World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:5, No:11, 2011

Refinement of Object-Z Specifications Using Morgan's Refinement Calculus

Mehrnaz Najafi, Hassan Haghighi

***Abstract*—**Morgan's refinement calculus (MRC) is one of the well-known methods allowing the formality presented in the program specification to be continued all the way to code. On the other hand, Object-Z (OZ) is an extension of Z adding support for classes and objects. There are a number of methods for obtaining code from OZ specifications that can be categorized into refinement and animation methods. As far as we know, only one refinement method exists which refines OZ specifications into code. However, this method does not have fine-grained refinement rules and thus cannot be automated. On the other hand, existing animation methods do not present mapping rules formally and do not support the mapping of several important constructs of OZ, such as all cases of operation expressions and most of constructs in global paragraph. In this paper, with the aim of providing an automatic path from OZ specifications to code, we propose an approach to map OZ specifications into their counterparts in MRC in order to use fine-grained refinement rules of MRC. In this way, having counterparts of our specifications in MRC, we can refine them into code automatically using MRC tools such as RED. Other advantages of our work pertain to proposing mapping rules formally, supporting the mapping of all important constructs of Object-Z, and considering dynamic instantiation of objects while OZ itself does not cover this facility.

***Keywords*—**Formal method, Formal specification, Formal program development, Morgan's Refinement Calculus, Object-Z

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

I. Introduction

BUILDING a reliable system is one of the main challenges when developing large, complex software systems [1]. Formal methods which refers to mathematically rigorous techniques and tools for the specification, design and verification of software systems, have offered a logical solution to the problem of software reliability [1, 2].

Every formal method requires a soundly based specification language. The popularity of the object-oriented programming approach has led to the adoption of a similar method for expressing encapsulation and reuse concepts in formal specifications [3]. Object-Z [4, 5] is a Z-based notation which provides specific constructs to facilitate specification in an object-oriented style [2].

Many contributions [1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14] have been so far proposed for developing programs from OZ specifications that can be categorized to animation [1, 2, 3, 6, 7, 8, 9, 10, 11, 12] and refinement [13, 14] methods. Rafsanjani and Colwill [6] described some basic rules for structural mapping from Object-Z to C^{++}, but these rules do not cover some specification constructs of Object-Z, such as precondition, postcondition, class invariants, visibility list, operation operators and some types of definitions like class variables and generic parameters. In [7], Fukagawa et al. has augmented the work of [4] by presenting two new rules that consider constructor for types of constants and template class for generic parameters.

In [8] it has been given another method to animate OZ by C^{++} which covers precondition, postcondition, class invariants, visibility list, some types of definitions like free types and class variables; however, it does not consider axiomatic definitions and multiple inheritance. Also, for a subset of operation operators including conjunction, negation, choice and parallel composition, rules are not proposed explicitly. In [9], we proposed an animation approach to develop C^{++} code from Object-Z specifications which considers several Object-Z concepts whose mappings have not been proposed in none of the previous works. For example, animation rules for all types of definitions like class union, axiomatic definitions and all of operation operators are proposed in [9].

In [3], a methodology for animating the Object-Z specification language using a Z animation environment is presented. The focus of this work is to model in Z a framework to manage the dynamic instantiation of objects and object references. Also, some other works exist in the literature which animates Object-Z specifications by other object-oriented programming languages, such as java [1, 10, 11], Eiffle [12], and Spec# [2]. Nevertheless, none of the mentioned animation methods did not pay attention to correctness, their mapping rules did not cover some important constructs of OZ such as all cases of operation expressions and also, they did not propose mapping rules formally which can help to prove the correctness of mapping.

On the other hand, Derrick and Boiten [13, 14] proposed a refinement approach in Object-Z which considers a correct refinement of a class schema to another class schema. However, fine-grained refinement rules, such as refinement at the operation schema level, are needed to do the refinement from design to code.

We use [3] to propose a methodology for refinement of OZ specifications using MRC. The main contribution of this methodology is to give an automatic path from OZ specifications to code. Our approach is to propose a translation function which maps OZ constructs to their MRC counterparts according to OZ concrete syntax [4], MRC abstract syntax which we define later, and also a framework in MRC in order

M. N. Faculty of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran (e-mail: m.najafi@mail.sbu. ac.ir).

H. H. Faculty of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran (phone: 9821-2990-4136;e-mail: h_haghighi@sbu. ac.ir).

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

to manage object references. Other contributions of our methodology are the ability to use fine-grained refinement rules, the introduction of dynamic instantiation of objects while OZ itself does not cover this facility, proposing translation rules formally and supporting mapping of important constructs, such as abbreviations in global paragraph while none of the related methods in the literature have not supported the animation\refinement of this construct.

The paper is organized in the following way. In section 2, the preliminaries of this work are reviewed by focusing on OZ concepts and constructs. In section 3, we provide the abstract syntaxes of those parts of both OZ and MRC in which we are concerned. Section 4 presents our methodology translating OZ constructs into their counterparts in MRC. Section 5 includes a case study showing the applicability and usefulness of our approach. Finally, the last section is devoted to the conclusion and some directions for future work.

II. Preliminaries

To mention the preliminaries of this paper, we first review the work of McComb and Smith who proposed a method of animating OZ which considers reference semantics and uses an existing Z animator in a way that is both automatable and transparent to the user [3]. In order to introduce concept "object" in Z, McComb and Smith modelled object identity with a free type whose name is *_identity* as follows:

_identity::= _null | _REF <<ℕ>>

The constructor "_REF" allows us to associate a unique object identity with each natural number being irrespective to the class the object belongs to. Value "_null" allows us to ignore parts of a specification for the purposes of animation. It acts as a placeholder for the reference to an object which has not been instantiated in the animation environment.

