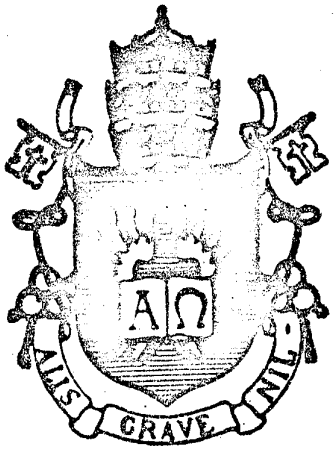

PUC

Series: Monografias em Ciência da Computação, No. 2/91

A METHOD FOR OBJECT-ORIENTED SPECIFICATIONS WITH VDM

Roberto Ierusalimschy

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453

RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação, 2/91
Editor: Carlos J. P. Lucena April, 1991

A METHOD FOR OBJECT-ORIENTED SPECIFICATIONS WITH VDM *

Roberto Ierusalimschy +

* This work has been sponsored by the Brazilian Government Office of Science and Technology.

+ On leave at the University of Waterloo, Ontario, Canada.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Tel.:(021)529-9386 Telex:31078 Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

A Method for Object-Oriented Specifications with VDM*

Roberto Ierusalimschy†
Pontifícia Universidade Católica
Rio de Janeiro – Brazil

May 14, 1991

Abstract

This paper presents a new specification method for Object Oriented Programming. The method is an extension of the method proposed by Jones, using VDM. The main additions are mechanisms for inheritance and nesting of specifications. Inheritance is used to achieve reusability, as supported by the object oriented paradigm, while nesting allows a better modularization of the state space of a system.

Keywords: Object Oriented Programming, Formal Specifications, VDM, Inheritance.

Sumário

Este trabalho apresenta um novo método de especificação para Programação Orientada a Objetos. Este método é uma extensão do proposto por Jones, com VDM. As principais contribuições são mecanismos para herança e aninhamento de especificações. Herança é usada principalmente para reutilização, como proposto pelo paradigma de orientação a objetos. Aninhamento permite uma melhor modularização do espaço de estados de um sistema.

Palavras-Chave: Programação Orientada a Objetos, Especificações Formais, VDM, Herança.

---

*This work has been supported by CNPq (The Brasilian National Research Council).
†on leave at the University of Waterloo (from Mar, 91 to Feb, 92).

1 Introduction

Since the beginning of the 80's Object Oriented Programming (OOP) has received more and more attention as a new programming paradigm. Certainly, one significant thrust in OOP has been programming languages. From its origins in Simula-67, to its prominence in Smalltalk-80, OOP has been guided by developments in object oriented programming languages (OOPLs).

Unfortunately, the evolution of OOPLs has been mainly ad hoc, without a solid ground of formalisms or methodologies. Usually, new features are justified only by means of examples. On the other hand, proposed object oriented methodologies do not use these features, thus neglecting facilities like inheritance and reference semantics. As a consequence, most of those methodologies used with OOP in reality are related to Abstract Data Types (ADT), and are better suited for non OOPLs such as Ada or Modula-2. [1]

This paper presents a new method for software specification, which has been developed to guide the design of a new OOPL, named O=M [Ier90, Ier91]. The method is mainly an extension of the method proposed by Jones [Jon86], adding facilities for inheritance of specifications and nesting, while keeping VDM's formal nature. Inheritance is used to support reuse of specifications, and also as a guide to inheritance in implementation, as supported by most OOPLs. Nesting is used here as a link between the monolithic point of view for states adopted by Jones, and an informal distributed one, supported by OOP; moreover, nesting allows simpler specification models for reference semantics. As a consequence of these features, we have specifications better fitted to the facilities provided by object oriented languages.

The paper is presented in six sections. The next section discusses the use of ADT specifications for object oriented languages. Section three shows how we introduce inheritance in formal specifications and in reuse of specifications. Nesting and its relationship with reference semantics are considered in section four. Then, section five gives an example of a specification for a File System, using those facilities. Finally, section six presents some conclusions.

[1] A good example is the methodology proposed in [Boo86].

2 Formal Specifications

Since their inception, OOPLs have been associated with Abstract Data Types. One of the first formal treatments for ADTs, from Hoare [Hoa72], was based on the class mechanism of Simula-67. It is undoubtedly recognized that data abstraction is a main characteristic of OOP. These facts, together with the success of ADTs in the field of formal software development, suggested this paradigm as a method for object oriented specifications.

At the end of the 70's, the so called *property oriented* methods for data type specifications gained attention, with particular emphasis on the algebraic and axiomatic specification methods. Their common characteristic is the absence of an explicit model in the definition of a type; the specification states only the properties that the operations over the type must fulfill. The main case for these methods, as opposed to the *model oriented* view proposed by Hoare, is that this kind of specification is *more abstract. More abstract* means that, by giving only the properties of a type, the specification allows a large class of models, instead of just one[2], avoiding biases towards particular implementations.

Recently, however, some results have changed this situation. One is the difficulty presented by property oriented methods for building many practical specifications. [Maj77] has already shown the problems for an algebraic specification of a Stack with access to its internal elements, and [BzT83] states that this type of difficulty happens for any data structure which allows operations on non border elements, such as trees and iterators. ???Another result is a method to know if a given model is abstract enough for a set of operations [Jon86]. The last point is whether we need so much abstract specifications.

