university of california • santa barbara

Department of Computer Science

TRCS85-15

M: A System Description Method

Bertrand Meyer

Department of Computer Science

August 1985

College of Engineering

M:

A SYSTEM DESCRIPTION METHOD

Bertrand Meyer

ABSTRACT

We sketch the principles of a method and notation for use by software designers to describe the functional characteristics of systems being planned or developed. The method is *implicit* since entities are described by their properties only; it is *object-oriented* since the descriptions emphasize classes of system objects over functions; it is *modular* since it provides ways to describe complex systems in a piecewise fashion; it is *iterative* since it encourages the stepwise refinement of system descriptions; it is *formal* while retaining some of the advantages of non-formal specification methods.

Table of Contents

M: A SYSTEM DESCRIPTION METHOD

Bertrand Meyer

1 - PURPOSE, SCOPE, CRITERIA

It is well-known in the software engineering community that the initial phases of the software lifecycle - specification and global design - are the crucial ones. They condition the smooth proceeding of the remaining phases and the quality of the eventual product.

In other engineering disciplines, a number of methods, notations and tools are available to support the corresponding phases. Design decisions can be *expressed*, *discussed*, *evaluated* and *recorded* using various mathematical techniques. No such widely accepted set of techniques exists in software engineering; this paper is an attempt to fill this gap.

The proposed approach comprises three components: a method, a notation and a set of tools. The method is called M. The associated notation is called LM. We shall outline the required computerized tools (TM), which have not been implemented.

We will first list the objectives and criteria that led to the design of M.

1.1 - Implicitness

In our view, the single most important feature of specifications is that they describe objects *implicitly*, not explicitly; in other words, a specification should state properties of objects, but not give a way to construct these objects, *even* an abstract construction, using mathematical concepts. This may also be expressed by saying that the role of a specification is to say what objects **have**, not what they **are**.

As an example of this distinction, consider first the following Pascal record type definition, a programming variant of the cartesian product of sets as known in mathematics:

```
type POINT =
      record
            x, y, z : real;
            speed : VECTOR
      end
```

Then consider the following characterization of *POINT* by four functions:

$x, y, z : POINT \longrightarrow REAL$

$speed : POINT \longrightarrow VECTOR$

These two ways of defining *POINT* may at first sight seem equivalent. The first, however, is explicit, whereas the second is implicit. This is because the first completely freezes the type *POINT*, defined as being "equal" to something; only with the second is it possible to add later a new property of *POINTs*, say a mass, without changing the initial definition:

$mass : POINT \longrightarrow REAL$

Although the difference between adding a new definition and changing an existing one may at first sight seem minor, the picture changes when viewed from a software engineering perspective. An essential issue in the management of software projects is how to avoid the constant un-shelving and redesign of previously baselined elements. It is thus much preferable to be able to work by addition rather than modification, leaving existing elements untouched whenever possible.

Such an incremental approach is particularly appropriate at the specification stage, when one is exploring issues and trying out different approaches. To make this possible, however, specifications must be written with the expectation that new elements will be added later. One should thus avoid premature freezing, and leave the descriptions as open as possible. It is essential to have a specification method that supports this process.

In fact, the conclusion of the specification step can be taken to be that time when one decides to freeze all the objects involved by equating them with the cartesian product of their attributes as defined so far. Then implicit definitions can be transformed (manually or automatically) into explicit ones similar in spirit to the above Pascal type definition. This will elaborated further in section 7 when we discuss how to use an M description as a basis for system implementation.

1.2 - Object-orientedness

Software systems may be described as devices that perform certain operations on certain objects. The description of a system may be structured around the objects or around the operations.

Using the objects (or rather the object types) as the basis for the description is preferable from a software engineering point of view. The reason is that, if one consider the whole lifecycle of a system, repeated changes will occur, so that many a system bears little resemblance at any given time to what it was a few months or years before. Practical experience shows, however, that in this constant evolution (which is the rule, rather than the exception, for most real systems), the basic objects manipulated by the system tend to remain more stable than the operations performed on them.

It is thus essential to recognize and specify early the essential categories of objects that occur in the system. In M, this is done by listing the **sorts** of the system at an early stage of the specification. The rest of the specification is concerned with expressing properties of these sorts by defining the applicable operations.

A sort may be understood as just a set in the ordinary mathematical sense.

1.3 - Syntax and Semantics

The description of the relational structure of a system, i.e. what objects are connected to what other objects and what operations apply to what objects, may be called the **syntax** of the system. Its **semantics**, on the other hand, is the description of the properties of the objects and the operations.

Describing semantics is a much more difficult task than describing syntax if one is to remain at the specification level. Many of the specification systems that have been successful in industry are mostly good at describing the syntax, and their attempts at including the semantics either use natural language or resort to an operational approach (that is to say, describe algorithms rather than abstract properties), thus departing from the true realm of specification. Formal specification techniques, on the other hand, make it possible to describe system semantics while remaining at the specification level, but they require much effort.

The method used in M is to divide the description of a system into several parts ("paragraphs" in the associated notation). The first stages are concerned with syntax, the later ones add semantics. The specification task is progressive; by writing the first, syntactical paragraphs, one may already gain some benefit from the method and associated tools. To obtain a more complete description, semantic properties will be grafted onto the basic stem.

1.4 - Modular features

One of the main reasons why formal specifications have not been more widely used is (in our opinion) the lack of tools and techniques to make the specification task more manageable. M includes simple features for two key aspects of modularity [12]: decomposability and composability.