Also, they considered an additional constant *self* to cater OZ notion of self reference; see [3] for more details. Also, they modelled in Z a framework to manage the dynamic instantiation of objects. The mentioned framework comprises an explicit reference table, i.e., finite mappings, where object identities are mapped to instances of a schema describing the state of each class (i.e., they modelled the state of class schema by schema in Z), and also, a schema whose name is *_ClassName_new* and is declared in order to model dynamic instantiation of objects operation; see [3] for more details. Finally, all operations of OZ specifications are transformed to Z operations based on the mentioned reference table.

As another preliminary of our work, we now review OZ concepts. In the following we show what appear in global paragraphs of OZ [4]:

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

Considering the above syntax, concepts of OZ which we cover in our methodology are [4, 5]:

Basic Type Definition: introduces one or more basic types by the inclusion of their names in a square-bracketed, comma-separated list.

Abbreviation Definition: introduces a type whose name is the identifier on the left-hand side of the definition and whose values are those of the expression on the right-hand side.

Free Type Definition: introduces a type whose name is the identifier on the left-hand side of the expression and whose values are given by the branches of the right-hand side of the definition.

Class: the major new construct in Object-Z is the class schema which captures the object-oriented notion of a class by encapsulating a single state schema, and its associated initial state schema, with all the operation schemas of the given state.

The structure of the class schema specification in Object-Z is shown below [4]:

```
┌──── ClassName [FormalParameters] ────
│ VisibilityList
│ InheritedClass(es)
│ LocalDefinition(s)
│ State
│ InitialState
│ Operations
└──────────────────────────────────────
```

Considering the above syntax, concepts which we use in our methodology are [4, 5]:

Visibility list: lists those features that are visible to the environment of an object of the class. In class schema "Queue" [4], shown in Fig. 1, count, INIT, Join and Leave are visible features.

Inheritance: when a class is inherited by another class in Object-Z, its definitions, i.e., state and initial state schemas and operations, are merged with those of the inheriting class.

State schema: is a nameless box with optional declaration and predicate parts. The predicate of a state schema is called the class invariants. Variables which are defined in the declaration part of the state schema are called state variables. In class schema "Queue", items and count are state variables, and this class schema does not have any invariants. State variables are categorized to:

1. Primary variables: may only be changed by an operation in which they are declared. In class schema "Queue", items and count are primary variables.
2. Secondary variables: may be changed by any operation.

Initial State: defines the initial states of a class, and its name is INIT. In class schema "Queue", items=<> and count=0 are predicates of its initial state schema.

Operations: defines the permissible changes in the state that an object of the class may undergo. In class schema "Queue", two operations Join and Leave are defined. Related concepts are:

1. *Operation schema*: is a named box which may consist of a Δ-list, a declaration and a predicate part. Δ-list consists of primary variables that the operation

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

may change when applied to an object of the class. The declaration part is for auxiliary variables needed to define the operation. They are generally input and output variables. The predicate part relates the possible states before the operation execution to the possible states after its execution. Class schema "Queue" has two operation schemas which are Join and Leave.

2. *Operation expression:* includes operation schema definitions and operation promotions defined later. An operation expression may be also an identifier with an optional rename list enabling previously defined operations to be modified or combined.
3. *Operation operators:* Object-Z has the following operation operators used to modify and combine operation expressions:
   - Conjunction (∧): used to model the simultaneous occurrence of two operations.
   - Parallel composition (|| or ||!): used to model communication between simultaneously occurring operations in both directions.
   - Choice (□): used to model the occurrence of an applicable operation among two operations. If both operations are applicable, then this operator selects one of them nondeterministically.
   - Sequential composition (;): used to model two operations occurring in sequence. It also allows the communication between the operations, but the communication is only possible in one direction.

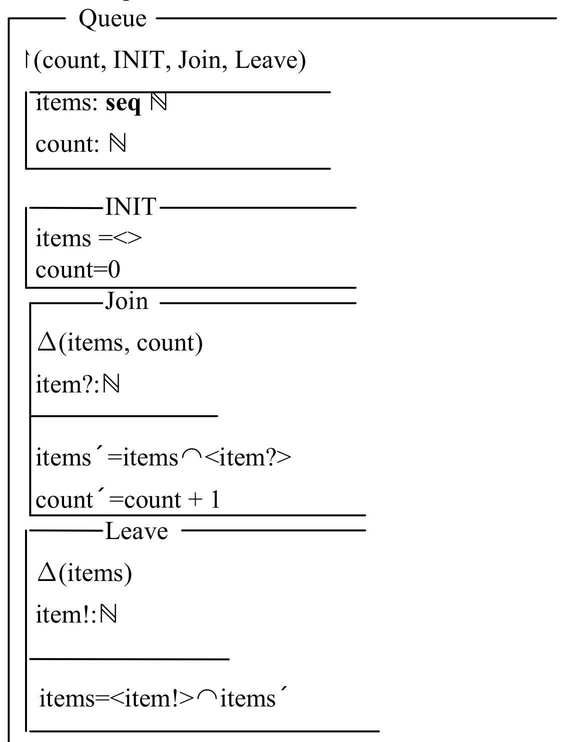


Fig. 1 Class schema Queue

Other important concepts in Object-Z are [4, 5]:

- Operation promotion: any class can have operations which model the application of operations to objects whose identities are declared as global constants. Such operations are said to promote the operations of the objects to operations of the class. An operation promotion is defined using the dot notation *a.Op*, where *a* is an expression which evaluates to the identity of the referenced object, and Op is the name of a visible operation of the related class.
- Attribute promotion: visible attributes of an object may be promoted to any scope in which the identity of that object may be referenced.

