Scientia Iranica D (2018) 25(6), 3415–3441

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
http://scientiairanica.sharif.edu

From Object-Z specification to Groovy implementation

F. Zaker, H. Haghighi*, and E. Nazemi

Faculty of Computer Science and Engineering, Shahid Beheshti University G.C., Tehran, 1983969411, Iran.

Received 11 July 2016; received in revised form 21 October 2017; accepted 4 August 2018

KEYWORDS
Formal program development;
Object oriented programming;
Animation;
Object-Z;
Groovy;
JVM.

Abstract. So far, valuable research studies have been conducted on mapping notations of object-oriented specification, such as Object-Z, in different object-oriented programming languages, such as C++. However, the results of selecting JVM-based programming languages for mapping have not covered most of basic Object-Z structures. In this paper, the Groovy language, as a dynamic JVM-based language, is selected to overcome some of the existing limitations. As the main contribution, the rules required for mapping Object-Z specifications to execute Groovy code are introduced. The proposed rules cover notions such as multiple inheritance, inverse specification of functions, functions defined on generic definitions, and free-type constructors. Previous methods have not covered these notions for the formal development of program from object-oriented specifications, regardless of the selected formal specification language and target programming language. In addition, in this paper, the parallel composition construct is mapped to a parallel, executable code to improve the faithfulness of the final code to the initial specification. A mapping rule for the class union construct is introduced, which has not yet been provided for any JVM-based language. Unlike previous works, instead of presenting the mapping rules in terms of natural languages, they are presented in terms of some formal mapping rules.

1. Introduction

Woodcock et al. [1] categorized 62 industrial projects into transport, finance, defense, telecom, nuclear, healthcare, and some other fields from November 2007 to December 2008 to which formal methods were applied. Fitzgerald et al. [2] reported on the application statuses of formal methods to 98 industrial projects in 2012 in almost all aforesaid categories. The increasing number of industrial projects to which formal methods have been applied shows the necessity of conducting researches on formal program development and similar research areas.

Due to the popularity of object-oriented programming approaches, there is a growing interest in utilizing object-oriented concepts, such as encapsulation and reuse, when applying formal methods [3]. Object-Z [4,5] is an extension of the Z notation [6], which allows for specification of large and sophisticated programs as sets of independent classes [7]. Some previous approaches have been proposed to develop object-oriented programs from Object-Z. An inclusive survey of the development of object-oriented programs from formal specifications is provided in [3].

We have found more than 63 published papers on the development of object-oriented programs from formal specifications, implying the importance of advances in this research field. For example, there are 18 papers on animating formal specifications; 11 papers used Object-Z [8-18]; seven used VDM and VDM++ [19-25] as the source specification languages. In addition, there are at least 32 works focusing on refinement of formal specifications from which 10

*. *Corresponding author.*
E-mail address: h_haghighi@sbu.ac.ir (H. Haghighi)

doi: 10.24200/sci.2018.20798

approaches refined Object-Z specifications [26-35], 5 methods applied VDM++ [36-40], and 17 papers were published for B, Event-B, and UML-B [41-57]. Among related works, 13 papers attempted to cover the whole process of refinement from specification to code: six using Object-Z [58-63] and seven using Event-B [64-70] as the source specification languages.

Among programming languages, JVM-based languages are prevalently used for the development of large-scale sophisticated software systems. According to Oracle's reports in 2014, over 9.3 million programmers worldwide used Java or other JVM-based languages at the time [71]. Groovy is a dynamic JVM-based programming language designed for working with internal DSLs by means of adding powerful meta-programming capabilities [72]. Groovy allows overriding of certain language constructs to facilitate numerical computation, because it uses a meta-class to control the behavior of each Groovy class. In Addition, Groovy can be compiled into Java bytecodes. Thus, integration with existing Java libraries is straightforward, and key Java optimization techniques can be applied to the compiled classes. Groovy supports integration with Java codes, allowing for joint compilation with existing Java sources. This is an important factor, which is not offered by other languages targeting JVM [73].

Previous surveys [3,74] concentrating on the development of object-oriented programs from formal specifications have revealed some weaknesses in this area. Some of these weaknesses are addressed in this paper. Considering Java as the destination language, the mapping in [7], with Object-Z as the source language, only focuses on the class structure. In addition, this mapping provides very simple rules and ignores implied preconditions. The mapping proposed in [75] only relies on simple and basic structures of Object-Z concerning the class structure, state schema, and operation schema. In [76], a relatively complete mapping from VDM to Java was proposed. However, the proposed mapping does not obviously cover object-oriented structures, since VDM-SL is not object oriented itself. This drawback was overcome partially in [77] by completing the mapping proposed in [76], adding object-oriented structures and using VDM++ as the source language. However, issues such as multiple inheritance with a large number of imposed constraints remain unsolved because Java does not permit multiple inheritance. In addition, the mapping proposed in [20] replaces the notion of parallel composition with concurrency. In [17-20], Event-B is used as the source language to develop Java code. Some tools were also developed in these studies. The OCB (Object-oriented Concurrent-B), which is very similar to Java as the destination language, was introduced in [17-19] to link the Event-B modeling language to the implementation code in the development process. In spite of using OCB as the intermediate language, these studies only focused on concurrency-related notions, such as processes and monitors [3].

Regarding C++ as the target language, in [78], a structured, yet imperfect, mapping was introduced from Object-Z to C++. The proposed mapping does not cover some specification constructs of Object-Z such as pre-condition, post-condition, class invariants, visibility list, operation operators, object containment, and some types of definitions, like class variables and generic parameters. In [79], the work of [78] has been supported by presenting two new rules that consider a constructor for constant types and a template class for generic parameters. These mapping rules were later completed in [9] by covering more Object-Z structures, such as class union, object aggregation, object containment, and some of the operation operators. These mapping rules have been formally presented and proved in [80]. However, some specification constructs, such as multiple inheritance, inverse specification of functions, functions on generic definitions, and constructors of free types remain unmapped. In addition, some specification constructs, such as parallel composition, are not mapped correctly.

Although many of the weaknesses in the development of programs from object-oriented specifications have been addressed [9,80], there are still unsolved problems. For example, multiple inheritance, generic definitions, and parallel composition notions are not covered by these studies. These problems occur due to the shortages in the destination languages in comparison to features of the used specifications languages. Moreover, in previous works, the readability of generated code is low, resulting in the increase of system maintenance costs. In the approach proposed in this paper, to improve the readability of the generated code, the AOP (Aspect-Oriented Programming) [81] model is used because it helps to separate pre-conditions and post-conditions from the body of schemas.

Since some of the existing weaknesses occur due to the shortcomings in destination languages, it is important to select a proper mapping destination language that enables us to overcome these weaknesses. Moreover, the destination language should be a language that is highly popular and acceptable among software developers. In this paper, Groovy is selected as a Java-based dynamic language to solve some of the aforementioned problems. Major contributions of this work are as follows. First, some new notions and constructs are included in the proposed mapping, which have not been covered by previous methods. These notions and constructs are listed below:

1. Multiple inheritance;
2. Inverse specification of functions (specification of inputs in terms of outputs);

3. Functions on generic definitions;
4. Free-type constructors (not constant values).

In addition, unlike previous works, parallel compositions are mapped on a parallel construct (not a concurrent construct). This approach improves the efficiency of the final code and its faithfulness to the initial specification. Moreover, the control of constraints is assigned to advisors [82] in the mapping rules. This helps avoid mapping specifications to complex and illegible codes.

As for another scientific contribution, this paper considers some notions other than the aforementioned ones, such as class union and free type (both constant values and constructors), which have not been addressed by previous JVM-based works.

The remainder of this paper is organized as follows. In Section 2, to increase the reader's familiarity with the Object-Z specification language, a simple specification of the credit card management system is investigated. In Section 3, some of the features of Groovy are explained. In addition, the main reasons are described to select this language as the destination language from the collection of JVM-based programming languages. Section 4 presents our mapping rules from Object-Z to Groovy and their applications through examples. In Section 5, the ability of the proposed mapping rules for producing object-oriented programs from Object-Z specifications is challenged through a case study. The last section is devoted to the conclusions and some directions for future work.

2. Object-Z as the source language

This section provides an illustrative example for relative understanding of Object-Z. The specification instance used in this section is similar to the case study presented in [9]. This specification, which is related to the credit card management system, will also be used as the case study of this paper.

2.1. Credit management system's specification

In specification of the credit card management system, we focus on the basic features of credit cards and their interactions. Withdrawal limit, card status, and ownership should be specified for each credit card. Hence, types and abbreviations are defined as shown in Listing 1.

We start the specification operation with the *CreditCard* class that introduces the possible features

$$[CUSTOMER]$$
$$limitValue == \{1000, 2000, 5000\}$$
$$Status ::= invalid \mid valid$$

Listing 1. Types and abbreviations.

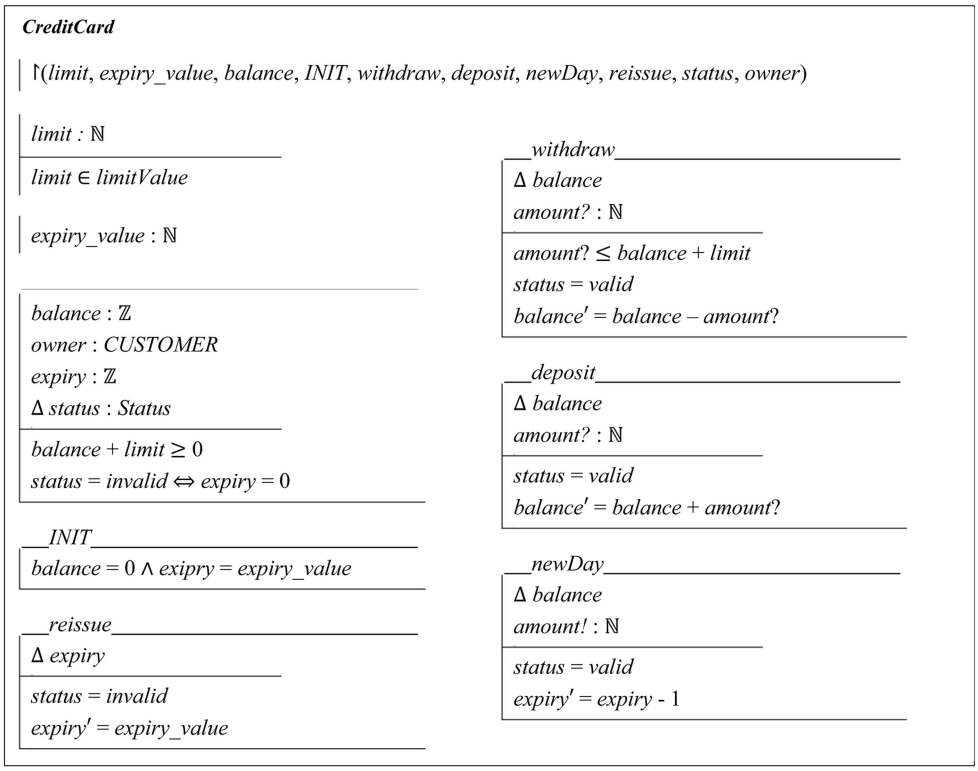
and operations of a credit card (Listing 2). A withdrawal *limit* is determined for each card, which is defined by the *limit* constant. The value of the limit constant is assigned by one of the possible values in *limitValue*. The number of days the credit card is valid is specified by a constant named *expiry_value*. The balance, owner, number of remaining days to the expiration date, and card status are specified by the *balance*, *owner*, *expiry*, and *status* variables in the state schema, respectively. Among these variables, the *status* variable is determined based on the number of remaining days to expiration; when this number is zero, the *status* variable becomes invalid.

To define a card, the *zero* value is assigned to the *balance* variable. This variable is expected to take a negative value. With each withdrawal from the credit card, the amount should be subtracted from the balance of the credit card. The minimum value of the *balance* variable is limited by the negative value of the *limit* constant. The *expiry* variable is assigned value with the *expiry_value* constant, and its value decreases by one at the end of each day via the *newDay* operation schema. Validity of the credit card is extended with the *reissue* operation schema, which assigns the *expiry_value* to the expiry variable. Withdrawal takes place with the *withdraw* schema, and depositing is specified by the *deposit* schema. Both of these operations are executable if the credit card status is set as *valid*. These schemas change *balance* in proportion to the requested operation, and the value is determined by the *amount* input parameter.