One of the main cases for object oriented programming is the change of emphasis from *"what the system does"* to *"what it does it to"* [Me88a]. This emphasis seems better fitted for the construction of an evolutionary system, as during the life-time of a system modifications are made to its behavior more frequently than to the concepts (or objects) it manipulates. Many times the designer over-specifies a type in anticipation of future updates. Inheritance is also centered around object states; usually subtypes change the functionality of the original type, keeping a similar internal structure.

This reasoning to justify OOP can also justifies model oriented speci-

[2]Although some algebraic methods, with the use of initial or final models, do not have this property (p.e. [ADJ78]).

fications. To build a specification only using operations is to give a great importance to them. Any future modification of the operations can lead to unanticipated changes on the specification and its properties. On the other hand, having an explicit model for a system (or for its types) allows the introduction of changes without great disturbance, as changes on the operations do not affect the model. Even model changes are easily managed, as explicit models are usually more intuitive than implicit ones. Whenever necessary, one can make a test to know whether the model is a minimal specification. For all those reasons, we argue that property oriented specifications are methodologically inconsistent with object oriented programming, while model oriented specifications and OOP use the same basic approach.

3 Inheritance

Inheritance is one of the main features of object oriented programming languages. In fact, some authors propose inheritance as the main feature, by giving the definition *Objects = Abstract Data Types + inheritance* (p.e. [DfT88]). Nevertheless, most of the development methods proposed as object oriented do not provide clear support for this feature. In this section we describe a mechanism to incorporate the concept of inheritance in an specification language framework.

Before proceeding, we would like to stress the difference between *subtyping*, *inheritance of specifications*, and *inheritance of implementations*. Here, type means specification. Subtyping is a relation "$\prec$" between types, meaning that, if $A \prec B$ (A is a subtype of B), thus all objects which satisfy the specification A also satisfy the specification B. In the framework of a programming language, where a type is only a partial specification, the subtype relationship usually takes into account only the type signatures, as these are all the compiler can check. However, in the framework of a specification language, subtyping means behavior compatibility[3]. Inheritance of specifications is a mechanism to reuse specifications, allowing the designer to combine and refine old specifications in order to build new ones. Finally, inheritance of implementations is mainly a mechanism for code (and data structure) reuse. It enables one to combine and refine code in order to build new implementations. Since Simula-67, most OOPLs identify these three concepts, in a way that a specification heir is always a subtype and

[3]According to the classification given in [WeZ88].

an implementation heir.[4] Some languages present only one of those concepts; Quest [Car89] has only subtyping, without a mechanism to declare heirs. Emerald [BeA87] has only inheritance of specifications, which also define subtype relationships. Smalltalk, without a type system, has only inheritance of implementations; the polymorphism of the language is not a consequence of the inheritance mechanism, but of late-binding and typeless variables.

There is no implicit relationship between subtyping and inheritance of specifications; one can use inheritance to create new specifications which are not subtypes of their ancestors, and, on the other hand, a type can be subtype of another one without being a heir of it. However, if the inheritance mechanism is such that every heir is also a subtype, then inheritance of specifications can be used not only for reuse of specifications, but for reuse of verification proofs as well. In a programming language framework, an inheritance mechanism dissociated from subtyping has little use, as it would achieve only reuse of signatures. Linking both concepts brings to the language a mechanism to build "subtypes by construction", enhancing polymorphism and type classification.

On the other hand, it is not useful to keep together the hierarchies for specifications and implementations. First, there are abstraction problems. A user of a class must know its specification ancestors, in order to know its complete behavior and type compatibility, but the user does not need to know the implementation ancestors of the class, because that is an implementation decision that must be hidden by the abstraction. Second, there is no reason to suppose that similar behaviors must be implemented by similar algorithms, or that similar data structures can not be used to implement incompatible specifications. Obviously, many times the implementation hierarchy will follow the specification one, but this must be a programmer decision, and not be imposed by the language. Third, the lack of distinction between both hierarchies create some confusion. For instance, many languages do not enforce behavior compatibility between subtypes with the argument that it is too restrictive and forbids many useful situations for code reuse. Behavior compatibility is a rule for specifications, not for implementations; with distinct hierarchies, one is free to reuse code, changing or deleting features in the subclasses, while maintaining behavior compatibility in the specification hierarchy.

[4] Two exceptions are Duo-Talk [Lun89] and O=M [Ier91], which have separate hierarchies for specifications and implementations.

In order to explain our inheritance[5] mechanism, the first step is to give a precise characterization for subtyping. The intuitive meaning of subtyping is that any object of a given type A is also an object of the supertypes of A. This characterization is also known as an *is-a* relationship. This implies that, if A is a subtype of B, then an object of A can be used wherever an object of type B is needed. An easy way to allow this is to provide a *projective function* which gets objects from A and returns B objects. Moreover, if one intends to use formal (or even informal) reasoning, then a subtype must keep all the properties of its ancestors.

