PUC

Monografias em Ciência da Computação
nº 27/92

A Formal Specification for a Hierarchy of Collections

Roberto Ierusalimschy

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22454-970
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 27/92
Editor: Carlos J. P. Lucena

Julho, 1992

A Formal Specification for a Hierarchy of Collections *

Roberto Ierusalimschy

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Abstract

This paper presents a formal specification for a hierarchy of types similar to the Collection hierarchy presented by the Smalltalk language. The specification method is an extension of VDM that supports inheritance of specifications, with the property that subtypes are behavior compatible with their parents. This formalism gives us a clear concept of behavior compatibility, that is used to justify our hierarchy structure and to compare inheritance of specifications, adopted here, with inheritance of implementations, adopted in Smalltalk.

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

Resumo

Este artigo apresenta uma especificação formal para uma hierarquia de tipos semelhante à hierarquia "Collection" apresentada pela linguagem Smalltalk. O método de especificação é uma extensão de VDM com suporte a herança de especificações, com a propriedade de subtipos terem sempre comportamento compatível com seus supertipos. Este formalismo nos dá um conceito preciso de compatibilidade de comportamento, que é usado para justificar nossa estrutura hierárquica e para comparar herança de especificações, empregada no artigo, com herança de implementações, adotada em Smalltalk.

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

A Formal Specification for a Hierarchy of Collections

Roberto Ierusalimschy
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro – Brazil*

July 14, 1992

Abstract

This paper presents a formal specification for a hierarchy of types similar to the Collection hierarchy presented by the Smalltalk language. The specification method is an extension of VDM that supports inheritance of specifications, with the property that subtypes are behavior compatible with their parents. This formalism gives us a clear concept of behavior compatibility, that is used to justify our hierarchy structure and to compare inheritance of specifications, adopted here, with inheritance of implementations, adopted in Smalltalk.

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

1 Introduction

Among the new concepts introduced by object-oriented languages, maybe the most important one is the concept of inheritance, and the associate notion of subclass (or subtype). Many authors consider inheritance as the distinction mark of object-oriented programming, and is common the "equation" *Objects* = *AbstractDataTypes* + *Inheritance* (e.g. [1]). On the other hand, inheritance is also a very controversial subject. There is nothing near a consensus about the meaning of inheritance, and is usually impossible to map type hierarchies written in an object-oriented language into a different language.

One of the sources of such diversity is the choice between multiple and single inheritance. In a hierarchy with multiple inheritance, a type can inherit properties from many different supertypes, while in a single hierarchy every type has at most one parent. Although single inheritance puts an artificial constrain in a type hierarchy, it is sometimes preferred because it avoids many conceptual difficulties posed by multiple inheritance, and also allows more efficient implementations.

Another reason for the diversity of inheritance concepts are the different degrees of compatibility between parents and children that a hierarchy may impose. Wegner & Zdonick [2] classifies the following compatibility levels, in order of increasing congruence:

cancel compatibility: A child has no commitment with its parents' definitions, and is able to redefine or even eliminate any inherited operation.

name compatibility: A child must have at least all operations defined for its parents, or operations with the same names. Apart the names, there is no other commitment between those operations.

*on leave at the University of Waterloo (from Mar, 91 to Feb, 92).

signature compatibility: A child must have at least all operations defined for its parents, and each child operation must have a type compatible with the type specified by the parent.

behavior compatibility: A child must have at least all operations defined for its parents, and the operations must have compatible behaviors.

Of course, the third criterion needs a concept of type, and it only can be applied in typed languages, while the forth criterion needs a clear definition of behavior in order to define "compatible behaviors".

In this paper, we present a formal definition for behavior compatibility, in the framework of VDM specifications [3]. We propose an extension to the specification language that allows inheritance of specifications, and we prove that, in our system, every type is behavior compatible with its parents, in the sense that it is a valid representation for them. Using this language, we specify a hierarchy of collections, inspired by the hierarchy *Collection* of the Smalltalk language [4]. We argue that behavior compatibility can be a powerful (and rigorous) mechanism to structure types. Also we show that, with proper design, behavior compatibility can be very flexible, and even allows cancellation of inherited operations.

The rest of the paper is organized as follows. The next section presents our extensions of VDM to support inheritance of specifications, and proves a lemma that relates it with behavior compatibility. Section 3 describes the collection classes in Smalltalk, and discusses some issues related with their formalization. In section 4 we show the formal specification for the hierarchy of collections. Section 5 discusses some problems arisen by multiple inheritance. Finally, section 6 draws some conclusions.

2 Inheritance of Specifications

In order to talk about behavior compatibility, we first need a clear concept of behavior. In this work we adopt a formal specification of a type as the main description of its behavior. The specification language we use is VDM, and we follow the specification method described in [3], with some extensions to support inheritance of specifications[1]. In this context, we say that a type S is behavior compatible with a type P if S is a valid representation of P.

In order to support inheritance, we adopted a more object-oriented syntax, that groups together the type declaration and all its operations. So, instead of writing in the usual way:

$$P :: f_1 : T_1$$

$$\ldots$$

where

$$\mathit{inv}\text{-}P(\mathit{mk}\text{-}P(f_1, \ldots)) \quad \triangleq \quad \mathit{invP}$$

O_1 $(\ldots)$
ext extO_1
pre preO_1
post postO_1

[1]The rationale for these extensions is presented in [5] and [6].

