A Mapping System from Object-Z to C++

M. Fukagawa T. Hikita * H. Yamazaki †
Dept. of Computer Science
Meiji University
Higashimita, Tama-ku, Kawasaki 214, Japan

Abstract

Object-Z is an extension of the formal specification language Z, augmenting the class concept as a structuring facility. This paper introduces and discusses a structural mapping system from Object-Z to the programming language C++, and reports on its implementation on UNIX. The structural mapping translates an Object-Z specification consisting of classes into class interfaces of C++ such as data members and prototypes of member functions. Thus it is not intended as a code generation system, but rather as a tool for analyzing specification (including syntax and type checking) and for aiding a software developer in obtaining code. Through the implementation of the mapping system several language features of Object-Z and C++ concerning object-orientation are clarified.

1 Introduction

The formal specification language Z is gaining popularity, and is accessible in several textbooks [9]. Object-Z is based on Z, augmenting the class concept as a structuring facility [3], [11]. Object-orientation in formal specification is one of the active research areas in software engineering [2], [11]. The programming language C++ is one of the more popular object-oriented languages [4].

This paper discusses a structural mapping system from Object-Z to C++. The idea of structural mapping was initially proposed by Rafsanjani and Colwill [8]. The structural mapping translates classes of an Object-Z specification into classes (more precisely, class interfaces) of C++ such as data members and headers of member functions. Thus it is not intended as a code generation system, but rather as a tool for analyzing specification (including syntax and type checking) and for aiding a software developer in obtaining code from specification. We consider that the usefulness of such a mapping system is fairly obvious as a tool for developing both specification and code.

In [8] the basic rules of the mapping were described, which were obtained through case studies. Their rules seem natural in regard to the language facilities of Object-Z and C++, and we employ their rules. We have implemented the major part of the mapping system, on which we report here. The implementation helped us to understand several subtle points of the mapping, especially those related to types and classes.

Our main contributions in this work are practical. Our implementation of a mapping system on Unix utilizes well-established compiling techniques for programming languages, with tools lex and yacc. Main points of the implementation are the treatment of generic parameters for generic schemas and classes, and that of state variables and member functions under inheritance. The implementation is still an ongoing project.

In the next section 2 we briefly review the languages Object-Z and C++. Section 3 offers a simple example of Object-Z specification and its mapping in C++. In section 4 we discuss the rules of structural mapping from Object-Z to C++, especially those related to types, predefined symbols of Z, and class inheritance. In section 5 several important points of the implementation of the mapping system are shown. Concluding remarks and future work are in section 6. Finally, in appendix two more examples of the mapping are included.

2 Object-Z and C++

We here assume the reader the basic knowledge of Z, Object-Z and C++. However, we very briefly review and summarize some of their important language features mainly concerning object-orientation, which will be needed for our realization of a mapping sys-

*e-mail: hikita@cs.meiji.ac.jp
†Currently, Hitachi Software Engineering, Ltd.

0-8186-6960-8/94 $04.00 © 1994 IEEE

tem.

2.1 Object-Z

Object-Z is an extension of Z, augmenting the class concept as a structuring facility. The structure of an Object-Z class is as Fig. 1, having as components constants, types, state schemas and operation schemas of Z. The concept of (multiple) class inheritance is available by specifying base classes.

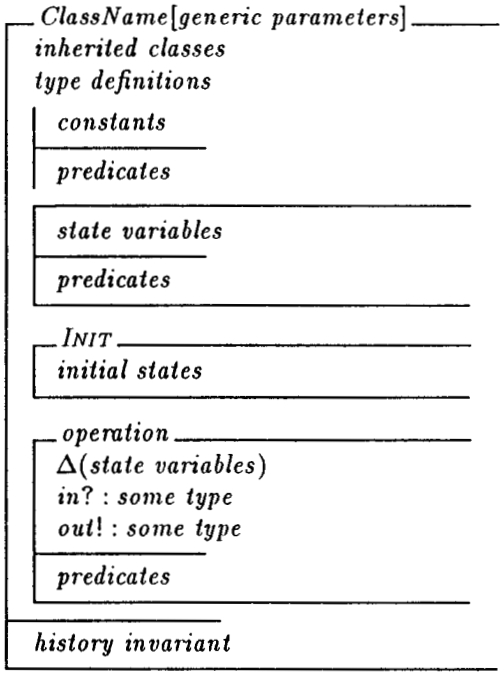


Figure 1: **Object-Z specification**

Notice in this form that there are small differences between the forms of schemas in a class of Object-Z and those of Z. Firstly, a state schema in an Object-Z class does not have a schema name. Secondly, the form of an initialization schema is different from Z. And thirdly, in the Δ notation in operation schemas (which indicates the change of values of states by the operation), each state to be modified is specified in Object-Z, while in Z the state schema name is specified in a Δ list.