III. Abstract Syntaxes of OZ and MRC

In this section we provide the abstract syntaxes of those parts of both OZ and MRC, in BNF notation, in which we are concerned.

A. Abstract Syntax of Object-Z

For a complete account of the concrete syntax of Object-Z see [4]. We are only interested in those parts used in our mapping.

Abstract syntax of Object-Z:

1) Specification

Specification ::= Paragraph
⋮
Paragraph

2) Global Paragraphs

We proposed the abstract syntaxes of global paragraphs and class in section 2. By considering the abstract syntax of global paragraphs or paragraph, abstract syntaxes of BasicTypeDefinition, AbbreviationDefinition and FreeTypeDefinition are as follows:

BasicTypeDefinition ::= [Identifier, …, Identifier]

AbbreviationDefinition ::= Abbreviation == Expression
Abbreviation ::= VariableName [FormalParameters]
| PrefixGenericName Identifier
| Identifier InfixGenericName Identifier

FreeTypeDefinition ::= Identifier::= Branch|…|Branch
Branch ::= Identifier
| VariableName <<Expression>>

3) Class Paragraphs

Visibilitylist ::= ↾(DeclarationNameList)

InheritedClass ::= ClassName[ActualParameters] [RenameList]

State ::= [Declaration [Δ Declaration][|Predicate]]
| [Δ Declaration [|Predicate]]
| [Predicate]

InitialState ::= INIT ≙ [Predicate]

Operation ::=
```
┌──── OperationName ──────────
│ DeltaList
│ [Declaration]
├─────────────────────
│ [PredicateList]
└──────────────────────────────
```

OperationName ≙ OperationExpression

4) Operation Expressions

OperationExpression ::= [Declaration [|Predicate]]
| [[Predicate]]
| [DeltaList[Declaration][|Predicate]]
| OperationExpression∧OperationExpression
| OperationExpression||OperationExpression
| OperationExpression[]OperationExpression
| OperationExpression;OperationExpression
| Expression.Identifier

B. Abstract Syntax of Morgan's Refinement Calculus

We could not find the official concrete syntax of the MRC in the literature. Taking our inspiration from [15], we built part of an abstract syntax of MRC which is needed for our methodology as follows:

Specification ::= ModuleSt
| TypeSt
| BasicTypeSt

ModuleSt ::= **module** Name
[**export** NameList]
[**import** NameList]
[**var** VarListType]
[**and** Pred]
[**procedure**
.
.
.
procedure]
initially Pred
end

TypeSt ::= **type** Name ≙ Type_expression

BasicTypeSt ::= [Identifier, …, Identifier]

procedure Name [(ParamList)] ≙ SpecSt

SpecSt ::= Frame [PreCondition, PostCondition]
| SpecSt ; SpecSt
| ProcCall ; SpecSt
| SpecSt ; ProcCall
| ProcCall

ProcCall ::= ProcName [(VarList)]

ParamList ::= **value** VarDecl, ParamList
| **result** VarDecl, ParamList
| **value result** VarDecl, ParamList
| **reference** VarDecl, ParamList

PreConcition::= Pred
PostCondition::=Pred
Frame::= :
|VarList:

IV. OUR REFINEMENT METHODOLOGY

For modelling the concept "object" in MRC, we consider a module whose name is *object*. Also, a new free type whose name is *identity* will be considered for modelling referential semantics as follows; for more details about this free type see [3] (this free type is similar to *_identity* in [3] which REF and null have the same functionalities as _REF and _null in *_identity*).

type identity ::= REF ℕ | null

Now we define a translation function $[|\ |]^T$ which translates each part of OZ abstract syntax into its counterpart in MRC abstract syntax.

Definition 1: Translation Function $[|\ |]^T$:

$[|\ |]^T$: **OZ→MRC** is a function from **OZ** to **MRC** which is explained in detail as follows:

A. Mapping of Global Paragraphs

Basic Type Definition: the same notion of basic type exists in MRC. Thus, the translation for basic types is as follows:

$[|$ BasicTypeDefinition $|]^T$ = $[|$ Identifier, …, Identifier $|]^T$= $[|$ Identifier, …, Identifier $|]^T$

Abbreviation Definition: We consider mapping of abbreviation definition in a case in which the right side of the abbreviation is in the form of computational expression and none of its expression elements is numeric, as follows:

$[|$ Abbreviation = = Expression $|]^T$ = **type** Abbreviation = $[|$ Expression $|]^T$ (which separates each operation expression element with '|')

Free Types: we propose the translation for free type definition when all of its branches are identifier, as follows:

$[|$ Identifier ::= $Branch_0|\dots|Branch_n$ $|]^T$= **type** Identifier = $[|Branch_0|]^T|\dots|[|Branch_n|]^T$

Class: $[|$ class schema (whose name is ClassName) $|]^T$ = **module** ClassName . . . **end**

B. Mapping of Class Paragraphs

Visibility list: we consider each declaration name in visibility list which is an operation name as an *export* name list. If class schema is a parent class schema, we must export all of its operations because a child class schema must inherit all of the operations of its parent class schema. Note that we do not export a declaration name which is not an operation name. This is due to the manner by which the mapping of state schema declarations will be proposed. So, the translation for this concept is as follows:

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

$[|$ VisibilityList $|]^T$ = **export** DeclarationName which is an operation name and is not INIT.

Inherited class: MRC does not have any direct corresponding concept for mapping of inheritance. An acceptable solution could be to use the "*uses*" relation. In this way, we must import all procedures of those module(s) corresponding to parent class(es) because a child class schema can use all of the operations of its parent class schema.