...
we are going to use the following notation:

Specification P
  $f_1 \colon T_1$
  ...
  $inv\text{-}P \;\triangleq\; invP$
  Operation $O_1\ (\ldots)$
  ext $extO_1$
  pre $preO_1$
  post $postO_1$
  ...
End P

The main extension to the language is a notation to declare a type as a heir of other types. We do this in a way such that a heir is always behavior compatible with its parents. Suppose we have the following specification:

Specification S
  Subtype of P_1
    rename $O^1_{rn_1}$ as $NO^1_{rn_1}$, $O^1_{rn_2}$ as $NO^1_{rn_2}, \ldots$
    redefine $O^1_{rd_1}, O^1_{rd_2}, \ldots$
  Subtype of P_2
    rename $O^2_{rn_1}$ as $NO^2_{rn_1}$, $O^2_{rn_2}$ as $NO^2_{rn_2}, \ldots$
    redefine $O^2_{rd_1}, O^2_{rd_2}, \ldots$
  ...
  $f^s_1 \colon T^s_1$
  $f^s_2 \colon T^s_2$
  ...
  $inv\text{-}S \;\triangleq\; invS$
  Operation $O^s_1\ (\ldots)$
  ext $extO^s_1$
  pre $preO^s_1$
  post $postO^s_1$
  Operation $O^s_2\ (\ldots)$
  ext $extO^s_2$
  pre $preO^s_2$
  post $postO^s_2$
  ...
End S

The above specification declares S as a *subtype* of P_i; P_i, in turn, is called a *supertype* of S. The following rules give the meaning of the above specification:

1. The actual fields of S are the join of all the fields from $P_1, \ldots,$ and S. If fields from different specifications have the same name and the same type (textually equal), then they are merged in one field. If fields from different specifications have the same name but different types, then there is an error condition (i.e. the meaning of the declaration is undefined).

2. The actual invariant of S is $\textit{inv-}P'_1 \wedge \ldots \wedge \textit{invS}$. The definition of $\textit{inv-}P'_i$ is as follows:

   $\textit{inv-}P'_i : S \to \mathbf{B}$

   $\textit{inv-}P'_i(s) \triangleq \textit{inv-}P_i(\textit{projP}_i(s))$

   where:

   $\textit{proj}_{P_i} : S \to P_i$

   $\textit{proj}_{P_i}(\textit{mk-}S(\ldots, f_1^{P_i}, \ldots)) \triangleq \textit{mk-}P_i(f_1^{P_i}, \ldots)$

   that is, $\textit{proj}_{P_i}$ is the orthogonal projection from S to P_i (remember that, by rule 1, S has all the fields from P_i).

   Notice that we use *invS* (without an hyphen) to denote the textual invariant, as written in the specification, while *inv-S* denotes the final logical function that results from the above operation. The same distinction applies to pre- and post-conditions.

   It is easy to verify that $\textit{inv-}P_i$ and $\textit{inv-}P'_i$ can be textually identical; the only difference between them is that the latter applies to S, and so it must "throw off" some fields. Notice that *invS* can refer to the fields inherited from other specifications.

3. The operations of P_i are inserted into S in the following way: first, the operations $O^i_{rn_1}, \ldots,$ are renamed as $NO^i_{rn_1}, \ldots$. Then, from this set, the operations $O^i_{rd_1}, \ldots$ are removed. The remaining operations are included in S with unmodified external lists, pre- and post-conditions. If operations inherited from different parents have the same name and the same definition (textually equal), then they are merged. If operations from different parents have the same name but different definitions, then there is an error condition.

4. For each operation in the redefinition list (i.e., each $O^i_{rd_j}$) there must be an operation O^s_k with the same name and same parameter list; this operation *redefines* $O^i_{rd_j}$ (each declaration O^s_k can redefine more than one inherited operation). Suppose that an operation O^s_k redefines the operations $O_{k_1}, O_{k_2}, \ldots$, inherited from the specifications $P_{l_1}, P_{l_2}, \ldots$. Then, its actual specification is a combination of its textual specification and the specifications of all O_{k_i}, according to the following rules:

   (a) The external list of O^s_k is the join of its textual external list ($\textit{extO}^s_k$) with the external lists of O_{k_i}. The list $\textit{extO}^s_k$ can not include fields inherited from any specification P_{l_i}. Intuitively, this is justified by the fact that an operation must have a behavior compatible with the operations it redefines. If the inherited definition asserts that some fields are not modified (by their absence in the external list), the new operation must keep this assertion.

   (b) The actual pre-condition of O^s_k is

   $\textit{preO}^s_k \vee \textit{pre-}O'_{k_1} \vee \ldots$

   Again, we have that

   $\textit{pre-}O'_{k_i} : S \to \mathbf{B}$

   $\textit{pre-}O'_{k_i}(s) \triangleq \textit{pre-}O_{k_i}(\textit{proj}_{P_{l_i}}(s))$

where P_{l_i} is the specification from where O_{k_i} is inherited.

We assume that when an operation is a redefinition, then the absence of an explicit pre-condition stands for false, instead of the usual true, so that the actual pre-condition simplifies to the conjunction of the inherited conditions.