Decomposability is concerned with techniques for dividing a large system into several simpler ones (the "top-down" component), and for postponing the description of some features in order to concentrate initially on the essential aspects. A realistic specification method should provide support for such a stepwise approach to specification; it should allow system descriptions to be iterative. This is essential to help specifiers master the overwhelming amount of detail that confronts them at the early stages of a project.

Composability is the ability to combine existing pieces of specifications when writing a new one (the "bottom-up" aspect). This property is particularly important in connection with one of the essential issues of software engineering, reusability, which is just as relevant for specification as for other phases of the lifecycle.

Features of the M method and the associated notation have thus been devised to allow for modular descriptions of systems. A system description may include an *interface* paragraph that describes the connection of the current specification with others, existing or yet to be written.

1.5 - Mathematical basis

The basic modeling tools used in the description of systems are the simple mathematical notions of sets and functions. Functions may be either total or partial; partial functions play an important role in connection with error situations.

1.6 - Errors and exceptional cases

The issue of how to deal with erroneous and exceptional cases plagues software design. Much of the complexity in requirements, specifications and design documents results from the need to account for various kinds of abnormal conditions (illegal inputs, etc.).

M offers no magic cure to this problem but emphasizes the need to keep the descriptions of normal and erroneous cases distinct. The aim is not to shun away from the inescapable necessity of dealing with the latter, but to keep the former simple and manageable.

To deal with exceptional cases, M relies on the mathematical concept of partial function. An *extension* paragraph provides a way to enlarge the domains of partial functions once a first version of the specification has been written.

1.7 - Tools

A comprehensive specification method such as M may only achieve its full potential if it is supported by good good computerized tools. The tools envisioned here are essentially *management* and *configuration* tools, used to keep track of the various specifications already written or under development. Examples are specification databases to retrieve previous specifications (e.g. by keywords); linkers to combine elements of system descriptions; structural editors to help in writing specifications; analyzers to check for consistency and other properties, both intra- and inter-systems; provers.

2 - INFLUENCES

Many of the features of the M method may be found in previous work. We list the conscious influences below, and confess without any shame to having stolen many ideas from other efforts. We do hope, however, that the whole is a little more than the sum of its parts.[1]

• The most direct influence was that of the Z specification language in its various incarnations ([1] being the last known one), with its emphasis on using simple mathematical concepts to model programming concepts and, in the later versions, facilities for modular system descriptions (chapters, classes). In a sense, M is nothing more than a restricted version of Z. Another work based on the same premisses is that of Sufrin [18, 19, 14].

• Another important source of fundamental insights was the work on VDM, particularly the presentation of the "rigorous approach" in [9], although some of the features of M (for example the emphasis on implicitness) depart significantly from VDM.

• The work on abstract data types was clearly a milestone in specification. To a certain extent, M is an attempt to make abstract data type techniques available to practitioners.

• M has many points in common with formal specification methods such as Special [16], FDM [10], Affirm[15]. The main difference is that the emphasis in M has been more on expressive features (facilitating descriptions) than on proofs. Also, we have aimed at a compromise between formality and usability, by permitting the users of the method to gain some benefits from a specification even if it has not been completed down to the last quantifier. Finally, M differs from Special and FDM in that no predefined notion of state is used; the mathematical basis is elementary set theory. A set representing possible states may be introduced explicitly if needed (as in the example below), but it is then treated just as any other set.

• Among formal methods, Clear [4, 6] stands apart with Z because of its emphasis on modular, composable specifications. Also along with Z, Clear is also particularly interesting in that it has been formally defined (in at least two different ways [5, 17]), a task that has yet to be undertaken for M.

• We have also drawn some lessons from less formal but industrially successful methods. In particular, systems such as ISDOS [20] and SREM [2] emphasize the use of specifications in project management, as repositories of essential information, and the role of tools.

• Many ideas come from programming languages. The syntax of the notation associated with M, called LM, follows the Algol-Pascal-Ada line. More importantly, modular features have been strongly influenced by programming languages: the description of objects was influenced by Simula and Smalltalk, the import-export clauses are not far in spirit from what may be found in Alphard, CLU, Modula or Ada. The idea of describing a system by successive "paragraphs" that yield successive approximations was conceived as a generalization of the Ada device of writing a package in two parts: a "specification" and a body.

[1] After presenting talks on this method, we heard comments such as "this is just VDM", "this is just Alphard", etc. Since more than one other method was involved, however, the validity of these comments is trivially disproved by *reductio ad absurdum*, following from the symmetry and transitivity of the "*is just*" relation.

3 - OVERVIEW OF THE BASIC PARAGRAPHS

A description of a system in M is expressed in the notation, LM, as a set of paragraphs. There is a recommended order for writing these paragraphs, given by figure 1.

An important part of a system specification is the interface paragraph, which gives the connection with other systems, thus permitting the modular approach to system description advertised above. This paragraph, which does not appear in figure 1, will be discussed in section 5.

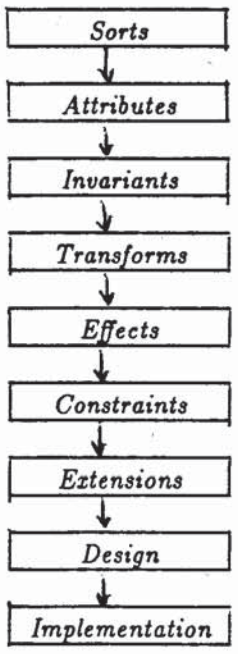

Figure 1: The Basic Paragraphs and their order.

