Direct definitions of operators: expression

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.



(a) Theory based Direct definitions of operators: predicate

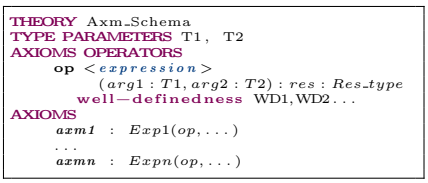


(b) A context for operators definitions: predicate

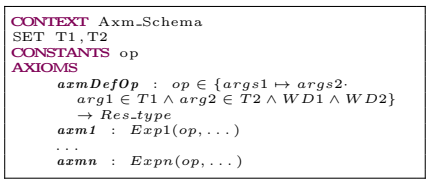
Fig. 3: Direct definitions of operators: predicates

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.



(a) A theory based definition of axiomatic operators.



(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.

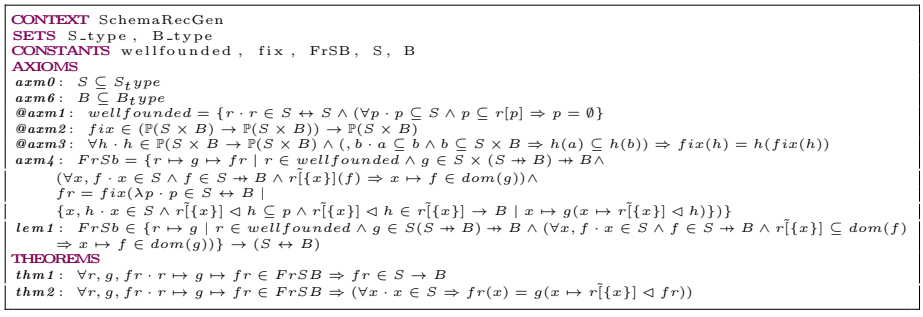


Fig. 5: A generic context with inductive sets definition operator $FrSb$.

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)

Towards CamilleX 3.0

Thai Son Hoang, Colin Snook, Asieh Salehi Fathabadi, Dana Dghaym, and Michael Butler

ECS, University of Southampton, Southampton, U.K.
{t.s.hoang, cfs, d.dghaym, a.salehi-fathabadi, m.j.butler}@soton.ac.uk

The CamilleX Framework [3] provides a textual representation of Event-B models for the Rodin Platform (Rodin) platform. It supports both (1) direct extensions of the Event-B syntax to support modelling extensions such as machine inclusion [2] and record structure [1], and (2) indirect extensions via *containment* mechanism to such as UML-B diagrams [4]. In this presentation, we will take a look at some of the remaining issues and proposal to address them in the next release of CamilleX.

Element Ordering. Currently, CamilleX relies on the Event-B Eclipse Modelling Framework (EMF) framework to store the semantics model of the Event-B machines and contexts. Modelling elements of the Event-B constructs are stored in different "collections", one for each carrier sets, constants, axioms, variables, invariants, events, and "extensions" (e.g., record structure). As a result, there is no ordering information is kept between the different modelling elements. For example, the current implementation of record structure generates the record-related invariants after all normal context axioms (similarly for records in a machine). This could cause problems when the order of the elements matter. Consider the following declarations of a record r with a field A of type S. Axiom @axm1 indicates the surjectivity of S with respect to field A.

```
// CamilleX context with Record
sets S
axioms
  @axm1: ∀s · s ∈ S ⇒ s ∈ ran(A)
record r
  A : one S
```

```
// Translated Rodin context
sets S r
constants A
axioms
  @axm1: ∀s · s ∈ S ⇒ s ∈ ran(A)
  // record field translation axiom
  @axm_r_A: A ∈ r → S
```

This translated model is ill-formed as the type for A can not be determined for axiom @axm1. We will need to be able to interleave the record declaration and axioms as necessary.

As result, the new version of Event-B EMF (currently under development) will store the modelling elements in a generic collection, named orderedChildren (the other collections will become derived attributes to minimise the impact of the changes). The syntax of CamilleX for XMachines and XContexts can be updated to allow the interleaving of modelling elements. We are working on updating the record-structure generator to take advantage of the new ordering.