The above characterization can bring one to compare the relationships between Type and Subtype and between Specification and Implementation. In fact, an implementation also satisfies the requirements stated previously. This similarity leads many authors to try to support separate specifications by means of supertypes (p.e. [Me88b, HlB87]). However, this is a misuse of inheritance: first, the mechanism for separate specifications intends to hide *"how to do"* from *"what to do"*, while the subtype mechanism is a way to organize specifications, without any change of abstraction level. Second, there is a subtle formal difference between both relationships. Although any implementation could be a valid subtype, not all subtypes are valid implementations, as a subtype does not need to conform with the *adequacy* criterion. This criterion states that any correct implementation must have a representation for each abstract object that the specification describes. This is not needed for subtypes and, many times, a subtype is defined exactly to restrict the valid objects of a given type (for instance, a subtype *Square* of a type *Rectangle*).

Besides these differences, the two relationships are close enough to allow us to use the formal definition presented by [Jon86] for implementations to fit our definition of subtyping. We say that an abstract type A is a subtype of an abstract type B, denoted by $A \prec B$, iff:

- there is a total function $proj_{A,B}: A \rightarrow B$, which allows to see states of objects of the type A as states of objects of the type B.
- For each function (or procedure) X over the type B, there is a function (or procedure) over A, with the same name, the same number of parameters, and such that its pre and post conditions satisfy the following equations:

$$- \forall x \in A \cdot \forall p \cdot pre\text{-}X_B(proj_{A,B}(x), p) \Rightarrow pre\text{-}X_A(x, p)$$

[5] From now on, we will use inheritance to mean inheritance of specifications.

– $\forall \overleftarrow{x}, x \in A \cdot \forall p \cdot pre\text{-}X_B(proj_{A,B}(\overleftarrow{x}), p) \wedge post\text{-}X_A(\overleftarrow{x}, x, p) \Rightarrow$
$post\text{-}X_B(proj_{A,B}(\overleftarrow{x}), proj_{A,B}(x), p)$

where p represents the operation parameters, and $\overleftarrow{x}$ the state before the operation. These formulae are similar to those for implementations, and simply state that wherever a B operation can be used, the equivalent A operation is also allowed. Following [Jon86], the parameter types are considered as a part of the pre and post conditions, so there is no need to put explicit rules for them.

If $A \prec B$, then every value of an A object can be viewed as a value of a B object, via the *proj* function. So, A objects can be treated as B objects, without the need to explicitly write the *proj* function.

It is important to note some consequences of this definition:

- A given type can have multiple supertypes.
- A very common case of subtyping occurs when the new subtype keeps the model structure of the original one, only adding new fields. In this case, the function *proj* is the orthogonal projection. When the type implementations follow the model, the inheritance mechanism found in OOPLs can be used to implement the subtype.
- The definition given above has no mention of initial states. So, it is possible to define new subtypes only putting stronger invariants. Of course, the new invariant can not refute any post-condition, otherwise the type can not be implemented.
- The input parameter types of the new operations can be supertypes of the parameter types of the old operations[6], as the new preconditions can be weaker than the old ones; conversely, the output parameter types can be subtypes of the old ones.
- It is trivial to show that the subtype relationship is transitive; the projection functions can be composed to give the new one, and the implications of the pre and post condition rules are transitive.

The next step, after a characterization of subtyping, is a definition of inheritance. As we intend to use inheritance for polymorphism and type

[6]This property is known as the *contravariance* property, the more restricted a type, the less restricted its operational parameter types.

classification, we will associate the subtype and inheritance relationships; any heir of a given type must be a subtype of this type. So, we would like a mechanism which allows us to reuse a given specification, but in such a way that the resulting type is always a subtype of the original one. Moreover, we would like to support multiple inheritance, keeping the subtype relationship for all parents.

The linguistic support for inheritance that we are going to introduce is specifically designed for composite types, as these are the most frequently used types in VDM specifications , and we can keep some parallelism with the inheritance mechanisms presented by programming languages. In order to accommodate the syntactic extensions presented here, and also to keep together models and operations, giving a more "object oriented" look to a specification, we use a different syntax for VDM specifications. To declare a type (or specification) B, and then another type A as a *subtype* (or *heir*) of B, we use the following notation:

Specification B
  fb_1: Tb_1
  ⋮
  fb_m: Tb_m
  inv-B $\triangleq$ p_B
  Procedure Pb_1 (...)
  ext wr fb_{w_1}
      ...
  pre *Pre-Pb*$_1$
  post *Post-Pb*$_1$
  ⋮
End B

Specification A
  Subtype of B
  fa_1: Ta_1
  ⋮
  fa_n: Ta_n
  inv-A $\triangleq$ p_A
  Procedure Pa_1 (...)

ext wr fa_{y_1}
...
pre *Pre-Pa*$_1$
post *Post-Pa*$_1$
⋮

End *A*

The subtype declaration in the type *A* makes the above definition of *A* equivalent to the following one:

Specification *A*
fb_1: Tb_1
⋮
fb_m: Tb_m
fa_1: Ta_1
⋮
fa_n: Ta_n
inv-A $\triangleq$ $p_A \wedge p_B$
Procedure Pb_1 (...)
ext wr fb_{w_1}
...
pre *Pre-Pb*$_1$
post *Post-Pb*$_1$
⋮
Procedure Pa_1 (...)
ext wr fa_{y_1}
...
pre *Pre-Pa*$_1$
post *Post-Pa*$_1$
⋮
End *A*

$proj_{A,B} : A \rightarrow B$

$proj_{A,B}(mk\text{-}A(b_1, \ldots, b_m, a_1, \ldots, a_n)) \triangleq mk\text{-}B(b_1, \ldots, b_m)$

It is important to note that, inside the new type, the inherited attributes (fields and operations) have the same status as the new ones. So, if there is any name collision between an inherited attribute and a new one, it must be treated as an error in the same way as if one defines two fields with the same name.