(c) The post-condition of O^s_k is

$postO^s_k \wedge (pre\text{-}O'_{k_1} \;\Rightarrow\; post\text{-}O'_{k_1}) \wedge \ldots$

As expected, the definition of $post\text{-}O'_{k_i}$ is as follows:

$post\text{-}O'_{k_i} : S \times S \to \mathbf{B}$

$post\text{-}O'_{k_i}(\overleftarrow{s}, s) \;\triangleq\; post\text{-}O_{k_i}(projp_{P_{l_i}}(\overleftarrow{s}), projp_{P_{l_i}}(s))$

Notice that, to avoid inconsistencies between $postO^s_k$ and the inherited behavior we need the condition over external lists (rule 4a). Otherwise, the implicit requirement about unchanged variables could be contradicted by $postO^s_k$.

The other operations of S, which are not redefining any inherited operation, are left unchanged.

5. All operations in S must fulfill the *satisfiability* proof obligation. This must be checked even for the inherited operations, because the new invariant can nullify this property.

Now, we can state the important property of the above definition.

Lemma: Apart from renames, a subtype S of a type P is a valid representation for the type P, that is, it satisfies the following properties:

1. there is a retrieve function from S to P,
2. S has all operations that P has, and
3. the operations in S model the correspondent operations in P.

Proof:

1. The obvious choice for a retrieve function from S to P is the orthogonal projection function $projp$. This function is total, as by rule 2 we can deduce that, for all $s \in S$, $inv\text{-}S(s) \vdash inv\text{-}P(projp(s))$.

   A subtle point here is the *adequacy criterion*[2]. Technically, our retrieve function can be not onto, because the invariant of S can put stronger restrictions over the fields of P. This fact corresponds to situations where a type has a level of indeterminacy that is reduced by a subtype. In a typical hierarchy this is very common, as many specifications are declared as generalizations of existing types, with a high degree of indeterminacy[3]. So, in our framework it is not worth pursuing adequacy. Instead, we will use the more general proof rules for operation modeling, that relies on a relation between abstraction and representation. The obvious choice for such relation is:

   $relp : P \times S \to \mathbf{B}$

   $relp(p, s) \;\triangleq\; p = projp(s)$

[2]This criterion states that a representation function must be onto, i.e., for each P value there must be at least one S value representing it.

[3]A good example is the type *Collection*, the root of the hierarchy presented in the next section.

	from $rel_P(\overleftarrow{p}, \overleftarrow{s}) \wedge pre\text{-}O_P(\overleftarrow{p}) \wedge post\text{-}O_S(\overleftarrow{s}, s)$	
1	$proj_P(s) \in P$	total-$proj_P$
2	$rel_P(proj_P(s), s)$	rel_P-defn($\wedge$-E(h))
3	$\ldots \wedge (pre\text{-}O'_P(\overleftarrow{s}) \Rightarrow post\text{-}O'_P(\overleftarrow{s}, s)) \wedge \ldots$	$post\text{-}O_S$-defn($\wedge$-E(h))
4	$\overleftarrow{p} = proj_P(\overleftarrow{s})$	rel_P-defn($\wedge$-E(h))
5	$pre\text{-}O_P(proj_P(\overleftarrow{s}))$	$\wedge$-E(h),4
6	$pre\text{-}O'_P(\overleftarrow{s})$	$pre\text{-}O'_P$-defn(5)
7	$post\text{-}O'_P(\overleftarrow{s}, s)$	$\Rightarrow$ -E($\wedge$-E(3),6)
8	$post\text{-}O_P(proj_P(\overleftarrow{s}), proj_P(s))$	$post\text{-}O'_P$-defn(7)
9	$post\text{-}O_P(\overleftarrow{p}, proj_P(s)) \wedge rel_P(proj_P(s), s)$	$\wedge$-I(8,2),4
	infer $\exists p \in P \cdot post\text{-}O_P(\overleftarrow{p}, p) \wedge rel_P(p, s)$	$\exists$-I(1,9)

Figure 1: proof of the result rule

2. The operations of P that do not appear in the redefinition list are straightly included in S (rule 3); the operations in the redefinition list are explicitly declared in S (rule 4).

3. The operations of P that do not appear in the redefinition list are textually identical in S and P, and so are valid models. Let us see the redefined operations. Suppose the operation O_S redefines an operation O_P. The general proof obligations for operation modeling are:

   - $\forall s \in S, p \in P \cdot rel_P(p, s) \wedge pre\text{-}O_P(p) \Rightarrow pre\text{-}O_S(s)$

   - $\forall \overleftarrow{s}, s \in S, \overleftarrow{p} \in P \cdot$
     $rel_P(\overleftarrow{p}, \overleftarrow{s}) \wedge pre\text{-}O_P(\overleftarrow{p}) \wedge post\text{-}O_S(\overleftarrow{s}, s) \Rightarrow \exists p \in P \cdot post\text{-}O_P(\overleftarrow{p}, p) \wedge rel_P(p, s)$

   The first proof obligation is trivial, if we remember that the retrieve function is the orthogonal projection and that $pre\text{-}O_S$ has the form $\ldots \vee pre\text{-}O'_P \vee \ldots$ (rule 4). A proof of the second one is presented in figure 1.

This lemma assures that a subtype, in our specification language, is always behavior compatible with its supertypes. More specifically, we have a "is-a" relationship between them. Every object of a given type (that is, that satisfies a given specification) also belongs to all supertypes of that type.