Identifier Declaration Taking advantage of the ordering allowing us to interleave the modelling elements, we can eliminate the block such as **axioms**, **invariants**,

events. Each element will be prefixed with a singular keyword, such as **axiom**, **invariant**, **event** (notice that the **event** keyword already exists). Moreover, identifier elements such as **constants**, **variables**, and parameters can be declared together with their types and their (initial) values. This allows all information related to the constants and variables in one place. For example, instead of

```
variables a
invariants
 @a−typeof: a ∈ ℕ
event INITIALISATION
begin
 @a−init: a := 0
end
```

we can have

```
variable a : ℕ := 0
```

and the relevant invariants can be generated accordingly.

Support Context Instantiation For context instantiation [5], we will need to distinguish between the abstract sets and constants that need to be instantiated and the properties of them that need to be proved during the instantiation. These elements can be added to the syntax of CamilleX for instantiated and instantiating contexts.

```
context c0
abstract sets S
abstract constants c
axiom A(S, c)
```

```
context d0
sets T
constants d
axiom A(S, c)
instantiates c0(T, d)
```

The translation from CamilleX will flatten the instantiated and instantiating contexts into the facility provided by the instantiation plug-in [5].

Acknowledgement. This work is supported by the following projects:

– HiClass project (113213), which is part of the ATI Programme, a joint Government and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture.
– HD-Sec project, which is funded by the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

References

1. Asieh Salehi Fathabadi, Colin Snook, Thai Son Hoang, Dana Dghaym, and Michael Butler. Extensible record structures in Event-B. In *ABZ 2021*, 2021.
2. Thai Son Hoang, Dana Dghaym, Colin F. Snook, and Michael J. Butler. A composition mechanism for refinement-based methods. In *ICECCS 2017*, pages 100–109. IEEE Computer Society, 2017.
3. Thai Son Hoang, Colin Snook, Dana Dghaym, Asieh Salehi Fathabadi, and Michael Butler. The CamilleX framework for the Rodin Platform. In *ABZ 2021*, 2021.
4. Mar Yah Said, Michael J. Butler, and Colin F. Snook. Language and tool support for class and state machine refinement in UML-B. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009*, volume 5850 of *Lecture Notes in Computer Science*, pages 579–595. Springer, 2009.
5. Guillaume Verdier and Laurent Voisin. Context instantiation plug-in: a new approach to genericity in Rodin. (in Rodin Workshop 2021), 2021.

Large Scale Biological Models in Rodin

Usman Sanwal[1], Thai Son Hoang[2], Luigia Petre[1], and Ion Petre[3,4]

[1] Faculty of Science and Engineering, Åbo Akademi University, Finland
[2] School of Electronics and Computer Science, University of Southampton, UK
[3] Department of Mathematics and Statistics, University of Turku, Finland
[4] National Institute for Research and Development in Biological Sciences, Romania

1 Introduction

Biological systems are typically very large and complex, so much that it is remarkably difficult to capture all the necessary details in one modeling step. The concept of refinement – gradually adding details to a model while preserving its consistency – is thus instrumental. We provide an Event-B based modeling framework for the modeling of large and complex biological systems. Other formal modeling methods such as process algebra, Petri nets and many others has also been used for biomodeling see [1], [3]. The advantage that Event-B brings is that it has refinement as the key concept of the development method. System details can be introduced in several steps and the tool manages all the links between all the intermediary models. Consistency of refinement ensures that all the properties of a model M_i are still valid in its direct refinement successor M_{i+1}. At each refinement step, one can focus on the new elements that are introduced and on their consistency with the previous model.

In this work, we model two biological systems using refinement in Event-B, i.e., we first model a simple, more abstract model of the system and then we add more details in a correct-by-construction manner. The two systems we address are the heat shock response and the ErbB signaling pathway. Modeling the heat shock response in Event-B succeeded before [4]: we started with the abstract model having 10 variables and 17 events and ended up with the concrete model having 22 variables and 57 events. Modeling the ErbB signaling pathway only succeeded earlier [2] for the abstract model, with 110 variables and 242 events. The concrete model would have 1320 events, which was not supported by Rodin. With our current approach we are able to handle such a big model.

2 Modelling reaction networks in Event-B

We model reaction networks as sets of biochemical reactions, where each reaction specifies its reactants, products, and possibly inhibitors and catalyzers. These reactions can be either reversible or irreversible and each reaction could also have an associated flux, describing the rate at which its products are produced and its reactants consumed. With these assumptions, a reaction r can be written as a rewriting rule of the form:

$$r: \quad m_1X_1 + m_2X_2 + ... + m_nX_n \rightarrow m'_1X_1 + m'_2X_2 + ... + m'_nX_n, \qquad \text{(R)}$$

where $\mathcal{S} = \{X_1, \ldots, X_n\}$ is the set of *reactants* and $m_1, ..., m_n, m'_1, ..., m'_n \in \mathbb{N}$ are non-negative integers.

