Scientia Iranica D (2013) 20(6), 1953–1977

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
www.scientiairanica.com

A formal mapping from Object-Z specification to C++ code

M. Najafi and H. Haghighi*

Faculty of Electrical and Computer Engineering, Shahid Beheshti University G.C., Tehran, P.O. Box 1983963113, Iran.

Received 13 May 2012; received in revised form 12 December 2012; accepted 25 June 2013

KEYWORDS
Formal program development;
Object-oriented programming;
Animation;
Object-Z;
C++.

Abstract. Object-Z is an extension of Z which provides specific constructs to facilitate specification in an object-oriented style. A number of contributions have been made so far to animate Object-Z with various object-oriented programming languages. However, none of the existing animation methods present their mapping rules formally. Also, none of these animation methods prove the correctness of their mapping rules. In our previous work, we informally presented an animation method to map Object-Z specifications into C++ code. In this paper, we propose a formal mapping from Object-Z specifications to C++ code. We also prove the correctness of the given mapping rules.

1. Introduction

Object-orientation is a popular approach which is applied to formal methods in order to express encapsulation and reuse concepts in formal specifications [1]. Object-Z [2,3] is an extension of Z [4] providing facility to define large and complex software as a collection of independent classes [5]. There are a number of approaches in the literature for developing object-oriented programs from Object-Z specifications.

Rafsanjani and Colwill [6] presented a method which maps Object-Z specifications to C++ code structurally; however, this method does not consider mapping of some constructs of Object-Z, such as precondition, postcondition, class invariants, visibility list, operation operators, object containment and some types of definitions such as class variables and generic parameters. In [7], Fukagawa et al. built upon the work of [6] to propose an approach for mapping Object-Z specifications to C++ code; indeed, they added two new rules to the work of [6] that consider constructor for types of constants and template class for generic parameters.

Johnston and Rose [8] presented another method which animates Object-Z specifications using C++. Although this method covers mapping of precondition, postcondition, class invariants, visibility list, some types of definitions like free types and class variables, it does not consider mapping of axiomatic definitions and multiple inheritance. In addition, the proposed mapping rules for some constructs are rather general; examples are visibility list and state schema.

Although some other works exist in the literature which animate Object-Z specifications by other object-oriented programming languages, such as Java [5,9], Eiffel [10] and Spec# [11], there are limitations in all of these works: none of them propose their mapping rules formally and prove the correctness of their mapping rules.

In our previous work [12-14], we presented an animation approach to develop C++ code from Object-Z specifications that covers the mapping of some Object-Z structures whose mappings are never been considered in the mentioned work. For example, we introduced animation rules for most of the global paragraph constructs, all types of definitions like class

*. *Corresponding author. Tel.: +982129904190;*
E-mail addresses: M.Najafi@mail.sbu.ac.ir (M. Najafi), h_haghighi@sbu.ac.ir (H. Haghighi)

union, object aggregation, object containment, all of the local definitions and operation operators. Also, animation rules were given with much more details facilitating automation capability.

In this paper, we present a new version of our animation approach to develop C++ code from Object-Z specifications which have the following advances in comparison to our previous work [12-14]:

1. We present our previous mapping rules [12-14] formally. In this way, the precision of presented rules is increased. In addition, the number of various cases, when mapping some constructs, is decreased.
2. We prove the correctness of our mapping rules; a contribution which has not been aimed with any of the existing animation methods.
3. We present translation of some constructs, such as local abbreviation definition (when its left-hand side is a variable name, and its right-hand side is in the form of class union or in the form of a set of sets of numbers) which have not been covered in our previous work [12-14].

The paper is organized in the following way: Section 2 presents formal mapping from Object-Z to C++. In Section 3, we prove the correctness of our method. In Section 4, we give a case study. In Section 5, we compare our method with other related methods based on a set of defined criteria. Finally, Section 6 concludes the paper.

It is worth mentioning that due to the space limitation, we only present the mapping of global paragraphs and class paragraphs except the visibility list. The formal mapping of other constructs, such as visibility list, definitions and promotion can be obtained easily. Also, we do not review related work in this paper; interested readers can refer to [14] for a complete description of related work. As the preliminaries of this paper, the readers should be familiar with Object-Z [2], C++ [15] and also our previous work [14].

2. Formal mapping of Object-Z to C++

Based on our previous work [12-14], we interact with the user in order to obtain the mapping of predicates, check whether operators have the same semantics, and increase the flexibility of our method by enabling the user to select one of the alternative mappings according to his or her preference; see the mapping of scope enrichment as an example. It is worth mentioning that some other work, such as [16,17], considers interaction with the user to increase the flexibility of the mapping method regarding user's opinion in code generation process. For modeling the user interaction, we introduce a new basic type UserDesc and a new free type UserInteraction_Description as follows:

[UserDesc]
UserInteraction_Description ::= NULL | Input <<Seq UserDesc>>

Value "NULL" allows us to show that no interaction with the user is performed. The constructor "Input" allows us to get an input from the user via its expression (i.e., Seq UserDesc).

Now we define a translation function $\mathcal{K}$ which translates each part of the Object-Z abstract syntax (see Appendix A for complete account of the abstract syntax of Object-Z) into its counterpart in C++ (using user interaction, if needed).

Definition 1. Translation Function $\mathcal{K}$:
$\mathcal{K}$: **Object-Z** $\times$ **UserInteraction_Description** $\rightarrow$ **C++** is a function from Object-Z $\times$ UserInteraction_Description to C++ which will be explained in detail in Subsections 2.1, 2.2, 2.3 and 2.4. Before beginning these subsections, it should be noted that:

1. When applying translation function $\mathcal{K}$, we assume that UserInteraction_Description has default value "NULL".
2. We consider each construct in Object-Z abstract syntax as a type throughout the paper. Also, we assume that each Object-Z construct, such as PredicateList and Declaration, is a set of elements; hence, we use set notations, such as membership throughout the paper.
3. We consider a set of functions (as utilities) to present translation function $\mathcal{K}$. These functions are illustrated in Table 1.
4. We define a set of predicates in order to present translation function $\mathcal{K}$. These predicates are illustrated in Table 2.

2.1. Mapping of specification

We did not present any mapping for specification in our previous work [12-14]. The translation of a specification is equivalent to translation of each of its paragraphs to their counterparts in C++ (user interaction may be needed to obtain the mapping of paragraphs in specification); hence, the translation of specification is as follows:

$\mathcal{K}$ (Specification, user_interaction) = $\mathcal{K}$ (ParagraphList, user_interaction) = $\forall$ p: ParagraphList $\bullet$ $\mathcal{K}$ (p, user_interaction)

2.2. Mapping of global paragraphs

Now we present the translation of each global paragraph construct as follows. Note that the scope of

Table 1. Utility functions used for formal mapping.

Function	Description
GetParam (VariableName v, Expression e)	Returns the parameters in 'e' needed to compute the value of 'v'.
GetParam (Identifier i, PredicateList p)	Returns the parameters in 'p' needed to obtain the value of 'i'.
GetComExp (Identifier i, PredicateList p)	Returns predicates of 'p' related to computing the value of 'i'.
GetAbElemType (Expression e)	We use this operation in the translation of abbreviation definition when its left-hand side is a variable name, and its right-hand side is a set definition along with a list of its elements. This operation returns the type of elements in 'e' (i.e., numeric or not_numeric).
GetSetKind (Expression e)	We use this operation in the translation of abbreviation definition when its left-hand side is a variable name, and its right-hand side is a set definition along with a list of its elements. This operation checks whether 'e' is in the form of flat set definition or non-flat set definition.
GetCardVar (Expression e, N i)	We use this operation in the translation of abbreviation definition when its left-hand side is a variable name, and its right-hand side is a set definition along with a list of its elements. This operation returns the number of elements defined in the 'i'th (i may be 0 or 1) level of sets in 'e'.
GetUniqueC++Name ()	Returns a unique name which is correct according to the C++ syntax.
GetRange (Expression e)	Returns the range value defined in 'e'.
GetPredExceptInitializations (PredicateList p, Declaration d)	Returns the predicates of 'p' except those used for initializing variables of 'd'.
GetSuperClass (ClassName c)	Returns super classes of class 'c'.
GetOpParam (ClassName c, OperationName op)	Returns the parameters of operation 'op' of class 'c'. We assume that the type parameter, whose array form is the output of this operation, has been already defined in C++.
GetOpSignature (ClassName c, OperationName op)	Returns the signature of operation 'op' of class 'c'. We assume that the type OpSignature, which is the output of this operation, has been already defined in C++.
GetItemType (ClassName c, Name n)	Checks if name 'n' that is defined in class 'c' is an attribute or a method; the result is as attribute or method.
GetClass (OperationExpression op)	Returns the class where the operation existing in 'op' is defined. We assume 'op' is in the form of operation promotion.
GetPre (ClassName c, OperationName op)	Returns the mappings of the first "if-conditions" in the body of operation 'op' of class 'c' without check_stateschema () (see this function later).
GetPost (ClassName c, OperationName op)	Returns the mappings of predicateList of operation 'op' of class 'c'.
Merge (Parameter[] p_1, Parameter[] p_2, operator o)	Merges the parameters 'p_1' and 'p_2' according to the semantics of operator 'o'.

Table 2. Predicates used for formal mapping.

Predicate	Meaning
OperatorDefinition (AxiomaticDefinition a)	Checks whether 'a' is in the form of operator definition.
OperatorIsInC++ (AxiomaticDefinition a)	Checks whether the operator, which is defined in 'a', already exists in the set of C++ operators. In other words, OperatorIsInC++ (a) is equivalent to OperatorDefinition (a) $\wedge$ a.Declaration.Identifier $\in$ **C++Op** **C++Op** is a set whose elements are all operators in C++ language.
SameSemanticOperatorInC++ (AxiomaticDefinition a)	Checks whether the operator, which is defined in 'a' and already exists in the set of C++ operators, has the same semantics as that of the existing operator in C++; we interact with the user in order to check whether operators have the same semantics or not. OperatorDefinition (a) $\wedge$ a.Declaration.Identifier $\in$ **C++Op** $\wedge$ user_interaction is 'the operator has the same semantics as that of the existing operator in C++'.
ExistsInClassUnion (ClassName c, Expression e)	We use this operation in the translation of abbreviation definition when its left-hand side is a variable name, and its right-hand side is in the form of class union. This operation checks whether class 'c' exists in class union expression 'e'.
ExistsOpInClass (ClassName c, OperationName op)	Checks whether the operation 'op' is defined in class 'c', i.e. op $\in$ c.OperationList
Noattrenamed (ClassName c, OperationName o)	None of the attributes of class 'c' which are used in operation 'o' are renamed.
CallOrpromotion (OperationExpression op)	Operation expression 'op' is in the form of operation call or operation promotion.
Comdirec (OperationExpression op_1, OperationExpression op_2)	Communication direction is from 'op_1' to 'op_2'. We assume that 'op_1' and 'op_2' are either in the form of operation call or operation promotion.
Globdef (AxiomaticDefinition ad)	'ad' is defined globally in the specification.
Statevariableschema (Schema s)	At least one state variable with type 's' is defined in Class*; we will define Class* later.

mappings of global paragraph constructs (except the mapping of axiomatic definition in the form of operator definition and generic definition in the form of operator definition) are the whole C++ code obtained from the mapping of the specification (i.e., global); see [14] for more details.