The sorts, attributes and transforms paragraphs describe essentially what we have called the syntax of a system; the other paragraphs give the semantics. It is important to note that M has been designed so that system descriptions may be incomplete; in particular, the TM tools should be able to cope with specifications where some paragraphs are missing.

The sorts paragraphs lists the basic classes of objects that are used in the system. For each sort, a list of some specific elements may be given.

The operations of the systems are classified as "attributes" or "transforms". In both cases, the underlying mathematical notion is that of (possibly partial) function. A simple attribute on a sort X is a function

$$f : X \longrightarrow Y$$

where Y is another sort. A function of this sort represents the possibility of accessing the value of a particular attribute defined on objects of sort X (like the x coördinate of "points" in section 1.1).

An attribute on sort X may also be non-simple, that is to say, involve parameters of sorts other than X. A non-simple attribute thus corresponds to a function of the form

$$f : X \times U_1 \times U_2 \times \cdots \times U_m \longrightarrow Y$$

for some sorts U_1, U_2, ..., U_m. .

Transforms, on the other hand, represent operations that may change objects of a given sort. Mathematically, a simple transform on sort X is a function of the form

$$f : X \longrightarrow X$$

but usually transforms will involve parameters other than the objects to be changed, i.e. they will correspond to mathematical functions of the form

$$f : X \times V_1 \times V_2 \cdots \times V_n \longrightarrow X$$

The invariants and effects paragraphs give the basic semantic properties associated with attributes and transforms, respectively:

- Invariants describe properties that the attributes of all objects must always satisfy, regardless of what operations (transforms) are applied to the objects.
- Effects describe the semantics of transforms by expressing for each sort X, each transform t on X and each attribute a on X, how (if at all) the value of a may change for an object of sort X when t is applied to it.

Both attributes and transforms may be partial functions, i.e. undefined for some values, corresponding to abnormal cases. The invariants and effects apply to the case when these functions are defined, that is to say, to the specification of the normal case.

The constraints paragraph gives the exact conditions under which each partial function is defined.

For some of these partial functions, the extension paragraph defines an alternate function, to be invoked instead of the corresponding primary function when an argument falls outside of the normal domain.

The design paragraph expresses the basic decisions made by the designer regarding the architecture of the implementation, by distributing the various elements of the system among modules.

The implementation paragraph achieves the transition from design to actual implementation.

4 - AN EXAMPLE

To show how the principles outlined in the previous section are applied in practice, we have chosen to illustrate the method and the notation through a particular example. Although small, this example cannot be characterized as a toy problem. It will allow us to present the essential aspects of M, with one very important exception, modular features, whose presentation is deferred to the next section.

4.1 - A Distributed File System

We consider the following problem. A computer network (figure 2) includes machines of diverse kinds, e.g. IBM computers running MVS, others running VM, Vaxes running Unix or VMS, etc. Users of these machines need to share files. This is the case, for example, when separate teams are coöperating on a particular project.

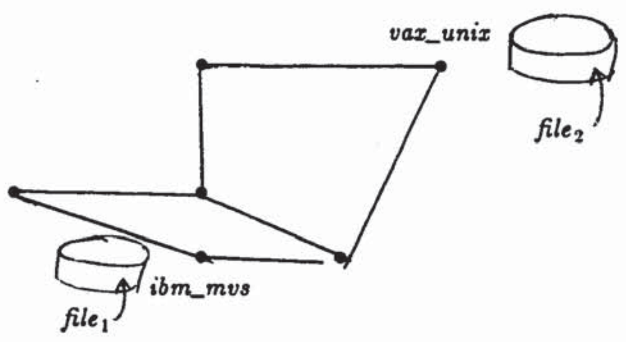

Figure 2: Files on a network

Thus a program running on a machine may need a file that resides on another. Since, however, this is a long-distance network, not a local-area one, it is impractical to let a program directly access a remote file; so what is needed is a set of tools for copying files back and forth over the net.

This immediately raises several problems. One is that various computer systems support various file types: for example, IBM MVS has a notion of "partitioned file" (a group of related sequential files, often a subroutine library), not supported by other systems; Unix and Multics have "directories", unknown on MVS or VM. This clearly puts restrictions on possible file transfers.

A more difficult problem is that of integrity: if we allow taking multiple copies of a file and then copying back updated versions, then the question arises of maintaining some control over possibly conflicting updates. Now the integrity of a file or set of files (database) cannot be defined *in abstracto*[2]: it depends on what you want to do with these files.

Thus we decide on the following policy: the tools we will design do not purport to solve the integrity problem, but they will make it possible for the designers of any particular application to implement any reasonable policy they define for application-dependent integrity control.

[2] It may, however, be definable in Abstracto.

In accordance with this idea, we decide on the following basic operations:

• *Copy*: this operation will copy a file from a given source computer to a given target computer.

• *Take*: this operation is as *Copy*, but preëmptive: once a *Take* operation has been successfully performed on a file *f*, no other program may perform a *Take* on that file until the file has been released by its temporary owner through one of the following two operations.

• *Return*: this operation copies back a previously "taken" file to its original source, taking into account any changes that may have been performed on the copy. The file becomes available again for further *Take* operations.

