Моделирование и анализ информационных систем. Т. 25, №6 (2018), с. 623–636
Modeling and Analysis of Information Systems. Vol. 25, No 6 (2018), pp. 623–636

DOI: 10.18255/1818-1015-2018-6-623-636

UDC 519.987

Translation from Event-B into Eiffel

Reznikova S., Rivera V., Lee J. Y., Mazzara M.

Received September 10, 2018
Revised October 10, 2018
Accepted November 1, 2018

Abstract. Formal modelling languages play a key role in the development of software: they enable users to specify functional requirements that serve as documentation as well; they enable users to prove the correctness of system properties, especially for critical systems. However, there is still an open question on how to map formal models to a specific programming language. In order to propose a solution, this paper presents a source-to-source mapping between Event-B models, a formal modelling language for reactive systems, and Eiffel programs, an Object Oriented (O-O) programming language. The mapping not only generates an actual Eiffel code of the Event-B model, but also translates model properties as contracts. The contracts follow the Design by Contract principle and are natively supported by the programming language. The mapping is implemented in the freely available Rodin plug-in EB2Eiffel. Thus, users can develop systems (i) starting with the modelling of functional requirements (properties) in Event-B, then (ii) formally proving the correctness of such properties in Rodin and finally (iii) by using EB2Eiffel to translate the model into Eiffel. In Eiffel, users can extend/customise the implementation of the model and formally prove it against the initial model. This paper also presents different Event-B models from the literature to test EB2Eiffel and its limitations. The article is published in the authors' wording.

Keywords: stepwise refinement, Design-by-Contract, formal modelling, reactive systems, Event-B, Eiffel

For citation: Reznikova S., Rivera V., Lee J. Y., Mazzara M., "Translation from Event-B into Eiffel", *Modeling and Analysis of Information Systems*, **25**:6 (2018), 623–636.

On the authors:
Sofia Reznikova, orcid.org/0000-0003-4616-2729, undergraduate student
Innopolis University,
1 Universitetskaya St., Innopolis 420500, Russia, e-mail: s.reznikova@innopolis.ru

Victor Rivera, orcid.org/0000-0002-1946-8979, Assistant Professor,
Innopolis University,
1 Universitetskaya St., Innopolis 420500, Russia, e-mail: v.rivera@innopolis.ru

JooYoung Lee, orcid.org/0000-0001-5421-730X, Assistant Professor,
Innopolis University,
1 Universitetskaya St., Innopolis 420500, Russia, e-mail: j.lee@innopolis.ru

Manuel Mazzara, orcid.org/0000-0002-3860-4948, Associate Professor,
Innopolis University,
1 Universitetskaya St., Innopolis 420500, Russia, e-mail: m.mazzara@innopolis.ru

Introduction

Modelling methodologies are used in software development to foresee how the resulting system will behave prior to building it. This stage might reveal hidden errors that can be handled at a smaller cost. This is of paramount importance since final users of systems are not aware of the consequences that malfunctioning systems might carry. There are different approaches to use these modelling methodologies (some are described in [6]), e.g. top-down and bottom-up approaches: using a top-down approach, one could think to start developing the system from a very abstract view point towards more concrete ones; in a bottom-up approach, on the other hand, one might think to start from a more concrete state of the system then add more functionality to it. The key point on both approaches is to always prove that properties of the systems hold.

Event-B is a formal modelling language for reactive systems, introduced by Abrial [1], which allows the modelling of complete systems. It follows the top-down approach by means of refinements. Event-B allows the creation of abstract systems and the expression of their properties. One can prove that the system indeed meets the properties then create a refinement of the system: same system with more details. It has been applied with success in both research and industrial projects, and in integrated EU projects aiming at putting together the two dimensions.

On the other side of the spectrum, following a bottom-up approach, one can work with the Eiffel programming language [8]. In Eiffel, one can create classes that implement any system. The behaviour of such classes is specified in Eiffel using contracts: pre- and post-conditions and class invariants. These mechanisms are natively supported by the language (as opposed to other programming languages). Having contracts, one can then verify that the implementation is indeed the intended (this can done statically by using a static verifier like Autoproof), and also one can track the specifications against the implementation [10]. This paper presents a series of rules to produce Eiffel programs from Event-B models, bridging both top-down and bottom-up approaches. An excerpt of the rules was presented elsewhere [15]. We also present a plug-in for Rodin, an IDE for Event-B, that implements the rules presented, the plug-in receives an Event-B model as input and produces the corresponding Eiffel classes equipped with contracts.