State: we consider a new type whose name is *ClassName_state* and is in the form of set comprehension. Set declaration consists of mapping of

- ✓ state declaration,
- ✓ inherited class (es) state declaration,
- ✓ a Boolean variable (Note that we must define type Boolean using type expression earlier),
- ✓ *Init* which will be used to accommodate the predicate *a.INIT* for an object *a* that will be evaluated as true precisely when that object is in its initial state,
- ✓ and a variable *self* in order to cater for OZ notion of self reference.

Set property consists of mapping of

- ✓ state predicate list,
- ✓ inherited class (es) state predicate list,
- ✓ and also a state invariant that is introduced to force the *Init* variable to be always equivalent to the predicate lists in initial state schema of class schema and its inherited class (es) schemas.

The translation of state is thus as follows:

type B ≙ False|True

$[|\text{State}|]^T$ = **type** ClassName_state ≙
{$[|\text{State.Declaration}|]^T$, Init: B, self: identity,
$[|\text{inheritedClass}_0\text{.State.Declaration}|]^T$,…,
$[|\text{inheritedClass}_m\text{.State.Declaration}|]^T$,

$|[|\text{State.predicateList}|]^T \wedge$
$[|\text{inheritedClass}_0\text{.State.predicateList}|]^T \wedge \ldots \wedge$
$[|\text{inheritedClass}_m\text{.State.predicateList}|]^T \wedge \text{Init} \Leftrightarrow$
$([|\text{InitialState.predicateList}|]^T \wedge$
$[|\text{inheritedClass}_0\text{.InitialState.predicateList}|]^T \wedge \ldots \wedge$
$[|\text{inheritedClass}_m\text{.InitialState.predicateList}|]^T$)}

For modeling the concept "object", we consider a variable in each module which is obtained through the mapping of class schema; the variable name is *ClassName_reftab* which holds objects of corresponding class schema (i.e., ClassName). This variable maps each object identity to its associated *ClassName_state* as follows:

ClassName_reftab: identity ↣ ClassName_state

Also, we consider a type whose name is *State_Object* which comprises reference tables (i.e., ClassName_reftab) of each class schema in the specification and a variable whose name is *state* with type *State_Object* in module *object*. The aim of declaring *State_Object* is to ensure that each identity is assigned to a unique object, and null is not mapped to any object. Declaration of this type is as follows:

type State_Object
{ClassName_0_reftab:identity↣ClassName_0_state, …,
ClassName_n_reftab: identity ↣ ClassName_n_state |
$\bigcap_{i=1}^{n}$ **dom** ClassName_i_reftab = ∅ ∧
$\bigcup_{i=1}^{n}$ **dom** ClassName_i_reftab⊆ ran REF }

Initial State: the mapping of initial state has been already considered in definition of *ClassName_state*.

Operation Schema: we map each operation schema to a procedure in the related module (i.e., that module obtained from the mapping of class schema which includes the mentioned operation schema). The procedure name is *ClassName_OperationSchemaName*. We propose its parts according to the abstract syntax of procedure in MRC presented in section 3. The parameters list of the procedure consists of an index with type *identity* which is the object identity that the procedure should be applied to; indexes for other objects which exist in *ClassName* and their operations are promoted in *ClassName* using type identity and the mapping of operation schema declaration, respectively. The procedure frame consists of *ClassName_reftab* and all of those operation schema declaration elements which are outputs. Also, its pre part (according to the abstract syntax of procedure in MRC) will be obtained by using the definition of getting precondition from postcondition, i.e., ∃ State′. Operation\output and its post part is predicateList itself (see the concrete syntax of operation schema in section 3).

$[|$ Operation Schema $|]^T$ = **procedure** ClassName_OperationSchemaName (**value** ClassName_index: identity, $[|$ Declaration $|]^T$) ≙ ClassName_reftab, DeclarationName which is in Declaration and is in the form of output: [[|∃ State′. Operation\declarations which are outputs $|]^T$, [| predicateList$|]^T$]

C. Mapping of Operation Expressions

Mapping of all cases of operation expressions are as follows:

1) [DeltaList[Declaration][|Predicate]]

We map this case to specification statement "w: [pre, post]" whose frame, i.e., w, is *ClassName_reftab* along with the output part of declaration. Also, its pre will be obtained by using the definition of getting precondition from *Predicate*, and its post is *Predicate* itself. Thus, the translation of this case is as follows:

$[|$ [DeltaList [Declaration] [|Predicate] $|]^T$ = ClassName_reftab, Declarations which are outputs: [[| ∃State′. [DeltaList [Declaration] [|Predicate]]\ Declarations which are outputs$|]^T$, [|Predicate$|]^T$]

2) [Declaration[|Predicate]]

This operation expression is a special case of the above one. Thus, its translation is similar to item 1:

[| [DeltaList *[*|Predicate*]* |] ^{T} = ClassName_reftab: [[| ∃State´. [DeltaList *[*|Predicate*]*] |] ^{T}, [|Predicate|] ^{T}]

3) [[Predicate]]

Similarly, this operation expression is a special case of item 1 above and thus its translation is as follows:

[|Predicate|]^T= ClassName_reftab:
[∃State´. [*[*|Predicate*]*] |] ^{T}, [|Predicate|] ^{T}]

For the following operation expressions, we consider a procedure whose name is ClassName_OperationName, and its parameters list consists of a variable whose name is ClassName_index with type *identity* and also those variables obtained from merging inputs and outputs of all operation expressions except parallel composition (hidden variables must be omitted from parameters list).