• *Free*: this operation makes a previously "taken" file available again for further *Take* operations. Changes performed on the copy are not reflected on the source.

• *Taken*: this operation is a query on the state of a file, which finds out whether or not the file is available for preëmptive copy (in a practical package, *Take* and *Taken* may have to be presented as a single primitive to ensure mutual exclusion).

One more design decision is needed here: how should a program reference the files it needs to access through the above primitives? In principle, a file residing on host *cmptr*, where its name (relative to the local file system of machine *cmptr*) is *local_name*, may unambiguously be identified, from any node of the network, by the pair <*cmptr*, *local_name*>.

This solution is not satisfactory, however, since it requires programmers to know precisely where each file resides on the network. Also, file naming conventions differ significantly on computer systems, and it is unpleasant to require, say, MVS programmers to know about Unix conventions or conversely. Finally, it seems wise to restrict applicability of the network file transfer operations (*Copy* and *Take*) to designated files, rather than allowing any program running on any machine to access any file on any other machine.

We thus introduce the notion of a **global name**. A file will only be available as source for the network transfer operations if it has been declared "global". When making a file global, one must give it a global name, which will be used to refer to the file if it is to be the source of a transfer operation. A new operation is thus needed:

• *Make_global*: this operation associates a global name to a file residing on a certain machine and makes this file available, through its global name, as source for the transfer operations (*Copy* and *Take*).

Clearly, global names must characterize global files uniquely over the whole network (whereas local names may be repeated: two different computers of the network may have a file called Jill). The *Make_global* operation may be implemented by creating an entry in a central catalog of global files: this catalog maintains the correspondence between global names and physical <*cmptr*, *local_name*> addresses. But other implementations may be conceived: for example, one might choose to have a specialized file server as one of the machines on the net, containing copies of all the global files. One of the roles of a useful specification is to express those properties of the system that are independent of the particular implementation chosen.

This concludes the first draft of our system specification. Of course, many details remain to be spelled out. Whereas natural language is quite adequate for discussing broad avenues of initial design, it does not suffice for the following steps, when things must be made precise, unambiguous and complete. Here formal specifications step in.

4.2 - Sorts

We begin our specification by its first paragraph, the list of sorts, given below. This is the sorts paragraph for our example system, which we call *DFS*, for "Distributed File System".

```
system DFS sorts
    COMPUTER ;
    FILE ;
    COMPUTER_TYPE has ibm_mvs, ibm_vm, vax_unix, apple_2_ms_dos, vax_vms, multics ;
    FILE_TYPE has sequential, direct_access, partitioned, directory ;
    FILE_MODE has global, nonglobal ;
    LOCAL_NAME ;
    GLOBAL_NAME ;
    FILE_CONTENT ;
    KEYWORD ;
    USER ;
    STATE ;
end system sorts ;
```

The sorts are the sets of values that may be taken by the various entities of the system being described. As a notational convention, we write sorts in uppercase and everything else in lower-case. Because of the emphasis on implicitness, we don't say much about each sort in the sorts paragraph: we give its name, and sometimes the name of some of its elements, that's all. There is no way at this stage to express that, say, a *POINT* has four components (as introduced in section 1.1), or (here) that a *FILE* is identified by a file descriptor with some concrete or even abstract structure.

COMPUTER and *FILE* are obviously needed as sorts. For the next two sorts, *COMPUTER_TYPE* and *FILE_TYPE*, we list some distinguished elements through the **has** clause. Note that there is no claim that these are the only elements (as with a Pascal type definition by enumeration): the sort is still open. The **has** clause implies, however, that the elements listed are presumed to be different.

A *FILE_MODE* makes it possible to determine whether a file has been made global.

We call *LOCAL_NAME* the sort containing all the names that may be used to identify files on the various computer systems involved. No specific property of this sort will be necessary at this level of the specification. *GLOBAL_NAME*, too, will not be described any further; this sort is used to describe possible global names for files that have been made global.

It is all nice to have names and modes associated with files, but of course if we want to describe the result of copy operations we must have the notion of *FILE_CONTENT*. Again, we need not specify this sort any further; it is enough that we can refer to it.

To "take" a file (preëmptive copy), one will need a keyword, used again to release it later. Hence the sort *KEYWORD*.

The sort *USER* is also needed for the *Take* operation: we shall need to record who has "taken" a given file. By "user", we actually mean a program rather than a person.

Finally, we need a sort *STATE* to describe the state of the complete distributed file system at any given time. The need for such a sort is a common, although not universal, occurrence in M specifications.

Here then is the first paragraph of our specification. The result achieved so far is modest but non zero: we have listed the categories of objects that play a role in our system. If we are lazy, or broke, or both, we might stop here and still benefit from having taken the trouble to write anything at all. This remark applies to each of the steps below, although we won't repeat it: an M specification may be partial, and the associated TM tools should be prepared to deal with it even if some paragraphs are missing. Of course, the full benefit of the method will only be obtained if the specification is complete, but one may already get partial results before.

4.3 - Attributes

We happen to be very courageous and enthusiastically undertake the rest of the specification. The next step is the attributes paragraph, given below (portions of lines beginning with two consecutive hyphens are LM comments).

