25952

OOSPEC: An Executable Object-Oriented Specification Language

Mohammad N. Paryavi
IBD Informatics, Inc.
P. O. Box 10660
Kansas City, MO 64188-0660
parcom@applelink.apple.com

William J. Hankley
Dept. of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506
hankley@cis.ksu.edu

Abstract: OOSPEC is a model-based specification language and development system. It is intended as a small system for use in introducing formal specifications for undergraduates. OOSPEC combines features from several other specification and programming languages, particularly VDM , Z, Eiffel, Ada, and Smalltalk. The environment and implementation are based (loosely) on Smalltalk. The environment supports a direct manipulation graphical interface for creation and evaluation of partial and full specifications. Objects are defined as instances of class specifications. Object data items and methods are modeled in terms of high level structures such as sets, sequences, and maps. Class methods are defined by pre- and post-assertions, although post- "assertions" have the side effect of binding new values of "out" data items. Correct syntax of specifications is directed by the editor. Type constraints and some other semantic constraints are checked as each assertion expression if defined. An example of a common banking system specification is presented.

1 Introduction

In the preface to his book on formal systems specification [INC 88], Ince writes:

> "Formal specification is the name given to the use of discrete mathematics for describing the function of both hardware and software systems. It is a subject which is gaining popularity in the United Kingdom and the United States; it was recently made a major component of the British Government's Alvey software engineering strategy. It is now starting to impinge upon the undergraduate and postgraduate curricula. Unfortunately, few books cover the subject at an introductory level. ..."

Ince provided an introductory text for formal specifications, but at the 1988 date, object oriented style for specifications was not covered.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copyright is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1995 ACM 0-89791-737-5

A further problem of bringing formal specifications into undergraduate curricula is that few supporting tools are available. Writing specifications is similar in concept to writing programs. U.S. students expect -- indeed, need -- an interactive environment and an executable language.

This paper presents a language for writing model-based specifications in an object-oriented form together with its supporting environment for development and evaluation (execution) of specifications. OOSPEC draws its basis from both VDM [BJO 82] and Z [DIL 90], in that data items are modeled in terms of high level structures such as sets, sequences, and maps and operations are specified by pre- and post-conditions about the state of data items expressed in predicate calculus notation [GRI 81]. The object oriented structure of OOSPEC is based on Smalltalk [GOL 83]. The imperative style of operation specifications follows from the specification part of Ada [ANS 83]. Class parameterization derives from Eiffel [MEY 88] and FOOPS [GOG 87] as well as Ada.

In order to achieve executable specifications, assertion predicates are given some aspects of procedural lanaguages; this follows in the same fashion that Prolog allows a procedural interpretation. OOSPEC predicates allow sequential, conditional, and iterative evaluation. What establishes OOSPEC as a specification language is that it uses high level structures to model data items.

The following sections describe general features of the OOSPEC language and the environment. A more complete presentation of the system [PAR 93] and an executable form of the interpreter are available from the author. Section 5 presents OOSPEC specification of a commonly used example.

2 OOSPEC Language

2.1 General Description

The basic unit of specification is the *class*. A class is a static specification of the structure and properties of a possible collection of objects that are *instances* of the class. Objects are dynamic run-time elements that will be created and modified during evaluation of a specification. The class of which an object is an instance is also referred to as the *type* of the object.

A class consists of data representation, in the form of *instance variables*, and *operations* that are defined over that data representation. An OOSPEC specification is composed of a collection of classes related by some hierarchy.

2.2 Objects

Objects are the basic building block of specifications. Simple objects are an instance of one of the primitive classes, as: 'a string' , 465 (integer), 35.7 (real), $A (character), < 1 2 3 > (a sequence of same type elements), { 1 2 3 } (a set of same type elements). Map types are included in the full language, but not yet implement.

A *named object* is an instance of a class that is represented by an identifier. Is is declared as:

accountNumber : INTEGER;

Named objects are used as *instance variable*, *temporary variables*, and *parameters*. Note that objects are created implicitly. Most object oriented languages require an explicit create message to the class of the object to be created. However, implicit creation fits better in the style of specification, particulary specification of modules to be implemented in imperative languages that use similar declaration of variables.