The semantics of Z and Object-Z is based on mathematical set theory, and objects and classes are treated as (named) sets.

In order to realize polymorphism in Object-Z, define an object as follows: $Obj :\downarrow C$. Then, one can assign to *Obj* not only objects of class C but also those of classes derived from C. Let D be a class derived from C, and let *Op* be the single name of the operations each belonging to C and D. Then, the operation $Obj.Op$ of the object bound at the time of execution is automatically selected.

2.2 C++

The programming language C++ is an extension of C, and many language features mainly related to object-oriented paradigm are added [4].

There are three parts in the members of a class of C++ concerning accessibility to these from outside: *public*, *protected* and *private*. And also, one can control the accessibility in a derived class when inheriting the members of a base class, using the same keywords *public*, *protected* and *private*. So that there are several, somewhat complicated, combinations of controling the accessibility of members in a class. And it also has a *friend* feature.

A pointer to an object of a derived class can be coerced to that pointing to the object of a base class, and vice versa. Especially, if one declares a member function of a base class to be *virtual*, when this function is called, the actual member function in a derived class is selected according to the class to which the object belongs, and thus polymorphism is realized.

Multiple inheritance is realized by declaring a base class to be a *virtual* base class.

2.3 Object models of Object-Z and C++

Rafsanjani and Colwill [8] clarified the essence and differences of the class concepts of the languages Object-Z and C++, summarizing these as stated in the above, and discussed object models of these two languages.

Concerning the hiding of members of a class, these two languages differ. The present version of Object-Z has no hiding facility and every member of a class is public, although it seems that discussion concerning this is going on. On the other hand, C++ has notably complicated hiding facility for members of a class as stated before.

3 Structural Mapping and Its Example

3.1 Structural mapping

Structural mapping translates an Object-Z specification into class interfaces of C++ programs. In gen-

eral, automatic code generation from specifications is a strongly desirable tool for software development, but in the case of Z and Object-Z it is not an easy task to construct such a tool. The automatic transformation from predicates on state variables and operations to code essentially needs some kinds of theorem proving techniques.

Our argument here is that the automatic generation of only class interfaces of code are sufficiently useful when translating specification to code by hand. Even in the specification phase it may be argued that declarations of states, operations and their types are essentially more important than fully specifying their conditions and invariants.

The byproducts of a mapping system are lexical, syntax and type checking facilities of specifications. These static checking are important tools, and Spivey's fuzz is such an example for Z [10].

3.2 An example of mapping

As an illustration of structural mapping, we here give an example of an Object-Z specification and its corresponding C++ code of declarations generated by our mapping system.

In the following are shown a specification of a birthday book, which is originally written in Z ([9], chap. 1) and is slightly modified in order to conform to Object-Z syntax. A class *BirthdayBook* is introduced which contains a state schema and three operation schemas. This is mapped to a C++ class, consisting of class interfaces, *i.e.* declarations of data members and headers of member functions of the class.

Brief explanations for some crucial points are in order. Classes `Power` and `PFun` (actually, templates) are prepared in C++ as a class library for realizing the Object-Z operators related to power sets and partial functions, respectively. In these classes other related Z symbols and operators like *dom* and # are prepared.

Note that all member functions (corresponding to Object-Z operations) are declared as virtual functions in C++. Also note the suffixes "_q" and "_x" of the parameters of the operations `AddBirthday` and the others. They correspond to the decorations " ? " and " ! " of variables in operation schemas of Object-Z.

In Appendix of this paper there are given two more examples of Object-Z specifications and the results of their mappings in C++. The first one shows a generic class $Stack[T]$, its derived class $IndexedStack[X]$, and the mapping of these two classes (this example of specification is taken from [3]). The second example is the mapping of multiple inheritance among classes.