To model reaction networks in Event-B: every reactant is modeled by a variable and every reaction is modeled by an event. Invariants ensure the correctness of each reactant and biological properties of interest, for instance the mass conservation rule that ensure that the number of certain reactants is constant.

Thus, $X_1, X_2, ..., X_n$ are the variables of the model, their type being specified by corresponding invariants. Initial values for all of these variables are set in the initialisation event. For each reaction r of the reaction network, we specify in its guard that it must have enough of each reactant in order for the reaction to be enabled, while the action of the event specifies the changes to happen to each variable. This general scheme can be applied to model any reaction network in Event-B. We have demonstrated its applicability for the case studies of heat shock response model and the ErbB signalling pathway model in [4] and [2] respectively.

In this work, we revisited the Event-B model of heat shock response model and the ErbB signalling pathway model to make them more scalable. We demonstrate here that how a particular modeling feature of Event-B – the common mathematical function – enables us to significantly reduce the concrete models sizes. The relation between the abstract and the concrete forms of a reactant is captured with a function. This enables us to model the concrete reactions more elegantly and concisely, and as a result, the total number of events in the refined model is reduced significantly. In the case of the heat shock response, the complete model is described through 21 events, instead of the 57 events of the model in [4]. The difference in the case of the ErbB model is drastic, as we now need only 242 events for the full model of the ErbB signaling pathway in Rodin, instead of 1320 events. We have successfully implemented the model in Rodin and all of the proof obligations were dischrged automatically. To the best of our knowledge, this is the largest-ever model built in Rodin.

References

1. Federica Ciocchetta and Jane Hillston. Process algebras in systems biology. In Marco Bernardo, Pierpaolo Degano, and Gianluigi Zavattaro, editors, *Formal Methods for Computational Systems Biology*, volume LNCS 5016, pages 265–312, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
2. Bogdan Iancu, Usman Sanwal, Cristian Gratie, and Ion Petre. Refinement-based modeling of the erbb signaling pathway. *Computers in Biology and Medicine*, 106:91 – 96, 2019.
3. Ina Koch, Wolfgang Reisig, and Falk Schreiber. *Modeling in systems biology: the Petri net approach*, volume 16. Springer Science & Business Media, 2010.
4. Usman Sanwal, Luigia Petre, and Ion Petre. Stepwise construction of a metabolic network in event-b. *Computers in Biology and Medicine*, 91(C):1–12, 2017.

Formal Verification of EULYNX Models Using Event-B and RODIN

Abdul Rasheeq[1,2] and Shubhangi Salunkhe[1,2]

DB Netz AG[1], Neovendi GmBH[2]

{shubhangi.salunkhe-extern, abdul.rasheeq-extern}@deutschebahn.com
{s.salunkhe, a.rasheeq}@neovendi.com

Abstract

Advancements in technology have improved the safety of railway transportation systems. However, railway operators face challenges when constructing and maintaining systems that use components from a variety of suppliers. Obsolescence may result in the need to replace complex components with equivalent parts while ensuring that the operation and safety of the overall system are not compromised.

This abstract presents a model-based systems engineering (MBSE) approach for the specification of Deutsche Bahn's railway interlocking system (RIS) to address two issues: The first issue is the separation of the life cycles of the interlocking core and the field elements to reduce the vendor lock-in risks when upgrading or renewing railway field elements such as a level crossing, point machine etc. The second issue is achieving the required assurance that safety properties are preserved by the specification.

In the EULYNX consortium, European railway infrastructure managers develop standard interfaces and subsystems for the next generation command, control and signaling (CCS) architecture. Model-based systems engineering (MBSE) is used to ensure soundness and completeness of the specified interfaces. In order to achieve this, we propose a diagrammatic MBSE framework so that safety compliant standardized models of the interface specifications can be handed over to the railway suppliers. In this framework, Infrastructure managers define the appropriate use case descriptions and modelling experts convert the use cases into executable SysML models using the Windchill Modeler3 tool. Subsequently, infrastructure managers evaluate whether the specified interfaces are sound regarding their intended use applying simulation-based testing. Modelling of the interlocking system interfaces using SysML which is a semi-formal language has already led to significant improvements in the quality of created specifications but does not allow formal verification of system properties. Our proposed framework enables the transformation of SysML models into the Event-B formal language to prove the safety requirements. Figure 1 depicts the overall verification and validation approach.