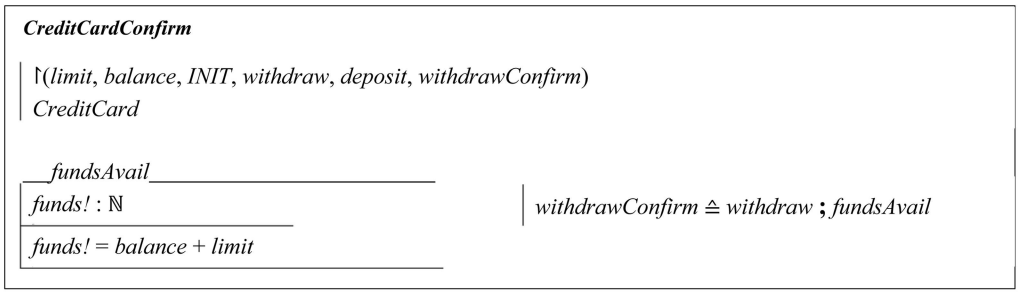
After specification of the *CreditCard* class, another class named *CreditCardConfirm* is specified as shown in Listing 3. The *CreditCardConfirm* class is derived from the *CreditCard* class. In this class, a new schema, called *withdrawConfirm*, is specified through sequential composition of the *withdraw* and *fundsAvail* schemas (which returns the credit card balance). The sequential composition is a binary operator used to model two operations occurring in a sequence [4]. Using this operator, the *withdrawConfirm* schema executes the withdrawal operation and, then, returns the remaining card balance.

Another class, named *CreditCardCount*, is defined which is derived from the *CreditCard* class (Listing 4). The *CreditCardCount* class changes the *withdraw* operation schema to present a new implementation of this operation schema from the parent schema using a new operational schema, called *incrementCount*. The objective of the *incrementCount* schema is to count the number of withdrawal transactions.

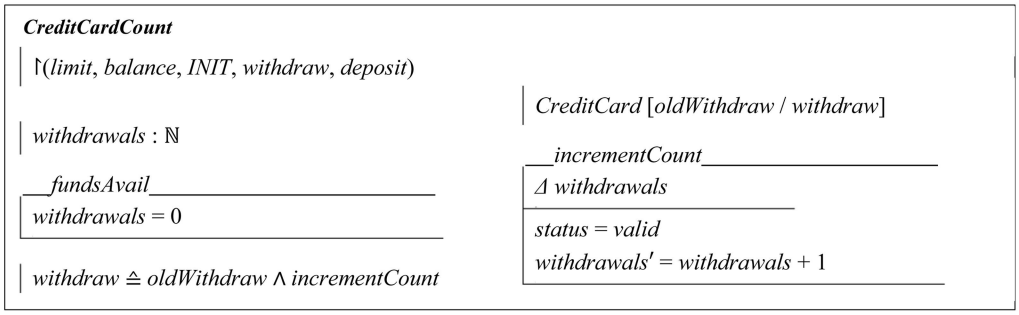
In Listing 5, a new class is specified which is derived from both the *CreditCardConfirm* and *CreditCardCount* classes. This class is suggestive of the well-known diamond problem in multiple inheritance. The diamond problem is an ambiguity that arises when two



Listing 2. *CreditCard* specification.



Listing 3. *CreditCardConfirm* specification.



Listing 4. *CreditCardCount* specification.

Listing 5. *CreditCardConfirmAndCount* specification.

classes B and C are derived from A, and class D is derived from both B and C. If there is a method in A that B and C override it and D does not override it, then it remains unknown which version of method D is derived: that of B, or that of C?

Finally, the *CreditCards* schema is defined that includes operations required for definition and discarding credit cards using the *CreditCard* schema. Due to space restrictions, the specification of the *CreditCards* schema is provided in Appendix A.

3. Groovy as the destination language

Groovy, as a new language, is based on the Java platform and adds many features that lead to the fame of Ruby [83]. Although this language is based on the main Java structure and is not different from Java on the byte code level, Groovy adds unique features to Java on the level of interaction with developers [84]. In addition, the development platform of grails uses Groovy for the service, controller, and domain class code [82]. The grails framework drastically increases the speed of development of organizational web-based software systems and even websites with a heavy workload. Nowadays, the grails platform is gaining more popularity among software developers. As for another advantage, using Groovy on the grails platform, the convention over configuration pattern (for detailed information, refer to Subsection 3.4) allows for generation of executable codes with suitable web-based user interfaces. This leads to considerable similarity of Groovy to the world of industry.

The distinctive features of Groovy, which convince us to select this language as the destination language for our mappings, will be described in the following subsections. These features facilitate the process of mapping and code generation.

3.1. Dynamism of the language

As a dynamic programming language, Groovy is capable of making many executive decisions at runtime, which are related to the compile time in typical programing languages. These decisions may lead to actions such as adding new code, extending objects and definitions, or modifying the type system [85] of Groovy. In general, it provides a full control over the fields, behaviors, and structures of objects involved in the program at runtime. As will be described in Subsection 4.2.5, the role of dynamism of the language is completely evident in mapping rules related to "renaming of operation schemas" in class inheritance.

3.2. Special structures of Groovy

One of the popular features of Groovy, as a dynamic language, is a structure called closure. Closure is a small code segment that could be stored and executed as a string. This structure is derived from the lambda expressions in functional programming languages. It receives a number of parameters as the input and executes commands in accordance with the inputs [85].

Lambda expressions are supported by all the structures and dynamic functions of Groovy [85]. This is one of the features added by Groovy to Java. This feature enables Java to implement facilities, such as universal and existential quantifications, in logic-based formal notations (e.g., Object-Z). This feature is not normally available in non-functional programming languages.

In addition, one of the most important features of Groovy is a set of classes and functions known as Map type. Typically, a Map is a collection of pairs (key, value). Each key is unique in the collection and is used to retrieve the corresponding value. All the entities defined in Groovy are considered a kind of Map due to the dynamic nature of this language. Hence, it is possible to add or delete an attribute to an entity anytime the developer desires [84]. In this paper, the most important application of this feature lies in the mapping of basic types, which will be discussed in Subsection 4.1.1.

Moreover, another feature of Groovy, called Mixin, solves the problem of multiple inheritance in Java [85]. According to this feature, we can add functions from any number of classes to destination classes at the runtime. The application of this feature is shown in the mappings presented in Subsection 4.2.5.

Another important feature of Groovy is the inclusive feature provided for metaprogramming. In Groovy, the developers have access to the structure of all classes at the runtime and can make any change to the desired structures [85].

3.3. Aspect-oriented model

Frameworks supporting Groovy (such as the spring framework) provide a feature called dependency injection [86]. This feature allows for applying various behaviors and attributes to functions and classes at the runtime using AOP advisors [82]. One of the most important challenges of generating executable codes from abstract specifications is the problem of pre- and post-conditions. As will be discussed in the following sections, this problem can be overcome using dependency injection feature such that the readability of the generated code is not far from the system specification.

3.4. Convention over configuration

One of the most important features of Groovy is the ability to use the convention over configuration pattern. The aim of this design model is to reduce the number of decisions that should be made by a developer during development of a software system [84]. This would result in simplifying the software development process

and preventing the personal deviations. This feature enables the descriptor to avoid plenty of details required for implementation. Most of the implementation details can be extracted from conventions defined on top of the specification language. For instance, naming of a class can provide lots of information about the application and role of the class in the system, related access levels, basic functions required by the class, and even the method for persisting instances of the class. These conventions could be mapped and implemented by Groovy or any other language supporting the convention over configuration pattern. Although the convention over configuration is not utilized in the mapping rules of this paper, it is one of the reasons for selecting Groovy as the mapping destination language. The convention over configuration pattern would largely contribute to inclusion of more features in future research studies.

3.5. Similarity to the specification language

As mentioned before, Groovy supports the closure structure, lambda expressions, and aspect-oriented model. These features along with many others make Groovy very similar to formal specification languages such as Object-Z. This improves readability and comprehensibility of Groovy code generated from Object-Z specifications.

4. Mapping of Object-Z to Groovy

This section presents the rules required for mapping Object-Z notations to Groovy. Subsection 4.1 provides mapping rules for global paragraphs, which are common in the Z specification language. In Subsection 4.2, the mapping of the class schema as a major new construct in Object-Z is described. Finally, Subsection 4.3 addresses the mapping rules for operational operators, playing the role of a composer of other predicates.

Supposing that *ObjectZSpec* is a set of formal specifications in Object-Z language and *GroovyCode* is a set of Groovy codes, mapping function $\mathcal{M}$: *ObjectZSpec* $\rightarrow$ *GroovyCode* is defined. Function $\mathcal{M}$ maps each specification in Object-Z to an executable code in Groovy. Each construct in Object-Z abstract syntax is considered as a type throughout the paper. In addition, it is assumed that each Object-Z construct is a set of elements; hence, set notations, such as membership, are used throughout the paper.

4.1. Mapping of global paragraphs

In the Object-Z specification language, each block of a specification placed in the root of specification document (except classes) is called a global paragraph. In this section, the mapping of global paragraphs is studied in six general categories: basic types, axiomatic definitions, generic definitions, abbreviation definitions, free types, and schemas.

$\mathcal{M}(GlobalParagraphs) =$

$\mathcal{M}(BasicTypeDefinitions) \cup$

$\mathcal{M}(AxiomaticDefinitions) \cup$

$\mathcal{M}(GenericDeinitions) \cup$

$\mathcal{M}(AbbreviationDefinitions) \cup$

$\mathcal{M}(FreeTypes) \cup \mathcal{M}(Schemas)$

The union sign in this paper is an associative binary operator defined as in the following:

$\forall\, a, b \subseteq \mathcal{M} \cdot a \cup b = \{x : \mathcal{M} | x \in a \vee x \in b\}$

4.1.1. Basic types

Every Object-Z specification begins with certain objects that are members of the basic types or given sets of the specification [4]. The basic type paragraphs are defined as follows [87]:

$[Type_1, Type_2, \cdots, Type_N]$.

A complex structure may be assumed for basic types in the implementation phase. Therefore, these types are mapped to Groovy classes to be able to develop them in the future. A separate file is created for each class. In the class related to each basic type, a static property is defined to keep instances of that class. We have:

$\mathcal{M}(BasicTypeDefinitions) =$

$\mathcal{M}([IdentifierList]) =$

$\forall\, i : Identifier | i \in IdentifierList \cdot (\mathcal{M}(i) =$

public class i {

private static $List < i > instanceList$

$=$ new $ArrayList < i > ()$

public static $List < i > getInstanceList()$

$\{instanceList\}$

public $i()\{instanceList.add(this)\}$

})■

Moreover, a list of the classes created for basic types is stored in the *Context* class (for more information about *Context* class, refer to Subsection 4.1.2) as follows:

public static List <Class> basicTypes = $[[Type_1], [Type_2], \cdots, [Type_N]]$.

Discussion on soundness. Object-Z allows the specifier to assign or retrieve values of basic types' fields without defining them explicitly. Since all classes in

Groovy inherit features of the Map structure, there is no need to define fields in the class structure in Groovy; this means that the provided mapping rule covers the semantics of basic types in Object-Z. In addition, the *basicTypes* list of the *Context* class and the *instanceList* of each class will be used to enforce the global constraints at the system level. For each class including the *basicTypes* list of the *Context* class, the integrity of the global constraints will be checked for each object in the *instanceList* of the class.

4.1.2. Axiomatic definition

An axiomatic definition introduces one or more global notions (constants, operators, symbols, or functions) by a list of declarations and an optional list of predicates constraining their values [9]. To allow the mapping of global variables and functions in Object-Z, a public class named *Context* is defined in Groovy that contains these variables and functions. The mapping for each case of axiomatic definitions [9] is presented in the following subsections:

$$\mathcal{M}(AxiomaticDefinitions) =$$

$$\mathcal{M}(ConstantDefinitions) \cup$$

$$\mathcal{M}(OperatorDefinitions) \cup$$

$$\mathcal{M}(SymbolDefinitions) \cup$$

$$\mathcal{M}(FunctionDefinitions)$$

4.1.2.1. Constant definition

Constant values are mapped to Groovy code by a combination of a private static variable and two public static functions (namely, get and set). The get function retrieves the value of a constant. The set function assigns a value to a constant. Consider [*constant*] in the following axiomatic definition:

[*identifier*] : [*type*]
[*constraints*]

The mapping is illustrated below:

$$\mathcal{M}(ConstantDefinitions) = \forall\, i : Constant$$

$$\cdot(\mathcal{M}(constant_i : type_i \ll constraints_i \gg)$$

$$= \text{private static } type_i\ constant_i$$

$$\text{public static } type_i\ \text{get}Constant_i()\{constant_i\}$$

$$\text{public static void set}Constant_i\ (type_i\ \text{value})\ \{$$

$$\quad \text{if } (!\mathcal{M}(constraints_i))\ \text{throw new Exception}()$$

$$\quad constant_i = \text{value}$$

$$\})\blacksquare$$

Discussion on soundness. Since static properties are shared between all instances of a class in Groovy, mapping constant values to static properties satisfies the semantics of constants in Object-Z. In addition, any access to the constant trying to set an unacceptable value will be blocked due to the constraints' control mechanism. Since the access level of $constant_i$ field is private, the required constraints are controlled in the body of $setConstant_i$ as the public setter function of $constant_i$ property. If the provided value does not meet the specified constraints, the raised exception prevents execution of the next line of the code, which is equivalent to the expected non-deterministic behavior in the same condition in Object-Z.