$[NAME, DATE]$

BirthdayBook

$known : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\, birthday$

INIT

$known = \emptyset$

AddBirthday

$\Delta(known, birthday)$
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

FindBirthday

$name? : NAME$
$date! : DATE$

$name? \in known$
$date! = birthday(name?)$

Remind

$today? : DATE$
$cards! : \mathbb{P}\, NAME$

$cards! = \{n : known \mid birthday(n) = today?\}$

```
#include "GlobalDefs.h"
class BirthdayBook{
protected:
//DeclPart
  Power< NAME > known;
  PFun< NAME, DATE > birthday;
public:
  BirthdayBook();
                // Null Constructor
  BirthdayBook(BirthdayBook& the_BirthdayBook);
                // Copy Constructor
  virtual ~BirthdayBook();
                // Destructor
  BirthdayBook&
    operator = (BirthdayBook& the_BirthdayBook);
                // Assignment Operator
  virtual void
    AddBirthday(NAME& name_q, DATE& date_q);
  virtual void
    FindBirthday(NAME& name_q, DATE& date_x);
  virtual void
    Remind(DATE& today_q, Power< NAME >& cards_x);
};
```

We have also tested, as a larger example, an Object-Z specification of the so-called Library Problem, developed in [6], [7]. (This problem is originally in "Problem Set for the Fourth Int. Workshop on Software Specification and Design, Monterey, USA, 1987.")

4 Structural Mapping

4.1 Basic rules

In [8] Rafsanjani and Colwill stated basic rules for the structural mapping from Object-Z to C++, which were obtained through their three case studies of rewriting Object-Z specifications to C++ code by hand. Since the rules seem natural and straightforward in regard to the current language facilities of Object-Z and C++ concerning object-orientation, we employ them. They are as follows (item 6 is new).

1. Constants and state variables in a class are mapped into the protected part of a C++ class.
2. All inheritances in Object-Z are mapped to public inheritances of C++.
3. In the case of multiple inheritance, a base class is mapped to a virtual base class of C++.
4. Operations in Object-Z classes are mapped to virtual functions in C++. The return values of the functions are of type void, and their parameters are passed by reference.
5. For each class of C++, a null constructor, a copy constructor, a destructor, an assignment operator, and invariants for constants are always supplied.
6. Constructors for types of constants are always supplied.

In item 1 the intention is that data members of a C++ class be encapsulated from outside, but in some cases it would be more preferable to map constant and state variables of an Object-Z class to the public part of a C++ class.

4.2 Types

The type system of Object-Z (or Z) is based on set theory, utilizing the set construction operations of power set, cartesian product, function space and schema types. This is semantically clear. However, power set and function space do not have direct counterparts in C++.

Moreover, all of these are generic. But genericity itself can be realized in C++ by the template construct. The generic symbol of power set is realized as a predefined class in C++, as

```
template <class T> class Power
```

Thus, our solution for realizing these type construction methods in C++ is simple; we map each of these type constructions directly to a (template) class of C++. There are of course many ramifications of actually realizing these classes in C++, which differ to each other in simplicity and efficiency. Some examples of class realization of function space are found in [6]. Apart from a mapping system, such a class library for Z operations seems important and desirable.

4.3 Inheritance

Object-Z allows multiple inheritance (like Fig. 2), and it should be realized in the mapping. In multiple inheritance in Object-Z specification, one has to determine a common ancestor class as a base class, starting from the far ends of derived classes. This procedure also applies in the case of multiple inheritance among generic classes. Appendix B shows an example of mapping for the case of multiple inheritance as Fig. 2.

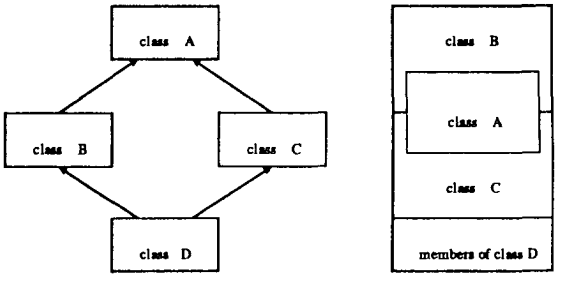


Figure 2: **Multiple inheritance**

When a class is an instance of a generic class, we must take into account the actual types for generic parameters. When determining a common ancestor class, instances of a generic class having exactly the same actual parameters of types are considered identical. Thus in this case we need a procedure for type equivalence checking in Object-Z.