It is important to note the difference between a subtype and an implementation. In the definition of subtypes we said nothing about initial states. The lemma tells us about the behavior of existing objects, but not about the creation of objects. In the specification of a subtype, one can stretch the invariant, narrowing the range of initial states. For instance, one can define a subtype *Square* of a type *Rectangle*, stating in the invariant the equality of width and height. On the other hand, it is obvious that *Square* is not a good implementation for generic rectangles.

However, it is not possible to build subtypes in unrestricted ways: the ultimate restriction for the range of admissible subtypes is posed by rule 5. It is not possible to stretch a post-condition more than allowed by the satisfiability criterion. Moreover, one can not put a new invariant that conflicts with the inherited post-conditions. So, the limits to modify a type are given not only by its model, but also by its operations. As an example, we can return to the type *Rectangle*. If the

- *Collection*
  - *Bag*
  - *IndexedCollection*
    - *FixedSizeCollection*
      - *Array*
      - *ByteArray*
      - *Interval*
      - *String*
        - *Symbol*
    - *OrderedCollection*
      - *SortedCollection*
  - *Set*
    - *Dictionary*
      - *IdentityDictionary*

Figure 2: The Smalltalk hierarchy of collections

type is immutable, then a subtype *Square* only restricts the range of initial states, and is correct. However, if *Rectangle* has an operation like *SET_WIDTH*, that changes the width for an arbitrary value and does not modify the height, than this operation would be unsatisfiable under the *Square* invariant. In that case, *Square* would not be a valid subtype of *Rectangle*.

3 Smalltalk Hierarchy × Formal Hierarchy

The Smalltalk library is organized as a hierarchy of classes, rooted in the class *Object*. All classes representing collections of objects, like sets and bags, are united in a subtree under the class *Collection*.

Before going on, it is important to notice that Smalltalk implements what we call *inheritance of implementations*. This means that the inheritance mechanism has no relationship with a concept of subtype or with the polymorphism of the language; instead, it is solely a mechanism for code reuse. As a consequence, Smalltalk presents only name compatibility between subclasses; moreover, as an operation can be redefined to rise an error if called, the language actually allows cancel compatibility. Implementation aspects, together with the restriction of single inheritance, guides the organization of the Smalltalk library hierarchy. The position of a class inside the hierarchy is mainly dictated by where it can reuse more code.

On the other hand, the formal hierarchy we are going to present is based on *inheritance of specifications*, that is, behavior compatibility. Moreover, our formal system allows multiple inheritance. So, it is expected that we have to modify the original hierarchy to fit the formal specification. Nevertheless, the final hierarchy encompasses all interesting classes under a root *Collection*, and it is rather similar to the Smalltalk structure.

The primary members of the *Collection* class hierarchy in Smalltalk[4] are shown in figure 2. The *Collection* class defines the basic functionality for all collections. *Bag*, *Set*, and *Array* have the expected meanings. *IndexedCollection* introduces operations for indexing elements by their positions inside the collection, and so introduces an order among the elements. *FixedSizeCollection* comprises the collections that have their size specified at creation time, while an *OrderedCollection*

[4]We are following the library presented in [7].

allows insertion of elements at both ends, growing as needed. The classes *ByteArray*, *String*, and *Symbol* are all special implementations for arrays, optimized to handle integers and characters; from a specification point of view they are similar to *Array*, and so we will not consider them here. Class *Interval* implements immutable sequence of integers in arithmetic progression. *Dictionary* corresponds to generic symbol tables, while *IdentityDictionary* differs from its parent only in the way it does key comparison, and will not be considered here, either.

In order to match the *Collection* tree with a formal hierarchy, and also to make use of multiple inheritance, we made the following modifications. First, as already explained, we removed the classes *ByteArray*, *String*, *Symbol*, and *IdentityDictionary*. On the other hand, we introduced the types *Stack*, *Queue* and *DoubleQueue*, mainly because we think no paper about abstract data types is complete without a stack[5].

The operation to remove a generic element poses some difficulties. Not all collections allow the deletion of a generic element: for instance, in a stack one can only remove the element at the top. We solved this point with multiple inheritance. We declared the remove operation in a subtype of *Collection*, called *RemovableCollection*. Any type that needs the operation can inherit it, while keeping all other parents.

The other changes were all in the *IndexedCollection* subtree. From a behavior perspective, many ordered collections are not compatible with an indexing specification. For instance, a sorted collection does not have an operation to put an arbitrary element in an fixed position — Smalltalk uses cancel compatibility to achieve that. Stacks and queues do not allow access to arbitrary elements, either. So, we declared *SortedCollection* as a sibling of *IndexedCollection*, and created a new type above them. As this new type encompasses all kinds of collections with an internal order, we took the name *OrderedCollection* for it, renaming the old *OrderedCollection* as *List* in our hierarchy. Finally, we took advantage of multiple inheritance to put *FixedSizeCollection* as a direct child of *Collection*, making it a sibling of *OrderedCollection*. Then, an array can be declared as an indexed collection with a fixed size, inheriting both behaviors.

4 The Specification of The Collection Hierarchy

In this section we present a formal specification for a hierarchy of collections, using the method presented in section 2.