Several translations have been proposed and implemented that go in the same direction as the work presented on this paper. In [7], Mèry and Singh present the `EB2ALL` tool-set that includes a translation from Event-B models to C, C++ and Java. Unlike our translation, `EB2ALL` provides support for a small part of Event-B's syntax, and users are required to write a final Event-B implementation refinement in the syntax supported by the tool. The Code Generation tool [4] generates concurrent Java and Ada programs for a tasking extension of Event-B. Unlike these tools, the work presented here does not require user's intervention, while it works on the proper syntax of the Event-B model. In addition, these tools do not take full advantage of the elements present in the source language, e.g. invariants. The work presented in this paper, in addition to generating source code, it generates contracts from the source language, making use of the Design-by-Contract approach. In [14, 13, 2], authors present a translation from Event-B to Java, annotating the code with JML (Java Modelling Language) specifications, and [12] shows its application. The main difference with the work presented here is the target language. We are translating to Eiffel which natively supports Design-by-Contract. In addition, Eiffel comes with different tools to prove Eiffel code statically (e.g. Autoproof [16]) that fully supports the language. Another difference is the translation of carrier sets. EventB2Java translates them as set of integers, hence it does not capture the essence of carrier sets. We translate carrier sets as user defined datatype.

The paper is structured as follows. Section 1. presents the needed background. Section 2. defines the translation rules whilst Section 3. describes their implementation as a Rodin plug-in. Section 4. shows some evaluations of the plug-in. Finally, Section 5. is devoted for conclusions, outlining potential improvements and directions for future work.

1. Preliminaries

1.1. Event-B

Event-B is a formal modelling language for reactive systems, introduced by Abrial [1], which allows the modelling of complete systems. Figure 1 shows the general view of an Event-B machine and context. Event-B models are composed of contexts and machines. Contexts define constants (written after constant in context C), uninterpreted sets (written after set in context C) and their properties (written after axioms in context C). Machines define variables (written after variables in machine M) and their properties (expressed as invariants after invariant in machine M), and state transitions expressed as events (written between events and the last end). The initialisation event gives initial values to variables.

```
machine $M$ sees $C$
variables $v$
invariants $label\_inv: \; I(s,c,v)$
events
  event $initialisation$
    then $A(s,c,v)$ end
  event $evt$
    any $x$
  where
    $label\_guard: \; G(s,c,v,x)$
  then
    $label\_action: \; A(s,c,v,x)$
  end
end
```

```
Context $C$
constant $c$
set $S$
axioms $X(s,c)$
end
```

Figure 1. General view of an Event-B machine and its context

An event is composed of guards and actions. The guard (written between keywords where and then) represents conditions that must hold for the event to be triggered. The action (written between keywords then and end) gives new values to variables

In Event-B, systems are modelled via a sequence of refinements. First, an abstract machine is developed and verified to satisfy whatever correctness and safety properties are desired. Refinement machines are used to add more detail to the abstract machine until the model is sufficiently concrete for hand or automated translation to code. Refinement proof obligations are discharged to ensure that each refinement is a faithful model of the previous machine, so that all machines satisfy the correctness properties of the original.

1.2. Eiffel

Eiffel is an Object-Oriented programming language that natively supports the Design-by-Contract methodology. The behaviour of classes is specified by equipping them with contracts. Each routine of the class contains a pre- and post-condition: a client of a routine needs to guarantee the pre-condition on routine call. In return, the post-condition of the procedure, on routine exit, holds. The class is also equipped with class invariants. Invariants maintain the consistency of objects. Contracts in Eiffel follow a similar semantics of Hoare Triples.