In the example of derived class *IndexedStack* in Appendix A the member functions `Push` and `Pop` in the corresponding class in C++ have the correct parameters though they are omitted in their specifications. Redefinition of operations in inheritance should also

be treated. In the case of inheritance in Appendix A no overloading occurs, so that we simply adopt as definition that of a base class.

5 Implementation of a Mapping System

5.1 Source format of Object-Z specification

We must choose a machine-readable format of Object-Z specifications for a mapping system. As a source text we here employ LaTeX source. It may contain plain texts other than proper Object-Z specifications as informal explanations.

The reason that we employ LaTeX format should be obvious. Tex style files for Z and Object-Z are already available, which can handle special symbols of Z and vertical and horizontal lines for boxes.

Z uses style file "fuzz.sty" [10], and Object-Z uses "oz.sty" [5]. These two use almost the same symbol names of Z. However, some symbols have different names in the two style files; the power set symbol $\mathbf{P}$ is `\power` in fuzz.sty, while it is `\pset` in oz.sty. But these differences can be easily absorbed in the scanning phase of the mapping process (explained later).

We found that an environment for a generic class of Object-Z is not included in oz.sty, so that we have prepared another style file "oz2.sty," which the specification writer should add. Thus, when preparing an Object-Z specification for the mapping system, one has to write in LaTeX:

```
\documentstyle[fuzz,oz,oz2]
  ..............
```

5.2 General plan of implementation

We use for preprocessing and scanning the lexical analyzer generator lex. And we also use for syntax analysis and transformation the syntax analyzer generator yacc. (Actually we used flex and bison, instead of lex and yacc, respectively.) Semantical functions for manipulating a name table and generating C++ code segment are written in C++.

In the current implementation of the mapping system the sizes of source code are approximately: 300 lines of lex text, 1,100 lines of yacc text, and 2,100 lines of semantic functions in C++.

The mapping process is as in Fig. 3.

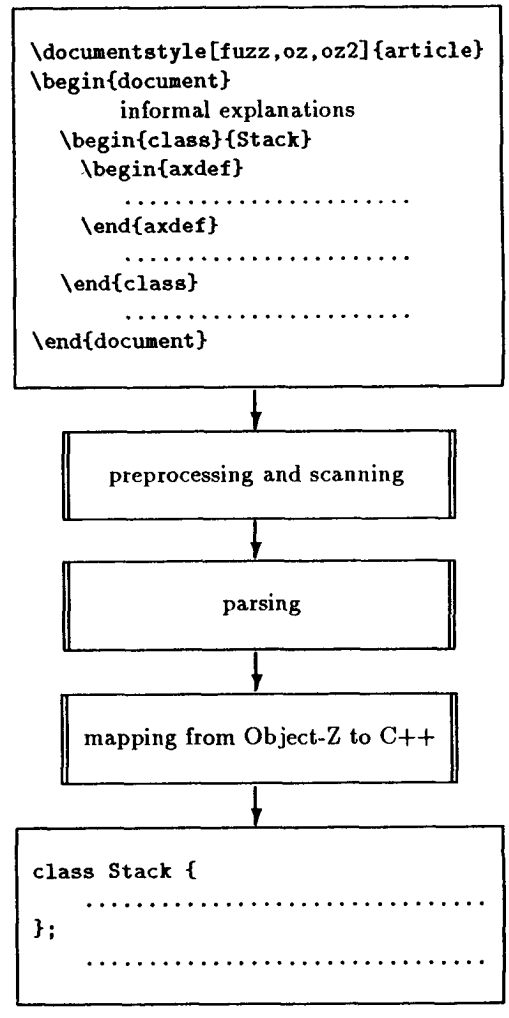


Figure 3: Mapping

5.3 Preprocessing and scanning

As preprocessing for mapping, some parts in input specifications are removed, which are the following.

1. LaTeX commands such as `\documentstyle{}` and `\begin{document}`.
2. Informal verbal explanations other than Object-Z specification. In LaTeX form they are the parts that are not within environments such as `\begin{class} ...\end{class}` and `\begin{zed} ...\end{zed}`.
3. Comments within specifications. They are in `\comment{...}`, `\comment*{...}`, and `\begin{zpar} ... \end{zpar}`.