It is very easy to show that an heir, defined according to the above mechanism, is also a subtype. It is also easy to see how the mechanism works for multiple inheritance. However, this mechanism is still very rigid, as 1) the inherited operations can not be modified in order to access the new fields, and 2) the possibility for multiple inheritance only works provided there is no name collision between fields or operations inherited from different ancestors.

In order to solve these problems, and to increase the flexibility of the inheritance mechanism, we have introduced the *Rename* and *Redefine* facilities. These facilities have been taken from the programming language Eiffel [Me88a], and adapted for the specification phase. The syntax for these facilities is as follows:

Specification *A*
  Subtype of *B*
    rename on_1 as $nn_1, \ldots, on_k$ as nn_k
    redefine $n_1, \ldots, n_l$
   fa_1: Ta_1
   $\vdots$
   fa_n: Ta_n
  (declarations for $n_1, \ldots, n_l$)
End *A*

The *Rename* facility is used to change the name of one or more inherited attributes (operation or field). Its most common application is to avoid name clashes, but it can also be used to give to the attribute a more appropriate name in the new environment. The semantics of a rename operation is only to replace all occurrences of the old name (on) by the new name (nn) in everything inherited, including the invariant and the pre- and post-conditions.

Note that these modifications do not affect the subtype relationship, and do not affect the *proj* function as well.

The *Redefine* facility is used to modify a given operation inherited from the original type , while keeping it compatible with the old one. For each operation in the redefine list, we must give a complementary definition, which is combined with the old one to create the new definition. The join process is such that the new definition always satisfies the subtype relation with the old one. It works as follows:

Suppose the old definition (in type B) is:

Procedure X (p: TP)
ext wr fb_{w_1}: Tb_{w_1}
...
wr fb_{w_j}: Tb_{w_j}
pre $pre\text{-}X_B$
post $post\text{-}X_B$

and we give the following complementary definition:

Procedure X (p: TP)
ext wr fa_{y_1}: Ta_{y_1}
...
wr fa_{y_k}: Tb_{y_k}
pre $pre\text{-}X_A$
post $post\text{-}X_A$

Then, the new definition is:

Procedure X (p: TP)
ext wr fb_{w_1}: Tb_{w_1}
...
wr fb_{w_j}: Tb_{w_j}
wr fa_{y_1}: Ta_{y_1}
...
wr fa_{y_k}: Tb_{y_k}
pre $pre\text{-}X_A \vee pre\text{-}X_B$
post $post\text{-}X_A \wedge (pre\text{-}X_B \Rightarrow post\text{-}X_B)$

The first thing to note is that this definition satisfies the criteria for subtyping. In fact:

- $pre\text{-}X_B \Rightarrow (pre\text{-}X_A \vee pre\text{-}X_B)$
- $(pre\text{-}X_B \wedge (post\text{-}X_A \wedge (pre\text{-}X_B \Rightarrow post\text{-}X_B))) \Rightarrow post\text{-}X_B$

Moreover, the original operation has an implicit post-condition stating that the variables not listed in the external list can not be modified, that is:

$$\bigwedge_{i \notin \{w_1,\ldots,w_j\}} \overleftarrow{fb_i} = fb_i$$

As the complementary definition can not have any *fb* field in its external list, the above condition is still satisfied by the new operation. It is important to note that any function definition which already satisfies the subtyping restrictions can be written as a complementary definition, without being modified by the join process, as:

- $(pre\text{-}X_B \Rightarrow pre\text{-}X_A) \Rightarrow (pre\text{-}X_A \Leftrightarrow (pre\text{-}X_A \vee pre\text{-}X_B))$
- $((pre\text{-}X_B \wedge post\text{-}X_A) \Rightarrow post\text{-}X_B) \Rightarrow$
  $(post\text{-}X_A \Leftrightarrow (post\text{-}X_A \wedge (pre\text{-}X_B \Rightarrow post\text{-}X_B)))$

In other words, the combination process transforms any definition into a new one which satisfies the subtype relationship; if the definition already satisfies this requirement, it is left unchanged.

Another use of the redefine facility is to allow the attributes inherited from different ancestors to be combined. As already stated, when there is a name collision between inherited names, it is treated like an error. However, sometimes one could want to join the attributes. This can be done redefining all the clashing attributes. Accordingly, the new operation will be the composition of all the old specifications and the complementary one.

Let us now see a small example of all those features. Suppose we have the type Stack, modeled as a sequence of values:

Specification *Stack*

val: T^*

Procedure *Push* (*e*: T)

ext wr val: T^*

post $val = e \curvearrowright \overleftarrow{val}$

Function *Pop* () e: T

ext wr val: T^*

pre len $val > 0$

post $(e = \text{hd}\,\overleftarrow{val}) \wedge (val = \text{tl}\,\overleftarrow{val})$

End *Stack*

and a type for iterating objects, as follows:

Specification *Iter*

s: T^*

c: $\mathbb{N}$

inv-Iter $\triangleq$ $0 \leq c \leq \text{len}\ s$

Procedure *Reset*

ext wr c: $\mathbb{N}$

post $c = 0$

Function *Next* () e: T

ext wr c: $\mathbb{N}$

rd s: T^*

pre $c < \text{len}\ s$

post $(c = \overleftarrow{c} + 1) \wedge (e = s[c])$

Function *End* () b: $\mathbb{B}$

ext rd c: $\mathbb{N}$

rd s: T^*

post $b = (c = \text{len}\ s)$

End *Iter*

With these definitions, we are able to define an IterStack, that is, a Stack with iteration facilities.