2.3 Messages

Specifications are executed by evaluating *message* expressions. Each message evaluates certain operations within objects and may modify the state of the objects. Generally, an OOSPEC message has one of the following forms:

object ←*operation_name* (*arg1, arg2, ...*)
... object is unchanged
or *object* ⇐*operation_name* (*arg1, arg2, ...*).
... object is changed

The only way an object can be in a new state is for one of its component objects to be bound to a new value using the *bind* message, as: name ⇐ bind ('John')

For messages with zero or one argument, the parentheses may be omitted.

Messages with no arguments are called *unary* messages, as: 'this is a string' ← size
< 3 5 2 6 > ← first

Binary messages are those that have only one argument, for example: 'hello' ← + ('there')
which may be abbreviated by omitting the ← symbol as:
'hello' + 'there' .
Binary messages are evaluated strictly from left to right; however, parentheses may be used to dictate the order of evaluation. Thus, the arithmetic message:
(3 ← + (4)) ← * (2)
may be written as: 3 + 4 * 2
It returns the result 14. We can, however, use parentheses to control evaluation order, so that the message:
3 + (4 * 2).
returns the result 11.

A *complex message* may have more than one argument, arguments that are themselves messages, or it may itself be an argument within another message. Examples of complex messages are:
< 3 5 2 6 8 1 > ← at ('hello' ← size)
and (3 + 4) ← > x .

All objects respond to the equality message =. Many classes define the relational message operators <, ≤, >, ≥, ≠ in their own context. For example, the class SEQUENCE defines the relational operators for comparing sequences while the class INTEGER defines the same relational operators for comparing integers.

Many objects understand messages that test their state, class, or condition. All such *predicate messages* evaluate to **true** or **false**. For example:
9 ← isOdd
name ← isString.

The class boolean also defines *proposition.* operators **and**, **or**, **not**, **equal**, and **implies.** For example, if **c** is an instance of class character, then the predicate that **c** is not a digit is: (c < $0) ← or (c > $9).

In previous examples, nested messages were always evaluated first. For *block* messages evaluation of nested argments is delayed.

The class boolean defines the conditional block messages *if-then* and *if-then-else* with delayed evaluation of their arguments. For example:
(x < y) ← if-then (name ⇐ bind ('John')) .

If **x < y** evaluates to **true**, then **name ⇐ bind ('John')** will be evaluated; in either case, the message is evaluate as **true**. Similarly,
(x < y) ← if-then-else (name ⇐ bind ('John') , name ⇐ bind ('Tom'))

must bind **name** to either 'John' or 'Tom'.

Three types of *iterator* block messages are provided: timesRepeat is defined for integer objects, whileTrue and whileFalse are defined over boolean objects. For example:
3 ← timesRepeat (x ⇐ bind (x + 1))

results in **x** being incremented by **3**.

2.4 Classes

A *class* is the basic unit of specification in OOSPEC. A simple example is shown in Figure 1. Objects of type STUDENT have the operation **register**,which when

evaluated with proper parameter values results in modification of values of **name** and **id** instance variables within the object.

For each class operation, **pre-** and **post-** *assertion* predicates specify, respectively, the required state of data items for the operation to be valid and the resultant state of data items after evaluation of the operation. Assertions are composed of messages to the related data objects. Because of the left-to-right order of evaluation of messages, sequential evaluation of conjuncts is implicit. For convenience to readers, assertions may be written in *sequential conjunctive form*. Thus, the post assertion for register could have been writen as:

name⇐bind (aName) ← and (id⇐bind (anId)) .

Model based specification languages such as VDM are grounded on the notion of specifying operations via pre- and post-conditions. Other languages such as Eiffel and annotated C++ [LEJ 91], use pre- and post-conditions to strengthen documentation or as a support mechanism for ensuring program correctness. In OOSPEC, the behavior of operations is entirely defined by pre- and post-condition assertions.

```
class: STUDENT;
document:
inherits from: ROOT
class invariants:
public
  instance variables
    name : STRING ;
    id   : INTEGER ;

  procedure register ( in aName : STRING; in anId :
INTEGER );
    pre
      ( aName ← size ≠ 0 ) ← and( anId ≠ 0 )
    post
      name⇐bind ( aName ),
      id⇐bind ( anId )

private

end class
```

Figure 1. A simple specification

More genrally, a class specification may be parameterized, as in Figure 2. To specialize a generic class for a particular type, an actual type name must be supplied, as:

s1 : STACK(INTEGER).