Example 1. Consider a constant specified as follows:

$myConstant : \mathbb{Z}$
$myConstant \% 2 = 0$

Based on the given constraint, values of the constant specified in the above paragraph should always be even. The following code is obtained after mapping:

```
private static int myConstant

public static int getMyConstant(){myConstant}

public static void setMyConstant (int value) {

    if (! (value %2 == 0)) throw new Exception()

    myConstant = value

}
```

As seen, a new value is assigned to *myConstant* only if this value meets the specified constraints. Any direct access to *myConstant* variable is prevented to ensure the right application of constraints.

4.1.2.2. Operator definition

All operators in Groovy have the corresponding functions. To provide a specific implementation for any operator on basic types, its corresponding function should be overridden in Groovy. The mapping rules should be provided for operator definitions in two different categories: unary and binary operators.

$$\mathcal{M}(OperatorDefinitions) =$$

$$\mathcal{M}(UnaryOperatorDefinitions) \cup$$

$$\mathcal{M}(BinaryOperatorDefinitions)$$

Consider [*operator*] as a unary operator in the following specification:

$\_[operator] :\ [type]$
$\forall\, t : [type].t\ [operator] \iff [constraints]$

The mapping rule is illustrated below:

$$\mathcal{M}(UnaryOperatorDefinitions) =$$

$$\forall\, i : Operator | i \in UnaryOperators$$

$$\cdot(\mathcal{M}(\_operator_i : type_i \ll \forall\, t : type_i \cdot t\ operator_i$$

$$\Longleftrightarrow constraints_i \gg) =$$

public class $type_i\{\mathcal{M}(operator_i)()$

$$\{\mathcal{M}(constraints_i)\})\blacksquare$$

Consider [*operator*] as a binary operator in the following specification:

$\_[operator]\_ : \ [type1] \leftrightarrow [type2]$
$\forall\, t_1 : [type_1], t_2 : [type_2].t_1[operator]t_2 \Longleftrightarrow [constraints]$

The mapping rule is illustrated below:

$$\mathcal{M}(BinaryOperatorDefinitions)$$

$$= \forall\, i : Operator | i \in BinaryOperators$$

$$\cdot(\mathcal{M}(\_operator_{i\_} : type_{1_i}, type_{2_i} \ll \forall\, t_1 :$$

$$type_{1_i}, t_2 : type_{2_i} \cdot t_1 operator_i\ t_2$$

$$\Longleftrightarrow constraints_i \gg) =$$

public class $type_{1_i}\{\mathcal{M}(operator_i)(type_{2_i} other)$

$$\{\mathcal{M}(constraints_i)\})\blacksquare$$

In Groovy, each operator has a specific predefined function that should be overridden in order to change its functionality (see [84] for a complete list of operators and their corresponding functions). In the provided mapping rule, $\mathcal{M}(operator)$ returns the name of the function, which is responsible for specifying the functionality of *operator*. Both unary and binary functions are defined in the class related to their first operand. The binary function receives the second operand in its parameters.

Discussion on soundness. After mapping, the operator's usage syntax remains exactly the same as its usage syntax in Object-Z. Supposing that $\mathcal{M}(constraints_i)$ preserves semantics of the constraints defined on the operator, there is no gap between the semantics of the resulting Groovy code and the original specification.

Example 2. Consider the definition of the following operator on the *Person* basic type:

$\_ == \_ : Person \leftrightarrow Person$
$\forall\, s, t : Person.s == t \Longleftrightarrow s.id = t.id$

This operator is mapped to a function in the class resulting from the mapping of the type located on the left side of the operator. Due to the existence of "==" operator, *equals* function is overridden as follows:

```
public class Person {

    int id

    boolean equals (other) {id == other.id}

}
```

If the type on the left side of the operator is mapped to a system type in Groovy (such as Integer, List, etc.), then the related function of the mapped system type is overridden at the runtime using the metaprogramming feature. See the renaming of operation schemas in Subsection 4.2.5 for an example about how a function of a class can be overridden at the runtime.

4.1.2.3. Symbol definition

To map symbols from Object-Z to Groovy, the notion of functions is utilized. Consider [*symbol*] as a symbol in the following specification:

$[symbol]\ [type_1] : [type_2]$
$\forall\, t : [type_1].[symbol]\ [t] = [expression]$

The mapping rule is as follows:

$$\mathcal{M}(SymbolDefinitions) =$$

$$\forall\, i : Symbol \cdot (\mathcal{M}(symbol_i type_{1_i} : type_{2_i}$$

$$\ll \forall\, t : type_{1_i} \cdot symbol_i\ texpression_i \gg) =$$

public static $type_{2_i}\ symbol_i\ (type_{1_i}\ t)$

$$\{\mathcal{M}(expression_i)\})\blacksquare$$

Symbol definition in Object-Z is very similar to the definition of functions in Groovy. Execution of a symbol in Object-Z is mapped to execute a function in Groovy with the same syntax. We map each symbol to a static function to make it executable without instantiating new objects.

Discussion on soundness. Each symbol definition is mapped to a function with the same input parameters in Groovy, and the syntax of executing the symbol definition remains the same in the mapping. Assuming that $\mathcal{M}(expression_i)$ maps the body of the symbol definition to the equivalent Groovy code, semantics of symbol definitions are preserved by the provided mapping rule.

Example 3. Consider a symbol specification as follows:

$square\ [\mathbb{Z}] : \mathbb{Z}$
$\forall\ x : \mathbb{Z}.square\ [x] = x * x$

Such a specification is mapped to a static function in the *Context* class as shown in the following:

```
public static int square (int x){x * x}
```

4.1.2.4. Function definition

The function mapping is very similar to the mapping of symbol definition. Function definitions are defined as static functions in the *Context class*. For example, consider the following specification for the square function:

$square : \mathbb{Z} \rightarrow \mathbb{Z}$
$\forall\ i, o : \mathbb{Z}.(i, o) \in square \iff o = i * i$

It should be mapped exactly similar to the mapping given for symbol *square* in the previous subsection. However, *square* is a simple case in which the output is explicitly specified in terms of the inputs. In case of inverse specifications where input variables are defined based on the output variable(s), the previous mappings are useless.

Consider the following syntax specifying an inverse function:

$[function] : [type_1] \rightarrow [type_2]$
$\forall\ t_1 : [type_1], t_2 : [type_2].(t_1, t_2) \in [function]$ $\iff [constraints]$

The mapping rule is illustrated below:

$\mathcal{M}(ReverseFunctionDefinition) =$

$\forall\ i : Function | i \in ReverseFunctions \cdot$

$(\mathcal{M}(function_i : type_{1_i} \rightarrow type_{2_i}$

$\ll \forall\ t_1 : type_{1_i}, t_2 : type_{2_i} \cdot (t_1, t_2 \in function_i$

$\iff constraints_i) \gg) =$

public static $type_{2_i} function_i(type_{1_i} t_1)$

$\{limitedRange(type_{2_i}) \cdot findResult$

$\{\mathcal{M}(constraints_i)\}\})\blacksquare$

The *findResult* function takes a closure as its input parameter and checks the constraints inside the closure against each value of the specified range. The *limitedRange* function is responsible for providing a limited range of $type_{2_i}$; the specifier may participate in selecting the limited range. The first value in the specified range that satisfies the given constraints is returned as the output of the *findResult* function.

Discussion on soundness. Although the method used in the mapping of inverse functions cannot find all possible solutions, it is able to find at least one return value for the function if the search space is limited properly. Supposing that the limited range of $type_{2_i}$ is selected wisely, the Groovy code resulting from applying the provided mapping rule is able to find one of the matching return values of the specified function.

Example 4. Consider the following specification of the *root* function:

$root : \mathbb{Z} \rightarrow \mathbb{Z}$
$\forall\ i, o : \mathbb{Z}.(i, o) \in root \iff i = o * o$

Such a specification is mapped to the following static function:

```
public static int root (int input)

    {(1..1000).findResult {it * it == input}}
```

To ensure the efficiency of the generated code, the range of integers at the time of mapping should be limited (if the descriptor has not done so). The reason is that there is a wide range of integer values that can be taken as an input by the root function. As seen in the code resulting from the above mapping, the integer values range from 1 to 1000. This range can be shortened or expanded in proportion to the descriptor's need. It is highly recommended for the descriptor to specify the required range of integers or other unlimited types. The keyword '*it*' inside the *findResult* function refers to the current element of the array or range. The descriptor can specify a different specific problem implementation for the *findResult* function. For instance, the utilization of a binary search method, instead of the default linear search method, can improve the performance of the *findResult* function.

4.1.3. Generic definition

A generic definition is a generic form of an axiomatic definition used to define a family of global notions (constants, operators, symbols, or functions) and is parameterized by its formal parameters [9]. Since the type of generic parameters is not known, the main challenge to the mapping of these definitions is to map functions specified by generic types [9]. However, this challenge could be resolved by the dynamism feature of Groovy. Based on this feature, there is no need to specify the type of input parameters of functions at the designing time.

Example 5. Consider the following specification:

$[X, Y]$

$first : X \times X \rightarrow X$
$\forall\ x : X, y : Y.first(x, y) = x$

This specification could be mapped to a function such as the following function in Groovy:

```
public static def first(x, y) {x}
```

As seen in the mapped function, the function's return type is unknown. This return type is determined at the runtime based on the type of the assigned value to the x input argument. It is the advantage of the dynamism feature of Groovy as the destination language. It should be noted that, in Groovy, the use of the *return* keyword is optional. In addition, the last accessed value in the body of a function is assumed as the default return value of that function.

Besides the problem mentioned above, another concern in the mapping of generic definitions is the mapping of generic constraints. In this case, two issues are raised:

1. Mapping the constraint;
2. Applying the constraint globally.

For the first issue, a static function, named *genericConstraint*, is defined in the *Context* class. All generic constraints are checked in this function. The mapping rule for this function is illustrated below:

```
public static void genericConstraint() {
    basicTypes.each {basicType− >
        basicType.instanceList.each {instance− >
        if(![constraints])throw newException()}
    }
}
```

In this mapping rule, by traversing the list of basic types defined in the *Context* class, all instances of each basic type class are checked against the specified constraints.

Example 6. Consider the following specification:

$[X]$

$left, right : X$
$left \neq right$

The required mapping is defined as follows:

```
public static void genericConstraint(){
    basicTypes.each {basicType− >
        basicType.instanceList.each {instance− >
            if (basicType.instanceList.count
            {it==instance}>1)
                throw new Exception()
            }
        }
    }
```

The *each* commands are necessary for all the specifications of generic constraints, while the internal *count* command is only a mapping of the particular constraint in this example.

Now, for the second mentioned issue, i.e., the global application of generic constraints, it is sufficient to use the annotations and aspects of Groovy. These annotations call the *genericConstraint* function as the pre- and post-condition to execute each function that changes the state of an object in the system. By using @*Transactional*, it is also possible to protect the integrity of the function to be executed. For this purpose, the function should be executed only when the constraints specified in *genericConstraint* are met (before and after the execution of the function). The @*Transactional* attribute includes the body of a function in a transaction. Hence, the raise of any exception in the function execution causes rollback for all applied changes to state variables.