A decision which significantly affects the appearance of M specifications was to systematically attach every attribute (and transform, see below) to one and only one sort. This raises no difficulty for what we have called "simple" attributes above, i.e. functions of the form

$$f : X \longrightarrow Y$$

Such a function will be included as part of the attributes "**on** *X*":

on *X* **attributes**

    .... ;
    f : *Y* ;
    .... ;

end *X* **attributes**

In our example, the attributes on sorts *FILE*, *COMPUTER* and *USER* fall into this category. In the general case, however, we have already mentioned that an attribute is mathematically a function of the form

$$f : X \times U_1 \times U_2 \times \cdots \times U_m \longrightarrow Y$$

We will also describe such a function as being an attribute "**on** *X*". To take the extra parameters into account, the definition of the attribute will be written as:

on *X* **attributes**

    .... ;
    $f\,(U_1, U_2, ..., U_m) : Y$;
    .... ;

end *X* **attributes**

Here, examples of such attributes are the attributes on sorts *COMPUTER_TYPE* (attribute *supporting*) and *STATE*.

Mathematically, the device that we apply to attributes is called "currying"; it consists in replacing (for $n \geq 0$) a function of $n+1$ arguments, f in our example, by a function f' of one argument (with values in X), yielding results that are functions of n arguments (in $U_1, U_2, ..., U_m$):

$$f : X \longrightarrow (U_1 \times U_2 \times \cdots \times U_m \longrightarrow Y)$$

There is a conscious dissymmetry in the convention chosen here, since we might just as well choose one of the U_i as the distinguished sort to which f is attached. The reason for this dissymmetry is the concern for modular, manageable descriptions. If we treat X and all U_i on equal footing, then we risk ending up with large, messy attributes paragraphs. Attaching each attribute to a distinguished sort makes it possible to divide the paragraph into a number of

```
system DFS attributes

    on FILE attributes
        locname : LOCAL_NAME total ;              -- The local name of a file
        host : COMPUTER total ;                   -- The machine where a file resides
        ftype : FILE_TYPE total ;                 -- Sequential file, directory etc.
    end FILE attributes ;

    on COMPUTER attributes
        make : COMPUTER_TYPE total ;              -- What brand is this computer: ibm_mvs, vax_unix...?
    end COMPUTER attributes ;

    on USER attributes
        where_running : COMPUTER total ;
    end USER attributes ;

    on COMPUTER_TYPE attributes
        supporting (FILE_TYPE) : BOOL total ;     -- Is this file type supported on this type of computer?

    on STATE attributes
        fcontent (FILE) : FILE_CONTENT total ;    -- Current contents of a file
        file_exists (LOCAL_NAME, COMPUTER) : BOOL total ;
                                                  -- Is there a file of that name on that computer?
        file_of_name (LOCAL_NAME, COMPUTER) : FILE partial ;
                                                  -- If so, what is it?
        used_globname (GLOBAL_NAME) : BOOL total ;
                                                  -- Has this global name been assigned to a file?
        globfile (GLOBAL_NAME) : FILE partial ;   -- If so, what file?
        mode (FILE) : FILE_MODE total ;           -- Has this file been made global?
        globname (FILE) : GLOBAL_NAME partial ;
                                                  -- If so, under what name?
        taken (GLOBAL_NAME) : BOOL partial ;      -- Has the file with this global name been reserved?
        owner (GLOBAL_NAME) : USER partial ;      -- If so, by whom?
        key (GLOBAL_NAME) : KEYWORD partial ;
                                                  -- and under what keyword?
    end STATE attributes ;

end system attributes ;
```

small sections, each corresponding to a sort.

This device is very close to a successful modularization technique for programming languages: the object-oriented approach to program design embodied by the Simula 67 and Smalltalk languages. The designers of Simula (followed by those of Smalltalk) introduced a conscious confusion between the notions of module and type: a module is the implementation of a data abstraction. This is in contrast with the somewhat looser notion of module found in Ada or Modula, where a module may be almost any grouping of elements (types, variables, procedures). The Simula-Smalltalk approach has some drawbacks, but it implements a very strong consistent view of modularity that in practice yields excellent system designs. These questions are further discussed in [12].

The notation used in LM to denote attributes of objects reflects the chosen dissymmetry: the argument corresponding to the distinguished sort will be written using dot notation (as for components of Pascal record types, properties of Simula reference variables etc.); the other arguments, if any, will be written in parentheses. Thus if *f* is an object of sort *FILE*, then its local name (an attribute defined in the "**on** *FILE*" section) will be written

s • *locname*

The host on which it resides will be written *s* • *host*, etc. Referring now to the "**on** *STATE*" section, the value of the attribute *file_exists* for a state *s*, a local name *l* and a computer *c* will be written

s • *file_exists* (*l*, *c*)

etc.

The last general remark necessary to fully understand the attributes paragraph is that attributes may be partial functions: some attributes may not be defined in all cases. Being partial is an important property, so every attribute definition must be followed by one of the two keywords **total** or **partial**. For any partial attribute, there will be an entry in the constraints paragraph (see section 4.7) describing the exact conditions under which the attribute is defined.

A few comments on the attributes of the example may be useful.

Note the difference between the attributes on *FILE* (properties of files which do not depend on the system state, like the host on which a particular file resides, its local name, its type, which are considered to be innate properties of the file) and the properties of files that are defined under *STATE* because they are state-dependent, like the content of a file.

On sort *USER*, attribute *where_running* gives the host on which a user (i.e. program) is being executed.

