PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 33/92

On the Modularization of Formal Specifications: the NDB Example Revisited

Carlos H. C. Duarte
Roberto Ierusalimschy
Carlos J. P. Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 33/92
Editor: Carlos J. P. Lucena Dezembro, 1992

On the Modularization of Formal Specifications: the NDB Example Revisited

Carlos H. C. Duarte
Roberto Ierusalimschy
Carlos J. P. Lucena

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

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
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br
techrep@inf.puc-rio.br (for publications only)

On the Modularization of Formal Specifications: The NDB Example Revisited

C. H. C. Duarte   R. Ierusalimschy   C. J. P. Lucena

(carlos, roberto, lucena@inf.puc-rio.br)

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente, 225, Rio de Janeiro, RJ, 22453, Brazil

Abstract

In this paper we study the formal specification of a DBMS formally based on the entity-relationship concept. We use the Norman's Database (NDB) example which has been explored by several authors in the recent literature. The system's operations and structure are described, by means of techniques, which extend VDM through nesting and inheritance. The extensions to the method and the resulting specification are presented. The advantages of the new approach are justified in the conclusion.

Keywords: Formal Methods, modularization, reuse, object orientation, inheritance, NDB challenge problem

Resumo

Neste artigo estudamos a especificação formal de um SGBD baseado formalmente no conceito de entidade-relacionamento. Usamos o problema-desafio NDB (Norman's Database) que tem sido explorado por vários autores na literatura recente. As operações do sistema e sua estrutura são descritos, usando técnicas que estendem VDM com aninhamento e herança. As extensões ao método e as especificações resultantes são apresentadas. As vantagens do novo enfoque são justificadas na conclusão.

Palavras-chave: Métodos Formais, modularização, reuso, orientação a objetos, herança, problema-desafio NDB

1 Introduction

Software design concepts from software engineering have begun to influence the notations used in the area of formal specifications. New constructs are presently being proposed specifically to deal with the issues of modularization and reuse of specifications in model based specification notations. The general trend is to integrate formal specification methods with practically proven software development process models.

A generally accepted approach is the extension of the "classical" model based notations, such as Z [Spive89] and VDM [Jones86] to accommodate generally accepted software design strategies developed in connection with established rigorous specification and development methods. So far, in most cases, the proposed extensions have been formulated in very informal terms. The recent debate centered around the NDB challenge problem (Norman's Database example [FACS92]) has served the purpose of illustrating the strengths and limitations of different formal methods as far as the issues of modularization and reuse are concerned ([Walsh90], [Fitzg90], [Hayes92]).

The issues of modularization and reuse are, of course, closely related. Modularization is associated to the specification of the concepts related to the notion of abstraction and it is achieved in model based formal methods through, for instance, the use of a hierarchy of types. In the approach proposed in [Fitzg90] it is illustrated that the use of modular specifications also supports the principle of separation of concerns while improving the understandability of specifications by addressing the problems of complexity and size of specifications. In [Hayes92] it is shown that modularization allows for the parameterization of related definitions which, in turn, produces the generalization of the problem at hand.

Reuse of specifications is a very relevant issue in software development processes. Once modular specifications are created, they can be used to generate other specifications by composition. The extensions to formal notations proposed by different authors to support reuse are, in a sense, complementary. They can be seen as a search for recurrent patterns in specifications [Fitzg90] or as the direct use of components out of libraries of components [Hayes92].

Our view about extensions to formal methods to support software design is influenced by our experience in the area of object oriented design [Ieru91,92]. It is also based on our belief that strict formal development is not viable in most cases and that the association of formal and informal development practices, as it occurs in all mature engineering areas, is feasible at the present with currently available technologies. For us, modularization is a concept associated to the hierarchy provided by inheritance. We claim that inheritance allows for a structured view of the abstraction levels that occur in specifications and that nesting allows for the independent meaning of objects when considered outside of their defining environments [Ieru91].

Some of the available rigorous requirement analysis techniques, such as the entity-relationship approach [Chen76], if applied by assuming that the design step will be carried out later with the support of a formal method such as VDM [Jones86], can lead to effective modularized/reusable specifications. In this paper we present the formal specification of a simple Data Base Management System (the NDB example) and illustrate how the entity-relationship approach can be associated to an object-oriented extension of VDM. The extension proposed is given a formal semantics in VDM and compared to the ones presented in [Fitzg90] and [Hayes92].

Next section describes the characteristics of the example Data Base Management System following [Walsh90]. The object-oriented extensions proposed to VDM are presented in sequel followed by the problem specification using the extended version of the method. We conclude by comparing the solution described to those proposed by the already mentioned authors.

2 NDB's problem description

In the NDB's data model, all information is handled by entities and binary relations between them. Each entity created has a name, and some may have a value. Each relation created also has a name and relates two (and only two) entities. One important peculiarity of the NDB is that its data model handles the meta-model, the conceptual model and the logical model.

The meta-model comprises two sets, one for entity-sets and one for relation-types. The elements of these sets belong to the conceptual model. At this level, each entity-set establishes one category for the entities that may belong to the BD's extension, while relation-types do the same for relations. Finally, the logical data model contains instances of entities and relations, and establishes associations between them.

Objects (entities and relations) at the conceptual level have other attributes. These attributes are status (describes when entities may be added or deleted), picture (defines the form of the entity values) and width (provides the length of the value) for entities; and name, fromset (the origin), toset (the destination) and maptype (the cardinality) for relations.

One particularization of concepts that may be imposed is that the relations should be normalized. This concept requires that the values taken by the relations be restricted to those imposed by rules dictated by the relation-type they belong to. These restrictions, also called functional dependencies, determines that each relation must satisfy a condition FS $\rightarrow$ TS, which implies that if the first entity in the relation belongs to the FS entity-set, then the other one must belong to TS.

3 Description of the Object-Oriented Extensions to VDM

In this section we describe the techniques first proposed in [Ieru91], which is going to be used in the NDB formal specification.