The root of the hierarchy is the type *Collection* (figure 3). That specification uses two auxiliary functions:

$$mpc : (T \xrightarrow{m} \mathbb{N}_1) \times T \to \mathbb{N}$$

$$mpc(m, e) \quad \triangleq \quad \text{if } e \in \text{dom } m \text{ then } m(e) \text{ else } 0$$

$$bag\text{-}size : T \xrightarrow{m} \mathbb{N}_1 \to \mathbb{N}$$

$$bag\text{-}size(m) \quad \triangleq \quad \sum_{i \in \text{dom } m} m(i)$$

Both operate over bags, implemented as a map that gives, for each element in the bag, its number of occurrences. *mpc* gives the number of occurrences of a given element inside a bag, and *bag-size* returns the total number of elements in a bag.

The basic structure of a collection is a bag, stored in the field *contents*. Other options would be a set or a sequence. A set would not allow us to define a generic *INSERT* operation, as one would

[5]It is amazing that the Smalltalk library does not include a class Stack. Maybe more amazing is the fact that it does not need stacks.

Specification *Collection*

$contents: T \xrightarrow{m} \mathbb{N}_1$
$maxSize: \mathbb{N}$

Operation *INSERT* $(e: T)$
ext wr $contents: T \xrightarrow{m} \mathbb{N}_1$
rd $maxSize: \mathbb{N}$
pre $\textit{bag-size}(contents) < maxSize$
post $e \in \mathrm{dom}\, contents \wedge mpc(contents, e) \geq mpc(\overleftarrow{contents}, e) \wedge$
$(\forall x \in T \cdot x \neq e \;\Rightarrow\; (mpc(contents, x) = mpc(\overleftarrow{contents}, x)))$

Operation *CONTAINS* $(e: T)\; b: \mathbb{B}$
ext rd $contents: T \xrightarrow{m} \mathbb{N}_1$
post $b = (e \in \mathrm{dom}\, contents)$

Operation *IS_EMPTY* $()\; b: \mathbb{B}$
ext rd $contents: T \xrightarrow{m} \mathbb{N}_1$
post $b = (\textit{bag-size}(contents) = 0)$

Operation *GET_ELEMENT* $()\; e: T$
ext rd $contents: T \xrightarrow{m} \mathbb{N}_1$
pre $\textit{bag-size}(contents) > 0$
post $e \in \mathrm{dom}\, contents$

Operation *SIZE* $()\; s: \mathbb{N}$
ext rd $contents: T \xrightarrow{m} \mathbb{N}_1$
post $s = \textit{bag-size}(contents)$

End *Collection*

Figure 3: Collection

Specification *RemovableCollection*
  Subtype of *Collection*
  Operation *REMOVE* $(e: T)$
  ext wr $contents: T \xrightarrow{m} \mathbf{N}_1$
  pre $mpc(contents, e) > 0$
  post $(\forall x \in T \cdot x \neq e \Rightarrow (mpc(contents, x) = mpc(\overleftarrow{contents}, x))) \wedge$
    $mpc(contents, e) = mpc(\overleftarrow{contents}, e) - 1$
End *RemovableCollection*

Figure 4: RemovableCollection

be able to deduce that the size of a collection does not change after the insertion of an already present element, an obviously false statement for generic collections. A sequence would be too biased for non-ordered collections, like sets. The other field of the type Collection is its maximum size.

Notice that, although we define a maximum size, we do not set an invariant to assure it. The check is made only in the pre-condition of the insertion operation. In this way, we can define unbounded collections just weakening this pre-condition to true. We do not do that in this hierarchy, because ultimately all real collections are bounded, and any program that uses a collection must be able to treat an overflow in a sensible way.

From the operations of *Collection*, the only one a little more complex is *INSERT*. We must make sure that its specification is flexible enough to allow redefinitions for all kinds of collections, but at the same time captures all the commonality of insert operations. The pre-condition only checks the available space. The post-condition has three terms: the first one asserts the presence of the inserted element in the final collection; the second asserts that the number of such elements does not decrease; and the third part assures that the operation does not affect other elements.

Not all collections support a generic remove operation, that is, one that can remove an arbitrary element. So we define this operation in a separate type, called *RemovableCollection* (figure 4). *RemovableCollection* inherits all operations from the basic collection, and adds the operation *REMOVE*. The pre-condition asserts the presence of the element to be removed, and the post-condition states that the number of such elements is decremented by 1 after the operation.

Both *Collection* and *RemovableCollection* are examples of what is called an *abstract class* in the Smalltalk terminology. They are not intended to have implementations, but rather to be used as supertypes for other types. Using them we can define the first two *concrete classes* of our hierarchy: *Set* and *Bag* (figure 5). Both are subtypes of *RemovableCollection*, as they allow the deletion of any contained element. For the type *Set*, the only modification is a stronger invariant, disallowing repetitions. For bags, the invariant is correct, but we must redefine the operation *INSERT* to make sure it always inserts one and only one element. Notice that the redefinition only specifies this new property, that is joined with the inherited properties according to the rules of section 2.

Our next specification is another abstract class, that encompasses all collections with an internal order for its elements (figure 6). To keep track of this order, we use a sequence, called *order*; the invariant assures the sequence and the bag (*contents*) have the same elements. The auxiliary function *seq-count* counts the number of occurrences of a given element inside a sequence.