In Figure 2, assertions are not shown; using the class editor (see environment section), they may be made visible by opening each operation declaration. The expanded form is shown in Figure 3.

```
class: STACK ( ItemType : GENERIC );
document:
inherits from: ROOT
class invariants:
public
  procedure push( in item : ItemType );
  procedure pop( );
  function top( ) : ItemType ;
  function empty( ) : BOOLEAN;
private
  s : SEQUENCE( ItemType );
end class.
```

Figure 2 Specification of a generic STACK, unexpanded.

```
class: STACK( ItemType : GENERIC );
document:
inherits from: ROOT
class invariants:
public
    procedure push( in item : ItemType );
      post
        s ⇐ append( item );

    procedure pop( );
      pre
        ( s ← size ) ← ≠ 0;
      post
        s ⇐ removeLast ;

    function top( ) : ItemType;
      pre
        ( s ← size ) ← ≠ 0;
      post
        top ⇐ bind ( s ← last );

    function empty( ) : BOOLEAN;
      pre
      post
        empty ⇐ bind ( 0 ← = ( s ← size ) ) ;

private
    instance variables
      s : SEQUENCE( ItemType );

end class
```

Figure 3. Expanded Specification of STACK

In these examples, structural parts of the class specification are shown by keywords even those parts that are empty. Parts of class specifications are explained in the following paragraphs.

The **documentation** section contains an English description of the class, along with dates of creation, modification, version, author, etc. are listed. The English description is optional but it is particularly useful to

beginning students. The class editor (see Environment section) requires some entry or update for every edit operation on the class specification.

The **inherits from** field contains the names of classes of which the class is a *subclass*. A class inherits or includes properties from its *superclass*. Further aspects of inheritance are presented in a later section.

Class invariants allow for specification of invariant properties of or constraints on data objects that are components of the class. The notion of class invariants dates back to Hoare's work on data type invariants [HOA 72]. The application of data type invariants to program design was investigated in [JON 80], [JON 86] and [JON 90]. An invariant of a class C should hold for every instance i of C at times when i is in a *stable state*, which is define as:

- at the time of creation of i,
- before an operation r of C is entered as a result of every message of the form $i \leftarrow r$
- after operation r completes evaluation.

Class invariants may be violated during evaluation of an operation, but after the evaluation is completed and the instance is in a stable state, the invariant must hold true. It is permitted for a class invariant to be violated during the evaluation of private operations. The reason for this relaxation of the rule is because private operations may be called from within public operations and that invariants are not required to hold true during the evaluation of a public operation. Class invariants for instance i of class C are checked before and after evaluation of each public operation when i is in a stable state. If the invariant does not hold, evaluation stops with an error message. An example of a class invariant is:

aSet ← forAll2(c1, c2, c1 ← ≠(c2) ← implies((c1 ← ssn) ← ≠(c2 ← ssn))) .

To support *information hiding*, a class is composed of **public** and **private** sections. The notion of information hiding dates back to Parnas' work [PAR 72] and is most evident in Ada [ANS 83] with the use of public and private. The public section of a class specifies those procedures, functions, and instance variables that make up the interface for objects defined by the class. Access to private entities is limited to the class in which they are defined.

A common feature of every object oriented language is the notion of **instance variables** [GOL 83] or in some langauges *attributes* [MEY 88]. Instance variables are data values held by objects. In OOSPEC, every instance variable must be declared with a type. A *public* instance variable may be accessed from other classes by sending a message with the same name as the instance variable. In order to guarantee that class invariants hold when objects are created, instance variables are initialized to a default zero or null value according to their types.

Operations in OOSPEC correspond to methods in other object oriented languages, except that **procedure** and **function** operations are distinguished. This distinction allows a specification to be readily understood as a specification of module in imperative languages such Ada or Pascal. A function operation must return a value of the specified type. A procedure operation does not have an explicit return value, but it may determine new values for instance variables or for the proceudre parameters. In addition, for use in composing assertion messages, each procedure operation is interpreted as a boolean expression -- that is, successful evaluation of a procedure message will result in the value **true**.

Operation parameters are of five *modes*:: **in**, **out**, **in-out**, **selector**, and **assertion**. The in, out, and in-out modes correspond to those defined in Ada. Selector and assertion parameters are used simultaneously within an operation declaration. An assertion parameter is used to pass a message to an operation and the message that is sent will make references to zero or more selector parameters. For example, if setOfItems is an instance of type SET, then the message:
setOfItems ← forAll2(item1, item2, item1 ← ≠ (item2))
contains two selector parameters, item1 and item2, which are referenced within the underlined message parameter.