4. Newline symbols (\\) that specify infix operators.

At the preprocessing phase all of these in the above are removed by lex.

5.4 Name table and related data structures

All the names appearing in an input specification in Object-Z are recorded in the name table. The structure of the name table is as in Fig. 4. For efficiency all the names can be accessed through a hashtable.

Global names (especially class names) are preserved in a linear list in the order they appear in the source specification (in order to check the scope of the names). Thus global names can be accessed both through a linear list and by hashing. Local names in classes and schemas are regarded as attributes of a global name of a class or a schema they belong to, and are linked in a list starting from a global name.

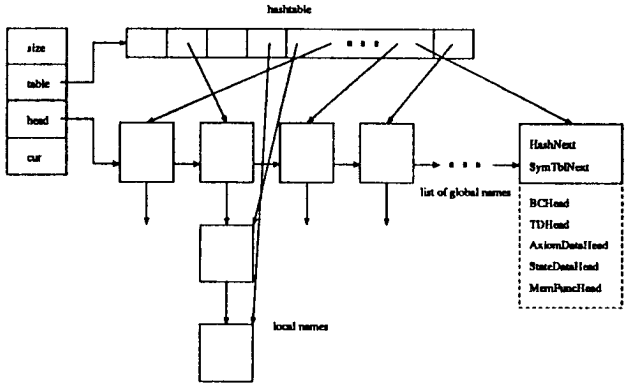


Figure 4: **Structure of name table**

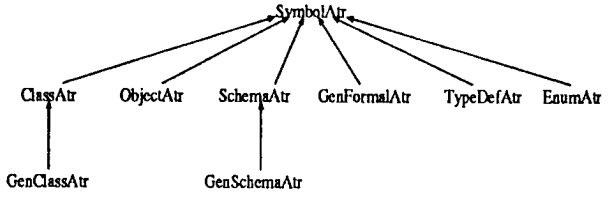


Figure 5: **Classes of name table entries**

There are several kinds of entries (class names, state names, operation names, ...) in the name table. We classify and describe these in C++ as a class hierarchy as in Fig. 5. We show for reference in Fig. 6 through Fig. 8 the contents of the entries of three particular kinds. The class `GenClassAtr` is derived from `ClassAtr`, which in turn is derived from `SymbolAtr`.

We also show here an important structure `BCNode` in Fig. 9 which is used to maintain base classes of a class.

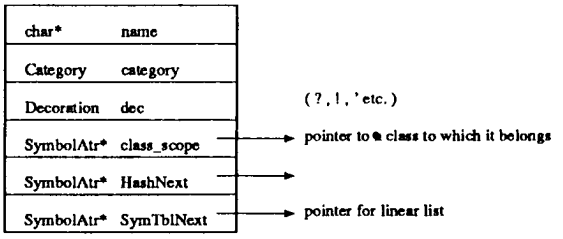


Figure 6: **Structure of SymbolAtr**

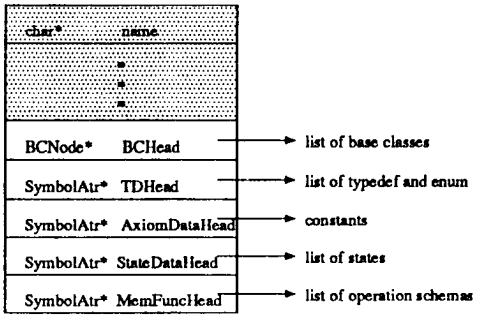


Figure 7: **Structure of ClassAtr**

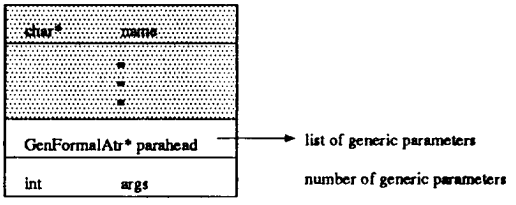


Figure 8: **Structure of GenClassAtr**

5.5 Parsing and mapping

We here show fragments of yacc texts parse.y in order to give the feeling of parsing and mapping. They include a part of syntax of Object-Z specification, and

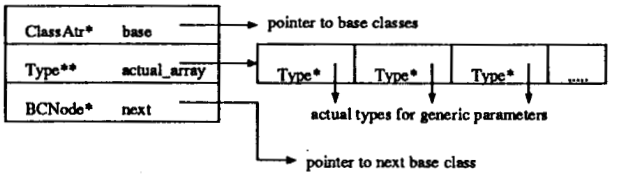


Figure 9: **Structure of BCNode**

also the semantic functions for actual mapping for each syntactic construct. The syntax rules of Z and Object-Z are taken from [9] and [3], respectively.