Figure 2 depicts an Eiffel class that implements part of a Bank Account. The name of the class is `ACCOUNT` and it appears right after the keyword **class**. In Eiffel, implementers need to list creation procedures after the keyword **create**.

```
class ACCOUNT create make
feature -- Initalisation
  make -- Initialise an empty account.
    do
      balance := 0
    ensure
      balance_set: balance = 0
    end
feature -- Access
  balance: INTEGER -- Balance of this account.
feature -- Element change
  withdraw (amount: INTEGER) -- Withdraw 'amount' from this account.
    require
      amount_not_negative: amount >= 0
      amount_available: amount <= balance
    do
      balance := balance - amount
    ensure
      balance_set: balance = old balance - amount
    end
invariant balance_not_negative: balance >= 0
end
```

Figure 2. Eiffel class

In Figure 2, `make` is a procedure of the class that can be used as a creation procedure. Class `ACCOUNT` structures its procedures in `Initialisation`, `Access` and `Element change`, by using the keyword **feature**. This structure can be use for information hiding (not discussed here). `balance` is a class attribute that contains the actual balance of the account. It is defined as an integer. Procedures in Eiffel are defined by given them a name (e.g. `withdraw`) and its respective arguments. It is followed by a head comment (which is optional). Procedures are equipped with pre- and post-conditions predicates. In Eiffel, a predicate is composed of a tag (optional) and a boolean expression. For instance, the pre-condition for `withdraw` (after the key work **require**) imposes the restriction on callers to provide and argument that is greater than or equal zero and less than or equal the balance of the account (amount_not_negative and amount_available are tags, identifiers, and are optionals). If the pre-condition of the procedure is met, the post-condition (after the key work **ensure**) holds on procedure exit. In a post-condition, the aid **old** refers to the value of an expression on procedure entry. The actions of the procedure are listed in between the key words **do** and **ensure**. The only action of `withdraw` procedure is to increase the value of `balance` by `amount`. Finally, The invariant is restricting the possible values for variables.

2. Translation

The translation is done by the aid δ : `Event-B` $\rightarrow$ `Eiffel`. δ takes an Event-B model and produces Eiffel classes. It is defined as a total function (i.e. $\rightarrow$) since any Event-B model can be translated to Eiffel. It uses two helpers: ξ translates Event-B Expressions or Predicates to Eiffel, and τ translates the type of Event-B variable to the corresponding type in Eiffel.

$$\frac{\begin{array}{l}\tau(v) = \mathtt{Type} \quad \xi(I(s,c,v)) = \mathtt{Inv} \quad \delta(\mathsf{events}\ e) = \mathtt{E} \\ \delta(\mathsf{event}\ \mathit{initialisation}\ \mathsf{then}\ A(s,c,v)\ \mathsf{end}) = \mathtt{Init}\end{array}}{\begin{array}{l}\delta(\mathsf{machine}\ M\ \mathsf{sees}\ C \\ \qquad \mathsf{variables}\ v \\ \qquad \mathsf{invariants}\ \mathit{label\_inv} :\ I(s,c,v) \\ \qquad \mathsf{event}\ \mathit{initialisation}\ \mathsf{then}\ A(s,c,v)\ \mathsf{end} \\ \qquad \mathsf{events}\ e \\ \quad \mathsf{end}) = \\ \mathbf{class}\ \mathtt{M}\ \mathbf{create}\ \mathtt{initialisation} \\ \mathbf{feature}\ \ {-}{-}\ \mathsf{Initialisation} \\ \quad \mathtt{Init} \\ \mathbf{feature}\ \ {-}{-}\ \mathsf{Events} \\ \quad \mathtt{E} \\ \mathbf{feature}\ \ {-}{-}\ \mathsf{Access} \\ \quad \mathtt{ctx\ :\ CONSTANTS} \\ \quad \mathtt{v\ :\ Type} \\ \mathbf{invariant} \\ \quad \mathtt{label\_inv:\ Inv} \\ \mathbf{end}\end{array}} \ (\mathtt{machine})$$

Figure 3. `machine` rule

Rule `machine` in Figure 3 is a high level translation. It takes an Event-B machine M and produces an Eiffel class `M`.