Temporaty variables may be introduced within the specification of any operation (although none appear in these examples). Like instance variables, temporary variables are data values held by objects. However, the value of a temporary variable is retained only during the evaluation of the operation.

2.5 Class Inheritance

OOSPEC supports three forms of inheritance: *hierarchical* inheritance, *specialized hierarchical* inheritance, and *specialized* inheritance.

Hierarchical inheritance is similar to that provided by Smalltalk. A class may be specified to be the *subclass* of an existing class, called the *superclass*. A subclass inherits all of its superclass' features except for its name. A subclass may redefine inherited operations. A subclass my amend inherited invariants. A subclass may define new instance variables, operations, and class invariants. However, a subclass may not redefine or add to the list of class parameters that it inherits from its superclass. Table 1. summarizes the rules for hierarchical class inheritance.

Specialized heiarchical inheritance is similar to hierarchical inheritance except that some class parameters must be specialized with existing types. For example, INTEGER-STACK could be created as a subclass of STACK(INTEGER) .

superclass features	inherited	redefined	amended
name	no	must	no
class parameters	yes	no	no
class invariants	yes	yes	yes
public instance variables	yes	no	yes
private instance variables	yes	no	yes
public operations	yes	yes	yes
private operations	yes	yes	yes

Table 1. Hierarchical inheritance rules

Finally, created objects inherit all but the private properties of their defining type. For example, suppose the class hierarchy of *BINARY-TREE* is:

BINARY-TREE
|
ORDERED-BINARY-TREE
|
INTEGER-BINARY-TREE

and let T1: INTEGER-BINARY-TREE. That is, ORDERED-BINARY-TREE is a restricted form of a BINARY-TREE; it is a subclass of BINARY-TREE. It inherits all the instance variables of BINARY-TREE, but it adds a further class invariant (the ordered property) and it redefines operations such as insert, remove, find, etc. to conform with the class invariant. INTEGER-BINARY-TREE is a specialized subclass of ORDERED-BINARY-TREE; it inherits everything from the specialized class ORDERED-BINARY-TREE (INTEGER). The object T1 inherits all public operations of INTEGER-BINARY-TREE and it satifies the class invariant which was inherited from ORDERED-BINARY-TREE.

As in other object-oriented languages, to explicitly reference an operation in the superclass the special object **super** is provided. Within the ORDERED-BINARY-TREE class, the message

super ← invariants

evaluates the invariants of BINARY-TREE in the context of ORDERED-BINARY-TREE. Similarly, the variable **self** is provided for accessing instance variables and operations inside the same class. For example, the message

self ← find(item)

evaluates the operation find in the same class.

3 The Environment

ENSPEC is a support environment for specifications developed in the language OOSPEC. It integrates a *graphical-direct-manipulation editor*, a *knowledge/data base* of prediates and other facts/rules, *consistency checking facilities*, *documentation facilities*, and most important, an *interpreter* and *degugger*. The complete description is presented in [PAR 93].

Figure 4. is a sample of what the ENSPEC menus look like. These particular menus are available during the composition (editing) of a class. The **class** menu consists of those commands that are relevant on a class while the **operation** menu contains commands related to actions performed on an operation.