4.1.4. Abbreviation definition

An abbreviation definition introduces a type whose name is the identifier on the left side of the definition and whose values are specified using an expression on the right side [9]. The abbreviation definition mapping can be studied in four major categories [9] as follows:

$$\mathcal{M}(AbbreviationDefinitions) =$$

$$\mathcal{M}(ComputationalExpressions) \cup \mathcal{M}(Sets) \cup$$

$$\mathcal{M}(ClassUnions) \cup \mathcal{M}(Ranges).$$

4.1.4.1. Computational expression

Computational expressions are easily mapped using the closure structure in Groovy. Consider $[variable]$ as a computational expression in the following specification:

$$[variable] == [expression].$$

The mapping rule is illustrated below:

$$\mathcal{M}(CompitationalExpressions) =$$

$$\forall\, i : Variable \cdot (\mathcal{M}(variable_i == expression_i) =$$

$$\text{def } variable_i = \{\mathcal{M}(expression_i)\})\blacksquare$$

To access the real-time value of the computational expression, it is sufficient to run the following command. This command calls the mapped closure and returns its output value:

variable()

Groovy closures are executable code blocks. Putting parentheses after a closure's name executes its body and assigns the return value to the variable placed on the left-hand side of the assignment; see [85] for more information about Groovy closures. The only difference between syntaxes of computational expressions in Object-Z and Groovy is the need to put parentheses after *variable* name in order to access the real-time value of the computational expression.

Discussion on soundness. In the Object-Z semantics, the real-time value of computational expressions should be returned on each call. It is supposed that $\mathcal{M}(expression_i)$ preserves the semantics of computational expression body. Since Groovy closures calculate the return value on each call, the proposed mapping rule preserves semantics of the computational expression.

4.1.4.2. Defining set

A set is defined the same as follows:

$$\mathcal{M}(Sets) = \forall\ i : Sets \cdot (\mathcal{M}(i) = (\text{def} i = \cup_{e \in i} e))\blacksquare$$

The def keyword in Groovy is used to declare variables without specifying their types. Set membership is also controlled by the *contains* function, which returns *true* if its input value is a member of the specified set.

Example 7. Consider the following set:

```
def variable = [10, 'string']
```

To control the membership of an element in the above set, it is sufficient to run the following command:

```
variable.contains ([element])
```

4.1.4.3. Class union

An abstract class is defined for the mapping of a union of classes. The classes included in the class union abbreviation are derived from the defined abstract class. This abstract class is also derived from the *Schema* class introduced in Subsection 4.1.6. The *instanceList* of the abstract class is the union of *instanceList* collections of its sub-classes.

Consider [*Union*] as the class union in the following specification:

$$[Union] == MemberClass_1 \cup \cdots \cup\ MemberClass_N$$

According to the explanations provided in this section, the mapping rule is as follows:

$$\mathcal{M}(ClassUnions) = \forall\ i : ClassUnion$$

$$\cdot(\mathcal{M}(MemberClass_{1_i} \cup \cdots \cup\ MemberClass_{N_i}) =$$

$$\text{public abstract class } i \text{ extends } Scheme\Big\{$$

$$\text{public static List} < i > \text{getInstanceList ()}$$

$$\Big\{\sum_{i=1}^{N} \text{MemberClass}_i \cdot \text{instanceList}\Big\}$$

$$\Big\}$$

$$\cup_{i=1}^{N} \text{public MemberClass}_i \text{ extends } i\ \{\})\blacksquare$$

Discussion on soundness. Using the provided mapping rule, each instance of a member class is also an instance of the *Union* class. In addition, the *instanceList* property of the *Union* class contains all instances of member classes; this is of help in controlling global constraints defined on instances of the *Union* class. In this way, the provided mapping rule preserves semantics of class union in Object-Z, while this construct is mapped to Groovy code.

4.1.4.4. Range

A range definition is mapped from Object-Z to Groovy with almost no change in the structure.

$$\mathcal{M}(Ranges) = \forall\ i : Ranges, start : i, end :$$

$$i | (\forall\ e : i.start \leq e \wedge end \geq e)$$

$$\cdot(\mathcal{M}(i) = (\text{def}_i = start..end))\blacksquare$$

4.1.5. Free type

A free-type definition introduces a type whose name is the identifier on the left side of the definition and whose values are determined by the branches of the right side [9]. If the right side only involves constant values (simple free type), the free type is mapped to an enumeration in Groovy. Otherwise, if constructor functions are used on the right side of a free-type definition (complex free type), it should be mapped to a class definition. The free-type construct can be mapped in two ways, depending on its right-side branches.

$$\mathcal{M}(FreeTypes) = \mathcal{M}(SimpleFreeTypes) \cup$$

$$\mathcal{M}(ComplexFreeTypes)\blacksquare$$

Consider [*freeType*] in the following simple free-type specification:

$$[freeType] ::= [constant_1] | \cdots | [constant_N].$$

The mapping rule is illustrated below:

$$\mathcal{M}(SimpleFreeTypes) = \forall\ i : FreeType | i$$

$$\in SimpleFreeType \cdot (\mathcal{M}(i$$

$$::= constant_{1_i} | \cdots | constant_{N_i}) = \text{enum } i$$

$$= \cup_{c=1}^{N_i} constant_{c_i})\blacksquare$$

Consider [*freeType*] in the following complex free type specification:

$[freeType] ::= [constant_1]|\cdots|[constant_N]|[func_1]$

$\ll [params_1] \gg |\cdots|[func_M]$

$\ll [params_M] \gg$

The mapping rule is illustrated below:

$\mathcal{M}(ComplexFreeTypes) = \forall\ i : FreeType|i$

$\in ComplexFreeType$

$\cdot(\mathcal{M}(i ::= constant_{1_i}|\cdots|constant_{N_i}|func_{1_i}$

$\ll params_{1_i} \gg |\cdots|func_{M_i} \ll params_{M_i} \gg =$

public class i\{

private $i()$

$\cup_{c=1}^{N_i}$public static i $getConstant_{c_i}()\{new\ i()\}$

$\cup_{c=1}^{M_i}$public static i $func_{c_i}(params_{c_i})\{new\ i()\}$

\})■

In this mapping, constant values are mapped to class properties. These read-only properties create and return a new instance of the class. For each constructor function, a corresponding function is introduced in the class. The type of parameters is determined from the type of the specified function's input parameters. Note that the access level of the default constructor of the class is defined as private in order to prevent creation of new instances of the class. Obviously, user intervention is required to complete the mapping of free types because no more implementation details can be extracted from their specifications.

Discussion on soundness. In the case of simple free types, which are only composed of constants, mapping to a simple enumeration in Groovy fulfills the requirements of the semantics of free types. In case of complex free types including constructor functions, regardless of the presence of constants, the branching logic is not deducible from the Object-Z specification. The provided mapping rule simply returns a new object of the class resulting from mapping the free-type specification to Groovy code. Of course, additional details are required to be provided by the specifier in order to produce the final implementation in Groovy. In general, the provided mapping rule covers all deducible semantics of simple and complex free types, and the rest is postponed to the next implementation phases.

4.1.6. Schema

A schema is a pattern of declaration and constraint [9]. The class construct is used to map the schema specifications. In this mapping, variables of the schema are defined as properties in the mapped class. Each schema should have a function that checks the consistency of constraints with current values of variables. For this purpose, an abstract class, named *Schema*, is designed such that all the mapped classes of each schema are derived from *Schema*. A sample implementation for class *Schema* is provided here:

```
public abstract class Schema{
    protected abstract boolean
    checkConstraintsInternal();
    public void checkConstraints
    (Closure addedCondition = {true}){
        if(!checkConstraintsInternal()
        | |!addedCondition()) throw new
          Exception()
    }
}
```

A function, called *checkConstraintsInternal*, is introduced in this class. The *Schema* class forces all the inherited classes to implement this function. The *checkConstraintsInternal* function returns *true* if the constraints of the schema are met; otherwise, the *checkConstraints* function, which is called after every change in variables, raises an exception. The *checkConstraints* function takes an input parameter of the closure type. The default value of this parameter is a closure that always returns *true*. This input parameter is later used to control pre-conditions of each operation schema of the class specification.

Two special annotations are defined to apply the constraints of the schema.

1. The @*CheckConstraintsAfter* annotation signs the functions and controls the constraints of the schema after the execution of each signed function. This annotation takes an optional parameter of the closure type that controls possible post-conditions of the related function;
2. The @*CheckConstraintsAround* annotation signs the functions and controls the constraints of the schema before and after the execution of each signed function. This annotation takes two optional parameters of the closure type: one for controlling the possible pre-conditions before the execution and another for controlling the possible post-conditions after the execution of the related function.

The next step is to define an interceptor to

control the above annotations. This interceptor has two responsibilities:

1. It executes the *checkConstraints* function before and after the execution of functions signed by *CheckConstraintsAround*. It may receive possible pre- and post-conditions from the annotation and send them to the *checkConstraints* function;
2. It executes the *checkConstraints* function after the execution of functions signed by *CheckConstraintsAfter*. It may receive possible post-conditions from the annotation and send them to the *checkConstraints* function.

Listing 6 shows the implementation of the mentioned annotations and interceptor. The interceptor uses AOP advisors to implement the desired functionality of each annotation.

Consider [*SchemaParagraph*] as a schema definition in the following specification:

$$\begin{array}{|l} [SchemaParagraph] \\ \hline [stateVariable_1] : [type_1] \\ \cdots \\ [stateVariable_N] : [type_N] \\ \hline [constraints] \\ \hline \end{array}$$

The mapping rule for such a schema is as follows:

$\mathcal{M}(Schemas) = \forall\, i : Schema$

$\cdot(\mathcal{M}(stateVariable_{1_i} : type_{1_i} \cdots$

$stateVariable_{N_i} : type_{N_i} \ll constraints_i \gg)$

$=$ public class i extends Schema {

$\cup_{c=1}^{N_i} type_{c_i} stateVariable_{c_i}$

$\cup_{c=1}^{N_i} type_{c_i}$ get$StateVariable_{c_i}()\{stateVariable_{c_i}\}$

$\cup_{c=1}^{N_i}$ @Transactional @CheckConstrainsAfter

public void set$StateVariable_{c_i}(type_{c_i}$ value)

$\{stateVariable_{c_i} =$ value$\}$

@Override protected boolean

checkConstraintsInternal()$\{\mathcal{M}(constraints_i)\}$

})■

Discussion on soundness. The most important consideration in mapping of a schema is assuring fulfillment of its constraints when one or more state variables change. Functions responsible for changing the schema variables and setter functions defined for state variables are defined as transactional functions. If constraints are not met, an exception is raised that prevents the assignment of values to variables through transaction rollback. This means that any change in the state of a schema preserves its constraints; otherwise, the behavior of the system is non-deterministic. Thus, semantics of schema declaration in Object-Z is preserved by the provided mapping rule.

Example 8. Consider the following specification for the *PowerSupply* schema:

$$\begin{array}{|l} PowerSupply \\ \hline contactor : \mathbb{Z} \\ status : \mathbb{Z} \\ \hline contactor = 1 \Rightarrow status = 0 \\ \hline \end{array}$$

Such a schema is mapped to the following class implementation in Groovy:

```
public class PowerSupply extends Schema{

    private int contactor

    public int getContactor(){contactor}

    @Transactional

    @CheckConstraintsAfter

    public void setContactor(int value)

    {contactor = value}

    private int status

    public int getStatus(){status}

    @Transactional

    @CheckConstraintsAfter

    public void setStatus(int value)

    {status = value}

    @Override

    protected boolean checkConstraintsInternal()

    {contactor! = 1| |status == 0}

}
```

4.2. Class

A major new construct in Object-Z is the class schema, which captures the object-oriented notion of a class by encapsulating a single state schema, its associated initial state schema, and all the operation schemas of the given state [9]. Each class in an Object-Z

```
public abstract class Schema{
    protected abstract bool checkConstraintsInternal();
    public void checkConstraints(Closure addedCondition = { true })
        if(!checkConstraintsInternal() || !addedCondition()) throw new Exception()
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CheckConstraintsAfter { Closure postCondition = { true } }

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface CheckConstraintsAround {
    Closure preCondition = { true }
    Closure postCondition = { true }
}

@Aspect
public class CheckConstraintsInterceptor {
   @Around("@annotation(CheckConstraintsAround)")
    public Object around(ProceedingJoinPoint point, CheckConstraintsArount annotation) {
        if(point.target instanceof Schema)
            point.target.checkConstraints(annotation.preCondition)
        Object result = point.proceed();
        if(point.target instanceof Schema)
           point.target.checkConstraints(annotation.postCondition)
        result;
    }

   @Around("@annotation(CheckConstraintsAfter)")
    public Object after(ProceedingJoinPoint point, CheckConstraintsAfter annotation){
        Object result = point.proceed();
        if(point.target instanceof Schema)
           point.target.checkConstraints(annotation.postCondition)
        result;
    }
}
```

Listing 6. Implementation of the AOP advisors.

specification is mapped to a class in Groovy with the same name. In Groovy, the access level to each entity in the final class is specified based on the rules presented in [9].