Specification *Set*

Subtype of *RemovableCollection*

$\textit{inv-Set} \triangleq \forall x \in \mathsf{dom}\, contents \cdot contents(x) = 1$

End *Set*

Specification *Bag*

Subtype of *RemovableCollection*

redefine *INSERT*

Operation *INSERT* $(e{:}\ T)$

post $mpc(contents, e) = mpc(\overleftarrow{contents}, e) + 1$

End *Bag*

Figure 5: Sets and Bags

Specification *OrderedCollection*

Subtype of *Collection*

rename *GET_ELEMENT* as *GET_FIRST*

redefine *GET_FIRST*, *INSERT*

$order{:}\ T^*$

$\textit{inv-OrderedCollection} \triangleq \forall x \in T \cdot \textit{seq-count}(order, x) = mpc(contents, x)$

Operation *GET_FIRST* $()\ e{:}\ T$

ext rd $order{:}\ T^*$

post $e = \mathsf{hd}\ order$

Operation *INSERT* $(e{:}\ T)$

ext wr $order{:}\ T^*$

pre false

post true

End *OrderedCollection*

Figure 6: OrderedCollection

```
**Specification** *Stack*
  **Subtype of** *OrderedCollection*
    rename $GET\_FIRST$ as $TOP$, $INSERT$ as $PUSH$
    redefine $PUSH$
  **Operation** $PUSH$ $(e: T)$
  post $order = [e] \frown \overleftarrow{order}$
  **Operation** $POP$ $()$ $e: T$
  ext   wr $order: T^*$
        wr $contents: T \xrightarrow{m} \mathbf{N}_1$
  pre   len $order > 0$
  post $order = \text{tl}\,\overleftarrow{order} \wedge e = \text{hd}\,\overleftarrow{order}$
**End** *Stack*
```

Figure 7: Stack

$$seq\text{-}count : T^* \times T \to \mathbf{N}$$

$$seq\text{-}count(s, x) \;\triangleq\; \text{if } s = [\,] \text{ then } 0 \text{ else}$$
$$\quad \text{if } x = \text{hd}\, s \text{ then } seq\text{-}count(\text{tl}\, s, x) + 1 \text{ else } seq\text{-}count(\text{tl}\, s, x)$$

There is a little trick in the redefinition of *INSERT*. To keep the invariant, *INSERT* must modify the field *order* according to *contents*. But where to put the new element depends on the kind of collection. A stack may need to insert the new element at the beginning of the sequence, while a queue would insert at the end. So, the redefinition of *INSERT* declares *order* in its external list to allow the maintenance of the invariant, but does not strengthen the post-condition; this is left for the subtypes. The redefinition of *GET_ELEMENT* to *GET_FIRST* is done because the latter seems to be more useful (but not available in unordered collections).

The simplest subtypes of *OrderedCollection* are *Stack* (figure 7) and *Queue* (figure 8). Each one redefines *INSERT* in order to specify the position of the inserted element, at the beginning (Stacks) or at the end (Queues) of the sequence. Also, each one specifies a particular operation to remove elements (*POP* and *REMOVE*). As they do not allow the deletion of a generic element, they can not inherit this operation from *RemovableCollection*.

Using stacks and queues, it is trivial to declare a double queue (figure 9). We only need to add one new operation, to remove elements from the end. Notice the use of the redefinition and rename facilities to join two inherited operations (*POP* and *REMOVE*) into one (*REMOVE_FIRST*).

Another useful subtype of *OrderedCollection* is *SortedCollection* (figure 10). For this type we assume that the basic type T has a total order, given by the infix function "$\leq_T$". The specification joins the basic properties from *OrderedCollection* with other properties from *Bag*, adding an invariant stating that the elements inside *order* must be sorted. From *Bag* comes the restriction that *INSERT* adds only one element to the collection. This property and the invariant are enough to completely specify the insertion; the redefinition is used again only to join both definitions (in this case we have not needed to rename them, as they already had the same name). *SortedCollection* also has a *REMOVE* operation, inherited from *Bag*. The redeclaration allows the operation to modify the field *order*. Again, the invariant is enough to ensure that this field is modified in the

Specification *Queue*
  Subtype of *OrderedCollection*
    redefine *INSERT*

  Operation *INSERT* $(e\colon T)$
  post $order = \overleftarrow{order} \frown [e]$

  Operation *REMOVE_FIRST* $()\ e\colon T$
  ext wr $order\colon T^*$
       wr $contents\colon T \xrightarrow{m} \mathbb{N}_1$
  pre $\mathsf{len}\ order > 0$
  post $order = \mathsf{tl}\ \overleftarrow{order} \wedge e = \mathsf{hd}\ \overleftarrow{order}$
End *Queue*

Figure 8: Queue

Specification *DoubleQueue*
  Subtype of *Queue*
    rename *INSERT* as *INSERT_LAST*
    redefine *REMOVE_FIRST*
  Subtype of *Stack*
    rename *PUSH* as *INSERT_FIRST*, *POP* as *REMOVE_FIRST*
    redefine *REMOVE_FIRST*

  Operation *REMOVE_FIRST* $()\ e\colon T$
  pre false
  post true

  Operation *REMOVE_LAST* $()\ e\colon T$
  ext wr $contents\colon T \xrightarrow{m} \mathbb{N}_1$
       wr $order\colon T^*$
  pre $\mathsf{len}\ order > 0$
  post $\overleftarrow{order} = order \frown [e]$
End *DoubleQueue*

Figure 9: DoubleQueue

```
Specification SortedCollection
  Subtype of OrderedCollection
    redefine INSERT
  Subtype of Bag
    redefine INSERT, REMOVE
  inv-SortedCollection ≜ ∀i, j ∈ inds order · i < j ⇒ c(i) ≤_T c(j)
  Operation INSERT (e: T)
  pre  false
  post true
  Operation REMOVE (e: T)
  ext  wr order: T*
  pre  false
  post true
End SortedCollection
```

Figure 10: SortedCollection

proper way.