Specification *IterStack*

Subtype of *Stack*

redefine *val*, *Pop*

```
Subtype of Iter
  rename s as val
  redefine val
Function Pop () e: T
ext   wr c: N
post  c = min(c~, len val)
End IterStack
```

As we want to join the fields *s* and *val*, we first rename one of them (*s* in this case), and then redefine both inherited *val* attributes, to avoid a clash. The redefinition of *Pop* is needed in order to keep the *Iter* invariant after the operation (otherwise a *Pop* could leave the iterator position outside the stack contents). It is only necessary to define the new properties of the *Pop* operation; the old ones are inherited.

4 Nesting

Nesting is a very useful concept to model many aspects of reality. Our perceptions of the real world are usually structured, with some concepts being used to build more complex concepts, and others being decomposed to simpler ones. Some examples are Bank–Branch–Account, and File-System–File–Character. Usually these sub-concepts can not exist by themselves; e.g., it is meaningless to have an Account without a Bank. On the other hand, they have their own meaning, in the sense that we can have relationships between these entities and entities outside their surrounding objects.

The idea of nesting resembles opacity, mainly because of the block structure of Algol-like languages. This kind of structure is not very compatible with the object oriented paradigm, which uses a flat state space, supported by reference semantics in OO programming languages. Moreover, as the OO paradigm already presents a mechanism for encapsulation (the object concept itself), sometimes the nesting concept is considered useless within OOP.

However, nesting does not need to mean opacity. First, nested components can be exported to be visible outside their surroundings. Second, we can use inheritance to decompose a specification in order to encapsulate only its specific parts. Using the File-System example, we can have a global GenericFile specification which is inherited and tuned by a File spec-

ification nested inside the File-System. The example in section 5 uses this approach. Moreover, nesting also presents a strong organizational aspect, connected with modularity. These characteristics can be very useful in an object oriented environment.

Here we show a specification facility that allows the modularization of a specification by means of nesting — a specification can be decomposed in components that can be described in a partially independent way. As with inheritance, the nesting facility extends VDM with concepts which can be mapped back to "standard" VDM.

The main idea in the mechanism is to use *maps* to keep nested objects as part of the state of their surroundings. For example, when we declare a Branch inside Bank, we mean that the Bank object has a map field where it keeps record of its branchs. In the same way, each branch has a map to keep its accounts. However, since nested objects have their own meaning, one must be able to write operations that act directly on these objects, even if the final result is a change in the surrounding state. In this way, we can keep the operations nearer their target objects. Without this facility, all operations would have to be put in the most external object.

Generally, a nested object is declared as follows:

Specification O
$fo_1 : To_1$
$\vdots$
Specification S
$fs_1 : Ts_1$
$\vdots$
$inv\text{-}S \triangleq p_S(s)$
End S
$inv\text{-}O \triangleq p_O(o)$
End O

and this declaration is translated to:

$S = \mathsf{N}$

Specification S_Obj

fs_1: Ts_1

$\vdots$

inv-S_Obj $\triangleq$ $ps(s)$

End *S_Obj*

Specification *O*

fo_1: To_1

$\vdots$

S_map: $S \xrightarrow{m} S\_Obj$

inv-O $\triangleq$ $po(o)$

End *O*

In words, the nested type is translated to an access key type (N), and the surrounding type gets a map between these keys and the "real" objects. Any operation over S can be translated to an operation over O, according to the following scheme:

Procedure X_S $(p: TP)$

ext wr fs_{w_1}: Ts_{w_1}

...

wr fs_{w_j}: Ts_{w_j}

wr fo_{y_1}: To_{y_1}

...

wr fo_{y_l}: To_{y_l}

pre *pre-X_S*

post *post-X_S*

is translated to:

Procedure X_O $(i: S; p: TP)$

ext wr *S_map*: $S \xrightarrow{m} S\_Obj$

wr fo_{y_1}: To_{y_1}

...

wr fo_{y_l}: To_{y_l}

pre $(i \in \text{dom } S\_map) \wedge pre\text{-}X_S(S\_map(i), p)$

$$\textbf{post}\ \exists o \in S\_Obj \cdot$$
$$post\text{-}X_S(\overleftarrow{S\_map}(i), o, p) \wedge (S\_map = \overleftarrow{S\_map} \dagger \{i \mapsto o\})$$

Accordingly, an operation call like $s.X(p)$ is translated to $X(s, p)$, that is, the receiver object is in fact the first parameter of the operation; we add in the pre-condition a check about the existence of that object. Note that the operation can access "external variables" $(fo_{w_1}, \ldots, fo_{w_l})$; the occurrence of these variables in the pre and post conditions remains unchanged by the translation.

A last point about the implicit use of maps for nesting is its relation to reference semantics[7]. The kind of maps we use to represent nested objects, with a natural number as index, can be translated directly to a programming language with reference semantics. The indexes are identified with the references, while the map range keeps the real objects.

5 An Example

In this section we present an example of a specification which makes use of all facilities presented above. The example specifies a simple file system.