4) OperationExpression$_1$ ∧ OperationExpression$_2$

We consider two cases for proposing translation function of this operation expression as follows:

- If neither of the operation expressions are in the form of promotion, and also the mapping of both of them are in the form of specification statements "w: [pre, post]" (we consider the left side of the conjunction as w_1: [Q_1, P_1] and the right side as w_2: [Q_2, P_2]), we use the conjunction notion in MRC for conjoining these two specification statements. Also, merging the variables of frames of the mentioned two specification statements are necessary. The translation function for this case is as follows (note that we use existential quantifier in pre because, in the beginning, both of Q_1 and Q_2 must be satisfied: Q_1 is satisfied for w_1 and it must be satisfied for at least one w_2, too. A similar concern should be considered for Q_2):

  [|OperationExpression$_1$ ∧ OperationExpression$_2$ |]^T
  = [| [|OperationExpression$_1$|]^T ∧
  [|OperationExpression$_2$|] ^{T} |] ^{T}
  = [| w_1: [Q_1, P_1] ∧ w_2: [Q_2, P_2] |] ^{T}
  = **merge**(w_1, w_2): [(∃ w_2. Q_1) ∧ (∃w_1.Q_2), P_1∧P_2]

- If both of the operation expressions are in the form of promotion, then we translate the conjunction to sequential composition as follows:

  [|OperationExpression$_1$ ∧ OperationExpression$_2$ |]^T =
  [|OperationExpression$_1$|] ^{T}; [|OperationExpression$_2$|] ^{T}

5) OperationExpression$_1$ || OperationExpression$_2$

We consider two cases for proposing translation function of this operation expression as follows:

- If neither of the operation expressions are in the form of promotion, and also the mapping of both of them are in the form of specification statement "w: [pre, post]" (we consider the left side of the conjunction as w_1: [Q_1, P_1] and the right side as w_2: [Q_2, P_2]), by considering the semantics of this version of parallel composition, we should hide those variables (i.e., communicating variables) which are inputs in one of the operation expressions and outputs in the other one and then equate corresponding variables; hence, we use the notion of renaming to equate these variables, hiding them in precondition and postcondition and subtracting them from frame variables. Below, *inputs* returns the set of input variables of an OZ operation, and *outputs* returns the set of output variables. The translation function is as follows:

  [| OperationExpression$_1$ || OperationExpression$_2$ |]^T
  = [| [|OperationExpression$_1$|]^T ||
  [|OperationExpression$_2$|] ^{T} |] ^{T}
  = [| w_1: [Q_1, P_1] || w_2: [Q_2, P_2] |] ^{T}
  = (w_1, w_2) - ({x_1, … , x_n} ∪ {y_1, … , y_m}) :
  [∃z_1, …, z_{n+m} •
  (∃ w_2. Q_1[z_1\$x_1$, …, z_n\$x_n$, z_{n+1}\$y_1$,…, z_{n+m}\$y_m$]) ∧
  (∃w_1.Q_2[z_1\$x_1$, …, z_n\$x_n$, z_{n+1}\$y_1$, …, z_{n+m}\$y_m$]),
  ∃z_1, …, z_{n+m} •
  (P_1 [z_1\$x_1$, …, z_n\$x_n$, z_{n+1}\$y_1$,…, z_{n+m}\$y_m$] ∧
  P_2[z_1\$x_1$, …, z_n\$x_n$, z_{n+1}\$y_1$, …, z_{n+m}\$y_m$])]

  Where {y_1, …, y_m} = *inputs* (op_2) ∩ *outputs* (op_1), and {x_1, …, x_n} = *inputs* (op_1) ∩ *outputs* (op_2).

- If both of the operation expressions are in the form of promotion, the translation is done similar to what we did for case 2 of conjunction.

6) OperationExpression$_1$ [] OperationExpression$_2$

We consider two cases for proposing translation function of this operation expression as follows:

- If neither of the operation expressions are in the form of promotion, and also the mapping of both of them are in the form of specification statement "w: [pre, post]" (we consider the left side of the conjunction as w_1: [Q_1, P_1] and the right side as w_2: [Q_2, P_2]), we use *choose* to simulate the nondeterminism selection between operation expressions. Thus, selecting the operation expression which must be executed will be done based on value r (see follows; note that parameter p that, generally speaking, shows the possibility of choosing of operation expressions must be determined). The translation function is as follows:

  [|OperationExpression$_1$ [] OperationExpression$_2$|]^T
  = [| [|OperationExpression$_1$|]^T []
  [|OperationExpression$_2$|] ^{T} |] ^{T}
  = [| w_1: [Q_1, P_1] [] w_2: [Q_2, P_2] |] ^{T}
  = **var** r: ℕ, **choose** r · **if** (r<p ∧ Q_1 → w_1: [Q_1, P_1] []
  r<p ∧ ¬Q_1→ w_2: [Q_2, P_2] []

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

r>=p ∧ Q_2→w_2: [Q_2, P_2] []
r>=p ∧ ¬Q_2→ w_2: [Q_2, P_2]
fi

- If both of the operation expressions are in the form of promotion, the mapping is as follows:

[|$OperationExpression_1$ [] $OperationExpression_2$|]^T
= **var** r: ℕ, **choose** r •
if r<p ∧ [| **pre** ($OperationExpression_1$) |]^T→
[| $OperationExpression_1$|]^T []
r<p ∧ ¬ [| **pre** ($OperationExpression_1$) |]^T →
[| $OperationExpression_2$|]^T []
r>=p ∧ [| **pre** ($OperationExpression_2$) |]^T →
[| $OperationExpression_2$|]^T []
r>=p ∧ ¬ [| **pre** ($OperationExpression_2$) |]^T →
[| **pre** ($OperationExpression_1$) |]^T
fi

7) $OperationExpression_1$;$OperationExpression_2$

The same notion of sequential composition exists in MRC. Thus, the translation function is as follows:

[| $OpeartionExpression_1$; $OperationExpression_2$ |]^T =
[|$OperationExpression_1$|]^T ; [|$OperationExpression_2$|]^T

8) Expression.Identifier

We propose the translation function for when *Expression* is a variable with type class schema, and *Identifier* is an operation name (i.e., operation promotion) as follows:

[|Expression.Identifier|]^T= Identifier (Expression, opi)

where opi denotes other parameters of *Identifier*, i.e., the parameter list of the operation whose name is *Identifier* except ClassName_index.