List 1 is the syntactic element `ClassBox`. In this part a class name entry is entered into the name table, and also the result of mapping is printed.

List 1 ClassBox in parse.y

```
1   ClassBox : BGNCLS LB Ident RB
2                {current_class
3                   = new ClassAtr
4                         ($<symbol_list>3);}
5              ClassField ClassBoxTail
6                {current_class->Print(0);
7                 id_table.insert(current_class);
8                 current_class = NULL;}
9            | error ENDCLS NL
10               {yyerror("class error\n");
11                current_class = NULL; yyerrok;}
12           ;
```

In List 2 constants, state variables and operations are entered into the name table as attributes of the class they belong to.

List 2 ClassField and Statement in parse.y

```
1   ClassField : Statement
2              | ClassField Statement
3              | error NL
4                  {yyerror("Statement error\n");
5                   yyerrok;}
6              ;
7   Statement  : /* empty */ NL
8                ..............................
9              | AxiomBox
10                 {current_class->
11                    insert_axiomdata
12                      ($<symbol_list>1);}
13             | StateBox
14                 {current_class->
15                    insert_statedata
16                      ($<symbol_list>1);}
17             | InitBox
18             | OpBox
19                 {current_class->
20                    insert_memberfunction
21                      ($<symbol_list>1);}
22             ;
```

6 Concluding Remarks

We have implemented a prototype version of a mapping system from Object-Z to C++. Our experience shows that the mapping system is a useful tool, when checking and analyzing Object-Z specification and rewriting specifications to C++ code.

The implementation of the mapping system is an on-going project, and several features of Object-Z (or rather Z) are yet unimplemented. These are: renaming, and several predefined function and relation symbols as a C++ class library. Also, our system lacks in some useful features in Object-Z and Z, especially those concerning schema calculus and specification refinement. The relationship between these Z features and structural mapping is left to be investigated, both theoretically and practically.

Acknowledgements

We thank K. Ishihata, S. Sano, and the referees for helpful comments on the manuscript.

References

[1] D. Duke and R. Duke, "Towards a semantics for Object-Z," in *VDM'90: VDM and Z : Formal Methods in Software Development*, LNCS, Vol. 428, pp. 244–261, Springer-Verlag, 1990.

[2] R. Duke, "Integrating formal methods with object-oriented software engineering," Proc. Joint Conf. on Software Engineering '93, pp. 3–10, 1993.

[3] R. Duke, P. King, G. Rose and G. Smith, "The Object-Z Specification Language: Version 1," TR 91-1, Dept. of Computing Science, Univ. of Queensland, Australia, 1991.

[4] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[5] P. King, "Printing Z and Object-Z LaTeX documents," Dept. of Computing Science, Univ. of Queensland, Australia, 1990.

[6] H. Miyazaki, K. Yatsu, M. Someya, S. Yamasaki and K. Kakehi, "The library problem (in Japanese)," in *Deductive Derivation of Programs, preliminary report*, pp. 239–362, Information Promotion Agency, Japan, 1992.

[7] H. Miyazaki, K. Yatsu, S. Yamada, H. Amano, H. Shibata, T. Hikita and H. Ishima, "Application of Object-Z to the library system (in Japanese)," in *Deductive Derivation of Programs*, pp. 51–221, Information Promotion Agency, Japan, 1993.

[8] G.-H. B. Rafsanjani and S. J. Colwill, "From Object-Z to C++: A structural mapping," in *Z User Workshop, London 1992*, pp. 166–179, Springer-Verlag, 1993.

[9] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1989; 2nd ed., 1992.

[10] J. M. Spivey, "The fuzz Manual," 2nd ed., 1992.

[11] S. Stepney, R. Barden and D. Cooper, eds., *Object Orientation in Z*, Springer-Verlag, 1992.