The goal of the techniques proposed here is to allow reuse of specifications and proofs, using object-oriented techniques. Therefore we adopt a syntax more related to this paradigm, and that groups together the type declaration and all its operations and functions. **Specification** and **End** are used to define the scope of the definitions we can make. Therefore, we can write

Specification S $f_1: T_1$ **Operation** $O\,(\ldots)$ ext $f_1: T_1$ pre ... post ... **End** S	instead of	$S :: f_1: T_1$ $O\,(\ldots)$ ext $f_1: T_1$ pre ... post ...

One of the extensions to Standard VDM is a notation to declare inheritance of specifications. Supose we have the following specification:

Specification S
Subtype of P_1
rename $O^1_{rn_1}$ as $NO^1_{rn_1}, \ldots, O^1_{rn_k}$ as $NO^1_{rn_k}$
redefine $O^1_{rd_1}, \ldots, O^1_{rd_l}$
⋮
Subtype of P_x
rename $O^x_{rn_1}$ as $NO^x_{rn_1}, \ldots, O^x_{rn_m}$ as $NO^x_{rn_m}$
redefine $O^x_{rd_1}, \ldots, O^x_{rd_n}$

$f_1: T_1$,
⋮
$f_y: T_y$
$inv\text{-}S \;\triangleq\; invS$

Operation O_1^s (...)
ext $extO_1^s$
pre $preO_1^s$
post $postO_1^s$

⋮

Operation O_z^s (...)
ext $extO_z^s$
pre $preO_z^s$
post $postO_z^s$
End S

The above specification declares S as a *subtype* of $P_1, \ldots, P_x$; each P_i in turn, is called a *supertype* of S. The following rules give the meaning of the above specification:

1. The actual components of S are the join of all components from $P_1, \ldots, P_x$ and S. If components from different specifications have the same name and type (textually equal), then they are merged in one component. If components from different specifications have the same name but diferent types, then there is and error condition.

2. The actual invariant of S is

$$invS \wedge (\bigwedge_{i=1}^{x} inv\text{-}P_i')$$

where the definition of $inv\text{-}P_i'$ is as follows:

$$inv\text{-}P_i' : S \rightarrow \mathbf{B}$$
$$inv\text{-}P_i'\ (s) \triangleq inv\text{-}P_i(projP_i(s))$$

and

$$projP_i : S \rightarrow P_i$$
$$projP_i\ (mk\text{-}S(\ldots, f_1^{P_i}, \ldots)) \triangleq mk\text{-}P_i(f_1^{P_i}, \ldots)$$

that is, $projP_i$, is the orthogonal projection from S to P_i (remember that, by rule 1, S has all the components from P_i).

Notice that we use $invS$ (without an hyphen) to denote the textual invariant, as written in the specification, while $inv\text{-}S$ 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 $inv\text{-}P_i$ and $inv\text{-}P_i'$ can be textually identical; the only difference between them is that the latter applies to S, and so it must "throw off" some components. Notice that $invS$ can refer to the fields inherited from other specifications.

3. The operations of P_i are insered into S in the following way: first, the operations $O_{rn_1}^i, \ldots, O_{rn_x}^i$ are renamed as $NO_{rn_1}^i, \ldots, NO_{rn_j}^x$. Then, the operations $O_{rd_1}^i, \ldots, O_{rd_y}^i$ are removed. The remainder operations are included in S with unmodified external lists, pre and post-conditions. If operations inherited from different parents have the same name and same definitions (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 the same parameter list; this operation *redefines* $O^{i}_{rd_j}$ (each declaration O^{s}_{k} can redefine more that one inherited operation). Suppouse that an operation O^{s}_{k} redefines the operations $O_{k_1}, \ldots, O_{k_m}$ from the specifications $P_{l_1}, \ldots, P_{l_m}$. Then, its actual specification is a combination of its textual specification and the specifications of all O_{k_i}, acoording to the following rules:

   (a) The external list of O^{s}_{k} is the join of its textual external list ($extO^{s}_{k}$) with external lists of O_{k_i}. The list $extO^{s}_{k}$ 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 componets 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

   $$preO^{s}_{k} \vee (\bigvee_{i=1}^{m} pre\text{-}O'_{k_i})$$

   and again, we have that

   $$pre\text{-}O'_{k_i} : S \to \mathbf{B}$$
   $$pre\text{-}O'_{k_i}\ (s) \triangleq pre\text{-}O_{k_i}(proj_{P_{l_i}}(s))$$

   where P_{l_i} is the specification from where O_k 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

   $$postO^{s}_{k} \wedge \bigwedge_{i=1}^{m}(pre\text{-}O'_{k_i} \Rightarrow post\text{-}O'_{k_i})$$

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

   $$post\text{-}O'_{k_i} : S \times S \to \mathbf{B}$$
   $$post\text{-}O'_{k_i}\ (\overleftarrow{s}, s) \triangleq post\text{-}O_{k_i}(proj_{P_{l_i}}(\overleftarrow{s}), proj_{P_{l_i}}(s))$$

   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 $postO^{i}_{k}$.

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

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

It is easy to prove the following lemma:

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.

The proof can be found in [Ieru92].

Parametric types can be specified by writing the parameter after the type name, and letting the specifications dependent on the instances of these parameters. The semantics for this construction is defined by textual substitution.

Another technique we have associated to the method is nesting of specifications. This thechnique allows decompositions to take place in a more natural way, since the perception of complex things is usually based on structuring concepts. Figures 1 and 2 we present the translation schema of one generic nesting construction into its equivalent flat VDM form. The nested specification is trasportated to outside, and we create a map in the other specification. This map have handles in its domain, and its range is a set of translated nested specifications. As we can see in the figures, the *S_Map* range represents a set of *S_Obj* instances in each O object, and the operations defined for the old S specification are translated into operations over the range of *S_Map*, using the handles that belong to the domain of this map.

To allow us to define each nested specification as an Abstract Data Type, which have its own components handled by the other specifications througth functions, each of which ensuring data hiding, we define **Function** as a mechanism to provide a inferface with other specifications. Its definition is writen below.