Basic type definition

According to our previous work [13,14], we map each basic type to a struct as follows:

$\mathcal{K}$ (BasicTypeDefinition) = $\mathcal{K}$ ([IdentifierList]) = $\forall$ i: Identifier $|i \in$ IdentifierList $\bullet$ ($\mathcal{K}(i)$ = struct i {};) $\square$

Axiomatic definition

Based on the form of an axiomatic definition (constant definition, operator definition, symbol (not operator) definition and function definition), we considered four cases for its mapping in [12,14]. We use concept "class union" in order to model these four cases. More precisely, for each form of the axiomatic definition, we introduce a new class whose instances are axiomatic definitions in the form of case description. For instance, we consider a new class whose name is "ADConstant-Definition" to model the axiomatic definition in the form of constant definition; in this way, four cases of axiomatic definition are modeled formally as follows:

AxiomaticDefinition = = ADConstantDefinition $\cup$ ADOperatorDefinition $\cup$ ADSymbolDefinition $\cup$ AD-FunctionDefinition

Also, we assume that the Declaration part of the axiomatic definition is in the form of "Identifier: Expression". Now we present the formal translation of AD: AxiomaticDefinition using the mapping rules given in our previous work [12,14]; of course, there is an exeption: when the operator already exists in C++, and its semantics is not the same as what exists in C++, we map "AD" to a global method

whose return type is Boolean; this method checks satisfaction of the semantics of the defined operator. In the following, "NM" means that "no mapping is needed". "PredicateListMapping" indicates the mapping of PredicateList part of "AD" obtained via user interaction. Also, "OZIdentifier" means that word "OZ" is concatenated with "Identifier" to build a new name. Similarly, "opIdentifier" denotes that word "op" is concatenated with "Identifier".

((AD ∈ ADConstantDefinition) ⇒ (𝒦 (AD,
PredicateListMapping) =
𝒦 ([Identifier: Expression ≪|PredicateList≫],
PredicateListMapping) = (const 𝒦 (Identifier:
Expression)

Boolean check_Identifier (){
if(!𝒦 (PredicateList, PredicateListMapping))
return false;
return true;})))

∨
(((AD ∈ ADOperatorDefinition) ∧ **OperatorIsInC++** (AD) ∧ **SameSemanticOperatorInC++** (AD)) ⇒ (𝒦 (AD) = NM))
∨
(((AD ∈ ADOperatorDefinition) ∧ **OperatorIsInC++** (AD) ∧ **!SameSemanticOperatorInC++** (AD)) ⇒ (𝒦 (AD) =
Boolean OZIdentifier (𝒦 (Expression)){
if (𝒦 (PredicateList, PredicateListMapping))
return true;
return false;}))
∨
(((AD ∈ ADOperatorDefinition) ∧ !**OperatorIsInC++** (AD)) ⇒
((𝒦 (AD, ' the left-most operand of the operator function is an object of the class of that operator '⌢ PredicateListMapping) =
𝒦 ([Identifier: Expression ≪|PredicateList≫], ' the left-most operand of the operator function is an object of the class of that operator ' ⌢ PredicateListMapping) =
(virtual Boolean opIdentifier (ClassName c, 𝒦 (Expression)){
if (𝒦 (PredicateList, PredicateListMapping))
return true;
return false;}))
∨
(𝒦 (AD, ' the left-most operand of the operator function is an object of a class different from the class of that operator ' ⌢ PredicateListMapping) =
𝒦 ([Identifier: Expression ≪|PredicateList ≫], '
the left-most operand of the operator function is an object of a class different from the class of that operator ' ⌢ PredicateListMapping) =
(typedef char [50] ClassName;
Boolean opIdentifier (ClassName c, 𝒦 (Expression)){
if (𝒦 (PredicateList, PredicateListMapping))
return true;
return false;}))))
∨
((AD ∈ ADSymbolDefinition) ⇒ (𝒦 (AD, PredicateMapping) =
𝒦 ([Identifier: Expression ≪|Predicate≫],
PredicateMapping) =
(#define Identifier (**GetParam** (Identifier,
Expression)) 𝒦 (Predicate, PredicateMapping))))
∨
((AD ∈ ADFunctionDefinition) ⇒ (𝒦 (AD, PredicateListMapping)=
𝒦 ([Identifier: Expression ≪|PredicateList≫],
PredicateListMapping) =
(void Identifier (𝒦 (Expression)){𝒦 (PredicateList, PredicateListMapping);})))

In the translation of axiomatic definition in the form of symbol (not operator) definition, we used Predicate instead of PredicateList because we assume this case of axiomatic definition has only one predicate. □

Generic definition

In our previous work [14], we considered two cases for the mapping of generic definition based on whether it is in the form of operator definition or not. Similar to what we did for modeling the four cases of axiomatic definition, we model the mentioned two cases of generic definition formally using the concept of "class union" as follows:

GenericDefinition= = GDOperatorDefinition ∪ GDNOperatorDefinition

Now we present the formal translation of GD: GenericDefinition using the mapping rules given in our previous work [14]. We map a generic definition that is in the form of operator definition and does not have formal parameters similar to the mapping of axiomatic definition which is in the form of operator definition; abbreviation "GPADOD" in the following implies that we use a translation similar to what presented for "global paragraph axiomatic definition in the form of operator definition". Note that we interact with the user in order to obtain the mapping of predicates, such as PredicateList part of GD (i.e., PredicateListMapping) and also preconditions (i.e., PredicateListPrecondition) and postconditions (i.e., PredicateListPostcondition) included in the PredicateList part of GD:

((GD ∈ GDOperatorDefinition) ∧ (GD = [Declaration ≪| PredicateList≫]) ∧ (GD ∈ ADOperatorDefinition))

$\Rightarrow \mathcal{K}$ (GD) = GPADOD
$\vee$
((GD $\in$ GDNOperatorDefinition) $\Rightarrow$
($\mathcal{K}$ ([Declaration $\ll$FormalParameters$\gg$
$\ll$|PredicateList$\gg$], PredicateListMapping $\frown$
DeclarationMapping $\frown$ PredicateListPrecondition
$\frown$ PredicateListPostcondition) =
$\mathcal{K}$ ([Declaration $\ll$[IdentifierList]$\gg$
$\ll$|PredicateList$\gg$], PredicateListMapping $\frown$
DeclarationMapping $\frown$ PredicateListPrecondition
$\frown$ PredicateListPostcondition) =
(template $\langle \forall i$: IdentifierList • class $i, \rangle$
Boolean **GetUniqueC++Name** () ($\mathcal{K}$
(Declaration, DeclarationMapping)){
if ($\mathcal{K}$ (PredicateList, PredicateListMapping $\frown$
PredicateListPrecondition)){
$\mathcal{K}$ (PredicateList, PredicateListMapping $\frown$
PredicateListPostcondition)
return true;}
return false;})))) $\square$

Abbreviation definition

In our previous work [12,14], we considered four cases for the mapping of abbreviation definition. Always, the left-hand side of the abbreviation definition is a variable name, but its right-hand side is in the form of a (an): computational expression, set definition along with a list of its elements, class union and range definition. The formal model of these cases of abbreviation definition is as follows:

AbbreviationDefinition = = $AD_1 \cup AD_2 \cup AD_3 \cup AD_4$

where AD_1, AD_2, AD_3 and AD_4 contain those instances of abbreviations whose right-hand side is in the form of computational expression, set definition along with a list of their elements, class union and range definition, respectively. In all four cases, the left-hand side is a variable name. In order to present the translation of abbreviations, we define new free types whose name are SetKind and AbElemType ("non-flat" refers to a set of ... of numbers):

AbElemType ::= numeric | not_numeric
SetKind ::= flat | non-flat

Now we map AbD: AbbreviationDefinition. To translate an abbreviation definition whose right hand side is a set along with the list of its members, we consider two cases in terms of whether the right hand set is a flat set (of non-number or number members) or a non-flat set. For non-flat sets, each member may be either a number or a set; if a member is set, it can be itself flat or non-flat similar to the above distinction. This story can be continued recursively. In our previous work [14], we only considered the mapping of flat sets. Now, we map a non-flat set to a linked list of records each of which corresponds to a set member. These records are defined by a struct, called Rec_VariableName; see the following obtained code in C++. The first attribute of this record contains a number if the corresponding member is a number; otherwise, it has no value. For a member that is a set itself, p2 points to a linked list obtained from the mapping of that set recursively. In addition, we have another pointer, called p1, to point to the next member of the set being mapped. For example, Figure 1 shows the mapping of $\{\{1,2\},3\}$ to this struct.

Also, we consider two methods "VariableName_Initialization", which recursively maps an Expression in Abbreviation Definition when it is in the form of a non-fat set, and "initialization_VariableName", which initialize VariableName (VariableName is the left hand side of AbD; see below). For simplicity, only the pseudo code of the mapping is given here (for this, many implementation details and required definitions are not given here; for example, we use type "Expression" in the signature of "VariableName_Initialization" without defining it in advance). Note that we interact with the user to initialize the elements of VariableName for obtaining the mapping of the Expression part of AbD when AbD $\in$ AD_1 (i.e., ExpressionMapping).

(((AbD $\in$ AD_1) $\Rightarrow$
($\mathcal{K}$ (AbD, ExpressionMapping) =
$\mathcal{K}$ (VariableName = = Expression,
ExpressionMapping) =
(#define VariableName (**GetParam** (VariableName,
Expression)) $\mathcal{K}$ (Expression, ExpressionMapping))))
$\vee$
((AbD $\in$ AD_2)$\Rightarrow$
((((**GetSetKind** (Expression) = flat) $\wedge$
(**GetAbElemType** (Expression) = numeric))
$\vee$
(**GetSetKind** (Expression) =non-flat)) $\Rightarrow$
($\mathcal{K}$ (AbD) =
$\mathcal{K}$ (VariableName = = Expression) =
(struct Rec_VariableName{
int value;
Rec_VariableName * p1;

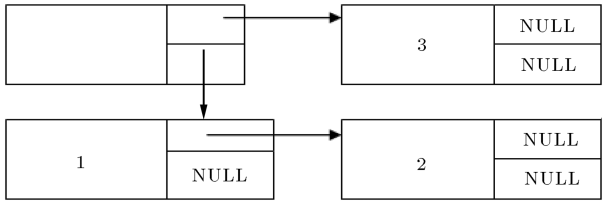

Figure 1. Mapping of $\{\{1,2\},3\}$ to linked list of Rec_VariableName elements.

```
    Rec_VariableName * p2;}VariableName;
  Rec_Variable* VariableName_Initialization (Expression
  s){
    Rec_VariableName * p;
    Rec_VaribaleName * pp;
    p=NULL;
    for each x in s{
      pp=new Rec_VariableName ();
      if (x is number){
            pp->value=x;
            pp->p2=NULL;}
      else{
            pp->p2=VariableName_Initialization (x);}
      pp->p1=p;
      p=pp;}}
  void initialization_VariableName ()
  {VariableName = VariableName_Initialization
  (Expression);}
)))∨
((GetSetKind (Expression) = flat) ∧
(GetAbElemType (Expression) = not_numeric) ⇒
   (K (AbD, ExpressionMapping) =
   K (VariableName = = Expression,
   ExpressionMapping) =
   (enum VariableName {K
   (Expression, ExpressionMapping)})))
   ∨
((AbD ∈ AD3) ⇒ (K (AbD) =
   K (VariableName = = Expression) =
   (struct VariableName { ∀ c: Class |
   ExistsInClassUnion (c, Expression) • c *
   GetUniqueC++Name (); })))
   ∨
((AbD ∈ AD4 ) ⇒ (K (AbD) =
   K (VariableName = = Expression) =
   (int [GetRange (Expression)]
   VariableName;)))□
```

Free type definition

We only present the mapping of free type definition when no constructor is used in its definition as follows (this case is a limitation of our work):

$\mathcal{K}$ (FreeTypeDefinition) = $\mathcal{K}$ (Identifier ::= BranchList)
= enum Identifier { ∀ b: Branch | b ∈ BranchList • □

Notation "|" used in the above mapping is the separator which is defined in enumerator syntax.