If *ct* is a computer type and *ft* is a file type, then

ct • *supporting* (*ft*)

is a boolean value (we assume the sort *BOOL* to be one of a small number of predefined sorts), true if and only if computer type *ct* supports file type *ft*. Thus we will expect *ibm_mvs* • *supporting* (*directory*) to be false, *vax_unix* • *supporting* (*sequential*) to be true, etc. (these properties will be expressed in the invariants paragraph).

On sort *STATE*, attribute *fcontent* gives the current contents of any file. Files will usually be accessed through their names, so we need to describe the correspondence between a file name and a file; this is achieved through attribute *file_exists*, which corresponds to the query "is there a file with a given name on a given computer?". In a state *s*, given a local file name *l* and a computer *c*, the file of name *l* on computer *c* is

s • *file_of_name* (*l*, *c*)

Note that attribute *file_of_name* is partial because there might be no file of name *l* on *c*. The precise condition under which *s* • *file_of_name* (*l*, *c*) is defined is that *s* • *file_exists* (*l*, *c*) be true; this condition will be expressed in the constraints paragraph of the specification.

If *g* is a global name, then *s* • *used_global_name* yields true if and only if name *g* has been assigned to a global file in state *s*. If this is the case, then this file may be obtained as *s* • *globfile* (*g*).

The "mode" of a file is *global* if and only if the file has been made global. If so, the file has a global name, obtained as *s* • *globname* (*g*).

Attribute *taken* applies to a global name and determines whether the file with that global name has been "taken" by a user in the current state; this attribute is partial because it only applies to global names which have been assigned to a file. If *s* • *taken* (g) is true for a global name g, then the user that has "taken" the corresponding file is given by *s* • *owner* (g) and and the key that was used to reserve it is *s* • *key* (g).

Note that because of the correspondence between global files and global names (attribute *globname* and *globfile*), the arguments of attributes *taken*, *owner* and *file* could have been chosen as *FILE*s rather than *GLOBAL_NAME*s.

4.4 - Invariants

The invariants express properties of the attributes which must always hold. The invariants paragraph for our example is given below.

```
system DFS invariants
     declare l : LOCAL_NAME, g : GLOBAL_NAME, c : COMPUTER, s : STATE, f : FILE ;
     i1 :   f • host • make • supporting (f • ftype) ;
     i2 :   s • file_of_name (l, c) • locname = l ;
     i3 :   s • globfile (s • globname (f)) = f ;
     i4 :   s • globname (s • globfile (g)) = g ;
     i5 :   s • used_globname (s • globname (f)) ;
     i6 :   s • mode (s • globfile (g)) = global ;

     j1 :   ibm_mvs • supporting (ft) = (ft ∈ {sequential, direct_access, partitioned}) ;
     j2 :   vax_unix • supporting (ft) = (ft ∈ {sequential, direct_access, directory}) ;
     j3 :   multics • supporting (ft) = (ft ∈ {sequential, direct_access, directory}) ;
     j4 :   vax_vms • supporting (ft) = (ft ∈ {sequential, direct_access, directory}) ;
     j5 :   apple_2_ms_dos • supporting (ft) = (ft ∈ {sequential, direct_access}) ;
end system invariants
```

Each invariant has a label (i_1, i_2, j_1, j_2, etc. in our example), which may be used to refer to it. Names are used in the invariants to denote objects of various sorts; they are introduced by a **declare** clause. By convention, any free variable is considered to be universally quantified, so that invariant i_1, for example, should be understood as if it was preceded by $\forall\ f \in$ *FILE*.

The meaning of the invariants should not be hard to understand. Invariant i_1 gives a consistency condition on file types: the brand of the computer on which file f resides (that is, *f* • *host* • *make*) must support the file type of f. Invariant i_2 is a consistency property on attributes *file_of_name* and *locname*: the name of the file of name l (on a computer c, in a state s) is l. Invariants i_3 and i_4 express that attributes *globfile* and *globname* are inverse of each other. Invariant i_5 expresses the relationship between *used_global_name* and *globname*, i_6 between *mode* and *globfile*.

Invariants j_1 to j_5 simply give the properties of attribute *supporting* by enumeration.

For the more interesting invariants (i_1 to i_6), the reader will have noticed that some of the functions involved are partial, so the meaning of equality must be made more precise. The appropriate interpretation is "weak equality": $a=b$ means "if both a and b are defined, then they are equal". (Recall that the precise specification of the domains of partial functions is deferred to the constraints paragraph).

Invariants play a very important role in expressing the fundamental properties of a system, those which must be preserved by any operation applied to its objects. The search for relevant invariants rewards the system designer with insights into the really important features. It also yields two important side benefits[3]:

- Invariants provide guidance for **testing**: the first thing to check when monitoring the behavior of the system, or a prototype of the system, on a set of test inputs, is whether any invariant is violated. This form of testing is effective because it goes right to the essential properties of the system, as opposed to "blind" testing.
- Invariants are useful for evolutive **maintenance**: to check whether a change to the software preserves the "essential semantics" of the system, one should go back to the original invariants and see if they still hold.

4.5 - Transforms

So far we have described only the static properties of our system. We come now to its dynamics, represented by transforms.