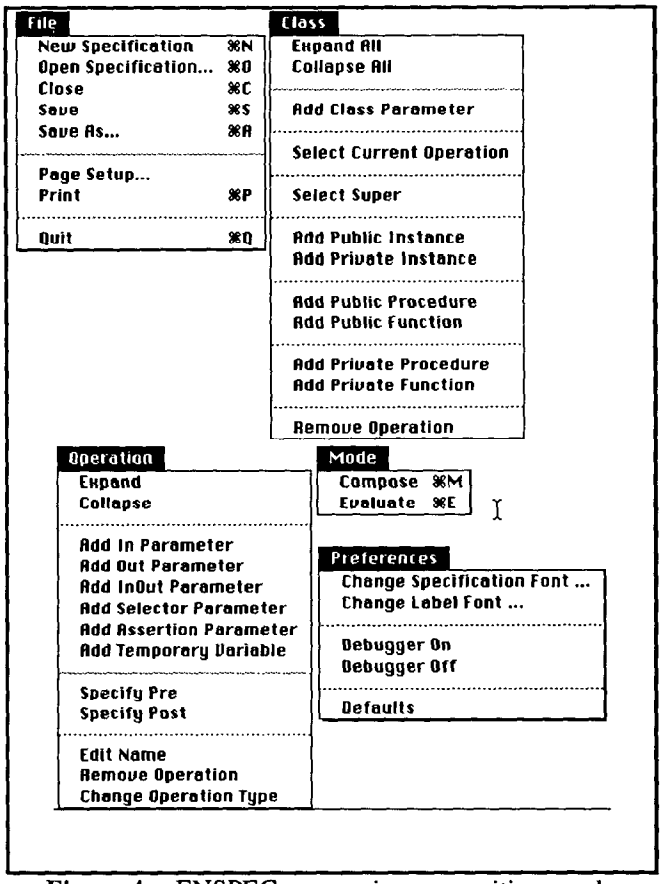

Figure 4. ENSPEC menus in composition mode.

Figure 5. shows an actual screen image of the prototype while editing the class queue. Here, the postcondition is being constructed interactively by selecting pieces from pop-up menus that appear.

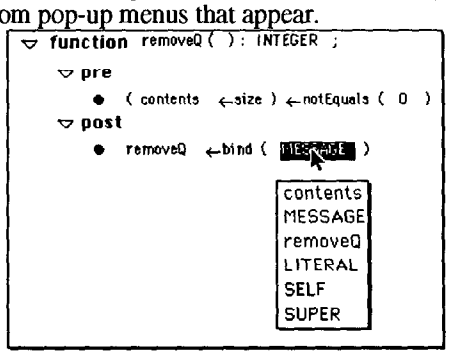

Figure 5. Editing a class

A class is evaluated through an Input/Output Frame. Figure 6 shows an example of an I/O Frame, in this case for the queue class. The I/O Frame lets the user enter values for parameters and trigger operations to evaluate. The interpreter then evaluates the operation and the result is displayed in output parameters. This allows for quick evaluation and testing of specifications.

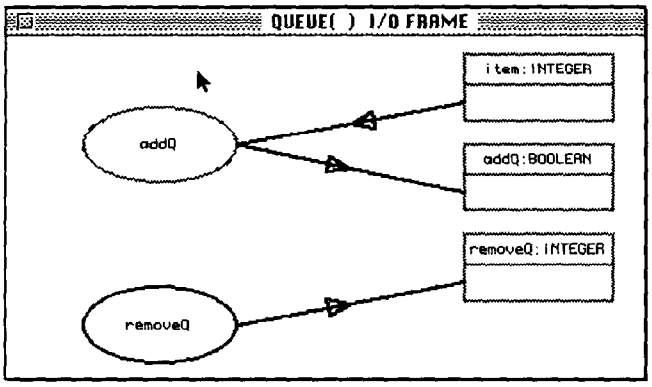

Figure 6. An Input/Output Frame

4 The Prototype

To demonstrate major features and functions of ENSPEC and its language OOSPEC, a prototype was built in Smalltalk.

In its current form, the ENSPEC prototype focuses on composing and evaluating specifications. The prototype deviates from the specification of OOSPEC in that only single left arrows are impemented in messages, and that in palace of using symbolic names of operations the prototype uses their mnemonic equivalent.

In the prototype, SET, SEQUENCE, INTEGER, REAL, BOOLEAN, Interpreter (I/O Frames and evaluation), Creation of new classes, and Modifying existing classes are fully implemented.

The following two tables list features and the current status of their implementation. Table 2 lists environment features and Table 3 lists language features.

Environment Features
implemented: SET, SEQUENCE, INTEGER, REAL, BOOLEAN, Interpreter, Class creation, Class modification
not implemented: CHARACTER, MAP, Assistance Facility, Knowledge/Database, Debugger, Documentation Facilities
partially implemented: Consistency Checking, Class Browser , Editor

Table 2. Implementation Status of Environment Features

Language Features
implemented: Specialized Inheritance, Functions & Procedures, Instance Variables, Public and Private Sections, in, out, in-out Parameters, Selector Parameters, Return Parameters, Temporary Variables, Precondition & Postcondition, Assertion, Literals, Simple Objects, Named Objects, Simple Messages, Unary Messages, Binary Messages, Complex Messages, Arithmetic Messages, Binding, Relational Messages, Conditional Messages, Boolean Messages, Iterative Messages,
not implemented: Spec. Hierarchical In., Hierarchical Inheritance, Class Document, Class Invariant, Predicate Messages, self & super
partially implemented: Consistency Checking, Class Parameters, Assertion Parameters, Naming Rules

Table 3. Implementation Status of Language Features

The prototype was built in Smalltalk/V Mac® [SHA 91] on a midrange Apple Macintosh® computer. Table 4 summarizes some measures of the ENSPEC prototype implementation.