Schema

If SchemaHeader is in the form of SchemaName, then the translation of schema is as follows ("PredicateListMapping" indicates the mapping of PredicateList part of Schema obtained via user interaction):

```
K (Schema, PredicateListMapping) =
K (SchemaName ≙ [Declaration
≪|PredicateList≫], PredicateListMapping) =
struct SchemaName { K (Declaration) };
Boolean check_SchemaName (SchemaName s){
   if (K (PredicateList, PredicateListMapping))
   return true;
   return false;}□
```

Class

The translation of class is as follows. Notice that we present the translation of class paragraphs in Subsection 2.3. Also, since we may need user interaction in the translation of class paragraphs, we consider user interaction in translation function here:

```
K (Class, user_interaction) = K (ClassName
≪FormalParameters≫, user_interaction)=
      [≪VisibilityList≫
       ≪InheritedClassList≫
       ≪LocalDefinitionList≫
       ≪State≫
       ≪InitialState≫
       ≪OperationList≫]
K (FormalParameters);
class ClassName: K (InheritedClassListClassName)
      {K (≪LocalDefinitionList≫, user_interaction);
       K (≪State≫, user_interaction);
       K (≪InitialState≫, user_interaction);
       K (≪OperationList≫, user_interaction)
       K (InheritedClassListRenameList)};
```

where $\mathcal{K}$ (InheritedClassListClassName) denotes the mapping of ClassName part of the InheritedClassList (see InheritedClassList abstract syntax in Appendix A). In addition, $\mathcal{K}$ (InheritedClassListRenameList) indicates the mapping of the remained parts of InheritedClassList.

In our previous work [12-14], we proposed templates for constructors and destructors; hence, they must be considered in class definition. Also, it is worth mentioning that the mapping of the VisibilityList is considered implicitly when mapping other constructs; see [14] for details.□

2.3. Mapping of class paragraphs

We present the translation of each class construct step by step as follows. For each class in Object-Z specification, we consider a corresponding class in C++ which is denoted by Class* throughout the paper.

Formal parameters

In our previous work [12,14], we considered a template class for each formal parameter. Thus, the translation of formal parameters is as follows (the mapping of each formal parameter has global scope):

```
K (FormalParameters) = K ([IdentifierList])=
template ⟨ ∀ i: IdentifierList • class i, ⟩ □
```

Local definitions

Basic type definition: Its translation is the same as of the global paragraph basic type definition. However, the result of translation must be considered within Class* definition.

$\mathcal{K}$ (BasicTypeDefinition) = $\mathcal{K}$ ([IdentifierList]) =
$\forall$ i: IdentifierList • ($\mathcal{K}$ (i) = struct i {};) □

Axiomatic definition: Similar to what we did for the mapping of global paragraph axiomatic definition, we consider four cases. However, mappings of all cases except axiomatic definition in the form of operator definition must be considered within Class* definition; see [14] for more details. Now we map AD: AxiomaticDefinition (see Subsection 2.2 Axiomatic Definition). If the axiomatic definition is in the form of a (an):

- Constant definition: In addition to map the PredicateList part of AD in the constructor of Class* [14], the following translation must be considered:

  (AD $\in$ ADConstantDefinition) $\Rightarrow$
  ($\mathcal{K}$ (AD, PredicateListMapping) =
  $\mathcal{K}$ ([Identifier: Expression $\ll$ | PredicateList$\gg$], PredicateListMapping) =
  (const $\mathcal{K}$ (Declaration)
  virtual Boolean check_Identifier (){
  if(!$\mathcal{K}$ (PredicateList, PredicateListMapping))
  return false;
  return true;}))

- Operator definition: The translation of local axiomatic definition in the form of operator definition is the same as that of global paragraph axiomatic definition. In the following, GPADOD abbreviates for "Global Paragraph Axiomatic Definition in the form of Operator Definition".

  (AD $\in$ ADOperatorDefinition)) $\Rightarrow$ ($\mathcal{K}$ (AD) = GPADOD)

- Symbol (not operator) definition: Its translation is the same as that of global paragraph axiomatic definition (when it is in the form of symbol definition); GPADSD abbreviates that translation:

  (AD $\in$ ADSymbolDefinition) $\Rightarrow$ ($\mathcal{K}$ (AD, PredicateMapping) = GPADSD)

- Function definition: The translation of this case of axiomatic definition is as follows:
  (AD $\in$ ADFunctionDefinition) $\Rightarrow$ ($\mathcal{K}$ (AD, PredicateListMapping)=
  $\mathcal{K}$ ([Identifier: Expression [|PredicateList|], PredicateListMapping) =
  (void Identifier ($\mathcal{K}$ (Expression)){$\mathcal{K}$ (PredicateList, PredicateListMapping);}))□

Abbreviation Definition: Similar to what we did for the mapping of global paragraph abbreviation definition, we consider four cases here. However, mappings of all cases must be considered within Class* definition. Now we map AbD: AbbreviationDefinition (see Subsection 2.2 Abbreviation Definition). Suppose that the left-hand side of the abbreviation definition is a variable name. In the following, we provide the mapping of the definition based on the various forms of its right-hand side:

- Computational expression: Its translation is the same as that of global paragraph abbreviation definition when its left-hand side is a variable name, and its right-hand side is in the form of computational expression; GPAbDCE denotes the translation of this case of global paragraph abbreviation definition: (AbD $\in$ AD_1) $\Rightarrow$ ($\mathcal{K}$ (AbD, ExpressionMapping) = GPAbDCE)

- Set definition along with a list of its elements:
  ((AbD $\in$ AD_2)$\Rightarrow$
  ((((**GetSetKind** (Expression) = flat) $\wedge$
  (**GetAbElemType** (Expression) = numeric)) $\vee$
  (**GetSetKind** (Expression) =non-flat)) $\Rightarrow$
  ($\mathcal{K}$ (AbD) = $\mathcal{K}$ (VariableName = = Expression) =
  (struct Rec_VariableName{
  int value;
  Rec_VariableName * p1;
  Rec_VariableName * p2;}VariableName;
  virtual Rec_Variable* VariableName_Initialization (Expression s){
  Rec_VariableName * p;
  Rec_VaribaleName * pp;
  p=NULL;
  for each x in s{
  pp=new Rec_VariableName ();
  if (x is number){
  pp->value=x;
  pp->p2=NULL;}
  else{
  pp->p2=VariableName_Initialization (x);}
  pp->p1=p;
  p=pp;}}
  virtual void initialization_VariableName ()
  {VariableName = VariableName_Initialization (Expression);})))$\vee$
  (((**GetSetKind** (Expression) = flat) $\wedge$
  (**GetAbElemType** (Expression) = not_numeric))
  $\Rightarrow$
  ($\mathcal{K}$ (AbD, ExpressionMapping) =
  $\mathcal{K}$ (VariableName = = Expression,
  ExpressionMapping) =
  (enum VariableName { $\mathcal{K}$ (Expression,

ExpressionMapping)}))))

- Class union: We did not present any mapping for this case of abbreviation definition in our previous work [12-14]. Nevertheless, the translation of this case of abbreviation definition is the same as that of global paragraph abbreviation definition when its left-hand side is a variable name, and its right-hand side is in the form of class union; GPAbDCU abbreviates that translation:
  $(AbD \in AD_3) \Rightarrow (\mathcal{K}\ (AbD) = GPAbDCU)$
- Range definition: Its translation is the same as that of global paragraph abbreviation definition when its left-hand side is a variable name, and its right-hand side is in the form of range definition; GPAbDRD abbreviates that translation:

  $(AbD \in AD_4) \Rightarrow (\mathcal{K}\ (AbD) = GPAbDRD)$ □

Free type definition: Its translation is the same as that of global paragraph free type definition; GPFD abbreviates the translation of global paragraph free type definition when it involves no constructor. Notice that the translation of local free type definition must be considered within Class* definition.
$\mathcal{K}$ (FreeTypeDefinition) = GPFD □

State

We consider the Declaration given before Delta notation in State abstract syntax as "Declaration1" (i.e., the part including primary variables) and the Declaration given after Delta notation as "Declaration2" (i.e., the part including secondary variables). Also, we separate two categories of secondary variables: those secondary variables that are not obtained via primary variables and other constants in PredicateList (i.e., D_{2NP}), and those secondary variables that are obtained via primary variables or other constants in PredicateList (i.e. D_{2P}). To model these two categories formally, we use a "class union" as follows:

$Declaration_2$: $D_{2P} \cup D_{2NP}$

Now, we present the translation of State which may need user interaction as follows. We interact with the user in order to obtain the mapping of PredicateList part of State (i.e., PredicateListMapping), global predicates (i.e., PredicateMapping) and the Declaration part of those generic definitions which are not in the form of operator definition (i.e., DeclarationMapping).

Note that CEL and FAS, which are used to obtain the mapping of secondary variables according to our previous work [14], abbreviate for "Computational Expression Length" (the expression in PredicateList used to compute the value of these variables) and "Frequency of Appearance in the Specification" (the frequency of appearance of these variables in the specification), respectively. Also, $\mathcal{K}$ (Dec) indicates the mappings of definitions; see [14] for details. Notice that the mapping of all parts of State must be considered within Class* definition:

$\mathcal{K}$ ([Declaration$_1$ [Δ Declaration$_2$] [|PredicateList|], CEL $\frown$ FAS $\frown$ PredicateListMapping $\frown$ PredicateMapping $\frown$ DeclarationMapping) =

(∀Dec: Declaration | Dec ∈ Declaration$_1$ • $\mathcal{K}$ (Dec)

(∀Dec: Declaration | Dec ∈ Declaration$_2$ •
((Dec ∈ D_{2np}) ⇒ $\mathcal{K}$ (Dec))
∨
((Dec ∈ D_{2p}) ⇒
($\mathcal{K}$ ([Dec$_{\ll}$| PredicateList$_{\gg}$], CEL $\frown$ FAS $\frown$ PredicateListMapping) =
$\mathcal{K}$ ([Identifier: Expression $_{\ll}$|PredicateList$_{\gg}$], CEL $\frown$ FAS $\frown$ PredicateList Mapping) =
(((CEL=NotLong) ∧ (FAS= High)) ⇒
(#define Identifier (**GetParam** (Identifier, PredicateList))
$\mathcal{K}$ (**GetCompExp** (Identifier, PredicateList), PredicateListMapping)))
∨
((FAS= Low) ⇒ ($\mathcal{K}$ (Dec);
virtual inline void set_Identifier (){
$\mathcal{K}$(**GetCompExp** (Dec, PredicateList), PredicateListMapping);}))
∨
(((CEL=Long) ∧ (FAS=High)) ⇒ ($\mathcal{K}$ (Dec);
virtual void set_Identifier (){
$\mathcal{K}$(**GetCompExp** (Dec, PredicateList), PredicateListMapping);})))

```
virtual Boolean check_stateschema (){
    if ((∀ P: Predicate • K (P,
    PredicateMapping)) && (∀ AD:
    AxiomaticDefinition | Globdef (AD) •
    check_axiomaticdefinition_name ()) &&
    K (GetPredExceptInitializations
    (PredicateList, Declaration2),
    PredicateListMapping) && GetSuperClass
    (Class*)::check_stateschema ()
    &&(∀ S:Schema |Statevariableschema (S) •
    check_SchemaName (S.SchemaName s)) &&
    (∀ GD: GenericDefinition | GD ∈
    GDNOperatorDefinition •
    genericdefinition_name (K (GD.Declaration,
    DeclarationMapping)))(
        return true;
      return false;}) □
```

Initial state

The translation of Initial State using our mapping rules

in [14] is as follows. We interact with the user in order to obtain the mapping of the PredicateList part of INIT (i.e., PredicateListMapping). We also apply PredicateList in constructor of Class*. In addition, method INIT must be considered within Class* definition:

$\mathcal{K}$ (INIT$\stackrel{\wedge}{=}$[PredicateList], PredicateListMapping) =
virtual Boolean INIT(){
return (check_stateschema () &&
$\mathcal{K}$ (PredicateList, Predicate
ListMapping) && **GetSuperClass**
(Class*)::INIT ()) } □

Operation schema

According to our previous work [12-14], we map operation schema to a virtual method whose name is the operation schema name itself as follows. Notice that we interact with the user in order to obtain the mapping of PredicateList part of operation schema (i.e., PredicateListMapping). The mapping of the operation schema must be considered within Class* definition:

((! **ExistsOpInClass** (Class*, OperationName)) ⇒
($\mathcal{K}$ (OperationName $\stackrel{\wedge}{=}$ [DeltaList ≪Declaration≫ ≪| PredicateList≫], PredicateListMapping) =
(virtual Boolean OperationName ($\mathcal{K}$ (Declaration), **GetOpParam** (**GetSuperClass** (Class*), OperationName)){
if (check_stateschema () &&
$\mathcal{K}$ (∃State* • [DeltaList ≪Declaration≫ ≪| PredicateList≫]\ Declaration°, PredicateListMapping) &&
GetOpSignature (**GetSuperClass** (Class*), OperationName)){
$\mathcal{K}$ (PredicateList, PredicateListMapping);
return check_stateschema();}
return false;})))
∨
((**ExistsOpInClass** (Class*, OperationName)) ⇒
($\mathcal{K}$(OperationName $\stackrel{\wedge}{=}$ [DeltaList ≪Declaration≫ ≪| PredicateList≫], PredicateListMapping) =
(virtual Boolean OperationName ($\mathcal{K}$ (Declaration)){
if (check_stateschema() &&
$\mathcal{K}$ (∃ State* • [DeltaList ≪Declaration≫ ≪| PredicateList≫]\ Declaration°, PredicateListMapping)){
$\mathcal{K}$ (PredicateList, PredicateListMapping);
return check_stateschema();}
return false;})))

where Declaration° and State* denote the outputs of the operation schema and State of Class*, respectively. □

Inheritance

In order to present the translation of rename list when mapping inheritance, we define a new free type as follows:

ItemType ::= attribute | method

Now, we translate inheritance as follows. Notice that in the case of method renaming, when at least one of the method attributes is renamed, we need to interact with the user to know how to map the renamed attribute (i.e., attribute_renaming). In other words, we get the mapping of the renamed attribute(s) based on the user interaction. Also, the mapping of both method renaming and attribute renaming must be considered within Class* definition:

$\mathcal{K}$ (InheritedClass) = $\mathcal{K}$ (ClassName ≪ActualParameters≫ ≪RenameList≫)=
$\mathcal{K}$ (ClassName ≪[ExpressionList]≫ ≪[RenameItemList]≫)=
(: public ≪virtual≫ ClassName < ∀ e: Expression | e ∈ ExpressionList • $\mathcal{K}$ (e),>

∀ item: RenameItem | (item ∈ RenameItemList) ∧ (item=newName/oldName) ∧ (**GetItemType** (ClassName, oldName)=method) ∧ Noattrenamed (ClassName, oldName) • $\mathcal{K}$ (item) =
(virtual Boolean newName (**GetOpParam** (ClassName, oldName)){
if (check_stateschema () && ClassName ::oldName (**GetOpParam** (ClassName, oldName))
return check_stateschema();
return false;})

∀ item: RenameItem | (item ∈ RenameItemList) ∧ (item=newName/oldName) ∧ (**GetItemType** (ClassName, oldName)=method) ∧ ¬ Noattrenamed (ClassName, oldName) • $\mathcal{K}$ (item, attribute_renaming)

∀ item: RenameItem | (item ∈ RenameItemList) ∧ (item=newName/oldName) ∧ (**GetItemType** (ClassName, oldName)=attribute) • $\mathcal{K}$ (item, attribute_renaming)

Note that we cannot present the result of $\mathcal{K}$ (item, attribute_renaming) because it depends on how to map attribute renaming which is determined by the user. □

2.4. Mapping of operation expressions

Suppose that we want to translate op $\stackrel{\wedge}{=}$ op_1 O op_2, where O is the operator and op_1 and op_2 are the operands. First, we define new free type "operator" which is used in function "Merge":

operator ::= $\wedge$ | [] | $\overset{9}{9}$ | || | $\bullet$ | $\neg$

We map conjunction and parallel composition operators in main method to boost threads [18]. In addition, we merge inputs and outputs of op_1 and op_2 and map the resulting parameters to attributes of ClassName because threads cannot have any input and output parameters. We also define new methods op_1conj and op_2conj whose bodies are promotions of op_1 and op_2; parameters of op_1conj and op_2conj are newly defined attributes for ClassName, provided that they have not been defined before. The mappings of other operators are the same as what we presented in our previous work [12-14]:

op $\triangleq$ op_1 $\wedge$ op_2.

(CallOrpromotion (op_1) $\wedge$ CallOrpromotion (op_2))
$\Rightarrow$
$\mathcal{K}$ (op $\triangleq$ op_1 $\wedge$ op_2) =
virtual Boolean op (){
void op_1conj () {op_1 (related_attributes_ClassName)};
void op_2conj () {op_2 (related_attributes_ClassName)};
if (check_stateschema() && **GetPre** (**GetClass** (op_1), op_1) && **GetPre** (**GetClass** (op_2), op_2)){
boost::thread conj_op_1_op_2 (&op_1conj);
boost::thread conj_op_2_op_1 (&op_2conj);
conj_op_1_op_2.join ();
conj_op_2_op_1.join ();}
else return false;} □

op $\triangleq$ op_1 [] op_2.

(CallOrpromotion (op_1) $\wedge$ CallOrpromotion (op_2))
$\Rightarrow$ $\mathcal{K}$ (op $\triangleq$ op_1 [] op_2) =
virtual Boolean op (**Merge** (**GetOpParam** (**GetClass** (op_1), op_1), **GetOpParam** (**GetClass** (op_2), op_2), [])){
int r =rand() % 2;
if (r = = 0){
if (check_stateschema ()){
if (!**GetPre** (**GetClass** (op_1), op_1)) return op_2 (**GetOpParam** (**GetClass** (op_2), op_2));
else return op_1 (**GetOpParam** (**GetClass** (op_1), op_1));}
return false;}
else {
if (check_stateschema ()){
if (!**GetPre** (**GetClass** (op_2), op_2)) return op_1 (**GetOpParam** (**GetClass** (op_1), op_1));
else return op_2 (**GetOpParam** (**GetClass** (op_2), op_2));}
return false;}} □

op $\triangleq$ op_1 $\overset{9}{9}$ op_2.

(CallOrpromotion (op_1) $\wedge$ CallOrpromotion (op_2))
$\Rightarrow$ $\mathcal{K}$ (op $\triangleq$ op_1 $\overset{9}{9}$ op_2) =
virtual Boolean op (**Merge** (**GetOpParam** (**GetClass** (op_1), op_1), **GetOpParam** (**GetClass** (op_2), op_2), $\overset{9}{9}$)){
Local_Params_Def
if (check_stateschema () && **GetClass** (op_1):: op_1 (**GetOpParam** (**GetClass** (op_1), op_1)) && **GetClass** (op_2):: op_2 (**GetOpParam** (**GetClass** (op_2), op_2))
return true;
return false;}

where Local_Params_Def denotes those parameters which are included in both parameters lists of op_1 and op_2; they are also outputs in op_1 and inputs in op_2. □

op $\triangleq$ op_1 || op_2.

(CallOrpromotion (op_1) $\wedge$ CallOrpromotion (op_2) $\wedge$ Comdirec (op_1, op_2)) $\Rightarrow$
$\mathcal{K}$ (op $\triangleq$ op_1 || op_2) = $\mathcal{K}$ (op $\triangleq$ op_1 $\wedge$ op_2) □

op $\triangleq$ op_1 $\bullet$ op_2.

(CallOrpromotion (op_1) $\wedge$ CallOrpromotion (op_2))
$\Rightarrow$
$\mathcal{K}$ (op $\triangleq$ op_1 $\bullet$ op_2, user_opinion) =
((user_opinion = 1) $\Rightarrow$ $\mathcal{K}$ (op $\triangleq$ op_1 $\wedge$ op_2)
$\wedge$
(user_opinion = 2) $\Rightarrow$
virtual Boolean op (**Merge** (**GetOpParam** (**GetClass** (op_1), op_1), **GetOpParam** (**GetClass** (op_2), op_2), $\bullet$)){
if (check_stateschema () && **GetPre** (**GetClass** (op_1), op_1) && **GetPre** (**GetClass** (op_2), op_2)){
GetPost (**GetClass** (op_1), op_1);
GetPost (**GetClass** (op_2), op_2);
return check_stateschema();}
return false;})

Note that in our previous work [13,14], we considered the user's opinion to obtain the final mapping of scope enrichment when at least one of its operands is in the form of operation promotion; we specified this kind of user interaction using "user_opinion" in the above mapping. The value of "user_opinion" could be "1" or "2" indicating the first option or the other one; to see options, refer to [13,14] □

op $\triangleq$ $\neg$ op.

(CallOrpromotion (op)) $\Rightarrow$
$\mathcal{K}$ (op $\triangleq$ $\neg$ op, PredicateListMapping) =
virtual Boolean op (**GetOpParam** (**GetClass** (op), op)){
if (check_stateschema () && $\mathcal{K}$ ($\exists$State* $\bullet$

```
(¬ op)\Declaration°, PredicateListMapping)){
        K (¬ (op. PredicateList),
        PredicateListMapping);
        return check_stateschema();}
return false;}
```

where "PredicateListMapping" indicates the mapping of negation of PredicateList part of op in the above translation. □

3. Proof sketch

Theorem 1 demonstrates the soundness of our mapping method by showing it refines Object-Z specifications into corresponding C++ code.

Theorem 1 (Soundness). For a given Object-Z specification ozs and a given C++ code cppc, if cppc = $\mathcal{K}$ (ozs), then we have ozs $\sqsubseteq^2$ cppc (The notation '$\sqsubseteq$' is used as the refinement notation.)

In the next subsection, we present our proof methodology.

3.1. Proof methodology

We use some proof obligations similar to what are proposed in B method [19] to prove that a refinement N refines a machine M in order to prove Theorem 1. The steps of our proof methodology are as follows:

1. We consider a correspondence between OZ specifications and machine notion in B. In other words, when determining each construct of B machine, we notice which parts of an OZ specification (and in which way) could participate, based on the semantics of OZ and machines in B.
2. We consider a correspondence between C++ code and refinement notion in B. In other words, when determining each construct of B refinement, we notice which parts of C++ code (and in which way) could participate, based on the semantics of C++ and refinements in B.
3. We introduce the machine and refinement obtained after doing steps 1 and 2 into Atelier-B [20].
4. Atelier-B makes three proof obligations [19] to show the introduced refinement refines the introduced machine. Finally, Atelier-B proves the obtained proof obligations.

The next two subsections describe the details of our proof method by explaining proof approach in B and details of correspondences mentioned in steps 1 and 2 above. Figure 2 summarizes our proof methodology.