Until now, we have reached at the following model in order to model the concept "object" in MRC:

type identity ::= null | REF ℕ

type B ≙ False | True

type ClassName_state ≙ {[|State.Declaration |]^T, Init: B,
self: identity, [|$inheritedClass_0$.State.Declaration|]^T,…,
[| $inheritedClass_m$.State.Declaration|]^T,
|[|State.predicateList|]^T ∧
[|$inheritedClass_0$.State.predicateList |]^T∧ …∧
[| $inheritedClass_m$.State.predicateList |]^T ∧ Init ⇔
([|InitialState.predicateList |]^T ∧
[|$inheritedClass_0$.InitialState.predicateList|]^T∧ …∧
[|$inheritedClass_m$.InitialState.predicateList |]^T)}

type State_Object
{$ClassName_0$_reftab:identity⤔$ClassName_0$_state, …,
$ClassName_n$_reftab: identity ⤔ $ClassName_n$_state |
$\bigcap_{i=1}^{n}$ **dom** $ClassName_i$_reftab = ∅ ∧
$\bigcup_{i=1}^{n}$ **dom** $ClassName_i$_reftab⊆ ran REF }

Also, we must import each operation which is promoted in the class schema, in its corresponding module.

In order to model dynamic instantiation of objects in MRC, we consider a procedure whose name is *ClassName_new.* This procedure has one parameter whose name is *obj* with type identity and in the form of call by value result (considering value result is due to the semantics of object in OZ [4]). This parameter is a new object identity that must be instantiated. Before instantiating the object of *ClassName*, we must instantiate all of the objects which are state variables in *ClassName*. Then, this procedure checks whether *obj* is a new identity and is not null. Also, *ClassName_new* considers a variable with type *ClassName_state* for *obj* which is in the initial state of *ClassName* and adds *obj* along with its associated *ClassName_state* value to *ClassName* reference table (i.e., *ClassName_reftab)*.

As we said earlier, we consider a variable *ClassName_reftab* in each module. After the instantiation of the new object, its object identity and *ClassName_state* value must be added to *ClassName_reftab*; hence, we consider a new procedure whose name is *ClassName_synch* in each module (which is obtained through the mapping of class schema). This procedure adds object identity and its *ClassName_reftab* (they are *ClassName_synch* parameters) to *ClassName_reftab* in the mentioned module. We call this procedure in *ClassName_new* after the instantiation of the new object. The complete model of modules *object* and *ClassName* in MRC are as follows (we consider *ClassName_reftab* = Ø as the initial predicate):

module object
import $ClassName_0$_synch, …, $ClassName_n$_synch
var state:State_Object
procedure $ClassName_0$_new (**value result** obj:identity)
≙
var r_1: identity
$ClassName_i$_new ($index_i$); …;$ClassName_j$($index_j$);
state: [obj≠ null ∧
obj ∉(**dom** $ClassName_0$_reftab ∪ … ∪
dom $ClassName_n$_reftab),
∃ r:$ClassName_0$_state. r.self=obj ∧ r.Init ∧ state=
{$state_0$.$ClassName_0$_reftab ∪ {obj ↦r},
$state_0$.$ClassName_1$_reftab, … ,
$state_0$.$ClassName_n$_reftab} ∧ r_1 = r] ;
$ClassName_0$_synch (obj, r_1)
⋮
procedure $ClassName_n$_new (**value result** obj:identity)
≙…
end

module ClassName
export DeclarationName which is in the visibility list and is an operation name.

import operations which are used in operation promotion in class schema ClassName and also all of the procedures of those module(s) corresponding to parent class(es).

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

var ClassName_reftab: identity ↣ ClassName_state

procedure ClassName_synch (**value** obj: identity, **value** r:ClassName_state)
ClassName_reftab: [true, ClassName_reftab = ClassName_reftab$_0$ ∪ (obj↦r)]

procedure ClassName_OperationSchemaName …

procedure ClassName_OperationName…

initially ClassName_reftab=∅
end

After mapping OZ specifications to MRC specifications using our translation function and model of concept “object”, tools such as RED [17] can be used in order to automate the refinement of resulting MRC specifications to final code.

V. Case study

Using the translation function described in the previous section, the following MRC specification is obtained from class schema “Queue” given in Fig. 1.:

type B ≜ True | False

type identity ≜ null |REF ℕ

type Queue_state ≜ {[|items: Seq ℕ, count: ℕ|]^T, Init: B, self: identity | Init ⇔ ([| items=<> ∧ count=0 |]^T) }

module Queue
export Queue_Join, Queue_Leave
var Queue_reftab: identity ↣ Queue_state