The transforms paragraph has several features in common with the attributes paragraph. In the same fashion as attributes, transforms will be curried, i.e. a transform function of the form

$$transf : X \times V_1 \times V_2 \cdots \times V_n \longrightarrow X$$

will appear as a transform "**on** X":

```
on X transforms
        ............................. ;
        transf (V₁, V₂, ..., Vₙ) ...... ;
        ............................. ;
end X transforms
```

As attributes, transforms are declared as either partial or total. Application of a transform is written using the same convention as for attributes: if x is an element of sort X, the object (of the same sort) resulting from applying transform *transf* to x, with arguments $v_1, v_2, ..., v_n$ is denoted

$$x \bullet transf\ (v_1, v_2, ..., v_n)$$

or just $x \bullet transf$ in the case of a simple transform with no arguments.

An important feature of transforms is that they are entirely specified by their effects on attributes. Let *transf* be a transform on sort X. When defining t, the M specifier must examine all attributes defined on X in the attributes paragraph, and determine for each such attribute *attr* whether application of *transf* may change the value of *attr*. In other words, one must specify whether

$$x \bullet transf(v_1, v_2, ..., v_n) \bullet attr\ (u_1, u_2, ..., u_m)$$

may or may not be different from

$$x \bullet attr\ (u_1, u_2, ..., u_m)$$

for arbitrary $v_1, v_2, ..., v_n, u_1, u_2, ..., u_m$. Here we are assuming that attribute *attr* has m arguments; the u_i argument list would be omitted for a simple attribute.

Every transform definition will thus be followed by the list of attributes that it may change in this fashion (preceded by the keyword **change**) as illustrated by the example below.

[3] I am indebted to J.-R. Abrial for these remarks.

```
system DFS transforms

    on STATE transforms
        make_global (FILE, GLOBAL_NAME) partial
            change mode, globfile, globname, used_globname ;
                - - Make this file global with this global name
        copy (GLOBAL_NAME, FILE) partial
            change fcontent ;
                - - Copy the contents of the file with this global name into this other file
        take (GLOBAL_NAME, FILE, USER, KEYWORD) partial
            change fcontent, taken, owner, key ;
                - - As "copy", but also preëmptive
        return (GLOBAL_NAME, USER, KEYWORD) partial
            change fcontent, taken, owner, key ;
                - - Copy back and release
        free (GLOBAL_NAME, USER, KEYWORD) partial
            change taken, owner, key ;
                - - Release without copying back
    end STATE transforms ;
end system transforms ;
```

This process of doing for each sort the complete "product" of transforms by attributes is an important part of M specifications. Note that for the reader of a specification the "**change** ..." list next to each transform definition is useful not only because it highlights attributes affected by the transform but also, just as importantly, because it makes it possible to infer what attributes may **not** possibly be impacted.

The precise description of *how* every impacted attribute is changed by a given transform is deferred to the next paragraph of the specification, the effects paragraph.

In our example, the five transforms (all on sort *STATE* at this stage of the specification) correspond to the operations introduced in the informal draft. For each of them, the reader should check the list of possibly changed attributes.

4.6 - Effects

Next we return to the semantics of the system by giving the effects of the various transforms. The structure of the effects paragraph leaves no place for hesitation: there must be one and exactly one entry for every item of every "**change**" list in the transform paragraph.

Precisely, if we have a transform entry of the form

on *X* **transforms**

    .... ;

    transf $(V_1, V_2, ..., V_n)$
        change ...., *attr*, ... ;

    .... ;

end *X* **transforms**

where *attr* is defined in the attributes paragraph as

on X **attributes**

> .... ;
>
> $attr\ (U_1,\ U_2,\ ...,\ U_m) : Y$;
>
> .... ;

end X **attributes**

then the effects paragraph must contain an entry of the form

$$x \bullet transf\ (v_1,\ v_2,\ ...,\ v_n) \bullet attr\ (u_1,\ u_2,\ ...,\ u_m) \quad = \quad E_{transf\,,\,attr}[v_1,\ v_2,\ ...,\ v_n,\ u_1,\ u_2,\ ...,\ u_m]$$

where $E_{transf\,,\,attr}$ [...] is an expression, usually involving the value of the attribute before the transform is applied, i.e. $x \bullet attr\ (u_1,\ u_2,\ ...,\ u_m)$. Here we are assuming a proper **declare** line for all the variables involved; recall that free variables are assumed to be universally quantified (that is, preceded by $\forall$).

The left-hand side of such an entry is entirely determined by the previous paragraphs of the specification; so the TM supporting tools should be able to construct it automatically. Of course, the right-hand side (expression E) can only be provided by the specifier.

The effects paragraph below describes precisely the result of the various operations in our example problem. To write it, we rely on the following useful notation. Let h be a function:

$$h : X \longrightarrow Y$$

Let $a \in X$ and $v \in Y$. We denote by

> $g =$ **replace** h **at** a **with** v

the function g that is identical to h except that its value for element a is v. In other words, for any $x \in X$:

> $g\ (x) =$ **if** $x = a$ **then** v **else** $h\ (x)$ **end if**

If h is a partial function, the domain of g is **domain** $(h) \cup \{a\}$.

The **replace...** form is not strictly part of the LM notation, but may be considered as a simple abbreviation (macro) for the **if...then...else..end if** expression, which is in LM. The former makes it possible to describe effects more clearly.

In a similar fashion, we denote by

> $g =$ **undefine** f **at** a

a function that is the restriction of f to **domain** $(f) - \{a\}$.

An important point should be noted regarding the meaning of the given effects in the case of partial functions. The transforms whose effects are given in the effects paragraph, and the attributes on which these effects are given, may be partial. The convention is that the effects described by the right-hand sides of the equalities in this paragraph are applicable only when the left-hand sides are defined. When a left-hand side is not defined, whether the corresponding right-hand side is defined or not does not matter; all bets are off.