Variables are translated as class attributes in class `M`. Event-B invariants are translated to Eiffel invariants. Both, Event-B and Eiffel, have similar semantics for invariants. Rule `context` generate an Eiffel class `CONSTANT` that contains the translation of Event-B constants and carrier sets defined by the user. Axioms, which restrict the possible values for constants are translated to invariants of this class. Constants in Event-B are entities that cannot change their values. They are naturally translated to Eiffel as **once** variables.

$$\frac{\begin{array}{l}\delta(\mathsf{axioms}\ X(s,c)) = \texttt{X}\\ \tau(c) = \texttt{Type}\end{array}}{\begin{array}{l}\delta(\mathsf{Context}\ \ C\\ \qquad \mathsf{constant}\ c\\ \qquad \mathsf{set}\ S\\ \qquad \mathsf{Axioms}\ X(s,c)\\ \quad \mathsf{end}) =\\ \textbf{class}\ \texttt{CONSTANTS}\\ \textbf{feature}\ \text{--}\ \mathsf{Constants}\\ \quad \mathsf{c}:\ \texttt{Type}\\ \qquad\quad \text{--}\ \text{`c' comment}\\ \qquad \textbf{once}\\ \qquad\quad \textbf{create}\ \texttt{Type}\ \textbf{Result}\\ \qquad \textbf{end}\\ \textbf{invariant}\\ \quad \texttt{X}\\ \textbf{end}\end{array}}\ (\texttt{context})$$

Carrier sets represent a new type defined by the user. Each carrier set is translated as an afresh Eiffel class so users are able to use them as types. Rule `cset` shows the translation. Parts of the class are omitted due to space. Class `EBSET [T]` gives an implementation to sets of type `T`. Class `S` inherits `EBSET [T]` due to the nature of carrier sets in Event-B.

$$\frac{\tau(s) = \texttt{Type}}{\begin{array}{l}\delta(\mathsf{Context}\ \ C\\ \qquad \mathsf{constant}\ c\\ \qquad \mathsf{set}\ S\\ \qquad \mathsf{Axioms}\ X(s,c)\\ \quad \mathsf{end}) =\\ \textbf{class}\ S\\ \textbf{inherit}\\ \quad \texttt{EBSET}\ [\texttt{Type}]\\ \dots\\ \textbf{end}\end{array}}\ (\mathsf{cset})$$

Rule `event` produces an Eiffel feature given an Event-B *event*. Parameters of the event are translated as arguments of the respective feature in Eiffel with its respective type. In Event-B, an event might be executed only if the guard is true. In Eiffel, the guard is translated as the precondition of the feature. Hence, the client is now in charge of meeting the specification before calling the feature. The semantics of the execution is handle now by the client who wants to execute the feature rather than the system deciding. The actual execution of the actions still preserve its semantics: execution of the actions is only possible if the guard is true. In Eiffel, for a client to execute a feature he needs to meet the guard otherwise a runtime exception will be raised: Contract violation.

Event-B event actions are translated directly to Eiffel statements. In Event-B, the before-after predicate contains primed and unprimed variables representing the before and after value of the variables. We translated the primed variable with the Eiffel key word **old**. Representing old value of the variable. For simplicity. the rule only takes into account a single parameter, a single guard and a single action. However, this can be easily extended.

```
$\xi(G(s,c,v,x)) = \texttt{G}$   $\xi(A(s,c,v,x)) = \texttt{A}$
$\tau(x) = \texttt{Type}$
------------------------------------------------------------ (event)
$\delta($event $evt$ any x
      where $label\_guard:\ G(s,c,v,x)$
      then $label\_action:\ A(s,c,v,x)$
      end$) =$
   evt(x : Type)
         -- 'evt' comment
      require
         label_guard: G
      do
         v.assigns(A)
      ensure
         label_action: v.equals(old A)
      end
```

Rule `init` below shows the translation of Event-B event *initialisation* to a creation procedure in Eiffel. The creation procedure initialises the object containing the constants definition. It also assigns initial values to variables taken from the initialisation in the *initialisation* event. In Eiffel, creation procedures are listed under the keyword **create**, as shown in rule `machine`. The **ensure** clause shows the translation of the before-after predicate of the assignment in Event-B.

$$\frac{\xi(A(s,c,v)) = \mathtt{A}}{\delta(\textsf{event}\ \mathit{initialisation}\ \textsf{then}\ \mathit{label}:\ A(s,c,v)\ \textsf{end}) =}\ (\mathtt{init})$$

```
δ(event initialisation
    then
      label :  A(s,c,v)
  end) =

  initialisation
        -- evt comment
    do
      create ctx
      v.assigns(A)
    ensure
      label: v.is_equal(old A)
    end
```

3. Implementation

Rules presented in previous sections are implemented as a Rodin plug-in. Machines are translated to Eiffel classes. Event-B Events and variables are features and variables of the translated class, respectively. Guards and actions are translated as preconditions and body of the features. Contexts are being translated as class constants. The translation takes advantage of the embedded Design-by-Contract mechanism defined in Eiffel, for instance, Machine invariants are naturally translated to class invariants.