4.2.1. Local specifications

The mapping process of local specifications of a class is very similar to that of global paragraphs. Local specification of basic types, axiomatic definitions, abbreviations, and free types are mapped exactly similar to global paragraphs. However, they are hosted by their parent class instead of the *Context* class.

4.2.2. State schema

Similar to the mapping of typical schemas, the state schema of a class is mapped onto a set of properties of the final class. Of note, if a class is derived from other classes, then the *checkConstraints* function (refers to Subsection 4.1.6) is defined as the combination of constraints of the child class and *checkConstraints* functions of the parent classes.

4.2.3. Initial schema

The initial schema lacks input variables and return values. It only assigns initial values to variables of the class state schema. Therefore, the initial schema of each class is mapped to the default (non-parametric) constructor of that class. However, if a class is derived from other classes, it should call their constructor functions in the body of its constructor. This point is discussed in more detail in Subsection 4.2.5. The initial schema is defined as a non-parametric function, called *ClassName_Constructor*, with no return value. This function is called in the constructor of the target class. It should be noted that, in the rest of the paper, wherever we mention a target object, it refers to the object intended to be mapped during the mapping process. Considering [*ClassName*] as the name of the target class, the mapping of the initial schema is as follows:

$\mathcal{M}(InitialSchema) =$

@Transactional public $Class()\{ClassConstructor()\}$

@CheckConstraintsAfter protected void

$ClassConstructor()\{\mathcal{M}(predicates)\}$■

4.2.4. Operation schema

Every operation schema in an Object-Z specification is mapped to a function of the same name in the

related class in Groovy. Each input of the schema is considered as an input parameter for the final function in Groovy. Considering the number of output parameters of an operation schema, three possible cases are studied.

$\mathcal{M}(OperationalSchemas) =$

$\mathcal{M}(NoOuputOperationSchemas)$

$\mathcal{M}(SingleOuputOperationSchemas) \cup$

$\mathcal{M}(MultipleOuputOperationSchemas) \cup$

Before describing these three cases, some naming conventions are explained here. In the mapping rule given in this subsection, $\mathcal{M}(params)$, $\mathcal{M}(preCondition)$, and $\mathcal{M}(postCondition)$ refer to the mapping of input variables, pre- and post-conditions of the target operation schema, respectively. $\mathcal{M}(expression)$ is a result of mapping those predicates of the operational schema, assigning values to the output parameters or explicitly specifying a change in the state variables of the target class. *accessor* specifies the access level of the mapped function. The process of selecting *accessor* is the same as that presented in [16].

The three possible cases for mapping operation schemas are as follows:

1. *The operation schema lacks any output.* The mapping rule for such an operation schema is as follows:

   $\mathcal{M}(NoOuputOperationSchemas) =$

   $\forall\ i : Schema || output(i) = 0 \cdot (\mathcal{M}(i) =$

   @CheckConstraintsAround(preCondition $=$

   $\mathcal{M}(preCondition), postCondition =$

   $\mathcal{M}(postCondition))accessor$ void

   $i(\mathcal{M}(params))\{\mathcal{M}(expressions)\})\blacksquare$

   In this case, the return type of the resulting function in Groovy is void, and no return value is expected from the resulting function. Pre- and post-conditions are automatically controlled using the CheckConstraintsAround attribute (see Listing 1);

2. *The operation schema has one output variable.* In this case, the final function in Groovy returns one value. The mapping rule for such a schema is illustrated below. In this rule, *type* refers to the type of the only output parameter of the operation schema. If it is not possible to detect the type of the return value, it is specified using the def keyword.

   $\mathcal{M}(SingleOuputOperationSchemas) =$

   $\forall\ i : Schema || output(i) = 1 \cdot (\mathcal{M}(i) =$

   @CheckConstraintsAround(preCondition $=$

   $\mathcal{M}(preCondition), postCondition =$

   $\mathcal{M}(postCondition))$

   $accessor\ \mathcal{M}(type) i(\mathcal{M}(params))\{$

   $\mathcal{M}(type) result$

   $\mathcal{M}(expressions) result$

   $\})\blacksquare$

   Since the operation schema returns a single value, the return value type of the resulting function is mapped using $\mathcal{M}(type)$. This function tries to extract the return value type of the operation schema using a simple switch-case statement. If $\mathcal{M}(type)$ function fails to detect the proper type, it will return def as the default type of the return value. Since Groovy is a dynamic programming language, it is possible to postpone the identification of the return value type to the runtime using the def keyword;

3. *The operation schema has more than one output variable.* In this case, the final function in Groovy returns a *Map* [85], which has keys with the same name as output variables of the operation schema. The mapping rule in this case is as follows:

   $\mathcal{M}(MultipleOuputOperationSchema) =$

   $\forall\ i : Schema || output(i)| > 1 \cdot (\mathcal{M}(i) =$

   @CheckConstraintsAroundpreCondition $=$

   $\mathcal{M}(preCondition, postCondition =$

   $\mathcal{M}(postCondition))$

   $accessor$ Map $i(\mathcal{M}(params))\{$

   $Map < String, Objec > result = [:]$

   $\mathcal{M}(expressions)$

   $result$

   $\})\blacksquare$

   At first, a Map variable named *result*, which takes strings as the key and objects as the value, is declared. Each assignment to an output variable is mapped to an assignment to a key of the *result* map. The name of the assigned key is the same as the

output variable name. Finally, the *result* variable is returned as the output value of the function.

Discussion on soundness. All the provided mapping rules for different types of operation schemas are using an attribute named @CheckConstraintsAround for ensuring pre- and post-conditions of the operator schema (see Listing 1 for implementation details of the @CheckConstraintsAround attribute). In addition, the @CheckConstraintsAround attribute checks fulfillment of global state constraints of the schema. Supposing that $\mathcal{M}(preCondition)$, $\mathcal{M}(postCondition)$, and $\mathcal{M}(expressions)$ map pre-conditions, post-conditions, and body of the operation schema onto proper Groovy codes, it is expected that the provided mapping rule preserves semantics of operation schemas in Object-Z.

4.2.5. Class inheritance

Since the Object-Z specification language supports multiple inheritance [4], this notion has to be covered in the mapping of classes to the Groovy programming language. To map the class inheritance, three possible cases are studied in the following:

$\mathcal{M}(Inheritance) = \mathcal{M}(SingleInheritance) \cup$

$\mathcal{M}(MultipleInheritanceWithSameRoot) \cup$

$\mathcal{M}(MultipleInheritanceWithDifferentRoots)$

1. *The child class is only derived from one other class.* Considering *SubClass* as the child class and *SuperClass* as its parent class, the mapping rule for this inheritance structure is illustrated below:

   $\mathcal{M}(SingleInheritance) =$

   $\mathcal{M}(SubClass \ll SuperClass \gg =$

   public class $SuperClass$ extends $Schema$

   $\{\mathcal{M}(SuperClassBody)\}$

   public class $SubClass$ extends $SuperClass$

   $\{\mathcal{M}(SubClassBody)\}$■

   In this case, the child class is derived from the parent class using the keyword extends in Groovy. The function resulting from mapping the initial schema of the parent class should be called in the constructor of the child class prior to the function resulting from mapping the initial schema of the child class. Through mapping, each instance of *SubClass* is an instance of *SuperClass*, too. All state variables and operation schemas of *SuperClass* are accessible in *SubClass*, and all constraints on state variables of *SuperClass* are controlled in the operations of the *SubClass*.

   As an example of this type of inheritance, see the mapping provided for the *CreditCardCount* (refer to Subsection 5.3) and *CreditCardConfirm* classes (refer to Subsection 5.4) in the case study.

2. *The child class is derived from two or more classes, and all the super classes eventually are derived from the same class.* In the following mapping rule, *SuperClass* represents the root class, *SubClass* represents the child class, $IntermediateClass_i$ represents the ith medial class, and N is the total number of medial classes.

   $\mathcal{M}(MultipleInheritanceWithSameRoot) =$

   $\mathcal{M}(SubClass \ll \bigcup_{i=1}^{N} IntermediateClass_i$

   $|IntermediateClass_i \ll SuperClass \gg \gg) =$

   public class $SuperClass$ extends $Schema$

   $\{\mathcal{M}(SuperClassBody)\}$

   $\cup_{i=1}^{N}$ public class $IntermediateClass_i$ extends

   $SuperClass\{\mathcal{M}(IntermediateClassBody_i)\}$

   @Mixin($\cup_{i=1}^{N} IntermediateClass_i$) public class

   $SubClass$ extends $SuperClass$

   $\{\mathcal{M}(SubClassBody)\}$■

   The child class is derived from the root class using the keyword extends in Groovy. The functions of the super classes are added to the child class using the Mixin attribute of Groovy [88]. In this case, the constructor function of each super class is added to the constructor function of the child class in order of participation of the corresponding classes in the inheritance tree. By using this mapping, all instances of *SubClass* are instances of *SuperClass* and each of intermediate classes using the method described in Subsection 4.2.5.2. In addition, all state variables and operations of intermediate classes are accessible in *SubClass*, and all constraints on state variables of intermediate classes and *SuperClass* are controlled in all operations of *SubClass*.

   The mapping of the *CreditCardConfirmAndCount* class in the case study (see Subsection 5.5) is an example for mapping multiple inheritance.

3. *The child class is derived from two or several classes, which do not have the same parent class:* In this case, the child class is derived from the *Schema* class, and the body of all the parent classes is added to the child class using the Mixin attribute. The mapping rule is illustrated below,

where $SuperClass_i$ refers to the ith parent class and $SubClass$ refers to the child class.

$\mathcal{M}(MultipleInheritanceWithDifferentRoots) =$

$\mathcal{M}(SubClass \ll \bigcup_{i=1}^{N} SuperClass_i \gg) =$

$\cup_{i=1}^{N}$ public $classSuperClass_i$ extends

$Schema\{\mathcal{M}(SuperClassBody_i)\}$

@Mixin($\cup_{i=1}^{N} SuperClass_i$) public class

$SubClass$ extends $Schema$

$\{\mathcal{M}(SubClassBody)\}$■

According to the pattern used in this paper to name the function related to the initial schema and due to the uniqueness of class names, the name of the function related to the initial schema of each class is unique. Hence, the child class can easily call the functions related to the initial schema of the parent classes. In the case of operation schemas, with the same name, the function related to the last class mentioned in the Mixinattribute is chosen. Similar to the previous mapping rule, all instances of *SubClass* are instances of all super classes using the method described in Subsection 4.2.5.2.

Discussion on soundness. The provided mapping rules for each studied inheritance case preserve the semantics of class inheritance in Object-Z specifications by ensuring the following required properties:

1. Instances created from each subclass are also instances of the parent classes (see Subsection 4.2.5.2);
2. State variables and operation schemas of the parent classes are available in the child classes;
3. Constraints defined on state variables of the parent classes are also controlled by the child classes;
4. Init schemas of the parent classes are called inside the initial schema of the child classes;
5. Multiple inheritance is supported;
6. Overriding and renaming of operation schemas are supported (see Subsection 4.2.5.1).

4.2.5.1. Operation schema renaming

One of the challenges in the class inheritance mapping is the renaming of operation schemas. In this case, an operation schema in the parent class is renamed, and a new implementation of that schema may be presented in the child class (often using the operation schema of the same name in the parent class). In this case, the schema's new body in the child class is implemented in a new private and uniquely named function. Then, the inherited function is renamed to the new specified name in the constructor of the target class. Finally, the uniquely named function is used to provide a new implementation for the renamed function. Consider the following specification:

$SubClass$

$|SuperClass$ $[newSchemaName/oldSchemaName]$

The mapping of the above renaming in the child class is performed by changing the name of the homonymous function of the parent class in the child class constructor. This task uses features of dynamism and metaprogramming of the Groovy dynamic language. The following is the rule for the mapping of schema renaming:

[new *SchemaName*]

= metaClass.getMetaMethod([*oldSchemaName*],

[*parameter_types*] as *Class*])

In Groovy, each class has a property, named *metaClass*, which allows accessing or modifying the definition of the class. A method named *getMetaMethod* of *metaClass* takes both a name and a list of parameter types and returns the matching method. [*parameter_types*] in this mapping refers to the list of Groovy classes corresponding to the type of input parameters in the target function. The mapping of the *CreditCardCount* class in the case study presented in Section 5 is an example of the mapping of schema renaming.