3.2. Proof in B method

Consider the following specification of machine M and refinement N:

MACHINE M(p)
CONSTRAINTS Constraints
CONSTANTS C_M
PROPERTIES $Properties_M$
VARIABLES V_M
INVARIANT Inv_M
INITIALIZATION $Init_M$
OPERATIONS
  y←op(x) =
  PRE $Pre_{op,\ M}$
  THEN $Def_{op,\ M}$
  END
...
END

REFINEMENT N
REFINES M
CONSTANTS C_N
PROPERTIES $Properties_N$
VARIABLES V_N
INVARIANT Inv_N
INITIALIZATION $Init_N$
OPERATIONS
  y←op(x)=
  PRE $Pre_{op,\ N}$
THEN $Def_{op,\ N}$
  END
...
END

Now, the proof obligations in B to prove that Refinement N refines Machine M are as follows (if S is a statement, and P is a predicate, then the notation [S]P denotes a predicate which is true of a given state precisely when executing S in that state is guaranteed to reach a final state in which P is true [19]):

1. $Constraints \wedge Properties_M \wedge Properties_N \rightarrow \exists(V_M, V_N) \bullet (Inv_M \wedge Inv_N)$
2. $Constraints \wedge Properties_M \wedge Properties_N \rightarrow [Init_N] \neg [Init_M] \neg Inv_N$
3. $Constraints \wedge Properties_M \wedge Properties_N \wedge Inv_M \wedge Inv_N \wedge Pre_{op,\ M} \rightarrow Pre_{op,N} \wedge [Def'_{op,N}] \neg [Def_{op,M}] \neg (Inv_N)$

It is worth noting that the third proof obligation should be repeated for each specified operation.

3.3. Details of correspondences

Now, we define correspondences mentioned in steps 1 and 2 of our proof methodology in order to enable Atelier-B to customize the three proof obligations given in Subsection 3.2. At first, we should consider the following cases in order to customize the third proof obligation:

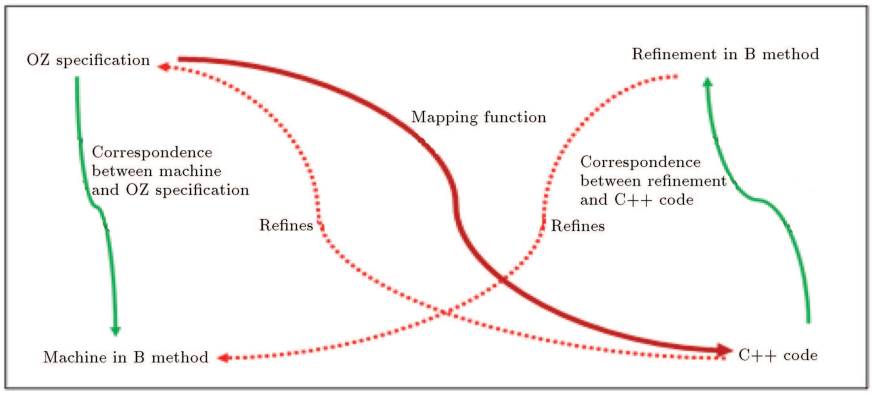

Figure 2. Our proof methodology.

- Case 1 (proof obligation 3.1): operation op is defined in the form of operation schema whose container class is a child class, and none of its parent classes has an operation with the same name as this operation. The proof of the two other cases which we considered in order to propose the mapping of operation schema in [14] (when the container class is not a child class, or operation op has delta-list and declaration part in its definition) can be done similarly by removing the predicates related to the parent class.
- Case 2 (proof obligation 3.2): operation op is defined in the form of operation schema and has the same name as inherited operation schemas.
- Case 3 (proof obligation 3.3): operation op is defined in the form of $op \triangleq op_1 \fatsemi op_2$; also, op_1 and op_2 are in the form of operation schema.
- Case 4 (proof obligation 3.4): operation op is defined in the form of $op \triangleq op_1\ O\ op_2$; also, op_1 and op_2 are in the form of operation schema, and O is the conjunction or parallel composition operator.
- Case 5: operation op is defined in the form of $op \triangleq op_1 \,[]\, op_2$; also, op_1 and op_2 are in the form of operation schema.
- Case 6 (proof obligation 3.6): operation op is defined in the form of $op \triangleq \neg\ op_1$ in which op_1 is in the form of operation schema.

1. Correspondence between Object-Z specifications and machines in B. Before presenting the correspondence, we should mention the following used conventions:
   - 1.1 "$Class^{\square}$" denotes the class in Object-Z specification for which the given proof obligations are being proved. To simplify the proof, we assume that "$Class^{\square}$" has only one parent class.
   - 1.2 "Declaration°" denotes the outputs of the operation whose definition is put before '\'.
   - 1.3 "Operation" indicates the definition of operation op. Similarly, "$Operation_1$" and "$Operation_2$" denote the definition of operation schemas op_1 and op_2, respectively.

Now, the correspondence is as follows. Cases 1 to 6 in parts $Pre_{op,M}$ and $Def_{op,M}$ are related to Cases 1 to 6 considered at the beginning of this subsection (Figure 3 shows part of the corresponding machine in Atelier-B):

Constraints = true

✓ $Properties_M$ = $Class^{\square}$.LocalDefinition.AxiomaticDefinition.PredicateList ∧ GlobalParagraph.AxiomaticDefinition.PredicateList ∧ GlobalParagraph.AbbreviationDefinition ∧ $Class^{\square}$.LocalDefinition.AbbreviationDefinition.PredicateList ∧ GlobalParagraph.Schema.PredicateList ∧ GlobalParagraph.GenericDefinition.PredicateList ∧ GlobalParagraph.Predicate

✓ Inv_M = $Class^{\square}$.State.PredicateList ∧ $Class^{\square}$.InheritedClass.State.PredicateList

✓ $Pre_{op,\,M}$ =

(1): ∃ ($Class^{\square}$.State)′ •$Class^{\square}$.Operation \ Declaration°

(2): (∃ ($Class^{\square}$.State)′ • $Class^{\square}$. Operation \ Declaration°) ∧(∃ ($Class^{\square}$. InheritedClass.State)′ •$Class^{\square}$. InheritedClass.Operation\Declaration°)

(3): (∃ ($Class^{\square}$.State)′ • $Class^{\square}$.$Operation_1$ \Declaration°) ∧ (∃ ($Class^{\square}$.State)′ • $Class^{\square}$.$Operation_2$ \Declaration°)

(4): (∃ ($Class^{\square}$.State)′ • $Class^{\square}$.$Operation_1$ \ Declaration°) ∧ (∃ ($Class^{\square}$.State)′ • $Class^{\square}$.$Operation_2$ \ Declaration°)

```
MACHINE
  M1 (Class_FormalParameters)

CONSTANTS
  Class_LocalDefinition_AxiomaticDefinition_Declaration,
  GlobalParagraph_AxiomaticDefinition_Declaration,
  GlobalParagraph_AbbreviationDefinition_Abbreviation,
  GlobalParagraph_Schema_Declaration,
  GlobalParagraph_GenericDefinition_Declaration

PROPERTIES
  Class_LocalDefinition_AxiomaticDefinition_PredicateList ∧
  GlobalParagraph_AxiomaticDefinition_PredicateList ∧
  GlobalParagraph_AbbreviationDefinition ∧
  Class_LocalDefinition_AbbreviationDefinition_PredicateList ∧
  GlobalParagraph_Schema_PredicateList ∧
  GlobalParagraph_GenericDefinition_PredicateList ∧
  GlobalParagraph_Predicate

VARIABLES
  Class_State_Declaration, Class_InheritedClass_State_Declaration,
  GlobalParagraph_Schema_Declaration, GlobalParagraph_GenericDefinition_Declaration

INVARIANT
  Class_State_PredicateList ∧ Class_InheritedClass_State_PredicateList

INITIALIZATION
  Class_InitialState_PredicateList ∧ Class_InheritedClass_ InitialState_PredicateList

OPERATIONS
  Boolean ← op1 (inputs and outputs) =
  PRE ∃ (Class_State)′ . Class_Operation\Declaration
  THEN Class_Operation_PredicateList
  END
END
```

Figure 3. Part of the corresponding machine in Atelier-B.

(5): $(\exists\ (\text{Class}^{\square}.\text{State})' \bullet \text{Class}^{\square}.\text{Operation}_1 \setminus \text{Declaration}^{\circ}) \vee (\exists\ (\text{Class}^{\square}.\text{State})' \bullet \text{Class}^{\square}.\text{Operation}_2 \setminus \text{Declaration}^{\circ})$

(6): $\exists\ (\text{Class}^{\square}.\text{State})' \bullet \text{Class}^{\square}.\ \neg\ \text{Operation} \setminus \text{Declaration}^{\circ}$

✓ $\text{Init}_{\text{M}} = \text{Class}^{\square}.\text{InitialState.PredicateList} \wedge \text{Class}^{\square}.\text{InheritedClass.InitialState.PredicateList}$

✓ $\text{Def}_{\text{op,M}} =$

(1): $\text{Class}^{\square}.\text{Operation.PredicateList}$

(2): $\text{Class}^{\square}.\text{Operation.PredicateList} \wedge \text{Class}^{\square}.\text{InheritedClass.Operation.PredicateList}$

(3): $\text{Class}^{\square}.\text{Operation}_1.\text{PredicateList} \wedge \text{Class}^{\square}.\text{Operation}_2.\text{PredicateList}$

(4): $\text{Class}^{\square}.\text{Operation}_1.\text{PredicateList} \parallel \text{Class}^{\square}.\text{Operation}_2.\text{PredicateList}$

(5): $\text{Class}^{\square}.\text{Operation}_1.\text{PredicateList} [] \text{Class}^{\square}.\text{Operation}_2.\text{PredicateList}$

(6): $\text{Class}^{\square}.\neg(\text{Operation.PredicateList})$

✓ $\text{C}_{\text{M}} = \text{Class}^{\square}.\text{LocalDefinition.AxiomaticDefinition.Declaration}, \text{GlobalParagraph.AxiomaticDefinition.Declaration}, \text{GlobalParagraph.AbbreviationDefinition.Abbreviation}, \text{GlobalParagraph.Schema.Declaration}, \text{GlobalParagraph.GenericDefinition.Declaration}$

✓ $\text{p} = \text{Class}^{\square}.\text{FormalParameters}$

✓ $\text{V}_{\text{M}} = \text{Class}^{\square}.\text{State.Declaration}, \text{Class}^{\square}.\text{InheritedClass.State.Declaration}, \text{GlobalParagraph.Schema.Declaration}, \text{GlobalParagraph.GenericDefinition.Declaration}$

2. Correspondence between C++ code and refinements in B. Before presenting the correspondence, we should mention the following used conventions:

2.1 "Operation_1" and "Operation_2" denote the

```
REFINEMENT M1_r

REFINES M1

CONSTANTS
  constant_Class_attributes, global_constants

PROPERTIES
  const_method && check_SchemaName (SchemaNames) && gen_method && K
  (GlobalParagraph_Predicate)

VARIABLES
  Class_attributes

INVARIANT
  Class_check_stateschema ()

INITIALIZATION
  Class_INIT ()

OPERATIONS
  Boolean ← op1 (inputs and outputs) =
  PRE Class_check_stateschema () && K (∃ (Class_State)' . Class_Operation\Declaration)
  THEN K (Class_Operation_PredicateList);
       If (Class_check_stateschema ()) return true;
       Return false;
  END
END
```

Figure 4. Part of the corresponding refinement in Atelier-B.

definition of operation schemas op_1 and op_2, respectively.

2.2 "Class$^{\square}$.attributes°" denotes the attributes of C++ class obtained from the mapping of Class$^{\square}$ except the constant attributes.

2.3 We substitute check_Identifier () in the mapping of axiomatic definition, when it is in the form of constant definition, with const_method.

2.4 We also substitute the method resulted from the mapping of generic definition, when it is not in the form of operator definition, with gen_method.

2.5 IO denotes the inputs and outputs of op. Also, IO_1 and IO_2 denote the inputs and outputs of op_1 and inputs and outputs of op_2, respectively.