Another part of the implementation required translation of the mathematical language of Event-B. There are in total 90 symbols denoting different mathematical formulas. Some of them exist natively in Eiffel while others had to be implemented.

The implementation was done in several steps: first the structure for the Rodin IDE plug-in was set-up, then the information about the model was retrieved from the database and, finally, the resulting elements were translated into Eiffel.

3.1. Event-B and Rodin Structure

The plug-in's functionality is realized based on extensions and extension points that define the point of contact between different programs. The package responsible for Rodin extensions is `org.rodinp.core`. It provides an interface via which different extensions can communicate and provide extension points.

Event-B package – `org.eventb.core.ast` – provides an Abstract Syntax Tree (AST) of the system modeled in Rodin. This is a tree representation of the syntactic structure of an Event-B model. A visitor has been implemented to traverse it. It translates the mathematical notation into Eiffel code step by step.

3.2. The General Structure of the Tool

The packages included in the tool are `plug-in` and `rodinDB` (implementation of the tool can be found in [11]). The `plug-in` package contains `GenCodeEiffel.java` that is the entry point (defines the order of the translation), and `Translator.java` that implements a Visitor (`ISimpleVisitor2`) that parses the formulas and traverses the AST. The `rodinDB` package contains `RodinDBElements.java` that deals with retrieving information from the Rodin Database.

Each Rodin project has a set of children of class `IRodinElement` which also have `IInternalElements` in them. Depending on what type the internal elements is, it is possible to retrieve the information regarding machines and contexts from Rodin database. The package includes 14 methods that handle this task.

3.3. Traversing the Abstract Syntax Tree

Rodin provides the Abstract Syntax Tree (AST) of an Event-B model. It is then necessary to implement a Visitor to traverse the AST to translate the model parts into Eiffel.

The next step after retrieving information from the Rodin database is to parse the received formulas and to organize them in a way suitable for Eiffel translation. The wrapper methods dealing with parsing are created for almost each method from the `rodinDB` package. The visitor implements *parsePredicate()*, that parses an Event-B predicate, *parseExpression()* that parses an Event-B expression and *parseAssignment()* that parses assignments.

As parameters they take a String-representation of a formula and launch the pass through the AST. The methods in `ISimpleVisitor2` are overridden to handle visits of different branches of the tree such as Atomic Expressions (covers standalone integers, natural numbers, empty sets, booleans and others), Binary Predicates (implication and equality), Becomes Equal To (assignment to a variable or a parameter), Associative Expressions (unions, intersections, backward and forward compositions, addition, multiplication) and many others. The full set of types is described in [1].

3.4. Translating into Eiffel

As each AST branch for a specific formula is visited, Eiffel code is generated and added to the `eiffelCode` Array List that aggregates the translation and then returns it to the code generator (`GenCodeEiffel.java`).

An excerpt of the visitor is shown in Figure 4. A free identifier is any variable from a machine or an event. First the method checks whether the translation is done to retrieve types or to translate event or machine parameters. Depending on the result, code is added to different places.

```
Override
public void visitFreeIdentifier (FreeIdentifier identifierExpression){
 if (eiffelType != null){
   eiffelType.add (identifierExpression.getName());
 }
   eiffelCode.add (identifierExpression.getName());
 }
}
```

Figure 4. Free Identifier Translation

4. Evaluation

Several Event-B models were used in the testing phase of the plug-in. This phase also captures how much and accurate (as without compilation errors in Eiffel) of the Event-B mathematical language gets translated. Table 1 shows the results for all the tested models: first column is the Event-B model taken from the literature; the second column indicates how much code is translated in terms of the language syntax. For instance, 94% of the `MIO` model is being translated, meaning the model is using Event-B syntax (6%) that cannot be translated due to the limitations of the tool (discussed later); and the third column is how much code results in compilation errors in Eiffel.