Number of Classes	41
Number of Methods	454
Number of Lines of code	5500
Application size	237K Bytes
Min. Memory space required	4MB
Development Effort	6 man months

Table 4. Measures of the prototype code

We are quite content with the performance of the prototype considering that it is an interpreter implemented on top of Smalltalk's interpreter running on an average personal computer. In fact, in tests of specifications the size of the banking example below, we were able to edit and evaluate classes with near instantaneous response from the prototype.

5 Banking Example

The following banking example presents actual screen images from the prototype.

The **BANK** Class

```
class: BANK ( );
document:
inherits from: ROOT
class invariants:
public
 ▽ procedure openAccount ( in initialDeposit : REAL ;
                           out accountId : INTEGER );
     temp vars
       newId : INTEGER ;
       newAccount : ACCOUNT ;
   ▽ pre
     • initialDeposit ←greaterThanOrEqual ( 100.0 )
   ▽ post
     • nextAccountId ←bind ( nextAccountId ←add ( 100 ) ) ,
     • newId ←bind ( nextAccountId ) ,
     • newAccount ←setId ( newId ) ,
     • newAccount ←activate ,
     • newAccount ←credit ( initialDeposit ) ,
     • accounts ←bind ( accounts
                ←union ( accounts ←makeASet ( newAccount ) ) )
     • accountId ←bind ( newId )

 procedure creditAccount ( in amount : REAL ;
                           in accountId : INTEGER );
   ▽ pre
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
   ▽ post
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
                ←and ( a ←credit ( amount ) ) )

 procedure debitAccount ( in amount : REAL ;
                          in accountId : INTEGER );
   ▽ pre
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
          ←and ( ( a ←balance? )
               ←greaterThanOrEqual ( amount ) ) )
   ▽ post
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
                ←and ( a ←debit ( amount ) ) )

 procedure closeAccount ( in accountId : INTEGER );
   ▽ pre
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
   ▽ post
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
          ←and ( ( a ←debit ( a ←balance? ) )
               ←and ( a ←inActivate ) ) )

 function accountBalance ( in accountId : INTEGER ) : REAL ;
   ▽ pre
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
   ▽ post
     • accounts ←exist ( a , ( a ← id? ) ← equals ( accountId ) )
          ←and ( accountBalance ←bind ( a ←balance? ) ) )

 function worthOfBank ( ) : REAL ;
   ▽ post
     • accounts ←forAll ( a , ( a ←active? )
          ←and ( worthOfBank
               ←bind ( worthOfBank
                    ←add ( a ←balance? ) ) ) )
private
  instance vars
    accounts : SET ( ACCOUNT );
    nextAccountId : INTEGER ;

end class: BANK
```

Figure 4. Specification of BANK

The **ACCOUNT** Class

```
class: ACCOUNT ( );
document:
inherits from: ROOT
class invariants:
public
  procedure credit ( in amount : REAL );
   ▽ pre
     • ( amount ←greaterThan ( 0.0 ) ) ←and ( active )
   ▽ post
     • balance ←bind ( balance ←add ( amount ) )
  procedure debit ( in amount : REAL );
   ▽ pre
     • ( amount ←greaterThan ( 0.0 ) ) ←and ( active )
   ▽ post
     • balance ←bind ( balance ←subtract ( amount ) )
  procedure activate ( );
   ▽ pre
     • ( id ←notEquals ( 0 ) ) ←and ( active ←equals ( false ) )
   ▽ post
     • active ←bind ( true )
  procedure inActivate ( );
   ▽ pre
     • ( id ←notEquals ( 0 ) ) ←and ( active )
   ▽ post
     • active ←bind ( false )

  procedure setId ( in anId : INTEGER );
   ▽ post
     • id ←bind ( anId )
  function id? ( ) . INTEGER ;
   ▽ post
     • id? ←bind ( id )
 ▽ function balance? ( ) : REAL ;
   ▽ pre
     • active ←equals ( true )
   ▽ post
     • balance? ←bind ( balance )
  function active? ( ) : BOOLEAN ;
   ▽ post
     • active? ←bind ( active )

private
  instance vars
    id : INTEGER ;
    balance : REAL ;
    active : BOOLEAN ,

end class: ACCOUNT
```