Function $g_s: T_{d_1} \times \ldots \times T_{d_k} \rightarrow T_r$
$g_s\ (p_{f_1}, \ldots, p_{f_k}) \triangleq p_s(f_{s_1}, \ldots, f_{s_n}, p_{f_1}, \ldots, p_{f_k})$

This declaration, inside a nested specification can be translated to the flat VDM as follows:

$g_s : S \times T_{d_1} \times \ldots \times T_{d_k} \rightarrow T_r$
$g_s(id, p_{f_1}, \ldots, p_{f_k}) \quad \triangleq \quad p_s(f_{s_1}(S\text{-}Map(id)), \ldots, f_{s_n}(S\text{-}Map(id)), p_{f_1}, \ldots, p_{f_k})$

Operation are similar to functions, but allow side-efects over the state of the object.

Operation $Z_s\ (p_{c_1}: T_{c_1}, \ldots, p_{c_k}: T_{c_k})$
ext wr $f_{sx_1}: T_{sx_1}$
...
wr $f_{sx_i}: T_{sx_i}$
wr $f_{oy_1}: T_{oy_1}$
...
wr $f_{oy_j}: T_{oy_j}$
pre *pre-*Z_s
post *post-*Z_s

Specification 0

$f_{o_1}: T_{o_1}$

$\vdots$

$f_{om}: T_{om}$

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

Specification S

$f_{s_1}: T_{s_1}$ $\vdots$

$f_{sn}: T_{sn}$

inv-S $\triangleq$ $p_s(s)$

Constructor X_s $(\ldots)$

pre ...

post ...

Destructor Y_s $(\ldots)$

pre ...

post ...

Operation Z_s $(\ldots)$

pre ...

post ...

Function $f_s: T_d \rightarrow T_r$

f_s $(\ldots) \triangleq \ldots$

End S

End 0

Figura 1: Nested specification

$$S = \mathbf{N}$$

$$\begin{aligned} S\_Obj \;::\; & self : S \\ & f_{s_1} : T_{s_1} \\ & \vdots \\ & f_{s_n} : T_{s_n} \end{aligned}$$

where

$$inv\text{-}S\_Obj() \;\triangleq\; p_s(s)$$

$$\begin{aligned} O \;::\; & S\_Map : S \xrightarrow{m} S\_Obj \\ & f_{o_1} : T_{o_1} \\ & \vdots \\ & f_{o_m} : T_{o_m} \end{aligned}$$

where

$$inv\text{-}O() \;\triangleq\; p_o(o) \wedge \forall id \in \mathrm{dom}\, S\_Map \cdot self(S\_Map(id)) = id$$

$X_o\ (\ldots)$

pre ...

post ...

$Y_o\ (\ldots)$

pre ...

post ...

$Z_o\ (\ldots)$

pre ...

post ...

$$f_s : S \times T_d \to T_r$$

$$f_s(\ldots) \;\triangleq\; \ldots$$

Figura 2: Translated specification

Notice the way the translation schema specifies that the operation modifies one value of the range of S_Map.

$Z_o\ (id: S, p_{c_1}: T_{c_1}, \ldots, p_{c_k}: T_{c_k})$
ext wr $S\_Map\ :\ S \xrightarrow{m} S\_Obj$
wr $f_{oy_1}\ :\ T_{oy_1}$
...
wr $f_{oy_j}\ :\ T_{oy_j}$
pre $id \in \mathrm{dom}\, S\_Map \wedge pre\text{-}Z_s(f_{sx_1}(S\_Map(id)), \ldots, f_{sx_i}(S\_Map(id)), f_{oy_1}, \ldots, f_{oy_j}, p_{c_1}, \ldots, p_{c_k})$
post $\exists o \in S\_Obj \cdot post\text{-}Z_s(f_{sx_1}(\overleftarrow{S\_Map}(id)), \ldots, f_{sx_i}(\overleftarrow{S\_Map}(id)), o, p_{c_1}, \ldots, p_{c_k}) \wedge$
$S\_Map = \overleftarrow{S\_Map} \dagger \{id \mapsto o\}$

We still need two other facilities, to allow the creation and destruction of objects. In what follows we present the specification of a generic creation operation, and its respective translation to flat VDM.

Constructor $X_s\ (p_{c_1}: T_{c_1}, \ldots, p_{c_k}: T_{c_k})$
ext wr $f_{sx_1}: T_{sx_1}$
...
wr $f_{sx_i}: T_{sx_i}$
wr $f_{oy_1}: T_{oy_1}$
...
wr $f_{oy_j}: T_{oy_j}$
pre $pre\text{-}X_s$
post $post\text{-}X_s$

$X_o\ (p_{c_1}: T_{c_1}, \ldots, p_{c_k}: T_{c_k})\ id: S$
ext wr $S\_Map\ :\ S \xrightarrow{m} S\_Obj$
wr $f_{oy_1}\ :\ T_{oy_1}$
...
wr $f_{oy_j}\ :\ T_{oy_j}$
pre $pre\text{-}X_s$
post $id \notin \mathrm{dom}\, \overleftarrow{S\_Map} \wedge \exists o \in S\_Obj \cdot post\text{-}X_s(o, f_{oy_1}, \ldots, f_{oy_j}, p_{c_1}, \ldots, p_{c_k}) \wedge$
$S\_Map = \overleftarrow{S\_Map} \cup \{id \mapsto o\}$

Finally, we define the removal of object instances:

Destructor $Y_s\ (p_{d_1}: T_{d_1}, \ldots, p_{d_k}: T_{d_k})$
ext wr $f_{sx_1}: T_{sx_1}$
...
wr $f_{sx_k}: T_{sx_k}$
wr $f_{oy_1}: T_{oy_1}$
...
wr $f_{oy_l}: T_{oy_l}$
pre $pre\text{-}Y_s$
post $post\text{-}Y_s$