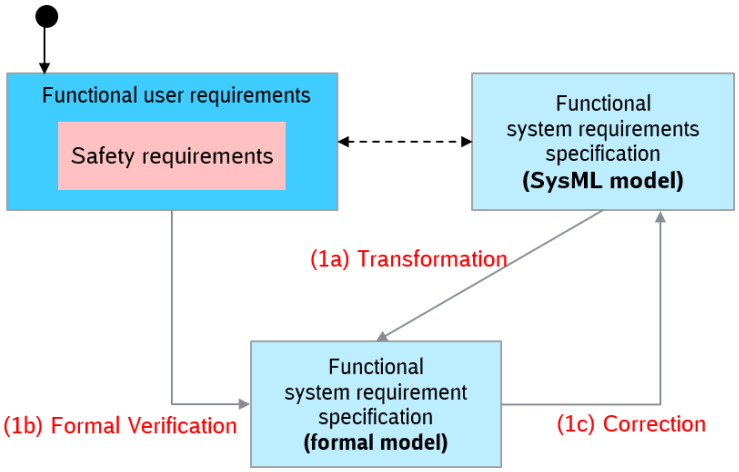
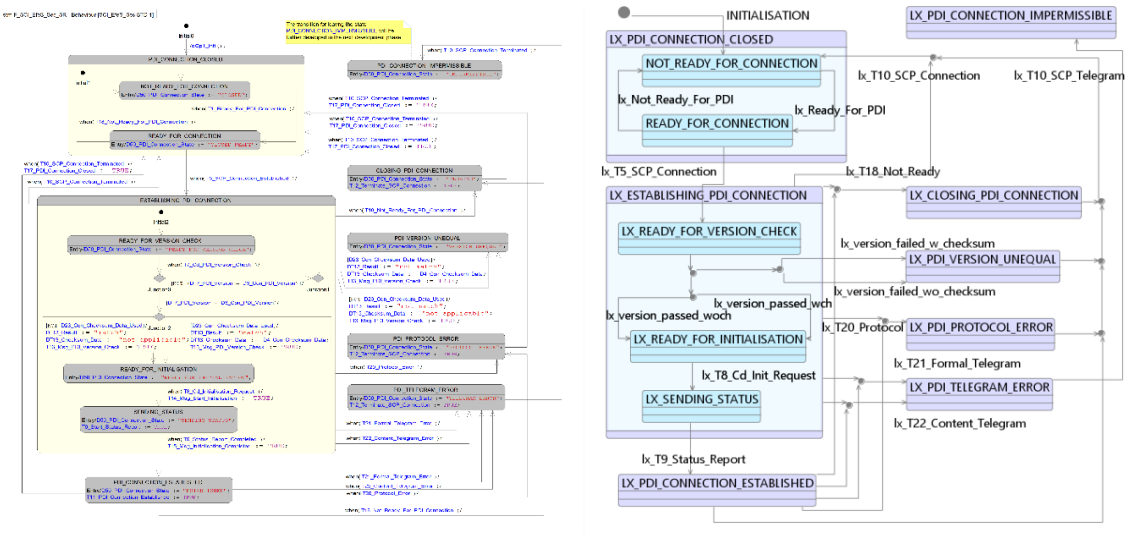


Figure 1: The principle behind formal modelling and verification

Initially, we achieved the transformation in step (1a) manually using UML-B which is a UML-like front end for Event-B and performed the formal verification of safety requirements. This transformation was applied on multiple EULYNX interface models. The following Use Case 1 shows one of the equivalent UML-B model of SysML model achieved using manual transformation.



Use Case 1: SysML Model and equivalent UML-B Model

During this formal transformation and verification, we observed that, the manual transformation of models is time consuming especially when the complexity of models increase (as we can see in Use Case 1). This led to the idea of having the automatic transformation, which will reduce the efforts require for manual transformation and makes the overall V&V approach more efficient.

The objectives of the automatic transformation are as follows: (1) The main objective is to propose a methodology and tool-chain to automate the transformation of SysML specification models into formal models (Event-B). (2) The traceability should be maintained between informal requirements and the modeled system, specifically for the safety properties. (3) The model should be verified against such safety