A THREE-LEVEL APPROACH TO THE DESCRIPTION OF DATA STRUCTURES, AND NOTATIONAL FRAMEWORK

Bertrand Meyer
EDF, Direction des Etudes et Recherches
1 avenue du Général de Gaulle 92141 Clamart FRANCE

1 - The Three Levels

When talking about data structures, whether local to a program or persistent over time, three different viewpoints are equally important. The first one is that of the user, who is interested in the external properties of a certain structure, more precisely, its noticeable behavior in response to outside effects (queries, requests for modifications, etc.). The second viewpoint is that of the language designer, who is in search of a small number of basic objects and building mechanisms which will allow for the description of complex objects in terms of simpler ones. The last view is that of the implementor, who must find efficient representations for the constructs thus described.

Based on this remark, a three-level description of data structures has been used by the author in previous work [6, 7, 8]. The three levels may be called :

- *functional* ;
- *constructive*, or *logical* ;
- *physical*.

The functional specification is an entirely implicit characterization of the structure at and by functions and properties of these functions. This "algebraic approach", as described by Liskov, Zilles, Guttag and others is now classical.

The logical description, on the other hand, is explicit : it provides a means to construct the structure, or instances of it, starting with a set of base objects and constructors. This mechanism remains, however, representation-independent, since the objects and constructors are purely mathematical entities, not computer-related elements. A logical description may thus be considered as an abstract implementation.

Several suitable frameworks exist for expressing such logical descriptions :

- Set theory, with constructors such as cartesian product and disjoint union ; this is basically Hoare's proposal in [3], and quite close to the type construction mechanisms in Algol 68 or Pascal;

- Codd's relational model ;

- recursive functions, as suggested by McCarthy [5] and more recently by Lehmann [4], also embodied in Gedanken.

- a single base type (product of zero types) with union and cartesian product as operators, and recursive definitions, as in [2].

Lastly, physical representation is concerned with the layout of objects in memory in accordance with the descriptions at the previous levels. It is only there that such concepts as flags, bytes, words, addresses, offsets, and so forth, appear.

In this approach, it should be noted that several functional specifications may be associated with with the same logical description, corresponding to various user views, protection levels, etc. ; and that a given logical description may produce many physical representations. This situation is pictured on Figure 1.

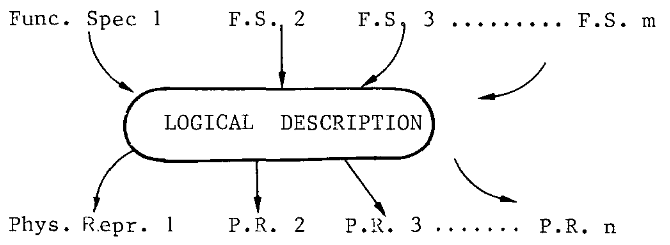

Figure 1 : Levels of Representation

Although not identical, and developed for different reasons, the three-level decomposition described here has much in common with the notion of the three "schemata" (external, conceptual, and internal) as used in Data Bases.

2 - Some notation

Several problems arise when trying to deal with the preceding view of data structures. One is to find a convenient notation at every level. Another is to devise means to obtain the "logical" and "physical" descriptions in a systematic way, starting form the functional specification.

For both purposes, the Z specification language [1] seems to be a useful tool. This formal language, whose detailed description falls outside the scope of this position paper, offers abstraction facilities which help in the stepwise description of data structures, and a framework for expressing complementary approaches.

The basic structuring mechanism is the class. A Z class describes some structure, e.g. a mathematical structure or an abstract data type, defined by elements of one or more of the following kinds:

- *primary attributes*, or components which must be provided to build any instance of the class ;

- *predicates*, which must be satisfied by these components ;

- *derived attributes*, which are also possessed by any instance of the class, but are deducible from the primary attributes.

In addition, most class definitions are generic, i.e. relative to some formal set parameters.

As an example, a "group" structure on some set X may be defined as a class having a generic parameter representing X ; two primary attributes, the internal operation and the neutral element ; predicates representing the group conditions ; and a derived attribute which is the inverse operation (minus). The definition in Z might look like the following :

```
Group[X] ≘ .

  class with
      oper : X * X → X ;
      zero : X ;
      minus : X * X → X
  where
      a : oper ∈ ASSOCIATIVE [X] ;
      n : zero ∈ NEUTRAL [X] (oper) ;
      i : oper ∈ INVERSIBLE[X] (zero)
  def
      minus ≘ inverse (oper, zero)
  end ;
```

All the attributes appear in the *with* clause, which plays the rôle of a type declaration. The *def* clause introduces the derived attributes. Here we assume that *ASSOCIATIVE*[X], the set of associative binary operators on X, as well as *NEUTRAL*[X] , *INVERSIBLE* [X], *inverse*, are defined elsewhere ; this is indeed done easily in Z (see the "basic chapters" in [1]).