Now, the correspondence is as follows. Cases 1 to 6 in parts $Pre_{op,M}$ and $Def_{op,M}$ are related to Cases 1 to 6 considered at the beginning of this subsection (Figure 4 shows part of the corresponding refinement in Atelier-B):

✓ $Properties_N$= const_method && check_SchemaName (SchemaName s) && gen_method && $\mathcal{K}$ (GlobalParagraph.Predicate)

✓ Inv_N= Class$^{\square}$.check_stateschema ()

✓ $Pre_{op,N}$=

(1): Class$^{\square}$.check_stateschema () && $\mathcal{K}$ (∃ (Class$^{\square}$.State)′ • Class$^{\square}$.Operation\Declaration°)

(2): Class$^{\square}$.check_stateschema () && $\mathcal{K}$ (∃ (Class$^{\square}$.State)′ • Class$^{\square}$.Operation\Declaration°) && Class$^{\square}$.InheritedClass.op (IO)

(3): Class$^{\square}$.check_stateschema () && op_1 (IO_1) && op_2 (IO_2)

(4): Class$^{\square}$.check_stateschema () && $\mathcal{K}$ (∃ (Class$^{\square}$.State)′ • Class$^{\square}$.$Operation_1$ \Declaration°) && $\mathcal{K}$ (∃ (Class$^{\square}$.State)′ • Class$^{\square}$.$Operation_2$ \ Declaration°)

(5): Class$^{\square}$.check_stateschema () && ($\mathcal{K}$ (∃ (Class$^{\square}$.State)′ • Class$^{\square}$.$Operation_1$ \ Declaration°) || $\mathcal{K}$ (∃ (Class$^{\square}$.State)′ • Class$^{\square}$.$Operation_2$ \ Declaration°))

(6): Class$^{\square}$.check_stateschema () && $\mathcal{K}$ (∃ (Class$^{\square}$.State)′ • Class$^{\square}$.¬Operation\Declaration°)

✓ $Init_N$= Class$^{\square}$.INIT ()

✓ $Def_{op,N}$=

(1): $\mathcal{K}$ (Class$^{\square}$.Operation.PredicateList); if (Class$^{\square}$.check_stateschema ()) return

true;
return false;

(2): $\mathcal{K}$ (Class$^{\square}$.Operation.PredicateList);
if (Class$^{\square}$.check_stateschema ()) return true;
return false;

(3): if (Class$^{\square}$.check_stateschema ()) return true;
return false;

(4): op_1 (IO_1) || op_2 (IO_2)

(5): Any v
v:=0 [] v:=1
if (v==0) then return op_1 (IO_1);
else return op_2 (IO_2);

(6): $\mathcal{K}$ (Class$^{\square}$.¬(Operation.PredicateList));
if (Class$^{\square}$.check_stateschema ()) return true;
return false;

✓ C_N=constant Class$^{\square}$ attributes, global constants

✓ V_N=Class$^{\square}$.attributes°

3.4. Proof in Atelier-B

Using the correspondences mentioned in the previous subsection, we introduced the resulting machine and refinement into Atelier-B. Then, Atelier-B made three proof obligations and proved the obtained proof obligations. To simplify the proof, in all proof obligations, we assumed that a correct method is used for mapping predicates from Object-Z to C++. We also assumed that an operation call with a Boolean return value is equivalent to a predicate with the same value.

4. Case study

In this section, we first present a specification of credit-card bank accounts system [14] in Object-Z. Then, we use our translation method in order to obtain C++ code from this specification.

4.1. Specification of credit-card bank accounts system

The aim of the credit-card bank accounts specification is to capture the basic functionality of credit-card account objects and their interactions. At first, the following global abbreviation and types are needed in the specification: Free type "Status" has two constants, namely "valid" and "invalid", and is used as a type to model validity of credit cards. Basic type "CUSTOMER" will be used later as a type to model holder(s) of credit card(s); "limitvalue" abbreviates for a set containing possible values which the account(s) cannot be overdrawn beyond them.

CreditCard

↾(*limit, expiry-value, balance*, INIT, withdraw, deposit, withdraw Avail, newday, reissue, *status*, *holder*)

limit: ℕ

limit ∈ limitvalue

expiry-value: ℕ

balance: ℤ
holder: CUSTOMER
expiry: ℤ
Δ
status: Status

balance + *limit* >= 0
status = *invalid* ⇔ *expiry* = 0

INIT

balance = *0* ∧ *expiry* = *expiry-value*

reissue

Δ (*expiry*)

status = *invalid*
expiry ′ = *expiry-value*
status ′ =*valid*

withdraw

Δ *(balance)*
amount?: ℕ

amount? <= *balance* + *limit*
status = *valid*
balance ′ = *balance* - *amount?*

deposit

Δ *(balance)*
amount?: ℕ

status = *valid*
balance ′ =*balance* + *amount?*

withdrawAvail

Δ *(balance)*
amount!: ℕ

status = *valid*
amount!= *balance* + *limit*
balance ′ = - *limit*

newday

Δ *(expiry)*

expiry ′ =*expiry-1*

limitvalue = = {1000, 2000, 5000}
Status ::= invalid | valid
[CUSTOMER]

Next, the class of a credit-card account object in isolation is specified: this class encapsulates the details of the state of a credit card and the operations it can undergo. Also, this specification introduces axiomatic definitions in the form of constant definition (i.e., the declaration of limit and its associated predicate list and also the definition of expiry-value). This class records four numbers, one an integer denoting the current balance of the account (this balance is of course often negative, indicating that the account is overdrawn) and the others are a natural number (a non-negative integer) denoting a fixed limit: the account cannot be overdrawn beyond this credit limit and the natural number denoting the fixed number of days before credit card expiration (i.e. expiry-value) and the integer number denoting how many days remained until credit card expiration (i.e. expiry), respectively. It is worth mentioning that expiry will be initialized with expiry-value in "INIT". Also, this class contains one more primary variable holder, which denotes the holder of credit card and secondary variable status, which denotes whether credit card is valid or not. If credit card is expired, then the status will become "invalid"; otherwise, it will be "valid". Finally, this class has the following operations:

1. "reissue": reissuing the credit card.
2. "withdraw": using the credit card to obtain funds for some purpose, e.g. to purchase goods.
3. "deposit": depositing money supplied by the environment (as variable amount?) into the account.
4. "withdrawAvail": withdrawing the total amount currently available (i.e., *balance* + *limit*).
5. "newday": decrementing the value of *expiry*.

The following specification (CreditCardConfirm class schema) uses inheritance to extend CreditCard class schema with additional features. This specification also introduces the sequential composition operator. This class schema defines two operations "fundsAvail" and "withdrawConfirm" that are not inherited from class CreditCard. The former results in the available funds, i.e., *balance* + *limit*. The overall consequence of the latter is to withdraw some amount from the account and output the value of the available funds that remain in the account.

CreditCardConfirm

↑ (*limit*, *balance*, INIT, withdraw, deposit, withdrawAvail, witdrawConfirm)
CreditCard

fundsAvail

funds!: ℕ

funds!= *balance* + *limit*

withdrawConfirm $\triangleq$ withdraw ⨟ fundsAvail

The following specification (CreditCardCount class schema) uses the notion of operation renaming and redefinition when inheriting CreditCard class schema. Also, the class CreditCardCount introduces the conjunction operator. This class schema defines one state variable, i.e. *withdrawals*, which keeps the number of times the operation "withdraw" has been applied. CreditCardCount also defines two operation "incrementCount" and "withdraw". The overall consequence of the latter is to withdraw some amount from the account and increment the value of *withdrawals*.

CreditCardCount

↑(*limit*, *balance*, INIT, withdraw, deposit, withdrawAvail)
CreditCard [oldWithdraw/withdraw]

withdrawals: ℕ

INIT

withdrawals = *0*

incrementCount

Δ (withdrawals)
status = *valid*
withdrawals ′ = *withdrawals* +*1*

withdraw $\triangleq$ oldWithdraw ∧ incrementCount

The following specification (CreditCards class schema) specifies a banking system consisting of an aggregate of credit-card account objects (of class CreditCard), each with the same limit on the magnitude of the overdraft permitted. Also, this specification introduces scope enrichment and parallel composition operators and uses operation promotion. This class schema has one state variable, i.e. *cards* whose value is a set of identities of objects of CreditCard. It also has a number of operations: "add" which specifies the addition of a new credit-card account object to *cards*, "delete" which specifies the deletion of an object from *cards*, "transferAvail" whose role is simply to specify the selection from cards of two distinct credit-card account objects between which funds transfer is to occur; the other operations is applied to particular credit-card account objects similar to the operations of CreditCard.

4.2. Obtaining C++ code

Now, we use our translation method in order to map the specification presented in Subsection 4.1 into a C++ code. We map each class schema to a C++ class with the same name as its associated class schema name. In this section, we only propose the C++ code for global data types and CreditCard class schema; see Appendix B for mapping of other classes. Also, as we said in our previous

CreditCards

↾(*commonlimit*, INIT, add, delete, withdraw, deposit, withdrawAvail, transferAvail, newday, reissue)

commonlimit: ℕ

commonlimit ∈ *limitvalue*

cards: ℙ CreditCard©

∀ c:*cards* • *c.limit* = *commonlimit*

INIT

cards = ∅

add

Δ (*cards*)
card?: CreditCard
customer?: CUSTOMER

card? ∉ *cards*
card?.limit = *commonlimit*
card?.INIT
card?.holder = *customer?*
cards′ = *cards* ∪ {*card?*}

delete

Δ (*cards*)
card?: CreditCard

card? ∈ *cards*
card?.status = invalid
cards ′ = *cards*\ {*card?*}

withdraw ≙ [*card?*: cards] • *card?*.withdraw
deposit ≙ [*card?*: cards] • *card?*.deposit
withdrawAvail ≙ [*card?*:cards] • *card?*.withdrawAvail
transferAvail ≙ [*from?*, *to?*: cards | *from?* ≠ *to?*] • *from?*.withdrawAvail || *to?*.deposit
newday ≙ [*card?*: cards] • *card?*.newday
reissue ≙ [*card?*: cards] • *card?*.reissue

work [14], we consider forward declaration of classes as follows:

```
class CreditCard;
class objectaggregation_CreditCard;
class CreditCards;
class CreditCardConfirm;
class CreditCardCount;
```

Moreover, we consider an *enum* "Boolean" to model return type of methods of classes globally (recall that return types of all mappings of operation schemas are "Boolean"). In addition, we consider an *enum* "Status" as mapping of the global free type "Status". We map basic type "CUSTOMER" to a *struct* whose name is "CUSTOMER". Furthermore, we map abbreviation *limitvalue* to an array and also we consider two global methods whose names are "check_limitvalue" and "initialization_limitvalue" according to the mapping of abbreviation definition in the form of set definition when all of its elements are numeric; see Subsection 2.2.

```
enum Boolean {false, true};
enum Status {invalid, valid};
struct CUSTOMER {};
int [3] limitvalue;

Boolean check_limitvalue (int var)
{ if (limitvalue[0] = = var || limitvalue[1] = = var ||
limitvalue[2] = = var) return true;
  return false;}

void initialization_limitvalue ()
{ limitvalue [0] = 1000;
   Limitvalue [1] = 2000;
   Limitvalue [2] = 5000;}
```

In order to map CreditCard class schema, we consider a new class whose name is CreditCard. Methods and attributes of this class are as follows:

Methods: We map each operation schema such as "withdraw", "deposit", "withdrawAvail" to a method whose name is the same as its associated operation schema name. It is worth mentioning that we consider method "INIT" for mapping Initialization schema. Also, we consider constructor and destructor according to their templates. It is worth mentioning that we call "check_limitvalue" in constructor according to the mapping of global abbreviations. Note that we will explain later why we must consider operator overloading '=' and "==" (we marked them with "user interaction" because the user must herself fill these operator overloading).