4.2.5.2. Membership problem

Another problem of the class inheritance mapping is that each instance of the child class has to be castable to the parent classes. Moreover, the inheritance has to be controllable using the *instanceOf* operator. The *instanceOf* operator checks whether an object is an instance of a specific class. The mentioned problem only occurs in Cases 2 and 3 of the class inheritance mapping. To overcome this problem, *instanceOf* and *asType* methods of the child class are overridden. The next example presents the solution to these problems.

Example 9. If a child class, named A, is derived from B, C, and D, then the following functions must be implemented in class A:

```
@Override boolean instanceOf(Class clazz)
 { [this.class, B, C, D].contains(clazz)}
@Override def asType (Class clazz) {
    if ([B, C, D].contains(clazz)){
       def newObject = clazz?.newInstance()
```

```
        copyProperties(this, newObject)
        newObject
    }
    else super.asType(clazz)
}
def copyProperties (source, target){
    source.properties.each {key, value− >
        if (target.hasProperty(key) && !(key in
        ['class', 'metaClass'])) target[key] = value
    }
}
```

In this example, *instanceOf* and *asType* methods are overridden. If the input class is B, C or D, the *instanceOf* method returns *true*. In this case, the *asType* method creates a new object of the input class, copies values of the properties of the current instance of A to the new object using the *copyProperties* method, and finally returns the new created object; otherwise, the *instanceOf* method returns *false*, and the *asType* method calls the *asType* method of its super class. In this example, the *copyProperties* method uses the metaprogramming feature of the Groovy language. At last, the following commands will yield the expected results as shown below:

```
A item = new A()
assert item.instanceOf(B)
assert item.instanceOf(C)
assert item.instanceOf(D)
B b = item as B; assert b != null
C c = item as C; assert c != null
D d = item as D; assert d != null
```

4.3. Operational operators

To map operational operators, the method proposed in [9] with slight modifications is presented. These modifications originate from the difference in application of pre- and post-conditions in C++ and Groovy. In the method proposed in this paper, the pre- and post-conditions are defined via the related annotations. Hence, the corresponding function is executed completely, or else it fails (i.e., none of its commands is executed).

4.3.1. Conjunction

Conjunctions in the form of $Op_1 \wedge Op_2$ are mapped as follows:

$\mathcal{M}(Conjunction) = \mathcal{M}(op_1 \wedge op_2)$

= bool conjunction = {op_1, op_2

→ try {op_1()&&op_2()} catch(ignored){false}

}

conjunction(op_1,op_1)■

This closure takes op_1 and op_2 functions as the inputs and calls the conjunction of these two functions. If an error occurs or the pre- or post-conditions of a function are not met, this command returns false.

Discussion on soundness. The order of evaluation of conjunction operands in Object-Z and Groovy is the same. Since the overall value of the conjunction is already determined, if the first operand is evaluated to be false, the second operand will not be evaluated. This is the reason why the semantics of the conjunction operator in Object-Z is preserved by the provided mapping rule.

4.3.2. Choice

Choice operations in the form of $Op_1 \vee Op_2$ are mapped as follows:

$\mathcal{M}(Choice) = \mathcal{M}(op_1 \vee op_2) =$

bool choice = {op_1, op_2 →

defindex=new Random().nextInt(2)

if try[op_1,op_2][index]))return

else tryOp([op_1,op_2][1-index])

}

boolean tryOp={op→try{op()}catch(ignored){false}}

choice(op_1,op_1)■

As seen, the above construct randomly tests one of the input functions and does not execute the other function if the first try succeeds. The second function is executed when the first function fails. This closure uses another closure named tryOp, which is responsible for preventing exceptions being raised when a closure cannot be executed.

Discussion on soundness. Since the choice closure selects an operator randomly to be executed, its behavior is non-deterministic and fulfills the semantics of the choice operator in Object-Z.

4.3.3. Sequential composition
Sequential compositions in the form of Op_1, and Op_2 are mapped using the following closure:

$\mathcal{M}(Sequential) = \mathcal{M}(op_1; op_2) =$

def sequential={op$_1$,op$_2$ →

defresult

tryOp{op$_1$();result=op$_2$()})

result

}

sequential(op$_1$,op$_1$)■

A closure named sequential has been defined. The sequential closure receives two functions as its input parameters and executes them sequentially. The value returned by this closure is derived from the second parameter.

Discussion on soundness. Groovy executes lines of code one by one in their present order. Since the operands of sequential composition are called in the same order in the sequential closure and the return value is determined based on the possible return value of the second operand, semantics of sequential composition in Object-Z is preserved by the introduced mapping rule.

4.3.4. Parallel composition
To provide a proper parallelism, parallel compositions in the form of $Op_1 \| Op_2$ are mapped using the following closure:

$\mathcal{M}(Parallel) = \mathcal{M}(op_1 || op_2 =$

void parallel = {op$_1$, op$_2$ → GparsPool.withPool

{op$_1$.callAsync()&&op$_2$.callAsync()}}

parallel(op$_1$,op$_1$)■

A new closure named parallel is defined. This closure executes op$_1$ and op$_2$ functions in parallel using the facilities provided by GparsPool [89] and returns a conjunction of the results.

4.3.5. Negation
Negation operators in the form of $\neg Op$ are mapped using the following closure. The body of this closure is similar to the result of the method presented in [9]:

$\mathcal{M}(Not) = \mathcal{M}(\neg \text{op}) =$

bool not={op!op()}

not(op)■

In Groovy, each non-zero and non-null value can be considered as true. We have defined a not closure, which receives a function in its parameters and negates the result of execution of that function.

5. Case study

This section evaluates the applicability of the proposed mapping rules. For this purpose, these rules are applied to the credit card management system (refer to Subsection 2.1) to generate Groovy code. Before starting to map the specification, an empty class named *Context* is created. This class will be used as a placeholder for global definitions and constraints in the next subsections.

5.1. Mapping the global paragraphs

The mapping of the specification to Groovy code starts with the global paragraphs. The first task involves mapping of the *Customer* basic type onto an empty class. Contents of this class could be implemented in proportion to the future needs. In this class, a constructor and a static variable named *instanceList* (which keeps instances of the class) are defined. The list of created instances is used to control the generic definitions. No generic constraint is involved in the mapping of the credit card management system specification. However, to develop a general pattern that could be used in automating the mapping process, the *instanceList* variable is incorporated in each basic type. See Listing 7 for the mapping of the *Customer* basic type.

In addition, the *Customer* class is added to the *basicTypes* list of the *Context* class. The *limitValue* set is mapped to a set in the *Context* class. Since *status* is specified as a free type, which only accepts constant values, it is sufficient to map this free type to a simple enumeration. See Listing 8 for the mapping of the *Context* class and the *Status* enumeration.

5.2. Mapping a simple class

After mapping the global paragraphs, the next step

```
public class Customer {
    private static List<Customer> instanceList = new ArrayList<Customer>()
    public static <Customer> getInstanceList() { instanceList }
    public Customer(){ instanceList.add(this) }
}
```

Listing 7. The mapping of the *Customer* basic type.

```
public class Context {
    public static List<Class> basicTypes = [Customer]
    public static limitValue = [1000, 2000, 5000]
}
enum Status { valid, invalid }
```

Listing 8. The mapping of the *Context* class and the *Status* enumeration.

is the mapping of the *CreditCard* class. According to the aforementioned mapping rules, all classes should be derived from a class named *Schema*. The *limit* and *expiryValue* constants are mapped to the corresponding static properties. The constraints of the state schema are only applied to the new value of the *limit* and *expiryValue* constants. For example, each time a value is assigned to the *limit* constant, the new value belongs to both of the set of natural numbers and the *limitValue* set. The *expiryValue* constant is also mapped similarly.

Of note, the state schema's constraints should be met before and after changing the value of each variable. To this end, the *CheckConstraintsAround* annotation along with the possible pre- and post-conditions of the operation schemas is applied to all the functions that change one or several properties of a class. As mentioned before, this annotation executes the *checkConstraints* function before and after the execution of each signed function. The *checkConstraints* function raises an exception if the conditions specified by *checkConstraintsInternal*, pre- or post-conditions are not met. The transactional nature of the functions that change the class state suggests that changes occur only if the state schema's constraints are met before and after the changes. In addition, the constructor of the target and set functions, assigning values to the variables of the state schema, are defined as transactional functions. After the execution of the body of these transactional functions, *CheckConstraintsAfter* is called to control the state schema's constraints.

To continue the mapping of the *CreditCard* class, a function named *CreditCard_Constructor* is defined for the initial schema of this class. The contents of the initial schema are mapped to the *CreditCard_Constructor* function. This function is defined as a protected function since there is no need to access it from the outside. However, it may be called in the default constructor of classes inheriting *CreditCard*. In addition, there is no need for this function to be transactional since transactional functions are called by the class constructor (which itself is a transactional function).

The *reissue*, *withdraw*, *deposit*, and it newDay functions are mapped, as shown in Listing 9. *CheckConstraintsAround* is applied to all these transactional functions. In some cases, *CheckConstraintsAround* is called with an additional parameter of the closure type since the target function may have pre-conditions that should be met prior to its execution. These pre-conditions are controlled by the state schema's constraints within the *checkConstraints* function. See Listing 9 for the mapping of the *CreditCard* class.

5.3. Mapping of a single inheritance

Single inheritance is used to inherit the *CreditCardConfirm* class from the *CreditCard* class. Similar to the previous functions, the *fundsAvail* function is mapped using the proposed mapping rules. To map the *withdrawConfirm* function, the sequential composition function (whose mapping is given in Subsection 4.3.3) is used. This function returns the output value of the last input closure. Therefore, the return type of the *fundsAvail* function is considered as the return type of the *withdrawConfirm* function. Since the *CreditCardConfirm* class lacks an initial schema, its constructor only calls the *CreditCard_Constructor* function of the parent class. See Listing 10 for the mapping of the *CreditCardConfirm* class.

5.4. Mapping of the renamed operation schema

The mapping of the *CreditCardCount* class is a bit more complicated than the previous ones. The reason is that the *CreditCardCount* class renames the *withdraw* function of its parent class and presents a new implementation for this function. The *withdraw* function is mapped to a private function in Groovy with a new unique name (*newWithdraw*). Next, in the constructor of the *CreditCardCount* class, the *withdraw* function of the parent class is renamed to *oldWithdraw*. The *oldWithdraw* function is incorporated into the body of the *newWithdraw* function. Afterwards, the new *withdraw* function, called the *newWithdraw* function, is added to the class instance in the constructor. The *withdrawals* state variable is also implemented similar to the previously mapped variables. See Listing 11 for the mapping of the *CreditCardCount* Class.

5.5. Mapping of multiple inheritance

The *CreditCardConfirmAndCount* class is derived from both the *CreditCardConfirm* and *CreditCardCount* classes. The *CreditCardConfirmAndCount* class also is derived from the *CreditCard* (root) class. As mentioned before, the inheritance is implemented with the *Mixin* attribute. Moreover, since the *withdraw* function in the *CreditCardCount* class is changed, this class is mentioned after the other super class (i.e.,

```
public class CreditCard extends Schema {

    private static int limit
    public static int getLimit() { limit }
    public static void setLimit(int value) {
        if (value >= 0 && Context.limitValue.contains(value)) limit = value
    }

    private static int expiryValue
    public static int getExpiryValue() { expiryValue }
    public static int setExpiryValue(value) { if (value >= 0) expiryValue = value }

    private int balance
    public int getBalance() { balance }
    @Transactional @CheckConstraintsAfter
    public void setBalance(int value) { balance = value }

    private Customer owner
    public Customer getOwner() { owner }
    @Transactional @CheckConstraintsAfter
    public void setOwner(Customer value) { owner = value }

    private int expiry
    public int getExpiry() { expiry }
    @Transactional @CheckConstraintsAfter
    public void setExpiry(int value) { expiry = value }

    private Status status
    public Status getStatus() { status }
    @Transactional @CheckConstraintsAfter
    public void setStatus(Status value) { status = value }

    @Override
    protected boolean checkConstraintsInternal()
    { expiry == 0 ? status == Status.invalid else status == Status.valid }

    @Transactional
    public CreditCard() { CreditCard_Constructor() }
    @CheckConstraintsAfter
    protected void CreditCard_Constructor(){ setBalance(0); setExpiry(expiryValue) }

    @Transactional @CheckConstraintsAround( preCondition = { status == Status.invalid } )
    public void reissue() { setExpiry(expiryValue) }

    @Transactional @CheckConstraintsAround( preCondition = {
            amount >= 0 && amount <= balance + limit && status == Status.valid })
    public void withdraw(int amount) { setBalance(balance - amount) }

    @Transactional @CheckConstraintsAround( preCondition = { status == Status.valid })
    public void deposit(int amount) { setBalance(balance + amount) }

    @Transactional @CheckConstraintsAround
    public void newDay() setExpiry(expiry - 1) }
}
```

Listing 9. The mapping of the *CreditCard* class.