Figure 5. Specification of ACCOUNT

6 Conclusions

OOSPEC was designed to be an object oriented specification language that is based on manipulation of sets, sequences, functions, relations, and predicates with quantifiers. Furthermore a support environment has been designed that makes learning and using OOSPEC easier. Since one of our objectives has been for ENSPEC to be

usable by students of introductory data structure/algorithms courses, careful attention has been paid to making the environment easy to use. That includes incorporating the familiar notions of functions and procedures and parameter passing from traditional imperative languages such as Ada and Pascal.

The prototype has successfully proven that it is possible to implement such an executable specification system on personal machines with an acceptable performance.

7 References

[ANS 83] ANSI and AJPO, Military Standard: Ada Programming Lanauge (Am. Nat. Standards Inst. and US Gov. Dept of Defense, Ada Joint Program Office), ANSI/MIL-STD-1815A-1983, Feb. 17, 1983.

[BJO 82] Bjorner, D. and Jones, C. B. *Formal Specification and Softwre Development.* Prentice/Hall, Englewood Cliffs, NJ, 1982.

[BOO 86] Booch, Grady. *Object-oriented development.* IEEE Transactions on Software Engineering. SE-12, 2, Feb. 1986, pp. 211-221.

[BOO 91] Booch, Grady. *Object-oriented Design.* Benjamin/Cummings, Redwood City, CA, 1991.

[COA 91-1] Coad, P. and Yourdon, E.; *Object-Oriented Analysis*, 2nd Ed., Prentice Hall, Englewood Cliffs,New Jersey, 1991.

[COA 91-2] Coad, P. and Yourdon, E.; *Object-Oriented Design.* Prentice Hall, Englewood Cliffs,New Jersey, 1991.

[COH 86] Cohen, B., Hardwood, W., and Jackson, M., *The Specification of Complex Systems*, Addison-Wesley, 1986.

[DIL 90] Diller, A. *Z: An Introduction to Formal Methods.* John Wiley & Sons, New York, 1990.

[ELL 90] Ellis, Margaret, A. and Stroustrup, Bjarne. *The Annotated C++ reference manual.* Addison-Wesley, 1990.

[GOG 87] Goguen, J. A. and Meseguer, J.; *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics.* In Research directions in object-oriented programming, B. Shriver and P. Wegner, (eds.). MIT Press, 1987, pp. 417-478.

[GOL 83] Goldberg, A. and Robson, D. *Smalltalk-80, the Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[HOA 72] Hoare, C.A.R. *Proof of Correctness of Data Representations*, Acta Informatica, vol. 1, pp. 271-281, 1972.

[INC 88] Ince, D. C.; *An Introduction to Discrete Mathematics and Formal System Specification.* Oxford Applied Mathematics and Computing Science Series. Clarendon Press, Oxford, 1988.

[JON 80] Jones, C. B. *Software Development: A Tigorous Approach*, Prentice-Hall International, Hemel Hempstead, 1980.

[JON 86] Jones, C. B. *Systematic Software Development using VDM*, Prentice-Hall International, Hemel Hempstead, 1986.

[JON 90] Jones, C. B. *Systematic Software Development using VDM.* Prentice Hall, New York, 1990.

[LEJ 91] LeJacq, Jean Pierre. *Function Preconditions in Object Oriented Software.* ACM SIGPLAN Notices, vol. 26, no. 10, Oct. 1991, pp. 13-18.

[LIS 86] Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development.* The MID Press, Cambridge, Massachusetts, 1986.

[MEY 88] Meyer, Bertrand. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[PAR 72] Parnas, D. L., *On Criteria to Be Used in Decomposing Systems into Modules*, CACM, vol. 14, no. 1, April 1972, pp. 221-227.

[PAR 93] Paryavi, M. N. *An Object-Oriented Formal Specification Language and Support Environment.* Ph.D. Dissertation , CIS Dept., Kansas State University, Manhattan, Kansas, 1993.

[SHA 91] Shafer, Dan and Ritz, Dean A. *Practical Smalltalk Using Smalltalk/V.* Springer-Verlag, New York, NY, 1991.

[WIR 90] Wirf-Brock, R., Wilkerson, B., and Wiener,L.; *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs,New Jersey, 1990.