Indexed collections are specified at figure 11. The main operations are AT, to access an element by its index (i.e., its position within the sequence), and PUT, to modify the element at a given position. Both operations check at the pre-condition that the index is inside the bounds of the sequence. The post-condition of PUT must assure that only the specified position is modified, without disturbing the other elements in the sequence.

A generic list structure can be defined only combining predefined behaviors — see figure 12. Such structure supports insertion and deletion at both ends (from type *DoubleQueue*), general indexing facilities (from *IndexedCollection*), and a remove operation (from *RemovableCollection*).

In order to specify a common array, we need a trick. At a first glance, arrays could not be a subtype of collection, because there is no way to insert an element into an array, but only to change elements: an array does not have "empty slots". However, we can disallow the $INSERT$ operation and still keep behavior compatibility with collections. This is done in the specification *FixedSizeCollection* (figure 13), simply stating in the invariant that such collections are always full, and so the pre-condition for an insertion can never be satisfied! Now, an array is just an indexed collection with a fixed size, and its specification is shown in figure 14. Notice that a type invariant must be true at the initial state of an object, but our specification does not define how to achieve this (and does not need to). A typical implementation may fill new arrays with an invalid or neutral initial value.

The specification for intervals is shown in figure 15. As a subtype of *FixedSizeCollection*, it has an invalid $INSERT$ operation. The invariant states that the sequence must be an arithmetic progression. The type T must be $\mathbf{N}$, $\mathbf{Z}$ or $\mathbf{R}$, but our language does not have a notation for this restriction. This specification describes a very unusual kind of collection, and it is defined here only because it is present in the Smalltalk hierarchy.

Our last specification is a dictionary, or a symbol-table (figure 16). Following the Smalltalk hierarchy, we declared it as a subtype of *Set*. The key to keep a compatible behavior is to identify the set with the keys in the dictionary. We rename most operations to reflect this concept. The

Specification *IndexedCollection*
  Subtype of *OrderedCollection*

  Operation $AT\ (i{:}\,\mathrm{N})\ e{:}\ T$
  ext rd $order{:}\ T^*$
  pre $i \in \mathsf{inds}\ order$
  post $e = order(i)$

  Operation $PUT\ (i{:}\,\mathrm{N}, e{:}\ T)$
  ext wr $order{:}\ T^*$
     wr $contents{:}\ T \xrightarrow{m} \mathrm{N}_1$
  pre $i \in \mathsf{inds}\ order$
  post $\mathsf{len}\ order = \mathsf{len}\ \overleftarrow{order} \land order(i) = e \land \forall j \in \mathsf{inds}\ order \cdot j \neq i \;\Rightarrow\; order(j) = \overleftarrow{order}(j)$

End *IndexedCollection*

Figure 11: IndexedCollection

Specification *List*
  Subtype of *DoubleQueue*
  Subtype of *IndexedCollection*
  Subtype of *RemovableCollection*
End *List*

Figure 12: List

Specification *FixedSizeCollection*
  Subtype of *Collection*
  $inv\text{-}FixedSizeCollection \;\triangleq\; bag\text{-}size(contents) = maxSize$
End *FixedSizeCollection*

Figure 13: FixedSizeCollection

Specification *Array*
  Subtype of *FixedSizeCollection*
  Subtype of *IndexedCollection*
End *Array*

Figure 14: Array

Specification *Interval*
  Subtype of *FixedSizeCollection*
  Subtype of *OrderedCollection*

  $\mathit{inv\text{-}Interval} \triangleq \exists a, r \in \mathbf{R} \cdot \forall i \in \mathsf{inds}\, \mathit{order} \cdot \mathit{order}(i) = i * r + a$

  Operation $AT\ (i\colon \mathrm{N})\ e\colon T$
  ext rd $\mathit{order}\colon T^*$
  pre $i \in \mathsf{inds}\, \mathit{order}$
  post $e = \mathit{order}(i)$
End *Interval*

Figure 15: Interval

```
Specification Dictionary
  Subtype of Set
    rename INSERT as INSERT_KEY, CONTAINS as CONTAINS_KEY
    redefine INSERT_KEY, REMOVE
   values: $T \xrightarrow{m} V$
  inv-Dictionary $\triangleq$ dom values = dom contents
  Operation INSERT_KEY (e: T)
  ext   wr values: $T \xrightarrow{m} V$
  pre   false
  post  true
  Operation REMOVE (e: T)
  ext   wr values: $T \xrightarrow{m} V$
  pre   false
  post  true
  Operation SET_VALUE (e: T, v: V)
  ext   wr contents: $T \xrightarrow{m} N_1$
        wr values: $T \xrightarrow{m} V$
        rd maxSize: N
  pre   $bag\text{-}size(contents) < maxSize \lor e \in \mathrm{dom}\ contents$
  post  $values = \overleftarrow{values} \dagger \{e \mapsto v\}$
  Operation GET_VALUE (e: T) v: V
  ext   rd values: $T \xrightarrow{m} V$
  pre   $e \in \mathrm{dom}\ values$
  post  $v = values(e)$
End Dictionary
```

Figure 16: Dictionary

new field, *values*, tracks the relationship between keys and values, and the invariant assures our identification between sets and keys. Notice that we could get a much simpler specification using only the field *value*, without any inheritance, but the point here is to show that a dictionary can be (formally) defined as a subtype of sets, according to the Smalltalk hierarchy.