CreditCardConfirm) in the *Mixin* attribute. Therefore, the *CreditCardConfirmAndCount* class inherits the *withdraw* function from the *CreditCardCount* class. In the constructor of the *CreditCardConfirmAndCount* class, the functions related to the initial schema of the parent classes are called in the order specified in the inheritance tree of the *CreditCardConfirmAndCount* class. See Listing 12 for the mapping of the *CreditCardConfirmAndCount* class.

The mapping of the *CreditCards* Class is given in Appendix B.

6. Conclusions and future work

In spite of the mapping rules introduced in previous works to map VDM++ specifications to Java code, the existing mapping rules aimed at JVM-based programming languages (as the mapping destination) do not properly cover the specification constructs of the source specification language. According to the high popularity of JVM-based languages, the necessity of developing highly complete mapping rules aimed at these languages becomes more evident.

```
@Mixin(CreditCard)
public class CreditCardConfirm extends CreditCard {

    @Transactional
    public CreditCardConfirm(){ CreditCard_Constructor() }

    @Transactional
    @CheckConstraintsAround
    public int fundsAvail() { def funds = balance + limit }

    @Transactional
    @CheckConstraintsAround
    public int withdrawConfirm (int amount)
    { sequential({ withdraw(amount) }, { fundsAvail() }) }
}
```

Listing 10. The mapping of the *CreditCardConfirm* class.

```
@Mixin(CreaditCard)
public class CreditCardCount extends CreditCard {

    private int withdrawals
    public int getWithdrawals() { withdrawals }
    @Transactional @CheckConstraintsAfter
    public void setWithdrawals(int value)
    { if (value >= 0) withdrawals = value }

    def oldWithdraw

    @Transactional
    public CreditCardCount(){
        oldWithdraw = metaClass.getMetaMethod('withdraw', [Integer] Class[])
        metaClass.withdraw = { int amount -> newWithdraw(amount) }

        CreditCard_Constructor()
        CreditCardCount_Constructor()
    }

    @CheckConstraintsAfter
    protected void CreditCardCount_Constructor(){ setWithdrawals(0) }

    @Transactional
    @CheckConstraintsAround( preCondition = { status == Status.valid })
    public void incrementCount(){ setWithdrawals(withdrawals + 1) }

    @Transactional
    private newWithdraw(int amount){
        conjunction({ oldWithdraw(amount) }, { incrementCount() })
    }
}
```

Listing 11. The mapping of the *CreditCardCount* class.

```
@Mixin(CreditCardConfirm, CreditCardCount)
public class CreditCardConfirmAndCount extends CreditCard {

    @Transactional
    public WithdrawConfirmAndCount(){
        CreditCard_Constructor()
        CreditCardConfirm_Constructor()
        CreditCardCount_Constructor()
    }
}
```

Listing 12. The mapping of the *CreditCardConfirmAndCount* class.

In addition to using a JVM-based language as the destination language for mapping, the following notions and constructs, which have not been covered by previous methods, have been addressed in this paper as follows:

1. Multiple inheritance is analyzed and supported by presenting mapping rules for three different possible case;
2. By limiting the expected range of the return value, it is possible to map inverse specifications of functions (i.e., specifications of inputs in terms of outputs);
3. It is possible to map generic functions by the dynamism feature of Groovy;
4. Although mapping rules for free-type constructors need the user's intervention, such rules are also presented in this work.

In addition, a mapping rule for the class union was provided, which has not been covered by previous JVM-based researches. In addition, unlike previous works, parallel compositions were mapped to a parallel construct (not a concurrent structure).

Finally, the control of constraints was assigned to advisors in the mapping rules. Consequently, using the aspect-oriented model, the readability of the result code was improved with annotations; in this way, the costs of future developments of the result code decreased.

We provided discussion on soundness of the proposed mapping rules and studied the applicability of the proposed mapping rules through a case study; however, a formal correctness proof of the rules will be presented in future work. As another limitation, this paper does not provide mapping rules for all the constraints existing in various constructs (including pre- and post-conditions of specified operations). In addition, the expressions existing in various constructs of the initial Object-Z specification are not covered in this paper.

The mapping approach proposed in this paper is also applicable to other similar destination programming languages such as Scala. However, due to the differences in the structural characteristics of each programming language, the mapping rules will be different in detail.

This work points out the following directions for future research:

1. A special tool or library could be developed to support the process of mapping from Object-Z to Groovy;
2. By using specific conventions, it is possible to specify the role of each class. For example, a class may be used as DTO (data transfer object) or a controller in the MVC (model - view - controller) design pattern. In such cases, there is no need for interaction with the user, and the descriptor is able to present all the required information using the agreed conventions. Therefore, by using Object-Z specifications, it is possible to develop even large-scale software systems based on the frameworks such as grails or Play. These powerful MVC frameworks, which use Groovy and Scala programming languages, apply the convention over configuration pattern.

References

1. Woodcock, J., Larsen, PG., Bicarregui, J., et al. "Formal methods: Practice and experience", *ACM Computing Surveys (CSUR)*, **41**(4), p. 19 (2009).
2. Fitzgerald, J., Bicarregui, J., Larsen, P.G., et al. "Industrial Deployment of Formal Methods: Trends and Challenges", In *Industrial Deployment of System Engineering Methods*, pp. 123-143, Springer (2013).
3. Najafi, M., Haghighi, H., and Zohdi Nasab, T. "A survey on formal, object-oriented program development approaches", *Scientia Iranica*, **22**(3), pp. 1001-1007 (2015).
4. Smith, G., *The Object-Z Specification Language*, Springer Science & Business Media (2012).
5. Rose, G., *Formal Object-Oriented Specification Using Object-Z*, Macmillan, United Kingdom (2000).
6. Woodcock, J. and Davies, J., *Using Z: Specification, Refinement and Proof*, Prentice Hall, NJ (1996).
7. Ramkarthik, S. and Zhang, C. "Generating Java skeletal code with design contracts from specifications in a subset of object-Z", *Computer and Information Science, 2006 and 2006 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse. ICIS-COMSAR 2006. 5th IEEE/ACIS International Conference on*, Honolulu, HI, pp. 405-411, IEEE Computer Society Press (2006).
8. Najafi, M. and Haghighi, H. "An animation approach to develop c++ code from object-z specifications", *Computer Science and Software Engineering (CSSE), 2011 CSI International Symposium on*, pp. 9-16, IEEE (2011).
9. Najafi, M. and Haghighi, H. "An approach to animate Object-Z specifications using C++", *Scientia Iranica*, **19**(6), pp. 1699-1721 (2012).
10. Najafi, M. and Haghighi, H. "An approach to develop C++ code from object-Z specifications", *2nd World Conference on Information Technology* (2011).
11. Ni, X. and Zhang, C. "Converting specifications in a subset of Object-Z to skeletal spec# code for both static and dynamic analysis", *Journal of Object Technology*, **7**(8), pp. 165-185 (2008).

12. Najafi, M. and Haghighi, H. "A formal mapping from Object-Z specification to C++ code", *Scientia Iranica. Transaction D, Computer Science & Engineering, Electrical Engineering*, **20**(6), p. 1953 (2013).

13. Rafsanjani, G.B. and Colwill, S.J. "From Object-Z to C++: A structural mapping", *Z User Workshop, London 1992*, pp. 166-179, Springer (1993).

14. Griffiths, A. "From Object-Z to Eiffel: a rigorous development method", *Technology of Object-Oriented Languages and Systems (TOOLS 18)*, pp. 293-308 (1995).

15. Ramkarthik, S. and Zhang, C. "Generating Java skeletal code with design contracts from specifications in a subset of object Z", *Computer and Information Science, 2006 and 2006 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse. ICIS-COMSAR 2006. 5th IEEE/ACIS International Conference on*, pp. 405-411, IEEE (2006).

16. Johnston, W. and Rose, G. "Guidelines for the manual conversion of Object-Z to C++", *SVRC Technical Report*, pp. 93-14 (1993).

17. Fukagawa, M., Hikita, T. and Yamazaki, H. "A mapping system from Object-Z to C++", *Software Engineering Conference, 1994. Proceedings., 1994 First Asia-Pacific*, pp. 220-228, IEEE (1994).

18. Wang, Z., Xie, M., and Zhao, Y. "Transform mechanisms of object-z based formal specification to Java", *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pp. 1-4, IEEE (2009).

19. Jackson, M. "Developing Ada programs using the Vienna development method (VDM)", *Software: Practice and Experience*, **15**(3), pp. 305-318 (1985).

20. Van Katwijk, J., Durr, E., and Goldsack, S. "Hybrid object-oriented real-time software development with VDM++", *Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on*, pp. 17-26, IEEE (1997).

21. Albalooshi, F. and Long, F. "Multiple view environment supporting VDM and Ada", *IEE Proceedings-Software*, **146**(4), pp. 203-219 (1999).

22. Moulding, M. and Newton, A. "Rapid prototyping from VDM specifications using Ada", *Automating Formal Methods for Computer Assisted Prototying, IEE Colloquium on*, pp. 11-21, IET (1992).

23. Chedgey, C., Kearney, S., and Kugler, H.-J. "Using VDM in an object-oriented development method for Ada software", *VDM'87 VDM-A Formal Method at Work*, pp. 63-76 (1987).

24. O'Neill, D. "VDM development with Ada as the target language", *VDM'88 VDM-The Way Ahead*, pp. 116-123 (1988).

25. Lou, Y. "VDM/C++: a design and implementation framework", Thesis, Concordia University (1994).

26. Liu, H. and Zhu, B. "Refactoring formal specifications in object-Z", *Computer Science and Software Engineering, 2008 International Conference on*, pp. 342-345, IEEE (2008).

27. McComb, T. "Refactoring object-z specifications", *FASE*, Berlin, Germany, pp. 69-83, Springer (2004).

28. McComb, T. and Smith, G. "Architectural design in Object-Z", *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pp. 77-86, IEEE (2004).

29. McComb, T. and Smith, G. "Compositional class refinement in Object-Z", *FM 2006: Formal Methods*, pp. 205-220 (2006).

30. McComb, T. and Smith, G. "A minimal set of refactoring rules for Object-Z", *Lecture Notes in Computer Science*, **5051**, pp. 170-184 (2008).

31. McComb, T. and Smith, G. "Introducing objects through refinement", *FM 2008: Formal Methods*, pp. 358-373 (2008).

32. Rasoolzadegan, A. and Barforoush, A.A. "A new approach to software development process with formal modeling of behavior based on visualization", *The Sixth International Conference on Software Engineering Advances* (2011).

33. Rasoolzadegan, A. and Barforoush, A.A. "Reliable yet flexible software through formal model transformation (rule definition)", *Knowledge and Information Systems*, **40**(1), pp. 79-126 (2014).

34. Ruhroth, T. "Refactoring Object-Z specifications", *18th Nordic Workshop on Programming Theory* (2006).

35. Smith, G. "Introducing reference semantics via refinement", *ICFEM*, pp. 588-599, Springer (2002).

36. Goldsack, S. and Lano, K. "Annealing and data decomposition in VDM", *ACM Sigplan Notices*, **31**(4), pp. 32-38 (1996).

37. Lu, J. "Introducing data decomposition into VDM for tractable development of programs", *ACM Sigplan Notices*, **30**(9), pp. 41-50 (1995).

38. Goldsack, S., Durr, E., and Plat, N. "Object reification in VDM++", *ICSE-17: Workshop on Formal Methods Application in Software Engineering Practice*, pp. 194-201 (1995).

39. Lano, K. and Goldsack, S. "Refinement of distributed object systems", *Proc. of Workshop on Formal Methods for Open Object-Based Distributed Systems*, Boston, MA, pp. 99-114, Springer (1997).

40. Lano, K. and Goldsack, S. "Refinement, subtyping and subclassing in VDM++", *Theory and Formal Methods*, **95**, pp. 1-446 (1994).

41. Butler, M., Abrial, J.R., Damchoom, K., et al. "Applying Event-B and Rodin to the filestore", *VSRNet, ABZ*, **152**, pp. 134-152 (2008).

42. Poppleton, M. "The composition of Event-B models", *ABZ*, **5238**, pp. 209-222 (2008).

43. Silva, R., Pascal, C., Hoang, T.S., et al. "Decomposition tool for event-B", *Software: Practice and Experience*, **41**(2), pp. 199-208 (2011).