The first item to specify is a type Collection, which exports objects to iterate over the collection. This is a typical specification one would expect to find in a specification library. The specification is shown in figure 1.

Usually, a collection can be modeled by a sequence. For collections where the order is irrelevant, one can strengthen the invariant to enforce a given order. All collections export a *Iterator* type, which allows multiple (and simultaneous) iterations over a collection. The iteration state is stored in the variable *position*. Besides that type, a collection can also export other operations not showed here.

Next, we have a specification for generic files (figure 2). This is another kind of specification one would expect to find in a library. The *GenericFile* type inherits from *Collection* the basic facilities for sequential reading. More general access is provided by means of a *Handle*. When a file is opened, an appropriate handle is created, and the handle takes care of the iteration state.

[7] By *reference semantics* me mean the characteristic of OOPL that all variables can store only object references, and not an object itself.

Specification *Collection*(T)

c: T^*

Specification *Iterator*

position: $\mathbb{N}$

inv-Iterator $\triangleq$ $(1 \leq position \leq \text{len } c + 1)$

Procedure *Reset*

ext wr *position*: $\mathbb{N}$

post $position = 1$

Function *Read* () r: T

ext wr *position*: $\mathbb{N}$

rd c: T^*

pre $position \leq \text{len } c$

post $r = c(\overleftarrow{position}) \wedge position = \overleftarrow{position} + 1$

Function *End* () e: $\mathbb{B}$

ext rd *position*: $\mathbb{N}$

rd c: T^*

post $e = (position > \text{len } c)$

End *Iterator*

⋮

End *Collection*

Figure 1: Collection Specification

Specification *GenericFile*
  Subtype of *Collection*(T)
    rename *Iterator* as *SeqReadHandle*

  Specification *Handle*

    position: $\mathbb{N}$

    Function *Read* () r: T
    ext wr *position*: $\mathbb{N}$
         rd c: T^*
    pre *position* $\leq$ len c
    post $r = c(\overleftarrow{position})$

    Procedure *Write* (d: T)
    ext wr *position*: $\mathbb{N}$
         wr c: T^*
    pre *position* $\leq$ len c
    post len c = len $\overleftarrow{c}$ $\wedge$
         $\forall i \in \mathbb{N} \cdot i \leq$ len $\overleftarrow{c}$ $\Rightarrow$ If $i = \overleftarrow{position}$
                                                  Then $c[i] = d$
                                                  Else $c[i] = \overleftarrow{c}[i]$

  End *Handle*
End *GenericFile*

Figure 2: Generic File Specification

The *GenericFile* only defines a generic handle, which defines basic writing and reading behavior. Both operations do not specify the value of *position* in the post-condition, allowing different subtypes to specify different restrictions. As a generic file does not know how to check the availability of disk space, the *Write* pre-condition only allows modifications; more specific pre-conditions, such as to allow the write operation to append values at the end of a file, are left to subtypes.

The file system specification is presented in figure 3. The first point to note is that a file system is itself a collection. As we usually want to iterate over file names (p.e. to search the directory), we can declare the file system as a subtype of *Collection*(*String*). We rename some attributes only to have more meaningful names.

A File system exports a file type, defined in figure 4. Besides the files, defined as *File_map*, and the directory list, inherited from *Collection*, the system must keep the relationship between names and files; this is achieved by the *dir* map. The following auxiliary function computes the free space on the system (the constant *MAX_SPACE* is the total space of the system).

$$fs : (File \xrightarrow{m} File\_Obj) \rightarrow \mathbf{Z}$$

$$fs(m) \triangleq MAX\_SPACE - \sum_{i \in \mathrm{dom}\, m} \mathrm{len}\, c(m(i))$$

The invariant of *FileSystem* assures that 1) all file names are in the directory, 2) all names have an associated file, 3) there are no repeated names or files, and 4) the files do not use more than the available space.

This specification presents only some basic operations. Other operations (like *DeleteFile*, *Rename*, etc) can be specified in a straightforward way. The *GetFile* operation just returns the file descriptor associated with the given name, and the *CreateFile* procedure creates an empty file, updating the *dir* map and *dirList* in order to keep the invariant.

The *File* type (figure 4) inherits from *GenericFile* its basic facilities, and tunes them for sequential and random access. The operations *OpenRandom* and *OpenSeq* provide for random and sequential access to a file.

For sequential access we define the handle *SeqHandle* (figure 5), which is a heir of both *Handle* and *SeqReadHandle* (the later has been inherited from *Collection*). The attributes *Read* and *position* are redefined only to be joined. The *Read* operation is completely defined by the inherited properties; *Handle* specifies the generic reading characteristics, and *SeqReadHandle* specifies the final value of *position*. The *Write* operation needs to specify

Specification *FileSystem*

Subtype of *Colection(String)*

rename *Iterator* as *Directory*, *c* as *dirList*

$dir: String \xrightarrow{m} File$

Specification *File*

(see figure 4)

End *File*

$inv\text{-}FileSystem \triangleq (\text{dom } dir = \text{rng } dirList) \wedge (\text{rng } dir = \text{dom } File\_map) \wedge (\text{len } dirList = \text{card dom } File\_map) \wedge (fs(File\_map) \geq 0)$