Attributes: We map *limit* and its associated predicate list and also *expiry-value* according to the mapping of axiomatic definition in the form of constant definition to *const* attributes *limit* and *expiry-value*; the associated predicate list of *limit* is mapped into the constructor. Also, we should consider input parameters for initializing *limit* and *expiry-value* (we name them l and m). Moreover, we map *holder* to an attribute whose name is "holder". Furthermore, we map *balance* and *expiry* which are primary variables to attributes whose names are "balance" and "expiry", respectively. We map status to a macro since it is a secondary variables obtained via the primary variable *expiry*. Finally, we should consider attribute *p_CreditCards*

according to the mapping of object containment in class "CreditCards" (i.e., cards: $\mathbb{P}$ CreditCard$_{©}$).

```
class CreditCard{
    public:
  /*[user interaction]*/
       #define status expiry=0?invalid:valid
       const unsigned int limit; int balance; CUSTOMER
       holder;
       const unsigned int expiry-value;
       virtual Boolean INIT ();
       virtual Boolean withdraw (unsigned int amount);
       virtual Boolean deposit (unsigned int amount);
       virtual Boolean withdrawAvail (unsigned int &
       amount);
       virtual Boolean newday ();
       virtual Boolean reissue();
       CreditCard (unsigned int 1);
       ~CreditCard();
       /*[user interaction]*/
       void operator=(CreditCard a);
       Boolean operator==(CreditCard a);
    protected:
       virtual Boolean check_stateschema();
       CreditCards * p_CreditCards;
       int expiry;};

/*[user interaction]*/
void CreditCard::operator=(CreditCard a)
{this→balance=a.balance;}

/*[user interaction]*/
Boolean CreditCard::operator==(CreditCard a)
{if(this→limit==a.limit && this→balance==a.balance)
     return true;
   return false;}

Boolean CreditCard::INIT ()
{if (check_stateschema() && balance==0 &&
expiry==expiry-value && status==valid) return true;
   return false;}

Boolean CreditCard::withdraw (unsigned int amount)
{if (check_stateschema() && amount==balance+limit
&& status==valid)
   {balance=balance-amount;
     if (check_stateschema()) return true;}
   return false;}

Boolean CreditCard::deposit (unsigned int amount)
{if (check_stateschema() && status==valid)
   {balance=balance+amount;
     if (check_stateschema()) return true;}
   return false;}

Boolean CreditCard::withdrawAvail (unsigned int &
amount)
{if (check_stateschema() && status==valid)
   {amount=balance+limit;
     balance=-limit;
     if (check_stateschema()) return true;}
   return false;}

Boolean CreditCard::reissue ()
{if (check_stateschema() && status==invalid)
   {expiry=expiry-value;
     if (check_stateschema()) return true;}
   return false;}

Boolean CreditCard::newday()
{if (check_stateschema())
   {expiry=expiry-1;
     if (check_stateschema()) return true;}
   return false;}

Boolean CreditCard::check_stateschema()
{if (balance+limit>=0)
   return true;
   return false;}

CreditCard::~CreditCard(){}

CreditCard::CreditCard (unsigned int 1, unsigned int
m):limit(1), expiry-value(m)
{if (!check_limitvalue (limit))
       CreditCard::~CreditCard();
else {
       balance=0;
       expiry=expiry-value;
       status=valid;}}
```

5. Discussion

In our previous work [14], we compared our mapping method with some prominent works [6-8] considering some criteria including Formal language coverage [21], Correctness [21], Interactivity [22], Transparency [22], Expressiveness [14], Tool support [14] and Programming language coverage [14]. Now, we update that part of this comparison affected by the new version of our mapping method.

5.1. Correctness

Unlike our method, none of the previous work [6-8] paid attention to prove the correctness of the mapping method.

5.2. Expressiveness

The previous work [6-8] represented their mapping rules only by the use of natural language while we presented our mapping method formally in addition to proposing templates for mappings in C++ language.

5.3. Tool support

Two of previous work, i.e. [6,7], have developed tools for supporting mapping methods. Authors in [8] did not propose any mapping rules for operation operators, but they used them in their case studies. To compare our method with that of [8], we can assert that we proposed all of our mapping rules *explicitly* and formally. Also, comparing our method with those of [6,7], we proposed our mapping rules formally. Having explicit and formal rules helps us to develop a supporting tool easily; the next section describes our tool status.

6. Conclusion and future work

In this paper, we presented a formal mapping from Object-Z specifications to C++ code based on the mappings given in our previous work [12-14]. Having the formal mapping in place, correctness of the mapping can be proved in an easier way, and the possibility to develop supporting tools for the approach is increased. It is worth mentioning that we are now in final stages of developing a tool which supports our mapping method using C++ (in Visual Studio) and MS SQL server; more precisely, we have so far developed parts regarding the mapping of global basic type definition, axiomatic definition, generic definition, abbreviation definition, schema, free type definition and some parts of class schema.

In continuing the paper, we have proposed the mapping of local abbreviation definition when its left-hand side is a variable name, and its right-hand side is class union; this case has never been covered in neither related work [6-8] nor our previous work [12-14]. Finally, we proved the correctness of our mapping method.

As our future work, we are going to:

1. Present appropriate rules to animate the remaining constructs of Object-Z, such as distributed operators and free types when constructors are used in the definition.
2. Finalize the development of the tool supporting our mapping method together with doing complexity analysis and benchmarks for the resulting tool.

References

1. McComb, T. and Smith, G. "Animation of object-Z specifications using a Z animator", *First International Conference on Software Engineering and Formal Methods*, pp. 191-201, IEEE Computer Society Press (2003).
2. Smith, G., *The Object-Z Specification Language*, Kluwer Academic Publishers, USA (2000).
3. Duke, R. and Rose, G., *Formal Object-Oriented Specification Using Object-Z*, Macmillan, UK (2000).
4. Woodcock, J. and Davies, J., *Using Z: Specification, Refinement, and Proof*, Prentice Hall (1996).
5. Ramkarthik, S. and Zhang, C. "Generating Java skeletal code with design contracts from specifications in a subset of Object-Z", *5th IEEE/ACIS International Conference on Computer and Information Science*, pp. 405-411, IEEE Computer Society Press (2006).
6. Rafsanjani, G. and Colwill, S.J. "From Object-Z to C++: a structural mapping", *Z User Meeting (ZUM'92)*, pp. 166-179, Springer-Verlag (1992).
7. Fukagawa, M., Hikita, T. and Yamazaki, H. "A mapping system from Object-Z to C++", *First Asia-Pacific Software Engineering Conference (APSEC94)*, IEEE Computer Society Press, pp. 220-228 (1994).
8. Johnston, W. and Rose, G. "Guidelines for the manual conversion of Object-Z to C++", *SVRC Technical Report 93-14*, The University of Queensland (1993).
9. Wang, Z. Xia, M. and Zhao, Y. "Transform mechanisms of Object-Z based formal specifi-cation to Java", *Computational Intelligence and Software Engineering (CiSE)*, pp. 1-4, IEEE Computer Society Press (2009).
10. Griffiths, A. "From Object-Z to Eiffel: a rigorous development method", *Technology of Object-Oriented Languages and Systems: TOOLS 18*, Prentice-Hall (1995).
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. "An animation approach to develop C++ code from Object-Z specifications", *International Symposium on Computer Science and Software Engineering*, pp. 9-16, IEEE Computer Society Press (2011).
13. Najafi, M. and Haghighi, H. "An approach to develop C++ code from Object-Z specifications", Accepted in *2nd World Conference on Information Technology* (2011).
14. Najafi, M. and Haghighi, H. "An approach to animate Object-Z specifications using C++", *Scientia Iranica*, **19**(6), pp. 1699-1721 (2012).
15. Deitel, H.M. and Deitel, P.J., *C++: How to Program*, 5th Edn., Prentice Hall (2005).
16. Méry, D. and Singh, N.K. "Automatic code generation from event-B models", *Second Symposium on Information and Communication Technology*, ACM Press, pp. 179-188 (2011).
17. Albalooshi, F. and Long, F. "Multiple view environment supporting VDM and Ada", *IEEE Proceedings Software*, **146**(6), pp. 203-219 (2002).
18. http://boost.org
19. Schneider, S., *The B-Method: An Introduction*, Macmillan (2001).
20. Atelier-B tool, Available at: http://www.atelierb.eu/en/atelier-b-tools/atelier-b-4-0/.

21. Breuer, P.T. and Bowen, J.P. "Towards correct executable semantics for Z", *Z Users Conference -ZUM*, pp. 185-209 (1994).

22. Utting, M. "Animating Z: interactivity, transparency and equivalence", *Second Asia-Pacific Software Engineering Conference (APSEC)*, pp. 294-303, IEEE Computer Society Press (1995).

Appendix A

Abstract syntax of Object-Z

In this section, we provide the abstract syntax of those parts of Object-Z, in BNF notation, in which we are concerned. For a complete account of the Object-Z syntax, see [2]; we are only interested in those parts used in our mapping:

In Object-Z, a specification is a set of paragraphs. The following shows what can appear in an Object-Z specification [2] (We use the notation of '$\ll\gg$' in order to specify optional parts of the syntax throughout the paper):

Specification

Specification ::= ParagraphList
ParagraphList ::= Paragraph
|Paragraph
ParagraphList

Paragraph ::= BasicTypeDefinition
| AxiomaticDefinition
| GenericDefinition
| AbbreviationDefinition
| FreeTypeDefinition
| Schema
| Class
| Predicate

Global paragraphs

BasicTypeDefinition ::= [IdentifierList]
IdentifierList ::= Identifier
| Identifier, IdentifierList

AxiomaticDefinition ::= [Declaration $\ll$|
PredicateList$\gg$]
GenericDefinition ::= [$\ll$FormalParameters$\gg$
Declaration $\ll$| PredicateList$\gg$]

AbbreviationDefinition ::= Abbreviation = =
Expression
Abbreviation ::= VariableName $\ll$FormalParameters$\gg$

FreeTypeDefinition ::= Identifier ::= BranchList
BranchList ::= Branch
| Branch | BranchList
Branch ::= Identifier
| VariableName $\ll$Expression$\gg$

Schema ::= SchemaHeader $\triangleq$ [Declaration $\ll$|
PredicateList$\gg$]
| SchemaHeader $\triangleq$ SchemaExpression
SchemaHeader ::= SchemaName $\ll$FormalParameters$\gg$

Class::= ClassName $\ll$FormalParameters$\gg$
[$\ll$VisibilityList$\gg$
$\ll$InheritedClassList$\gg$
$\ll$LocalDefinitionList$\gg$
$\ll$State$\gg$
$\ll$InitialState$\gg$
$\ll$OperationList$\gg$]

FormalParameters ::= [IdentifierList]

Class paragraphs

Visibilitylist ::= $\uparrow$(DeclarationNameList)
DeclarationNameList ::= DeclarationName
| DeclarationName,
DeclarationNameList

InheritedClassList ::= InheritedClass
| InheritedClass,
InheritedClassList
InheritedClass ::= ClassName $\ll$ActualParameters$\gg$
$\ll$RenameList$\gg$
ActualParameters ::= [ExpressionList]
ExpressionList ::= Expression
| Expression, ExpressionList

RenameList ::= [RenameItemList]
RenameItemList ::= RenameItem
| RenameItem, RenameItemList
RenameItem ::= DeclarationName / DeclarationName
LocalDefinitionList ::= LocalDefinition
| LocalDefinition
LocalDefinitionList
LocalDefinition ::= BasicTypeDefinition
| AxiomaticDefinition
| AbbreviationDefinition
| FreeTypeDefinition

State ::= [$\ll$Declaration$\gg$ $\ll\Delta$ Declaration$\gg$ $\ll$|
PredicateList$\gg$]
| [Δ Declaration $\ll$| PredicateList$\gg$]
| [PredicateList]

InitialState ::= INIT $\triangleq$ [PredicateList]

OperationList ::= Operation
| Operation
OperationList

Operation ::= OperationName $\triangleq$ [$\ll$DelataList$\gg$

≪Declaration≫ ≪| PredicateList≫]
| OperationName $\triangleq$ [Declaration ≪| PredicateList≫]
| OperationName $\triangleq$ [≪ PredicateList≫]
| OperationName $\triangleq$ OperationExpression

OperationName ::= Identifier

DeltaList ::= Δ (DeclarationNameList)

Operation expressions
OperationExpression ::=
[≪DeltaList≫≪Declaration≫≪| Predicate≫]
| [Declaration≪|Predicate≫] | [≪Predicate≫]
| Identifier ≪RenameList≫
| OperationExpression \(DeclarationNameList)
| OperationExpression ∧OperationExpression
| OperationExpression ||OperationExpression
| OperationExpression ||!OperationExpression
| OperationExpression [ ]OperationExpression
| OperationExpression ⨾ OperationExpression
| OperationExpression • OperationExpression
| Expression.Identifier | (OperationExpression)

Appendix B

Mapping of CreditCardConfirm, CreditCardCount and CreditCards class schemas to C++ code

To map CreditCardConfirm class schema, we consider a new class whose name is CreditCardConfirm. Methods and attributes of this class are as follows:

Methods: This class inherits all of the features of CreditCard class. In addition to the inherited methods, this class has two new methods "withdrawConfirm" and "fundsAvail" which are obtained from the mapping of "withdrawConfirm" and "fundsAvail" operation schemas, respectively. The method "fundsAvail" has one output *funds* which, as we stated earlier, should be in the form of call by reference. The method "withdrawConfirm" has one input and one output which are obtained from the mapping of sequential composition operator. More precisely, the operation schema "withdraw" has the input *amount*, and the operation schema "fundsAvail" has the output *funds*; merging these two parameters results in two parameters *amount* and *funds* for "withdrawConfirm".