$Y_o\ (id: S, p_{d_1}: T_{d_1}, \ldots, p_{d_k}: T_{d_k})$
ext wr $S\_Map\ :\ S \xrightarrow{m} S\_Obj$
wr $f_{oy_1}\ :\ T_{oy_1}$
...
wr $f_{oy_l}\ :\ T_{oy_l}$
pre $id \in \mathrm{dom}\, S\_Map \wedge pre\text{-}Y_s(f_{sx_1}(S\_Map(id)), \ldots, f_{sx_i}(S\_Map(id)), f_{oy_1}, \ldots, f_{oy_j}, p_{d_1}, \ldots, p_{d_n})$
post $post\text{-}Y_s(f_{sx_1}(\overleftarrow{S\_Map}(id)), \ldots, f_{sx_i}(\overleftarrow{S\_Map}(id)), f_{oy_1}, \ldots, f_{oy_j}, p_{d_1}, \ldots, p_{d_k}) \wedge$
$(S\_Map = \{id\} \triangleleft \overleftarrow{S\_Map})$

To improve the OO style of our extension, we adopt the usual dot notation for the specification of operation and function calls. So we write $id.name(p_1, \ldots, p_n)$ for $name(id, p_1, \ldots, p_n)$, where id is an object identificator.

4 The NDB's Specification

This section describes a formal specification for the NDB, obtained through the application of the OO techniques described in the previous section. Our objective here is to argue that it is possible to produce a specification for the example problem at least as modular as the ones discussed in [Fitzg90] and [Hayes92].

Those previous works propose three main styles of specifications for NDB: one that only considers binary relations, another that considers the use of a generic n-ary relation specification to specify the same problem, and, finally, one that considers typed-relations and normalization. We cover all three cases in this sequence.

To begin with, we propose a diagramatic representation for the example problem, which allows us to provide informal views of the structure of the specification. This informal conceptual model is built through the use of simple "box-and-arrows" diagrams. In each box there is an object name and each arrow has a relation name ("is_a" or "comp_of"), whose semantics is associated to the formal constructions we have proposed for the method. Each box, which represents an object, has two parts, one containing the components and the other containing the operations and functions of the object.

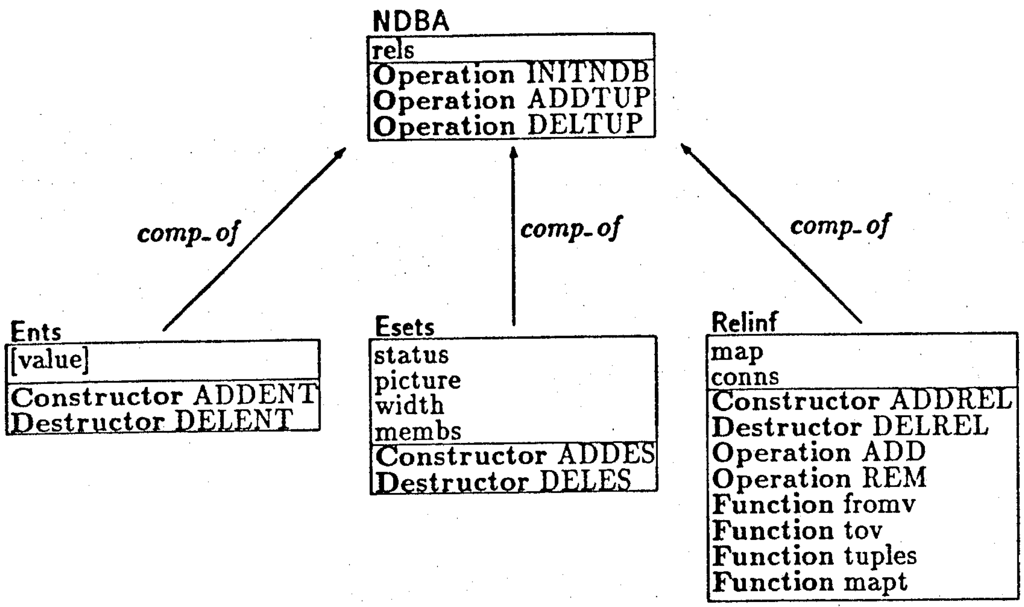

Figura 3: NDBA informal model

We have stated in section 2 that the NDB's data model is constructed by means of entities, relations, entity-sets and relation-types. Usually, these concepts generate two kinds of maps between formal objects: one relating indexes to objects, and the other creating relations between objects. Maps of the first kind are translated, in our specifications, to nested types. We can see this nesting in figure 3, in association with the "comp_of" relations.

Since the informal definition of the problem is well known, we can now formulate the formal specification of the NDB. In figure 4 we present the *NDBA* specification, which defines a binary-relation based DataBase. Then, in figures 5, 6 and 7 we present the specification of NDB relations, entities and entity-sets, respectivelly. In these figures, *Status*, *Picture* and *Width* are not defined further, and the following type specifications are needed:

$$Maptype = \{1{:}1, 1{:}M, M{:}1, M{:}M\}$$

$$\begin{array}{lrcl} Pair :: & fv & : & Ents \\ & tv & : & Ents \end{array}$$

$$\begin{array}{lrcl} Reltype :: & fs & : & Esets \\ & ts & : & Esets \\ & name & : & [Name] \end{array}$$

Specification *NDBA*

$rels \colon Reltype \xrightarrow{m} Relinf$

Specification *Relinf*

(*see figure* 5)

End *Relinf*

Specification *Ents*

(*see figure* 6)

End *Ents*

Specification *Esets*

(*see figure* 7)

End *Esets*

$inv\text{-}NDBA \;\triangleq\; \forall rt \in \mathrm{dom}\, rels \cdot \{fs(rt), ts(rt)\} \subseteq \mathrm{dom}\, Esets\_Map$