Table 1. Summary of the results

Model	Translated	Compiled
Social Event Planner	100%	98%
MIO	94%	88%
Binary Search	100%	92%
Linear Search	100%	100%
Reversing the Array	78%	78%
Sorting the Array	100%	96%
Finding a minimum	94%	94%
Square root	100%	100%

Social Event Planner: This model was used for testing during the implementation stage. This model is described in more detail in [14]. It is a model for planing social events. The functionality includes creating events, inviting people to them and setting up permissions for inviting other people. This model defines its own sets ($PERSON$ and $CONTENTS$). They are translated as Eiffel user-defined classes.

An example Event-B event `create_account` is shown on Figure 5. ANY declares parameters of the event, WHERE denotes guards (necessaries conditions to hold for the event to be triggered) of the event, THEN are the actions of the event. Figure 6 depicts the output of the plug-in. The output is a translation of Event-B event (in Figure 5) into Eiffel. There are two **require** statements corresponding to two guards (from Event-B) that ensure that the variables belong to the sets they are supposed to. The **do** statements assign translated expressions to the variables `contents, persons, owner and pages`, initializing them with an appropriate type. In Eiffel, **create** s a keyword used to create instances a classes, similar to **new** in Java or C++.

Class `EBSET` is a class created specifically for the translation of sets from Event-B. It inherits most of Eiffel set's functionality but allows more flexibility. It is part of `eb_math_lang` package that also includes natural numbers (`EBNAT`), integers, ranges and relations.

```
ANY
    c1
    p1
WHERE
    grd1 : $p1 \in PERSON \setminus persons$
    grd2 : $c1 \in CONTENTS \setminus contents$
THEN
    act1 : $contents := contents \cup \{c1\}$
    act2 : $persons := persons \cup \{p1\}$
    act3 : $owner := owner \cup \{c1 \mapsto p1\}$
    act4 : $pages := pages \cup \{c1 \mapsto p1\}$
END
```

Figure 5. *create_ account* event

```
create_account (p1: PERSON; c1: CONTENTS)
   require
      grd1: PERSON.difference (persons).has (p1)
      grd2: CONTENTS.difference (contents).has (c1)
   do
     contents.assigns ((contents).union (create {EBSET[CONTENTS]}.singleton
(c1)))
      persons.assigns ((persons).union (create { EBSET[PERSON]}.singleton
(p1)))
       owner.assigns ((owner).union (create  {EBREL[CONTENTS,PERSON]}.vals
(<<(create { EBPAIR[CONTENTS,PERSON]}. make (c1, p1))>>))
       pages.assigns ((pages).union (create  {EBREL[CONTENTS,PERSON]}.vals
(<<(create {EBPAIR[CONTENTS,PERSON]}. make(c1, p1))>>))
   end
```

Figure 6. Eiffel Code for *create_ account* event

MIO – Bus Transportation System: This model includes several entities: buses, bus stations, people, doors and sensors. It regulates bus transportation and is described in [3]. There are six refinements with each adding new entities and concepts. It is an extensive model with 21 actions in the initialisation event and 15 invariants for the last refinement.

Binary and Linear Search: These two models represented *binary* and *linear* search algorithms. The former one looks for a number, dividing each subsection into two, and the latter goes through a set of numbers in a linear way.

The binary search included three events apart from the initialisation (`inc`[rement], `dec`[rement] and `found`). The model for linear search included a similar `found` event as well as `progress` event that continued to search for the number if the correct one is not found.

In general, the tool is able to translate 86% of the 90 symbols that can be used with Event-B. Although the remaining 14% is not implemented yet, it is noted that they occur rarely in the models.

A common problem for most models (e.g. Social Event Planner, MIO, Binary Search and Sorting the Array) that results in compilation errors is due to the type translation of variables. For instance, the Event-B assignment $owner := owner \cup \{cmt \mapsto owner(rc)\}$ with a set extension ($\{\}$) including more than two expressions poses a problem as the types of the sets should be included into the Eiffel code before visiting the variable in the tree. Therefore, instead of a type, Java's `null` keyword is returned (as the tool is written in Java and for Eiffel this is an undefined keyword). This problem occurs for set extensions (declaring a set between two curly brackets), backward and forward composition of functions.