Attributes: This class has no attribute.

```
class CreditCardConfirm: public CreditCard{
  public:
     virtual Boolean withdrawConfirm (unsigned int amount, unsigned int & funds);
     CreditCardConfirm (unsigned int 1);
     ~CreditCardConfirm ();
  private:
     virtual Boolean fundsAvail (unsigned int & funds);
};

CreditCardConfirm::~CreditCardConfirm(){}

CreditCardConfirm::CreditCardConfirm(unsigned int 1):
limit(1)
{if (!(limit==1000 || limit==2000 || limit==5000))
    CreditCardConfirm::~CreditCardConfirm();
  else balance=0;}

Boolean CreditCardConfirm::fundsAvail (unsigned int &
funds)
{if (CreditCard::check_stateschema())
  {funds=balance+limit;
    if (check_stateschema() return true; }
  return false;}
Boolean CreditCardConfirm::withdrawConfirm (unsigned
int amount, unsigned int & funds)
{if (withdraw(amount) && fundsAvail(funds)) return
true;
   return false;}
```

To map CreditCardCount class schema, we consider a new class whose name is CreditCardCount. Methods and attributes of this class are as follows:

Methods: This class inherits all of the features of CreditCard class. In addition to the inherited methods, this class has the new method "incrementCount" which is obtained from the mapping of "incrementCount" operation schema. In order to map the renaming of the operation schema "withdraw", we consider a method whose name is "oldWithdraw" and has the same inputs and outputs as those of "withdraw" in CreditCard class. We map "withdraw" in CreditCardCount according to the mapping of conjunction. Finally, this class has the method "INIT" which is obtained from the mapping of "INIT" in CreditCardCount considering the inherited "INIT".

Attributes: In addition to the inherited attributes, this class has new attribute *withdrawals* obtained from the mapping of *withdrawals* in CreditCardCount.

```
class CreditCardCount: public CreditCard{
   public:
      virtual Boolean INIT();
      virtual Boolean withdraw (unsigned int amount);
        CreditCardCount(unsigned int 1);
            ~CreditCardCount();
   private:
      unsigned int withdrawals;
```

```
        virtual Boolean incrementCount();
        virtual Boolean oldWithdraw(unsigned int
        amount);};

CreditCardCount::~CreditCardCount(){}

CreditCardCount::CreditCardCount(unsigned int 1):
limit(1)
{ if (!(limit==1000 || limit==2000 || limit==5000))
          CreditCardCount::~CreditCardCount();
     else {balance=0;
   withdrawals=0;}}

Boolean CreditCardCount::INIT()
{ if( CreditCard::INIT() && withdrawals==0)
    return true;
         return false;}

Boolean CreditCardCount::incrementCount()
{ if(check_stateschema() && status==valid)
  {withdrawals=withdrawals+1;
     if(check_stateschema())
       return true;}
  return false;}

Boolean CreditCardCount::oldWithdraw(unsigned int
amount)
{if (check_stateschema() && CreditCard::
withdraw(amount))
  { if (check_stateschema())
    return true;}
  return false;}

Boolean CreditCardCount::withdraw(unsigned int
amount)
{ if ( oldWithdraw(amount) && incrementCount())
      return true;
   return false;}
```

In order to map CreditCards class schema, we consider a new class whose name is CreditCards. Methods and attributes of this class are as follows:

Methods: This class has methods "add", "delete", "withdraw", "deposit", "withdrawAvail", "transferAvail", "newday" and "reissue" which are obtained from the mapping of "add", "delete", "withdraw", "deposit", "withdrawAvail", "transferAvail", "newday" and "reissue" in CreditCards class schema, respectively. For instance, we consider the mapping of withdraw according to the mapping of scope enrichment. The preconditions of the operation are the conjunction of promo-tion and RHS of scope enrichment (We assume that the user selects the second alternative for mapping this operation; see Subsection 2.4). This operation does not have any postconditions. The method inputs are *card* and *amount* which are obtained from merging the inputs of the left-hand side of scope enrichment and *card?*.withdraw.

Attributes: We map *commonlimit* and its associated predicate list according to the mapping rule for axiomatic definitions. Also, we consider the new class objectaggregation_CreditCards for mapping of cards which is in the form of object aggregation with object containment. We map *cards* according to the mapping of object aggregation with object containment to attributes *cards* and *p_CreditCards* in classes CreditCards and CreditCard, respectively. In order to handle '=' and "==" which are needed for correct compiling and running objectaggregation_CreditCard class, we must overload these operators for CreditCard class; hence, we considered these operator overloading in CreditCard class earlier.

```
class CreditCards{
 public:
    const unsigned int commonlimit;
    virtual Boolean INIT();
    virtual Boolean add(CreditCard * card);
    virtual Boolean delete(CreditCard * card);
    virtual Boolean withdraw(CreditCard * card,
    unsigned int amount);
    virtual Boolean deposit(CreditCard * card, unsigned
    int amount);
    virtual Boolean withdrawAvail (CreditCard * card,
    unsigned int & amount);
    virtual Boolean transferAvail (CreditCard * from,
    CreditCard * to);
    virtual Boolean newday (CreditCard * card);
    virtual Boolean reissue (CreditCard * card);
    CreditCards(unsigned int 1);
    ~CreditCards();
private:
    objectaggregation_CreditCard * cards;
    virtual Boolean check_stateschema();};

Boolean CreditCards::INIT()
{if (check_stateschema() && cards→GetListSize()==0)
    return true;
  return false;}

Boolean CreditCards::check_stateschema()
{ /*[user interaction]*/
  Boolean f=true;
  for(int i = 0; i <cards→GetListSize(); i + +)
    if((cards→GetNodeValue(i))→limit==commonlimit)
      f=false;
  return f;}
```

```
Boolean CreditCards::add(CreditCard * card,
CUSTOMER customer)
{if (check_stateschema() && !cards→SearchValue(card)
&& card→limit==commonlimit && card→INIT()
            card→holder==customer){
           cards→AddToList(card);
           if (check_stateschema()) return true;}
   return false;}

Boolean CreditCards::delete(CreditCard * card)
{if(check_stateschema() && cards→SearchValue(card)
&& card→status==invalid){
       cards→RemoveFromList(card);
       if(check_stateschema()) return true;}
   return false;}

Boolean CreditCards::withdraw(CreditCard * card,
unsigned int amount)
{/*[user interaction]*/
   if(check_stateschema() && cards→SearchValue(card)
     && card→withdraw(amount)){
        if (check_stateschema())
          return true;}
   return false;}

Boolean CreditCards::deposit(CreditCard * card,
unsigned int amount)
{/*[user interaction]*/
   if(check_stateschema() && cards→SearchValue(card)
    && card→deposit(amount)){
        if(check_stateschema())
         return true;}
    return false;}

Boolean CreditCards::withdrawAvail(CreditCard * card,
unsigned int & amount)
{/*[user interaction]*/
   if(check_stateschema() &&
cards→SearchValue(card)
    && card→withdrawAvail(amount)){
        if(check_stateschema())
         return true;}
   return false;}

Boolean CreditCards::transferAvail(CreditCard * from,
CreditCard * to)
{/*[user interaction]*/
   unsigned int amount;
   if(check_stateschema() &&
cards→SearchValue(to)&&
cards→SearchValue(from) && from != to &&
    from→withdrawAvail(&amount) && to→
    deposit(amount)){
       if(check_stateschema())
        return true;}
   return false;}

Boolean CreditCards::newday (CreditCard * card)
{if(check_stateschema() && cards→SearchValue(card)
&&card→newday())
  {if(check_statschema()) return true;}
  return false;}

Boolean CreditCards::reissue(CreditCard * card)
{if (check_stateschema() && cards→SearchValue(card)
&& card→reissue())
   {if (check_stateschema()) retrun true;}
   return false;}

CreditCards::CreditCards(unsigned int 1):commonlimit(1)
{if (!check_limitvalue (commonlimit))
   CreditCards::~CreditCards();
 else{
   /*[user interaction]*/
   cards=new objectaggregation_CreditCard(1000,
   commonlimit);}}

CreditCards::~CreditCards()
{delete cards;}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class objectaggregation_CreditCard{
    private:
       CreditCard * elements;
       int size;
       int max;
    public:
        objectaggregation_CreditCard(int s, unsigned int
1);
       ~objectaggregation_CreditCard();
       void AddToList(CreditCard * value1);
       void RemoveFromList(CreditCard * value1);
       int GetListSize();
       Boolean SearchValue(CreditCard * value1);
       void Delete();
       CreditCard * GetNodeValue(int index); };

objectaggregation_CreditCard::objectaggregation_Credit-
Card(int s,unsigned int 1)
   {max=s;
     elements=new CreditCard(1) [max];
     size=-1;}

objectaggregation_CreditCard::~objectaggregation_Credit-
Card ()
   {delete[ ] elements;}

void objectaggregation_CreditCard::AddToList
(CreditCard * value1)
   {if(size< (max − 1))
     {elements[size+1]=*value1;
```

```
        size++;}}

void objectaggregation_CreditCard::
RemoveFromList(CreditCard * value_1)
   {if(SearchValue(value_1))
     {int index=0;
       for(int i = 0; i <=size; i + +)
         if(elements[i]==*value_1)
           index=i;
       for(i=index; i <=size-1; i + +)
         elements[i]=elements[i + 1];
       size--;}}

int objectaggregation_CreditCard::GetListSize()
   {return (size+1);}

Boolean objectaggregation_CreditCard::
SearchValue(CreditCard * value_1)
   {for(int i = 0; i <=size; i + +)
       if(elements[i]==*value_1)
         return true;
     return false;}

void objectaggregation_CreditCard::Delete()
        {size=-1;}
CreditCard * objectaggregation_CreditCard::
GetNodeValue(int index)
   {if(index⇐size)
     {return &(elements[index]);}
     return NULL;}
```

Biographies

Mehrnaz Najafi received both her MSc and BSc degrees in Computer Engineering-Software from Shahid Beheshti University, Iran, in 2012 and 2010. Her research interests are formal program development and formal verification.

Hassan Haghighi is an assistant professor in the faculty of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran. He received his PhD degree in Computer Engineering-Software from Sharif University of Technology, Iran, in 2009. His main research interest is using formal methods in the software development life cycle.