Operation *ADDTUP* $(f, t \colon Ents, rt \colon Reltype)$

ext rd $rels \colon Reltype \xrightarrow{m} Relinf$

wr $Relinf\_Map \colon Relinf \xrightarrow{m} Relinf\_Obj$

pre $rt \in \mathrm{dom}\, rels \wedge rels(rt).pre\text{-}ADD(f, t)$

post $rels(rt).post\text{-}ADD(f, t)$

Operation *DELTUP* $(f, t \colon Ents, rt \colon Reltype)$

ext rd $rels \colon Reltype \xrightarrow{m} Relinf$

wr $Relinf\_Map \colon Relinf \xrightarrow{m} Relinf\_Obj$

pre $rt \in \mathrm{dom}\, rels \wedge rels(rt).pre\text{-}REM(f, t)$

post $rels(rt).post\text{-}REM(f, t)$

Operation *INITNDBA* ()

ext wr $rels \colon Reltype \xrightarrow{m} relinf$

wr $Esets\_Map \colon Esets \xrightarrow{m} Esets\_Obj$

wr $Ents\_Map \colon Ents \xrightarrow{m} Ents\_Obj$

wr $Relinf\_Map \colon Relinf \xrightarrow{m} relinf\_Obj$

post $rels = \{\,\} \wedge Esets\_Map = \{\,\} \wedge Ents\_Map = \{\,\} \wedge Relinf\_Map = \{\,\}$

End *NDBA*

Figura 4: NDBA specification

Specification *Relinf*

map: *Maptype*
conns: *Pair*-set

$inv\text{-}Relinf \triangleq mapt(conns) \wedge \exists rt \in \mathrm{dom}\, rels \cdot (rels(rt) = self \wedge \forall r \in conns \cdot$
$(\mathrm{let}\ mk\text{-}Pair(f, t) = r\ \mathrm{in}$
$f \in membs(Esets\_Map(fs(rt))) \wedge t \in membs(Esets\_Map(ts(rt)))))$

Constructor *ADDREL* (m: *Maptype*, rt: *Reltype*)
ext rd $Eset\_Map: Eset \xrightarrow{m} Eset\_Obj$
wr $rels: Reltype \xrightarrow{m} Relinf$
pre $\{fs(rt), ts(rt)\} \subseteq \mathrm{dom}\, Esets\_Map \wedge rt \notin \mathrm{dom}\, rels$
post $map = m \wedge conns = \{\,\} \wedge rels = \overleftarrow{rels} \cup \{rt \mapsto self\}$

Destructor *DELREL* (rt: *Reltype*)
ext wr $rels: Reltype \xrightarrow{m} Relinf$
pre $rt \in \mathrm{dom}\, rels \wedge conns = \{\,\} \wedge rels(rt) = self$
post $rels = \{rt\} \triangleleft \overleftarrow{rels}$

Operation *ADD* (f, t: *Ents*)
pre $mapt(conns \cup \{mk\text{-}Pair(f, t)\})$
post $conns = \overleftarrow{conns} \cup \{mk\text{-}Pair(f, t)\}$

Operation *REM* (f, t: *Ents*)
pre $mk\text{-}Pair(f, t) \in conns$
post $conns = \overleftarrow{conns} - \{mk\text{-}Pair(f, t)\}$

Function $fromv: Pair \rightarrow Ents$

$fromv\ (tu) \triangleq fv(tu)$
pre $tu \in conns$

Function $tov: Pair \rightarrow Ents$

$tov\ (tu) \triangleq tv(tu)$
pre $tu \in conns$

Function $tuples: \rightarrow pair\text{-set}$

$tuples\ () \triangleq conns$

Function $mapt: D\text{-set} \rightarrow \mathbf{B}$

$mapt\ (c) \triangleq \nexists t1, t2 \in c \cdot t1 \neq t2 \wedge$
cases map of
M:M $\rightarrow$ false
M:1 $\rightarrow fv(t1) = fv(t2)$
1:M $\rightarrow tv(t1) = tv(t2)$
1:1 $\rightarrow fv(t1) = fv(t2) \vee tv(t1) = tv(t2)$
end

End *Relinf*

Figura 5: Relinf specification

Specification *Ents*

$value\colon [Value]$

$inv\text{-}Ents \;\triangleq\; \exists x \in \mathrm{rng}\, Esets\_Map \cdot self \in membs(x)$

Constructor $ADDENT\ (v\colon [Value], m\colon Ents\text{-}\mathrm{set})$

$\mathrm{ext}\quad \mathrm{wr}\ Esets\_Map\colon Esets \xrightarrow{m} Eset\_Obj$

$\mathrm{pre}\quad m \subseteq \mathrm{dom}\, Esets\_Map$

$\mathrm{post}\ value = v \wedge Esets\_Map = \overleftarrow{Esets\_Map} \dagger$
$\{es \mapsto \mu(\overleftarrow{Esets\_Map}(es), membs \mapsto membs(\overleftarrow{Esets\_Map}(es)) \cup \{self\}) \mid es \in m\}$

Destructor $DELENT\ ()$

$\mathrm{ext}\quad \mathrm{rd}\ Relinf\_Map\colon Relinf \xrightarrow{m} Relinf\_Obj$
$\qquad \mathrm{wr}\ Esets\_Map\colon Esets \xrightarrow{m} Esets\_Obj$

$\mathrm{pre}\quad \forall rl \in \bigcup\{tuples(r) \mid r \in \mathrm{rng}\, Relinf\_Map\} \cdot fromv(rl) \neq self \wedge tov(rl) \neq self$

$\mathrm{post}\ Esets\_Map = \overleftarrow{Esets\_Map} \dagger$
$\{es \mapsto \mu(\overleftarrow{Esets\_Map}(es), membs \mapsto membs(\overleftarrow{Esets\_Map}) - \{self\}) \mid es \in \mathrm{dom}\, \overleftarrow{Esets\_Map}\}$

End *Ents*

Figura 6: Ents specification

As it could be expected, the state invariant has been broken down in small parts, each of which associated to the state of the object type it most refers to. For example, in the specification presented in [Walsh90], there is a declaration

$$\mathrm{dom}\, em = \bigcup \mathrm{dom}\, esm$$

which states that the set of the entities must be the same as the set which contains all the entities that belong to an entity-set. This intention is captured in our specification by *inv-Ents* and *inv-Esets*.