Function *GetFile* $(name: String)\ f: File$

ext rd $dir: String \xrightarrow{m} File$

pre $name \in \text{dom } dir$

post $f = dir(name)$

Procedure *CreateFile* $(name: String)$

ext wr $dirList: String^*$

wr $dir: String \xrightarrow{m} File$

wr $File\_map: File \xrightarrow{m} File\_Obj$

pre $name \notin \text{rng } dirList$

post $dirList = \overleftarrow{dirList} \curvearrowright name \wedge$

$\exists f \in File \cdot f \notin \text{dom } File\_map \wedge dir = \overleftarrow{dir} \cup \{name \mapsto f\} \wedge$

$File\_map = \overleftarrow{File\_map} \cup \{f \mapsto mk\text{-}File\_Obj([\,], \{\,\})\}$

Function *FreeSpace* $()\ f: \mathbb{Z}$

ext rd $File\_map: File \xrightarrow{m} File\_Obj$

post $f = fs(File\_map)$

End *FileSystem*

Figure 3: File System Specification

Specification *File*
Subtype of *GenericFile*

Specification *SeqHandle*
(see figure 5)
End *SeqHandle*

Specification *RanHandle*
(see figure 6)
End *RanHandle*

Function *OpenRandom* () *rh*: *RanHandle*
ext wr $RanHandle\_map: RanHandle \xrightarrow{m} RanHandle\_Obj$
post $rh \notin \text{dom } \overleftarrow{RanHandle\_map} \wedge$
$RanHandle\_map = \overleftarrow{RanHandle\_map} \cup$
$\{OpenRandom \mapsto mk\text{-}RanHandle\_Obj(1)\}$

Function *OpenSeq* () *sh*: *SeqHandle*
...
End *File*

Figure 4: File Specification

Specification *SeqHandle*
Subtype of *SeqReadHandle*
redefine *Read*, *position*
Subtype of *Handle*
redefine *Read*, *position*, *Write*
Function *Read* () r: T
Procedure *Write* (d: T)
ext rd $File\_map$: $File \xrightarrow{m} File\_Obj$
pre $position = \text{len}\ c + 1 \wedge fs(File\_map) > 0$
post $position = \overleftarrow{position} + 1 \wedge$
$(position = \text{len}\ \overleftarrow{c} + 1 \Rightarrow c = \overleftarrow{c} \frown d)$
Procedure *ReWrite*
ext wr $position$: $\mathbb{N}$
wr c: T^*
post $position = 1 \wedge c = [\,]$
End *SeqHandle*

Figure 5: Sequential Handle

the final value of *position*, and allows append operations. The operations *Reset* and *End* are inherited from *SeqReadHandle* without modification.

For random access we define *RanHandle* (figure 6). As a heir of *Handle*, it only needs to specify the final value for position in the operations *Read* and *Write*, and define a procedure *Seek* to set up a random position.

6 Related Works

This section discusses other works related to inheritance of specifications and nesting in object oriented approaches.

Cardelli [Car84] has presented a semantic description of subtyping in a programming language framework. Within this framework, his description introduces a clear concept of subtyping and inheritance, but with no support for complete specifications. Moreover, his work makes no distinction

Specification *RanHandle*
  Subtype of *Handle*
    redefine *Read*, *Write*
  Procedure *Write* (*d*: *T*)
  post $position = \overleftarrow{position}$
  Function *Read* () *r*: *T*
  post $position = \overleftarrow{position}$
  Procedure *Seek* (*r*:ℕ)
  ext wr *position*:ℕ
       rd *c*: T^*
  pre $1 \leq r \leq \text{len } c$
  post $position = r$
End *RanHandle*

Figure 6: Random Handle

between inheritance of specifications and implementations; as a consequence, it does not support multiple inheritance for abstract data types.

[Lea90] presents an interesting method which incorporates inheritance in the specification language LARCH. Its concept of subtype is similar to ours. The main deficiency of that proposal is that it can not handle mutable objects, which is a significant case in real OO systems. This deficiency seems to be caused by its use of a property oriented method; model oriented methods, like VDM, handle mutable objects without difficulty.

Another definition for inheritance of specifications in a model oriented framework is presented in [SPB90], using Z as the specification language. This work concentrates on concurrency aspects of OOP. Its definition of inheritance is not associated with a concept of subtype or another concept related with verification concerns, but with a way to combine process specifications.

The facilities to rename and redefine inherited features were borrowed from the programming language Eiffel [Me88a]. (LARCH [GHW85] presents a strong rename facility, too.) Eiffel has also the criteria of weaker preconditions and stronger post-conditions to redefine functions in subtypes. However, the language does not enforce the criteria; furthermore, those conditions must be written over the implementation, breaking the abstraction. In spite of the weaker precondition criterion, a routine redefinition in Eiffel can restrict its input parameter types [Mey89], creating inconsistencies in the type system.

The origins of nesting facilities in an object oriented environment can be found in the programming language Euclid [Lam77]. Although it is not a true object oriented language, Euclid allows *module* objects to be created dynamically and export types and operations. The programming language Beta [Mad86] also incorporates nesting facilities in an OO framework. [BuZ88] presents some extensions to the notion of nesting in order to improve its use in OO languages. Nevertheless, none of these proposals is intended for specifications. Jalote [Jal89] presents a method that uses nesting to implement object oriented specifications, but nested objects are hidden from the outside world.