If the correct types are included into the Eiffel code manually, the compilation returns no errors. As Table 1 shows, these types of errors are not very common, e.g. for Social Event Planner there are only three cases of such statements, for MIO, Binary Search and Sorting the Array there is only one.

Other models (namely, reversing an array and finding a minimum) require those parts of the mathematical language that have not been implemented (Bound Declarations and Bound Identifiers that are used as variables in Quantified Predicates – $\forall$ and $\exists$). This is where most of their errors occur.

The list of Event-B models used in this phase and their translation to Eiffel using the plug-in can be found in [11].

5. Conclusion

We presented a series of rules to transform an Event-B model to an Eiffel program. The translation takes full advantage of all elements in the source by translating them as contracts in the target language. Thus, no information on the behaviour of the system is lost. These rules shows a methodology for software construction that makes use of two different approaches. We also presented a Rodin plug-in that implements the translation. The plug-in enables users to take advantages of Event-B (e.g. refinement) to then translation to Eiffel, which provides an actual implementation of the model, to take advantages of the language (e.g. Design by Contract). The main limitation of the plug-in is that no proof of soundness has been carried out. This paper shows a proof-of-concept and opens up a direction to carry out with the proof.

In order to be able to fully automate the translation, type retrieval for variables that are further down the AST need to be implemented. This corresponds to 14% of the Event-B mathematical language. One of the challenges is to translate choice from set ($x :\in S$) that arbitrarily chooses a value from the set S and choice by predicate ($z :| P$) that arbitrarily chooses values for the variable in z that satisfy the predicate P. For this, we plan to make use of ProB or Constraint Programming to assign values that satisfy a predicate. Another direction is related to the translation of Proof Obligations (POs), mathematical formulas, automatically generated by Rodin, to Eiffel. POs need to be proven in order to ensure that a machine is correct [5]. They can be proved either automatically or interactively. By translating Proof Obligations into Specification Drivers [9] it will be possible to formally verify the translated Eiffel code against its contracts.

Formal modelling belongs to the technical domain to specify functional requirements. In order to prove correctness of system properties users need to be properly trained and understand technical aspects of formal models or, at least, of programming languages. In order to bridge the gap between business perspective and technical perspective, it would be necessary to provide a more comprehensive modelling framework that is left for future investigation [17].

References

[1] Abrial J.-R., *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, New York, 2010.

[2] Cataño N., Rivera V., “EventB2Java: A Code Generator for Event-B”, *NASA Formal Methods. NFM 2016*, Lecture Notes in Computer Science, **9690**, Springer, Cham, 2016, 166–171.

[3] Cataño N., Rueda C., “Teaching formal methods for the unconquered territory”, *Teaching Formal Methods. TFM 2009*, Lecture Notes in Computer Science, **5846**, Springer, Berlin, Heidelberg, 2009, 2–19.

[4] Edmunds A., Butler M., “Tool support for Event-B code generation”, *Workshop on Tool Building in Formal Methods*, Wiley and Sons, Quebec, Canada, 2010.

[5] Hallerstede S., “On the purpose of Event-B proof obligations”, *Abstract State Machines, B and Z*, Lecture Notes in Computer Science, **5238**, Springer, Berlin, Heidelberg, 2008, 125–138.

[6] Mazzara M., “Deriving specifications of dependable systems: toward a method”, *12th European Workshop on Dependable Computing (EWDC)*, 2009.

[7] Méry D., Singh N. K., “Automatic code generation from Event-B models”, *Proceedings of the Second Symposium on Information and Communication Technology, SoICT ’11*, ACM, New York, 2011, 179–188.

[8] Meyer B., “Applying "design by contract"”, *Computer*, **25**:10 (1992), 40–51.

[9] Naumchev A., Meyer B., “Complete contracts through specification drivers”, *CoRR*, 2016, abs/1602.04007.

[10] Naumchev A., Meyer B., Rivera V., “Unifying requirements and code: An example”, *Perspectives of System Informatics – 10th International Andrei Ershov Informatics Conference, PSI 2015*, Revised Selected Papers, Kazan and Innopolis, Russia, 2015, 233–244.