Another interesting point is related to the choice of which components should be nested, and which ones should not. Clearly, those belonging to an object type must be nested in it. The components that should not be nested are those that define relationships between objects, e.g., rels in *NDB* specification presented in the work cited above.

The specifications of creation and destruction of object instances are quite simple in our version of the method, because part of then are implicit from the semantics of **Constructor** and **Destructor**. Only the initialization values for the object (as can be seen in the constructor *ADDES*) and the influence of these operations over other objects (which is shown in the destructor *DELES*) remain to be specified.

Specification *Esets*
  status: *Status*
  picture: *Picture*
  width: *Width*
  membs: *Ents*-set

$$inv\text{-}Esets \triangleq membs \subseteq \mathrm{dom}\, Ents\_Map$$

Constructor *ADDES* $(s: Status, p: Picture, w: Width)$
post $status = s \wedge picture = p \wedge width = w \wedge membs = \{\}$

Destructor *DELES* ()
ext rd $rels: Reltype \xrightarrow{m} Relinf$
pre $\nexists rt \in \mathrm{dom}\, rels \cdot fs(rt) = self \vee ts(rt) = self$
End *Esets*

Figura 7: Esets specification

A generalization to NDB relations can be defined by specifying a type which is independent of this problem. Since it has some of the characteristics of relations, we can redefine *Relinf* as a subtype of this new type (indirectly), making a new specification for NDB, called *NDBB*. This is shown in figure 8.

There is no need to rewrite the other specifications, because the changes done in *Relinf* are hidden in its structure, by the use of functions. So, instead of the old version of *Relinf*, we use the specification presented in figure 11 to specify NDB, thogether with those in figures 9 and 10, for *Relation* and *BinaryRelation*. These specifications illustrate the use of parametric types.

In figure 9, a type *Fsel* is used, but we left it undefined there, because it can be any set of constants which denote relation selectors. That is not the case in figure 10, since the relations have cardinality equal to 2, and to define the properties of the relation atributes, we must use selector names.

When we generalize Relinf and create Relation, our mental activity is concentrated on the identification of the portions of the structure and the operations that are candidate to become a new object. In this and in other similar cases, the invariant may need to be broken, and the more general object may acquire some of its parts. This situation happens with the old *Relinf* invariant, which have one part aquired by BinaryRelation (the statement in which *mapt* is used), and the other aquired by the new *Relinf*.

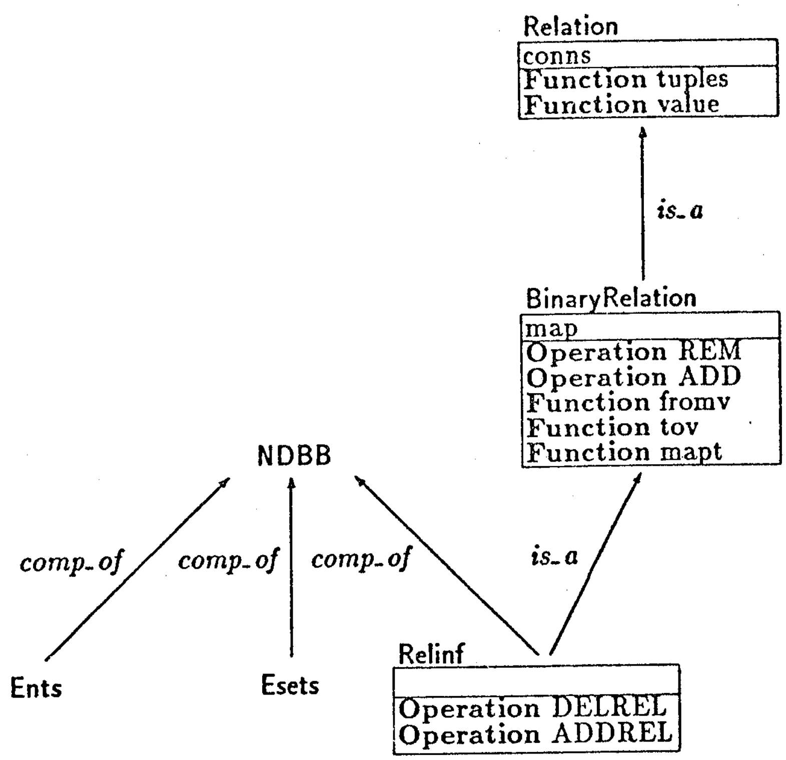

Figura 8: NDBB informal model

Specification *Relation*(D)

$\quad Tuple = Fsel \xrightarrow{m} D$

$\quad conns$: *Tuple*-set

$inv\text{-}Relation \ \triangleq\ \forall t1, t2 \in conns \cdot \mathrm{dom}\, t1 = \mathrm{dom}\, t2$

Function *tuples*: $\rightarrow$ *tuple*-set

$tuples\ () \triangleq conns$

Function *value*: $(Tuple \times Fsel) \rightarrow D$

$value\ (t, sel) \triangleq t(sel)$

pre $\ t \in conns$

End *Relation*

Figura 9: Relation specification

Specification $BinaryRelation(D)$

Subtype of $Relation(D)$

map: $Maptype$

$inv\text{-}BinaryRelation \triangleq (\forall t \in conns \cdot \mathsf{dom}\, t = \{\mathrm{FS}, \mathrm{TS}\}) \wedge mapt()$

Operation $ADD\ (f, t: D)$

pre $mapt(map, conns \cup \{\mathrm{FS} \mapsto f, \mathrm{TS} \mapsto t\})$

post $conns = \overleftarrow{conns} \cup \{\mathrm{FS} \mapsto f, \mathrm{TS} \mapsto t\}$

Operation $REM\ (f, t: D)$

pre $\{\mathrm{FS} \mapsto f, \mathrm{TS} \mapsto t\} \subseteq conns$

post $conns = \overleftarrow{conns} - \{\mathrm{FS} \mapsto f, \mathrm{TS} \mapsto t\}$