7 Conclusions

We have extended VDM specifications in order to deal with object oriented specifications. We have argued that model oriented approaches to specifi-

cations, like VDM and Z, are better fitted to the object oriented paradigm than property oriented specifications.

Some features of the object oriented model, like data abstraction with persistence and instance visibility, are already present in model oriented methods. Our extensions allows the method to deal with inheritance of specifications and nesting of objects. Nesting also allows a better treatment for reference semantics, a mechanism found in most OOPLs.

The proposed inheritance mechanism allows the definition of "subtypes by construction", which means that types defined as heirs automatically have behavior compatible with their ancestors. In a programming language with independent hierarchies for specification and implementation, that feature does not present an obstacle to code reuse. At the same time, subtypes with behavior compatibility makes possible the use of formal reasoning in object oriented programs.

Specifications can be build in a modular way with the help of nesting. Together with inheritance, nesting does not preclude specification reuse, as shown in the File-System example.

Acknowledgments

The author would like to thank C. J. Lucena and J. L. Rangel, for fruitful discussions on early stages of the work, and D. Cowan for his comments on this paper.

References

[ADJ78] J. Goguen, J. Thacher, E. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, *Current Trends in Programming Methodology IV*, Prentice-Hall, 1978.

[BeA87] A. Black et al. Distribution and Abstract Data Types in Emerald, *IEEE Transactions on Software Engineering SE-13*(1), 1987, pp. 65–76.

[Boo86] G. Booch. Object-Oriented Development, *IEEE Transactions on Software Engineering SE-12*(2), 1986, pp. 211–21.

[BuZ88] P. Buhr & C. Zarnke. Nesting in an Object Oriented Language is NOT for the Birds, in S. Gjessing, K. Nygaard (eds), *ECCOP'88 Proceedings*, Springer-Verlag, 1988, pp. 128–45. (LNCS 322)

[BzT83] A. Berztiss & S. Thatte. Specifications and Implementations of Abstract Data Types, *Advances in Computers 22*, 1983, pp. 296–353.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance, in D. MacQueen, G. Kahn, G. Plotkin (ed.) *Semantics of Data Types : International Symposium*, Springer-Verlag, 1984. (LNCS 173)

[Car89] L. Cardelli. *Typeful Programming*, notes of IFIP Advanced Seminar on Formal Description of Programming Concepts, Petropolis – Brazil, 1989.

[DfT88] S. Danforth & C. Tomlinson. Type Theories and Object-Oriented Programming, *ACM Computing Surveys 20*(1), 1988, pp. 29–71.

[GHW85] J. Guttag, J. Horning, J. Wing. *Larch in Five Easy pieces*, Digital Systems Research Center, Palo Alto, 1985.

[GoR83] A. Goldberg & D. Robson. *Smalltalk-80: The Language and its Implementation*, Adison-Wesley, 1983.

[HlB87] D. Halbert & P. O'Brien. Using Types and Inheritance in Object-Oriented Programming, *IEEE Software 4*(5), 1987.

[Hoa72] C. Hoare. Proof of Correctness of Data Representations, *Acta Informatica 1*(4), 1972, pp. 271–81.

[Ier90] R. Ierusalimschy. *O=M: Uma Linguagem Orientada a Objetos para Desenvolvimento Rigoroso de Programas*, PhD thesis, Dep. Informática, PUC-Rio, 1990.

[Ier91] R. Ierusalimschy. *The O=M Programming Language*, Monografias em Ciência da Computação 3/91, Dep. Informática, PUC-Rio, 1991.

[Jal89] P. Jalote. Functional Refinement and Nested Objects for Object-Oriented Design, *IEEE Transactions on Software Engineering SE-15*(3), 1989, pp. 264–70.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*, Prentice-Hall International, 1986. (Series in Computer Science)

[Lam77] B. Lampson et al. Report on the Programming Language Euclid, *Sigplan Notices 12*(2), 1977.

[Lea90] G. Leavens. Reasoning about Object-Oriented Programs that Use Subtypes, *Sigplan Notices 25*(10), 1990, pp. 212–23. (OOPSLA/ECOOP'90 Proceedings)

[Lun89] C. Lunau. Separation of Hierarchies in Duo-Talk, *Journal of Object-Oriented Programming 2*(2), 1989, pp. 20–26.

[Mad86] O. Madsen. Block Structure and Object Oriented Languages, *Sigplan Notices 21*(10), 1986, pp. 133–42.

[Maj77] M. Majster. Limits of the "Algebraic" Specification of Abstract Data Types, *Sigplan Notices 12*(10), 1977, pp. 37–42.

[Me88a] B. Meyer. Eiffel – A Language and Environment for Software Engineering, *The Journal of Systems and Software 8*(3), 1988, pp. 199–246.

[Me88b] B. Meyer. *Object-Oriented Software Construction*, Prentice-Hall International, 1988. (Series in Computer Science)

[Mey89] B. Meyer. You Can Write, But Can You Type?, *Journal of Object-Oriented Programming 1*(6), 1989, pp. 58–70.

[SPB90] S. Schuman, D. Pitt, P. Byers. *Object-Oriented Process Specification*, Technical Report CS-90-01, Computing Sciences, University of Surrey, 1990.

[WeZ88] P. Wegner & S. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like, *ECOOP'88 Proceedings*, Springer-Verlag, 1988, pp. 55–77. (LNCS 322)