44. Hoang, T.S. and Abrial, J.-R. "Event-B decomposition for parallel programs", *International Conference on Abstract State Machines, Alloy, B and Z*, Berlin, Germany, pp. 319-333, Springer (2010).
45. Hoang, T.S. and Abrial, J.-R. "Event-B development of the FindP program", *Technical Report/* Swiss Federal Institute of Technology Zurich, Department of Computer Science, **653**, pp. 1-22 (2009).
46. Pascal, C. and Silva, R. "Event-B model decomposition", DEPLOY Plenary Technical Workshop (2009).
47. Butler, M. "Incremental design of distributed systems with Event-B", *Engineering Methods and Tools for Software Safety and Security*, **22**, p. 131 (2009).
48. Said, M., Butler, M., and Snook, C. "Language and tool support for class and state machine refinement in UML-B", *FM 2009: Formal Methods*, pp. 579-595 (2009).
49. Said, M.Y. "Methodology of refinement and decomposition in UML-B", Thesis, University of Southampton (2010).
50. Abrial, J.-R., Cansell, D., and Méry, D. "Refinement and reachability in event_b", *ZB 2005: Formal Specification and Development in Z and B*, pp. 129-148, Springer (2005).
51. Abrial, J.-R. and Hallerstede, S. "Refinement, decomposition, and instantiation of discrete models: Application to Event-B", *Fundamenta Informaticae*, **77**(1-2), pp. 1-28 (2007).
52. Hallerstede, S., Leuschel, M., and Plagge, D. "Refinement-animation for event-B-towards a method of validation", *ASM*, pp. 287-301, Springer (2010).
53. Jones, C. "RODIN deliverable D19 intermediate report on methodology", *Technical Report Series-* University OF Newcastle upon Tyne Computing Science, **990** (2006).
54. Lano, K. and Haughton, H. "Specification in B: An introduction using the B toolkit", *World Scientific* (1996).
55. Silva, R. and Butler, M., *Supporting Reuse Mechanisms for Developments in Event-B: Composition* (2009).
56. Silva, R. and Butler, M. "Supporting reuse of Event-B developments through generic instantiation", *International Conference on Formal Engineering Methods*, pp. 466-484, Springer (2009).
57. Abrial, J.-R. "A system development process with Event-B and the Rodin platform", *International Conference on formal engineering methods*, pp. 1-3, Springer (2007).
58. Stevens, B. "Implementing Object-Z with perfect developer", *Journal of Object Technology*, **5**(2), pp. 189-202 (2006).
59. Qin, S. and He, G. "Linking object-z with spec", *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pp. 185-196, IEEE (2007).
60. Kimber, T.G. "Object-Z to perfect developer", Thesis, Department of Computing, Imperial College London (2007).
61. Derrick, J. and Boiten, E. "Refinement in Z and Object-Z. formal approaches to computing and information technology", *Formal Approaches to Computing and Information Technology (FACIT)*, Springer-Verlag, **21**(22), p. 134 (2001).
62. Derrick, J. and Boiten, E. "Refinement of objects and operations in Object-Z", In *Formal Methods for Open Object-Based Distributed Systems IV*, pp. 257-277, Springer (2000).
63. Najafi, M. and Haghighi, H. "Refinement of Object-Z specifications using Morgan's refinement calculus", *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, **5**(11), pp. 1347-1356 (2011).
64. Méry, D. and Singh, N.K. "Automatic code generation from Event-B models", *Proceedings of the Second Symposium on Information and Communication Technology*, Hanoi, Vietnam, pp. 179-188, ACM (2011).
65. Méry, D. and Singh, N.K. "EB2J: Code generation from Event-B to Java", *14th Brazilian Symposium on Formal Methods: Short Papers*, p. 7 (2011).
66. Edmunds, A., Rezazadeh, A., and Butler, M. "Formal modelling for ada implementations: tasking Event-B", *Reliable Software Technologies-Ada-Europe 2012*, pp. 119-132 (2012).
67. Edmunds, A. and Butler, M. "Linking Event-B and concurrent object-oriented programs", *Electronic Notes in Theoretical Computer Science*, **214**, pp. 159-182 (2008).
68. Edmunds, A. "Providing concurrent implementations for Event-B developments", Thesis, University of Southampton (2010).
69. Edmunds, A. and Butler, M. "Tasking Event-B: An extension to Event-B for generating concurrent code", PLACES (2011).
70. Edmunds, A. and Butler, M. "Tool support for Event-B code generation", *Workshop on Tool Building in Formal Methods*, Qúebec, Canada, J. Wiley and Sons (2010).
71. Rommel, C. "Using Java to control IoT development costs", Oracle, http://www.oracle.com/us/solutions/internetofthings/iot-development-cost-wp-2872509.pdf (2016).
72. Sateanpattanakul, S. and Walairacht, A. "Jgroovy-an extensible java programming language with groovy", *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*, pp. 1139-1144, IEEE (2010).
73. Kaewkasi, C. and Gurd, J.R. "Groovy AOP: a dynamic AOP system for a JVM-based language", *Proceedings of the 2008 AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies*, p. 3, ACM (2008).

74. You, J., Junquan, L., and Xia, S. "A survey of formal methods using in software development", *Information Science and Control Engineering 2012 (ICISCE 2012), IET International Conference on*, Shenzhen, pp. 1-4 (2012).

75. Wang, Z., Xia, M., and Zhao, Y. "Transform mechanisms of object-Z based formal specification to Java", *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pp. 1-4, IEEE, Computer Society Press (2009).

76. Charatan, Q. and Kans, A., *Formal Software Development from VDM To Java*, Palgrave Macmilan (2003).

77. "The VDM++ to Java code generator", http://www.vdmtools.jp/en/modules/tinyd2/index.php?id=2/javacgmanppa4E.pdf (2013).

78. Rafsanjani, G. and Colwill, S.J. "From Object-Z to C++: A structural mapping", In *Z User Workshop*, London 1992, pp. 166-179, Springer, London, UK (1992).

79. Fukagawa, M., Hikita, T., and Yamazaki, H. "A mapping system from object-Z to C++", *Software Engineering Conference, 1994. Proceedings.*, 1994 First Asia-Pacific, Tokyo, pp. 220–228, IEEE Computer Society Press (1994).

80. Najafi, M. and Haghighi, H. "A formal mapping from Object-Z specication to C++", *Scientia Iranica. Transactions D, Computer Science & Engineering*, **20**(6), pp. 1953 (2013).

81. Kiczales, G., Lamping, J., Mendhekar, A., et al. "Aspect-oriented programming", *European Conference on Object-Oriented Programming*, pp. 220-242, Springer (1997).

82. Klein, D., *Grails: A Quick-Start Guide*, Progmatic Bookshelf (2009).

83. Bächle, M. and Kirchberg, P. "Ruby on rails", *IEEE Software*, **24**(6), pp. 105-108 (2007).

84. Koenig, D., King, P., Laforge, G., et al., *Groovy in Action*, Manning (2007).

85. Barclay, K. and Savage, J., *Groovy Programming: An Introduction for Java Developers*, Morgan Kaufman (2010).

86. Fowler, M. "Inversion of control containers and the dependency injection pattern", http://martinfowler.com/articles/injection.html (2004).

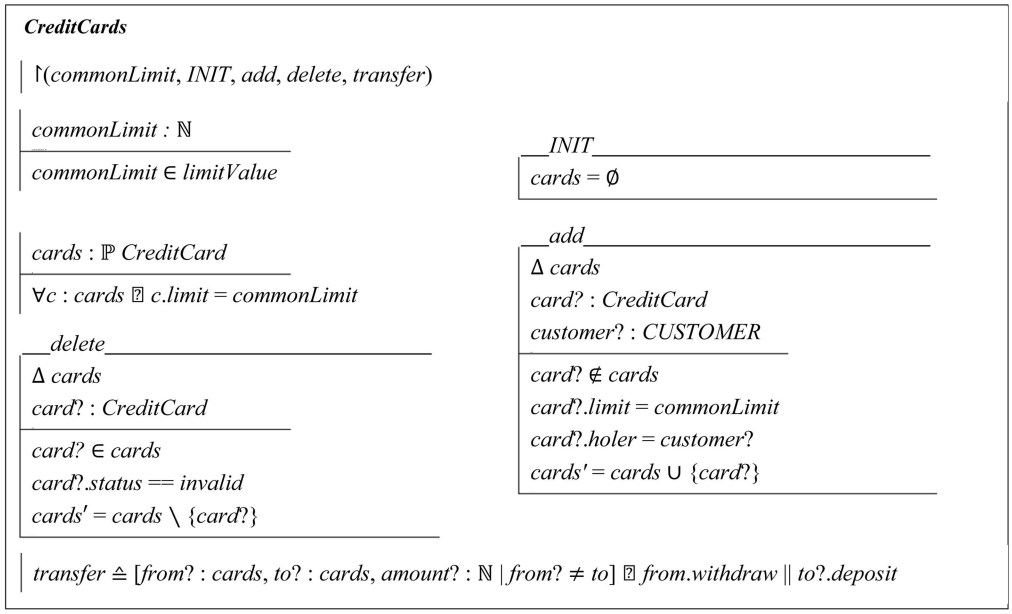
87. Kimbler, G.T., *Object-Z to Perfect Developer*, Imperial College, London (2007).

88. Adamovich, A. and Fiandesio, L. "Implementing multiple inheritance in Groovy", In *Groovy 2 Cookbook*, A. Adamovich and L. Fiandesio, Eds., Packt Publishing Ltd. (2013).

89. Adamovich, A. and Fiandesio, L. "Concurrent programming in Groovy", In *Groovy 2 Cookbook*, A. Adamovich and L. Fiandecio, Eds., Packt Publishing Ltd. (2013).

Appendix A

In the *CreditCards* class (Listing A.1), the *commonLimit* constant defines the limitation shared by all credit cards. The *cards* set contains the defined credit cards. The *add* operation schema defines a new card for a customer. The *delete* operation schema discards an invalid credit card. The *transfer* operation schema transfers a certain amount of money from one card to another.



Listing A.1. *CreditCards* specification.

```
public class CreditCards extends Schema {

    private int commonLimit
    public int getCommonLimit(){ commonLimit }
    public void setCommonLimit(int value) {
        if (value >= 0 && Context.limitValue.contains(value)) limit = value
    }

    private Set<CreditCard> cards
    public Set<CreditCard> getCards() { cards }
    @Transactional
    @CheckConstraintsAfter
    public void setCards(Set<CreditCard> value) { cards = value }

    @Override
    protected boolean  checkConstraintsInternal() {
        !cards.any { c.limit != commonLimit }
    }

    @Transactional
    public CreditCards() {
        CreditCards_Constructor()
    }

    @CheckConstraintsAfter
    protected CreditCards_Constructor() {
        setCards(new Set<CreditCard>())
    }

    @Transactional @CheckConstraintsAround( preCondition = {
            !cards.contains(card) && card.limit == commonList && card.owner == customer })
    public void add(CreditCard card, Customer customer) { cards.add(card) }

    @Transactional @CheckConstraintsAround( preCondition = {
            cards.contains(card) && card.status = Status.invalid })
    public void delete(CreditCard card) { Cards.remove(card) }

    @Transactional @CheckConstraintsAround( preCondition = {
            cards.contains(from) && cards.contains(to) && from != to && amount >= 0 })
    public void transfer (CreditCard from, CreditCard to) {
        parallel({ from.withdraw(amount) }, { to.deposit(amount) })
    }
}
```

Listing B.1. The mapping of the *CreditCards* class.

Appendix B

Listing B.1 shows implementation of the *CreditCards* class based on the *CreditCard* class and its sub-classes. In this class, the *cards* set is of the *Set* type and accepts elements of the *CreditCard* type. Similar to other classes mapped in Section 5, the properties and functions of this class are easily implemented based on the proposed mapping rules.

Biographies

Farzin Zaker received his MSc degree in Computer Engineering-Software from Shahid Beheshti University, Tehran, Iran in 2013 and is currently a PhD student in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include formal methods in the software development life cycle and distributed autonomic systems.

Hassan Haghighi received his PhD degree in Computer Engineering-Software from Sharif University of Technology, Iran in 2009 and is currently an Associate Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include formal methods in the software development life cycle, software testing, and software architecture.

Eslam Nazemi received his PhD degree in Industrial Engineering and Information Technology in 2005 and is currently an Associate Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include self-* software engineering, large scale software development and self- adaptive software quality.