Function $fromv$: $Tuple \rightarrow D$

$fromv\ (tu) \triangleq value(tu, FS)$

Function tov: $Tuple \rightarrow D$

$tov\ (tu) \triangleq value(tu, TS)$

Function $mapt$: $D\text{-set} \rightarrow \mathbf{B}$

$$
\begin{aligned}
mapt\ (c) \triangleq\ & \forall t1, t2 \in c \cdot t1 \neq t2 \wedge \\
& \mathsf{cases}\ map\ \mathsf{of} \\
& \quad \mathrm{M{:}M} \rightarrow \mathsf{true} \\
& \quad \mathrm{M{:}1} \rightarrow t1(\mathrm{FS}) = t2(\mathrm{FS}) \Rightarrow t1(\mathrm{TS}) = t2(\mathrm{TS}) \\
& \quad \mathrm{1{:}M} \rightarrow t1(\mathrm{TS}) = t2(\mathrm{TS}) \Rightarrow t1(\mathrm{FS}) = t2(\mathrm{FS}) \\
& \quad \mathrm{1{:}1} \rightarrow t1(\mathrm{FS}) = t2(\mathrm{FS}) \Leftrightarrow t1(\mathrm{TS}) = t2(\mathrm{TS}) \\
& \mathsf{end}
\end{aligned}
$$

End $NRelation$

Figura 10: BinaryRelation specification

Specification *Relinf*

Subtype of *BinaryRelation(Ents)*

$$inv\text{-}Relinf \;\triangleq\; \exists rt \in \mathsf{dom}\, rels \cdot ((rels(rt) = self) \wedge \forall t \in conns \cdot fromv(t) \in Esets\_Map(fs(rt)) \wedge tov(t) \in Esets\_Map(ts(rt)))$$

Constructor *ADDREL* $(m: Maptype, rt: Reltype)$

ext rd $Esets\_Map: Esets \xrightarrow{m} Esets\_Obj$

wr $rels: Reltype \xrightarrow{m} Relation$

pre $\{fs(rt), ts(rt)\} \subseteq \mathsf{dom}\, Esets\_Map \wedge rt \notin \mathsf{dom}\, rels$

post $map = m \wedge conns = \{\,\} \wedge rels = \overleftarrow{rels} \cup \{rt \mapsto self\}$

Destructor *DELREL* $(rt: Reltype)$

ext wr $rels: Reltype \xrightarrow{m} Relation$

pre $rt \in \mathsf{dom}\, rels \wedge conns = \{\,\} \wedge rels(rt) = self$

post $rels = \{rt\} \lhd \overleftarrow{rels}$

End *Relinf*

Figura 11: Relinf specification using BinaryRelation

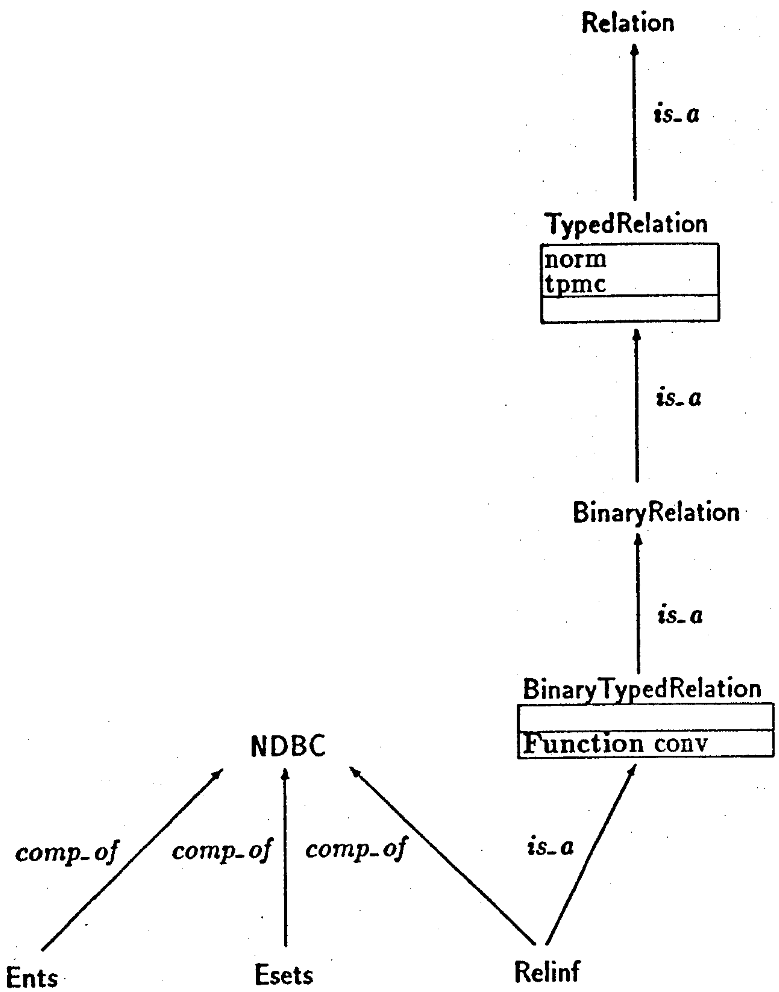

Figura 12: NDBC informal model

The opposite activity, specialization, can also takes place, as ilustrated in figure 12. It requires taking the BinaryRelation type and determining a more restrictive invariant, to produce *BinaryTyped Relation*. Specialization occurs in another situation, when it is necessary to agregate new components to a type. We can observe this by looking *Relation* and *TypedRelation* in figure 12.

Figures 13 and 14 specify *TypedRelation* and *BinaryTypedRelation*, respectively. It is necessary to define the type *Norm* as follows, which is used there to comprise all the possible kinds of normalization it is possible to define over the atributes of the relations in the DataBase.

$Norm = (Fsel\text{-set} \times Fsel)\text{-set}$

The other specifications need to be rewrited a little, leting *BinaryRelation* be a subtype of *Typed Relation*, instead of *Relation*, and leting *Relinf* be a subtype of *BinaryTypedRelation*, instead of *BinaryRelation*.