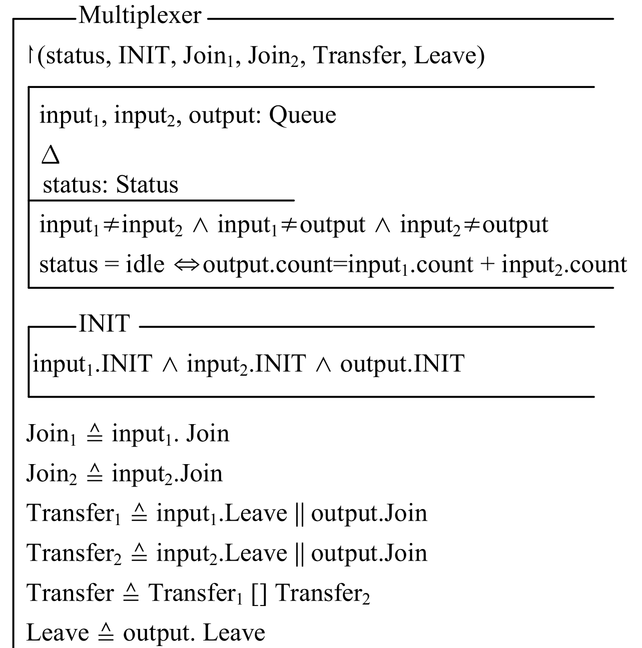
procedure Queue_synch (**value** obj: identity, **value** r:Queue_state) Queue_reftab: [true, Queue_reftab = Queue_reftab$_0$ ∪ (obj↦r)]

procedure Queue_Join (**value** Queue_index: identity, **value** [|item: ℕ|]^T)≜
Queue_reftab: [[| ∃ items′: Seq ℕ, count′: ℕ, item?:ℕ. items′=items ⁀ <item?> ∧ count′=count + 1|]^T, [| items′=items ⁀ <item?> ∧ count′=count+1|]^T]

procedure Queue_Leave (**value** Queue_index:identity, **value result** [|item:ℕ|]^T)≜
Queue_reftab, item: [[| ∃ items´: Seq ℕ, count´: ℕ, item!:ℕ. items= <item!> ⁀ items′|]^T, [| items= <item!> ⁀ items′|]^T]
end

To apply our approach for considering objects and the related framework in MRC, we map class schema “multiplexer” [4] which is as follows:

Status ::= idle | busy


Multiplexer
↾(status, INIT, $Join_1$, $Join_2$, Transfer, Leave)

$input_1$, $input_2$, output: Queue
Δ
status: Status

$input_1 \neq input_2 \wedge input_1 \neq output \wedge input_2 \neq output$
status = idle ⇔ output.count=$input_1$.count + $input_2$.count

INIT
$input_1$.INIT ∧ $input_2$.INIT ∧ output.INIT

$Join_1$ ≜ $input_1$. Join
$Join_2$ ≜ $input_2$.Join
$Transfer_1$ ≜ $input_1$.Leave || output.Join
$Transfer_2$ ≜ $input_2$.Leave || output.Join
Transfer ≜ $Transfer_1$ [] $Transfer_2$
Leave ≜ output. Leave

Mapping of this class schema is as follows:

type Status ≜ idle | busy

type Multiplexer_state ≜ {[|$input_1$, $input_2$, output: identity, status: Status|]^T, Init: B, self: identity |
[| $input_1 \neq input_2 \wedge input_1 \neq output \wedge input_2 \neq output$ ∧ status=idle⇔output.count = $input_1$.count + $input_2$.count |]^T ∧ Init ⇔
([| $input_1$.INIT ∧ $input_2$.INIT ∧ output.INIT |]^T)}

module Multiplexer
export Multiplexer_Join$_1$, Multiplexer_Join$_2$, Multiplexer_Transfer, Multiplexer_Leave
import Queue_Join, Queue_Leave
var Multiplexer_reftab: identity ↣ Multiplexer_state

procedure Multiplexer_synch (**value** obj: identity, **value** r:Multiplexer_state) Multiplxer_reftab: [true, Multiplexer_reftab = Multiplexer_reftab$_0$ ∪ (obj↦r)]

procedure Multiplexer_Join$_1$ (**value** Multiplxer_index: identity, **value** Queue_index:identity, **value** [| item: ℕ|]^T)
≜ Queue_Join ($input_1$, item)

procedure Multiplexer_Join$_2$ (**value** Multiplxer_index: identity, **value** Queue_index:identity, **value** [| item:ℕ|]^T)
≜ Queue_Join ($input_2$, item)

procedure Multiplexer_Leave (**value** Multiplexer_index: identity, **value** Queue_index:identity, **value result** [| item: ℕ|]^T)
≜ Queue_Leave(output, item)

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100

procedure Multiplexer_Transfer$_1$ (**value** Multiplexer_index: identity)≙
Queue_Leave(input$_1$, item) ; Queue_Join(output, item)

procedure Multiplexer_Transfer$_2$ (**value** Multiplexer_index:identity)≙
Queue_Leave(input$_2$, item) ; Queue_Join(output, item)

procedure Multiplexer_Transfer (**value** Multiplexer_index:identity)≙
var r:ℕ, **choose** r• **if** r<1000 ∧ [|**pre**(Transfer$_1$)|]^T →
Multiplexer_Transfer$_1$(Multiplexer_index) []
r<1000 ∧ ¬[|**pre**(Transfer$_1$)|]^T →
Multiplexer_Transfer$_2$(Multiplexer_index) []
r>=1000 ∧ [|**pre**(Transfer$_2$)|]^T →
Multiplexer_Transfer$_2$(Multiplexer_index) []
r>=1000 ∧ ¬[|**pre**(Transfer$_2$)|]^T→
Multiplexer_Transfer$_1$(Multiplexer_index) **fi**
end