An important property of classes is the existence of "derive/synthesize" relationships between different class definitions. In particular, a previously defined class A can be used as a prefix to a new class B which will inherit its attributes and properties, much in the same way as in the programming language SIMULA 67. The notation is a slight refinement of the preceding one :

```
B ≘
  class A with
      .....
  where
      .....
  def
      .....
  end
```

In the process of defining such subclasses, some primary attributes may become derived (by being included in the *def* clause at some lower level). This is known as partial binding.

This mechanism is very useful for the top-down description of complex systems or data structures. In particular, it allows for a more systematic description of abstract data types than the ones which have been published, which contain much repetition between slightly different structures. This point is developed at length in [9].

Classes may also be combined in a bottom-up fashion by union and join operations. However, the top-down approach seems to be of particular interest for our matter. The process of designing and describing a data structure could be viewed as starting with the construction of a class ; then refining into subclasses, all this being at the "functional" level and mostly implicit ; the part of Z which is of use here is the "class sublanguage" which has been sketched above.

Then at some point one must stop throwing in new attributes, and freeze the functional design. The problem of devising a compatible abstract representation, or logical description, arises then. This can be done entirely within the framework of Z by using its more classical "set sublanguage", which is essentially the language of set theory (with sets, functions, relations etc. as objects, and cartesian product, composition etc. as operations) and has implicitly been used above.

It may be noted that once the functional specification has thus been frozen, then in theory a logical description may be obtained in a straightforward way by taking the cartesian product of the types of the primary attributes in the corresponding class definition, or rather a subset of it in order to take the predicates into account. That is, the logical description corresponding to

```
class with
    p1 : P1 ; p2 : P2 ; ..... ;
    d1 : D1 ; d2 : D2 ; ...
where
    pred (p1, p2, .....)
def
    d1 ≘ ... ; d2 ≘ ... ; ...
end
```

will be

```
set x for x : P1 * P2 * ..... where
        pred (proj1 (x), proj2 (x), .....)
end
```

projI being the *I*-th projection.

3 - Conclusion

Two related points are developed in this position paper : the need for a description of data structures using three different levels, seen as complementary rather than competitive ; and the use of Z to serve as a vehicle for expressing such descriptions. Z, which has been used much in the same way to model the dual concept - programs - [10] appears to be a powerful framework for expressing and transforming formal specifications.

Bibliography

[1] J.R. Abrial, S.A. Schuman, B. Meyer : *Specification Language* ; Proc. Summer School on the Construction of Programs, Belfast 1979 ; Cambridge University Press, 1980. See also : J.R. Abrial, *The Specification Language Z : Syntax and "Semantics"* ; Internal report, Oxford University, Computing Laboratory, 1980.

[2] W. Burge : *Recursive Programming Techniques* ; Addison-Wesley, 1976.

[3] C.A.R. Hoare : *Notes on Data Structuring* ; in *Structured Programming* (Dahl, Dijkstra, Hoare), Academic Press, 1972.

[4] D.J. Lehmann : *Modes in ALGOL Y* ; University of Southern California, 1977.

[5] J. McCarthy : *Basis for a Mathematical Theory of Computation* ; in *Computer Programming and Formal Systems* (Braffort and Hirschberg, Eds.) ; North-Holland, 1963.

[6] B. Meyer et C. Baudoin : *Méthodes de Programmation* ; Eyrolles, Paris, 1978 (English translation to appear).

[7] B. Meyer : *Description des Structures de Données* ; Bulletin de la Direction des Etudes et Recherches EDF, série C, 2, 1976, pages 81-90.

[8] B. Meyer : *Types abstraits, Spécifications fonctionnelles, et Décomposition des Programmes* ; Journées SESORI sur la Synthèse, Manipulation et Transformation de Programmes, Saint-Rémy de Provence, 1978.

[9] B. Meyer : *Méthode et Notation pour les Types abstraits* ; AFCET-GROPLAN, Lamoura, 1980.

[10] B. Meyer : *A Basis for the Constructive Approach to Programming* ; IFIP World Computer Congress, Tokyo, 6-9 October 1980 (to appear).