IndexedStack $[X]$

$Stack[X]$

$index : \mathbb{N}_1$

$items \neq \langle\rangle \Rightarrow index \in \mathrm{dom}\ items$

SetIndex

$\Delta(index)$
$n? : \mathbb{N}_1$

$n? \in \mathrm{dom}\ items$
$index' = n?$

Push

$\Delta(index)$

$items \neq \langle\rangle \Rightarrow index' = index + 1$

Pop

$\Delta(index)$

$index = 1 \land items' = \langle\rangle \Rightarrow index' = 1$
$index \neq 1 \Rightarrow index' = index - 1$

Appendix.

A Generic Class Stack

Stack $[T]$

$max : \mathbb{N}$

$items : \mathrm{seq}\ T$

$\#items \leqslant max$

INIT

$items = \langle\rangle$

Push

$\Delta(items)$
$item? : T$

$\#items < max$
$items' = \langle item? \rangle \frown items$

Pop

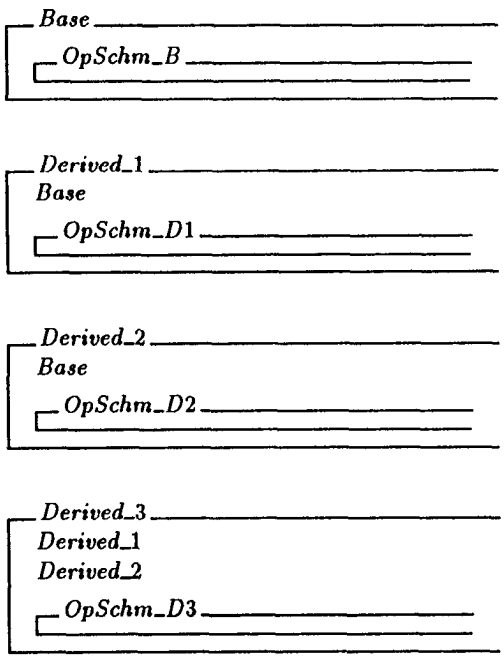
$\Delta(items)$
$item! : T$

$items \neq \langle\rangle$
$items = \langle item! \rangle \frown items'$

```
#include "GlobalDefs.h"
template < class T >
class Stack{
protected:
//AxiomPart
  unsigned int max;
//DeclPart
  Seq< T > items;
public:
  Stack();
  Stack(Stack< T >& the_Stack);
  Stack(unsigned int& the_max);
  virtual ~Stack();
  Stack&
    operator = (Stack< T >& the_Stack);
  int inv(unsigned int& the_max);

  virtual void Push(T& item_q);
  virtual void Pop(T& item_x);
};

template < class X >
class IndexedStack : public Stack< X >{
protected:
//DeclPart
  unsigned int index;
public:
  IndexedStack();
  IndexedStack
    (IndexedStack< X >& the_IndexedStack);
  IndexedStack(unsigned int& the_max);
  virtual ~IndexedStack();
```

```
  IndexedStack& operator
    = (IndexedStack< X >& the_IndexedStack);
  int inv(unsigned int& the_max);

  virtual void SetIndex(unsigned int& n_q);
  virtual void Push(X& item_q);
  virtual void Pop(X& item_x);
};
```

B Multiple Inheritance




```
#include "GlobalDefs.h"
class Base{
protected:
public:
  Base();
  Base(Base& the_Base);
  virtual ~Base();
  Base& operator
    = (Base& the_Base);
  virtual void OpSchm_B();
};
```

```
class Derived_1 : virtual public Base{
protected:
public:
  Derived_1();
  Derived_1(Derived_1& the_Derived_1);
  virtual ~Derived_1();
  Derived_1& operator
    = (Derived_1& the_Derived_1);
  virtual void OpSchm_D1();
};
class Derived_2 : virtual public Base{
protected:
public:
  Derived_2();
  Derived_2(Derived_2& the_Derived_2);
  virtual ~Derived_2();
  Derived_2& operator
    = (Derived_2& the_Derived_2);
  virtual void OpSchm_D2();
};
class Derived_3
  : public Derived_1, public Derived_2{
protected:
public:
  Derived_3();
  Derived_3(Derived_3& the_Derived_3);
  virtual ~Derived_3();
  Derived_3& operator
    = (Derived_3& the_Derived_3);
  virtual void OpSchm_D3();
};
```