Specification *TypedRelation*(D)

Subtype of *Relation*(D)

$Tpmc = Fsel \xrightarrow{m} D\text{-set}$

$norm: Norm$

$tpmc: Tpmc$

$$inv\text{-}TypedRelation \triangleq (\forall m \in conns \cdot \mathrm{dom}\, m = \mathrm{dom}\, tpmc \wedge (\forall x \in \mathrm{dom}\, m \cdot m(x) \in tpmc(x))) \wedge$$
$$(\forall tp \in norm \cdot \mathrm{let}\ mk\text{-}Norm(s,f) = tp\ \mathrm{in}$$
$$s \cup \{f\} \subseteq \mathrm{dom}\, tpmc \wedge$$
$$(\forall t1, t2 \in conns \cdot s \lhd t1 = s \lhd t2 \Rightarrow t1(f) = t2(f)))$$

End *TypedRelation*

Figura 13: TypedRelation specification using Relation

Specification *BinaryTypedRelation*(D)

Subtype of *BinaryRelation*(D)

$$inv\text{-}BinaryTypedRelation \triangleq \exists rt \in \mathrm{dom}\, rels \cdot tpmc = \{\mathrm{FS} \mapsto fs(rt), \mathrm{TS} \mapsto ts(rt)\} \wedge$$
$$norm = conv(map)$$

Function $conv: Maptype \to Norm$

$conv\ (ty) \triangleq$ cases ty of

$\mathrm{M{:}M} \to \{\,\}$

$\mathrm{M{:}1} \to \{(\{\mathrm{FS}\}, \mathrm{TS})\}$

$\mathrm{1{:}M} \to \{(\{\mathrm{TS}\}, \mathrm{FS})\}$

$\mathrm{1{:}1} \to \{(\{\mathrm{TS}\}, \mathrm{FS}), (\{\mathrm{FS}\}, \mathrm{TS})\}$

end

End *BinaryTypedRelation*

Figura 14: BinaryTypedRelation specification using BinaryRelation

5 Conclusions and Future Work

The extensions to VDM proposed in the present paper do lead to modular and reusable specifications. The semantics of the proposed extensions was formally given in terms of established VDM constructs . The new constructs capture the notion of object-oriented design as defined by established design theories (decomposition by form as opposed to functional decomposition [Maher90]) and made viable by existing object-oriented programming languages. The proposal is contrasted with some of the recent solutions presented in the literature and illustrated by the same NDB "challenge problem".

In the case study described we were able to highlight a characteristic of object oriented specifications which has a direct parallel in the Z notation [Spive89]. When states are specified in model based specification methods they need to be integrated later to other states which are, eventually, more general than the originally specified states. In this situation, the initially defined operations need to be generalized. In Z this generalization is called a promotion. It is achieved by means of the schema calculus which is part of the Z notation. The same issue occurs in the case of nesting since it is also a composition operation. In this case the state of the objects are composed of all the nested object components. This is guaranteed by the semantics of the proposed object oriented features . In other words, promotion is implicitly defined in the techniques illustrated before therefore reducing the effort required for the expression of a formal specification.

Since we have used functions to handle the interface between objects the resulting specification displayed a high degree of independence between objects. When changes were needed during the case study to the point of requiring a restructuring of the specification they were confined to the interfaces. In other words there was no need to redefine other types when a type specification was changed.

Inheritance also provided the reuse of specifications. It takes place when a new type needs to be specified and some of its characteristics can be imported from an already defined type. If the existing specification is sufficiently general the new type can be defined as a subtype.

The use of our extended version of VDM in association with the informal entity-relationship approach simplified considerably the understanding of the problem. The structure of the informal diagrams of the semi-formal method were derived directly from the characteristics of the new features included in VDM.

In the continuation of the present work we are trying to associate concurrency features to the set of proposed object oriented features.We are also working on the creation of a method to allow the separated reification of the types. Right now we need to translate the specifications produced to Standard VDM to then apply the conventional reification method. Of course, this is still far from ideal.

Referências

[Andre91] Andrew, D. and Ince, D., *Practical Formal Methods with VDM*, Ed. McGraw-Hill, 1991.

[Chen76] Chen, P.P.S., The Entity-Relationship Model - Towards a Unified View of Data, *ACM Transactions on Database Systems*, Vol. 1, No. 1, 9-36, Mar. 1976

[FACS92] *Formal Aspects of Computing*, No. 4, Vol. 1, Jones, Cliff and Cooke, John (editors), Ed. Springer International, 1992

[Fitzg90] Fitzgerald, J. S. and Jones, C. B., Modularizing the Formal Description of a Database System, In: *VDM'90 - Formal Methods in Software Development*, Editor:Bjorner, D., Horae, C. A. R. and Langmaack, H., 189-210, Ed. Springer Verlag, 1990

[Hayes92] Hayes, I., VDM and Z: A Comparative Study, in [FACS92], 76-99

[Ieru91] Ierusalimschy, R., *A Method for Object-Oriented Specifications for VDM*, Série Monografias em Ciência da Computação, Pontifícia Universidade Católica, Rio de Janeiro, Brazil, 1991

[Ieru92] Ierusalimschy, R., *A Formal Specification of a Hierarchy of Collections*, Série Monografias em Ciência da Computação, Pontifícia Universidade Católica, Rio de Janeiro, Brazil, 1992

[Jones86] Jones, C. B., *Systematic Software Development*, Ed. Prentice-Hall International, 1986

[Maher90] Maher, M. L., Process Models of Design Syntesis, *AI Magazine*, 49-58, Winter 1990

[Spive89] Spivey, J. M., *The Z Notation: A Reference Manual*, Prentice Hall International, 1987

[Walsh90] Walshe. A., NDB: the formal specification and rigorousus design of a single-user database system, In: *Case Studies in Sistematic Software Development*, Editors: Jones, C. B. and Shaw, C. F., 11-45, Ed. Prentice-Hall International, 1990