Object framework is as follows:

type State_Object ≙
{Queue_reftab: identity⤔Queue_state,
Multiplexer_reftab: identity⤔ Multiplexer_state |
(**dom** Queue_reftab ∩ **dom** Multiplexer_reftab) = Ø ∧
(**dom** Queue_reftab ∪ **dom** Multiplexer_reftab) ⊆
ran REF }

module object
import Queue_synch, Multiplexer_synch
var state:State_Object
procedure Queue_new (**value result** obj:identity) ≙
var r$_1$
state: [obj≠ Null ∧ obj ∉(**dom** Queue_reftab ∪ **dom** Multiplexer_reftab),
∃ r:Queue_state. r.self=obj ∧ r.Init ∧ state=
{state$_0$.Queue_reftab ∪ {obj ↦r},
state$_0$.Multiplexer_reftab } ∧ r$_1$ = r] ; Queue_synch (obj, r$_1$)

procedure Multiplexer_new (**value result** obj:identity) ≙ **var** r$_1$
Queue_new(state$_0$.input$_1$);Queue_new(state$_0$.input$_2$);
Queue_new(state$_1$.output) ;state: [obj≠ Null ∧ obj ∉(**dom** Queue_reftab ∪ **dom** Multiplexer_reftab),
∃ r:Multiplexer_state. r.self=obj ∧ r.Init ∧ state=
{state$_0$.Multiplexer_reftab ∪ {obj ↦r},
state$_0$.Queue_reftab} ∧ r$_1$ = r] ; Multiplexer_synch (obj, r$_1$);
end

VI. Conclusions and Future Work

In this paper, we proposed a methodology for refinement of OZ specifications using MRC. First, we proposed a translation function for mapping OZ constructs into their MRC counterparts, and then we modeled concept "object" in MRC. Finally, we proposed how to model dynamic instantiation of objects in MRC while OZ itself does not cover this facility. In this way, in order to refining OZ specifications, one should first map an OZ specification into its equivalent specification in MRC using our translation function and our model of concept "object". Then, the resulting MRC specification can be refined automatically using MRC refinement tools such as RED.

Unlike existing animation approaches [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12], our methodology pays attention to the correctness using MRC refinement laws, and also our translation function covers mapping of abbreviations, free types and basic types in global. Comparing our methodology with the refinement approach of [13, 14], our methodology supports fine-grained refinement using MRC laws. Of course, Morgan's Refinement framework is open ended so that makes it easy to add coarse-grained refinement laws (e.g. module refinement law) to it. Finally, we can use tools such as RED to automate the refinement of OZ specifications using MRC indirectly.

Nevertheless, some other constructs of OZ, such as distributed operators, other cases of abbreviation definition like when the right side of the abbreviation is in the form of set definition, generic definition and finally other cases of free types (when constructors are used in the definition), are still open considering our mapping process.

In our future work, we are going to

1. extend our translation function in order to map the remaining constructs of Object-Z,
2. show the applicability of our method using a new case study including a larger subset of Object-Z,
3. show the automation of our methodology using one of MRC tools such as RED, and finally
4. add refinement laws for module construct in MRC.

References

[1] S. Ramkarthik, and C. Zhang, "Generating java skeletal code with design contracts from specifications in a subset of object-z, "in 5th IEEE/ACIS International Conference on Computer and Information Science, , pp. 405-411, 2006.
[2] X.Ni, and C.Zhang, "Converting specifications in a subset of object-z to skeletal spec# code for both static and dynamic analysis, " in Journal of Object Technology, Vol. 7, No. 8, pp.165-185, 2008.
[3] T.McComb, and G.Smith, "Animation of object-z specifications using a z animator, "in First International Conference on Software Engineering and Formal Methods, pp. 191-200, 2003.
[4] G. Smith, The Object-Z Specification Language: Advances in Formal Methods, Kluwer Academic Publishers, 2000.
[5] R. Duke, and G. Rose, Formal Object-Oriented Specification Using Object-Z, Macmillan, UK, 2000.
[6] G. Rafsanjani, and S. J. Colwill, "From object-z to c++: a structural mapping," in Z User Meeting (ZUM'92), Springer-Verlag, pp. 166-179, 1992.
[7] M. Fukagawa, T. Hikita, and H. Yamazaki, "A mapping system from object-z to c++," in 1st Asia-Pacific Software Engineering Conference (APSEC94), IEEE Computer Society Press, pp. 220-228, 1994.

[8] W. Johnston, and G. Rose," Guidelines for the manual conversion of object-z to c^{++}, " in SVRC Technical Report 93-14, 1993.
[9] M. Najafi, and H. Haghighi, "An animation approach to develop c^{++} code from object-z specifications, " in International Symposium on Computer Science and Software Engineering, pp. 9-16, 2011.
[10] C. Fischer, Combination and implementation of processes and data: from csp-oz to java, PhD Thesis. University of Oldenburg, 2000.
[11] Z.Wang, M.Xia, and Y.Zhao, "Transform mechanisms of object-z based formal specification to java," in International Conference on Computational Intelligence and Software Engineering, pp. 1-4, 2009.
[12] A. Griffiths, "From object-z to eiffel: a rigorous development method," in Technology of Object-Oriented Languages and Systems: TOOLS 18, Prentice-Hall, 1995.
[13] J. Derrick, and E. A. Boiten, "Refinement of objects and operations in object-z, " in Formal Methods for Open Object-based Distributed Systems IV, pp. 257-277, Kluwer Academic Publishers, 2000.
[14] J. Derrick, and E. A. Boiten, Refinement in z and object-z: foundations and advanced applications, Formal Approaches to Computing and Information Technology (FACIT), 1st edition, Springer-Verlag, 2001.
[15] C. Morgan, Programming from specifications, Prentice Hall, 1990.
[16] J. Woodcock, and J. Davies, Using z: specification, refinement, and proof, Prentice-Hall, 1996.
[17] D. A. Carrington, and K. A. Robinson, "Tool support for the refinement calculus, "in Computer-Aided Verification, Vol. 3 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 381-394, American Mathematical Society, 1991.

International Science Index, Computer and Information Engineering Vol:5, No:11, 2011 waset.org/Publication/2100