[11] Reznikova S., *Innopolis thesis*, 2018, https://github.com/sonyareznikova/InnopolisThesis.

[12] Rivera V., Bhattacharya S., Cataño N., "Undertaking the tokeneer challenge in Event-B", *2016 IEEE/ACM 4th FME Workshop on Formal Methods in Software Engineering*, 2016, 8–14.

[13] Rivera V., Cataño N., "Translating Event-B to JML-specified Java programs", *Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC'14*, ACM, New York, 2014, 1264–1271.

[14] Rivera V., Cataño N., Wahls T., Rueda C., "Code generation for Event-B", *Int. J. Softw. Tools Technol. Transf.*, **19**:1 (2017), 31–52.

[15] Rivera V., Lee J. Y., Mazzara M., "Mapping Event-B machines into Eiffel programming language", *Proceedings of 6th International Conference in Software Engineering for Defence Applications – SEDA 2018*, Rome, Italy, 2018.

[16] Tschannen J. et al., "AutoProof: Autoactive functional verification of object-oriented programs", *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2015*, Lecture Notes in Computer Science, **9035**, Springer, Berlin, Heidelberg, 2015, 566–580.

[17] Yan Z. et al., "BPMO: semantic business process modeling and WSMO extension", *IEEE International Conference on Web Services (ICWS 2007)*, 2007, 1185–1186.

Резникова С., Ривера В., Ли Д. Й., Маццара М., "Перевод моделей Event-B в Eiffel", *Моделирование и анализ информационных систем*, **25**:6 (2018), 623–636.

DOI: 10.18255/1818-1015-2018-6-623-636

Аннотация. Формальные языки моделирования играют важную роль в разработке программного обеспечения, так как позволяют пользователям, во-первых, определять функциональные требования, которые также служат документацией для проекта, а во-вторых, доказывать корректность свойств систем, что особенно важно для критических систем. Однако не существует четкого понимания того, как сопоставить формальную модель и определенный язык программирования. В качестве решения данной проблемы авторы статьи предлагают использовать возможность source-to-source соответствия между моделями, описанными на языке Event-B (языке моделирования для реактивных приложений и систем), и программами на объектно-ориентированном языке программирования Eiffel. Предложенное решение не только автоматически генерирует соответствующий модели на Event-B код на Eiffel, но также переводит свойства модели в виде контрактов. Контракты соответствуют принципу Design-by-Contract и нативно поддерживаются в Eiffel. Реализация решения доступна как плагин EB2Eiffel в Rodin (среде разработки для Event-B). Таким образом, пользователи могут разрабатывать различные системы, начиная с моделирования функциональных требований (свойств) в Event-B, затем формально доказывая корректность этих свойств в Rodin и, наконец, используя EB2Eiffel для перевода модели на язык программирования. Используя Eiffel, пользователи могут расширять и модифицировать реализацию модели и доказывать корректность измененной модели относительно ее оригинальной, изначально переведенной версии. Также в статье описан процесс тестирования EB2Eiffel разными моделями, написанными на Event-B, и представлены ограничения плагина. Статья публикуется в авторской редакции.

Ключевые слова: пошаговое улучшение систем, Design-by-Contract, формальное моделирование, реактивные приложения, Event-B, Eiffel

Об авторах:
Резникова Софья, orcid.org/0000-0003-4616-2729, студент-бакалавр, Университет Иннополис,
ул. Университетская, 1, Иннополис 420500, Россия, email: s.reznikova@innopolis.ru

Ривера Виктор, orcid.org/0000-0002-1946-8979, доцент, Университет Иннополис,
ул. Университетская, 1, Иннополис 420500, Россия, email: v.rivera@innopolis.ru

Ли Джу Йонг, orcid.org/0000-0001-5421-730X, доцент, Университет Иннополис,
ул. Университетская, 1, Иннополис 420500, Россия, email: j.lee@innopolis.ru

Маццара Мануэль, orcid.org/0000-0002-3860-4948, профессор, Университет Иннополис,
ул. Университетская, 1, Иннополис 420500, Россия, email: m.mazzara@innopolis.ru