5 Some Difficulties with Multiple Inheritance

Whenever a system allows multiple inheritance, it arises the possibility of different kinds of conflicts among the heritage. Object Oriented programming languages have introduced many mechanisms to cope with these conflicts, ranging from single inheritance (e.g. Smalltalk [4]) to some scheme of priorities (e.g. Common Lisp [8]) to facilities for renaming and redefining (Eiffel [9]). Obviously, single inheritance is not a solution to multiple inheritance. The scheme of priorities adopted by some languages can lead to unexpected results in more intricate hierarchies. Moreover, although some systems allow some form of control over the solution (for instance, by means of changing the order of the subtype declarations), such control is very restricted. So we have adopted here a solution based on explicit statements to avoid conflicts, the *renames* and *redefines* declarations. Nevertheless, it is important to keep in mind the drawbacks of those solutions.

Redefinition is a very powerful mechanism to enhance the flexibility of inheritance, even in a system with single inheritance. In our system, the semantics of redefinitions is given in such a way that subtypes are always consistent with their parents. But we still can have some problems. Suppose we try to define a new specification, which is a subtype of both *Stack* and *Queue* (pages 12 and 13). This will lead us to an unfeasible specification, as the post-conditions from both *INSERT* operations are incompatible. The strange thing is that these operations were originally the same, the *INSERT* operation from type *Collection*, that being inherited through two different pathes became incompatible.

The mechanism for renaming has more problems. The main use for renaming is to avoid name clashes between operations inherited from different parents. This is a very important facility in order to be able to join specifications written by different groups. However, the facility is not compatible with the notion of subtyping (remember that in the lemma of section 2 we did not consider renamings). A partial solution, adopted by the programming language Eiffel, is to keep track of the renames when viewing a type as a supertype. For example, if a variable of type *Collection* contains a *Stack*, when we refer to the operation *INSERT* the system automatically translate it to a reference to the operation *PUSH*. However, such solution does not work sometimes. Referring again to the above example, if the variable contains a *DoubleQueue*, the same reference to the operation *INSERT* can be translated to *INSERT_FIRST* or *INSERT_LAST*, giving different results.

6 Conclusions

We have presented a formal definition for behavior compatibility. We have extended the VDM notation to support inheritance of specifications, in a way that an heir is always behavior compatible with its ancestors. Based on this extension, we have built a hierarchy of collections, resembling the tree of collection classes in the Smalltalk library. Our hierarchy uses multiple inheritance, and follows a strict "is-a" discipline: an implementation always satisfies the formal specifications of all supertypes of its basic type.

We argue that behavior compatibility is not too restrict, mainly when a language supports separate hierarchies for types and implementations (like Duo-Talk [10] and O=M [11]). In the implementation tree, parents can be chosen according to a criterion of code reuse. On the other hand, the specification hierarchy must be based strictly in behavior considerations; this gives an objective criterion to organize the hierarchy and permits the use of formal methods for reasoning about programs with inheritance.

A key point in our method is the use of a specification language that supports indeterminacy, like VDM. This feature allows definitions of very general types in the first levels of a hierarchy; these types specify only what is known or needed at that level. Then subtypes can be defined narrowing the range of indeterminacy of the inherited behavior, while adding new behaviors (in the form of new fields and new operations). As we have shown in the collection example, this method presents great flexibility, and in some cases a proper design can even cancel inherited operations. Moreover, the method allows reuse of specifications, as shown in the definitions of *Array* and *List* — both types only combine inherited specifications to define their behavior.

A point we have neglected in our work is iteration. Collections in Smalltalk offer a variety of methods to iterate over their elements. However, all those methods are based on *block* parameters, i.e., the iterator method receives as a parameter the operation to be performed over the elements of the collection. Our specification language does not support this kind of second order functions, and so we have not included those operations in the formal specifications. In [5] we propose the use of iterator objects[6] to cope with iterations without needing second order facilities, and we also show how to include such mechanisms in a formal specification for ordered collections.

References

[1] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–71, 1988.

[2] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP'88 Proceedings*, pages 55–77, 1988.

[3] Cliff B. Jones. *Systematic Software Development using VDM.* International Series in Computer Science. Prentice Hall, second edition, 1990.

[4] Adele Goldberg and Dave Robson. *Smalltalk-80 : The Language and its Implementation.* Addison-Wesley, 1983.

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

[6] Roberto Ierusalimschy. A method for object-oriented specifications with VDM. Monografias em Ciência da Computação 2/91, PUC-Rio, Rio de Janeiro, Brazil, 1991.

[7] Digitalk, Inc., Los Angeles - CA. *Smalltalk/V Windows - Tutorial and Programming Handbook*, 1991.

[8] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification.* ANSI Common Lisp, 1988. Doc. 88-003, X3J13 Standards Committee.

[9] Bertrand Meyer. Eiffel - a language and environment for software enginnering. *The Journal of Systems and Software*, 8(3):129–46, 1988.

[6]This concept is somewhat similar to streams in Smalltalk.

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

[11] Roberto Ierusalimschy. The O=M programming language. Monografias em Ciência da Computação 3/91, PUC-Rio, Rio de Janeiro, Brazil, 1991.