Note, however, that whenever a left-hand side is defined, then the corresponding right-hand side must also be defined since its evaluation is required to obtain the value of the left-hand side. This consistency problem will be studied in section 6.6.

```
system DFS effects
    declare l : LOCAL_NAME, g : GLOBAL_NAME, c : COMPUTER,
        s : STATE, f : FILE, k : KEY, u : USER ;
    s • make_global (f, g) • mode = replace s • mode at f with global ;
    s • make_global (f, g) • globfile = replace s • globfile at g with f ;
    s • make_global (f, g) • globname = replace s • globname at f with g ;
    s • make_global (f, g) • used_globname = replace s • used_globname at g with true ;
    s • copy (g, f) • fcontent = replace s • fcontent at f with s • globfile (g) • fcontent ;
    s • take (g, f, u, k) • fcontent = replace s • fcontent at f with s • globfile (g) • fcontent ;
    s • take (g, f, u, k) • taken = replace s • taken at g with true ;
    s • take (g, f, u, k) • key = replace s • key at g with k ;
    s • take (g, f, u, k) • owner = replace s • owner at g with u ;
    s • return (g, f, u, k) • fcontent = replace s • fcontent at s • globfile (g) with f • fcontent ;
    s • return (g, f, u, k) • taken = replace s • taken at g with false ;
    s • return (g, f, u, k) • owner = undefine s • owner at g ;
    s • return (g, f, u, k) • key = undefine s • key at g ;
    s • free (g, f, u, k) • taken = replace s • taken at g with false ;
    s • free (g, f, u, k) • owner = undefine s • owner at g ;
    s • free (g, f, u, k) • key = undefine s • key at g ;
end system effects ;
```

4.7 - Constraints

So far we have been treading on rather unsteady ground since our specification contains partial functions and we have all but ignored undefined values. This method is useful for concentrating on the basic cases first, but of course at some point we must say exactly when operations are applicable and where they are not. This is the object of the constraints paragraphs.

In this paragraph, we look back at the definitions of attributes and transforms, and we include an entry for each function that has been introduced as partial (again, the TM tools should guide us here by automatically providing the list of entries to be filled). Each entry will thus correspond to a partial function f (attribute or transform), previously defined as being "on X" for some sort X, possibly with parameters in sorts A_1, A_2, ..., A_n ; the entry will be written in the form

```
system DFS constraints
    declare l : LOCAL_NAME, g : GLOBAL_NAME, c : COMPUTER,
        s : STATE, f : FILE, k : KEY, u : USER ;
    s in domain file_of_name for l, c iff s • file_exists (n, c) ;
    s in domain globfile for g iff s • used_globname (g) ;
    s in domain globname for f iff s • mode (f) = global ;
    s in domain taken for g iff s • used_globname (g) ;
    s in domain owner for g iff s • taken (g) ;
    s in domain key for g iff s • taken (g) ;
    s in domain make_global for f, g iff not s • used_globname (g)
    s in domain copy for g, f iff
        s • globfile (g) • ftype = f • ftype and not (s • taken (s • globname (f)))

    s in domain take for g, f, u, k iff
        s • globfile (g) • ftype = f • ftype and not (s • taken(s • globname (f)))
        and not s • taken (g) ;

    s in domain return for g, u, k iff
        s • taken (g) and s • owner (g) = u and s • key (g) = k ;

    s in domain free for g, u, k iff   - - Same conditions as for return
        s • taken (g) and s • owner (g) = u and s • key (g) = k ;

end DFS constraints ;
```

x in domain f **for** $a_1, a_2, ..., a_n$ **iff** P

where P is a condition on $x, a_1, a_2, ..., a_n$, defining the constraints that must be satisfied by these arguments to ensure that $x \bullet f\ (a_1, a_2, ..., a_n)$ is defined.

One of the benefits of a formal specification is that it forces the software designer to give precise answers to some questions that are very important for the behavior of the eventual system. We have an example here with the constraints on such transforms as *copy* and *take*. Although it was stated in the informal draft specification (section 4.1) that *take* is preëmptive but *copy* is not, another problem was not addressed: is one permitted to perform a *copy* whose source is a file that has been reserved by a *take* operation? Here we cannot escape this question. The choice described below is to authorize a *copy* from a source that has been "taken"; but a *copy* or *take* operation may not use a reserved file as its **target**. Formal notations naturally lead to asking (and answering) such important questions.

An important point to note is the convention used when conditions on the domain of a partial function refer to other partial functions. The convention is that the expression $f\,(a) = g\,(b)$, where f and g may be partial functions, is a shorthand for

$(a \in$ **domain** (f) **and** $b \in$ **domain** $(g))$ **and then** $f\,(a) = g\,(b)$

where **and then** is the non-commutative **and** operator (yielding false if its first operand is false, regardless of whether its second operand is defined or not). Thus the condition for *owner* below, for example, should be understood as *s* **in domain** *owner* **for** *g* **iff** *s* • *used_globname (g)* **and then** *s* • *taken (g)* ;

This device, which significantly simplifies the expression of constraints, corresponds to a particular logic for dealing with undefinedness, analyzed more precisely in section 6.5 below.

4.8 - Extensions

Partial functions provide a simple mathematical tool for describing computations which should not be attempted. We find this approach preferable to the alternative way of dealing with errors by using explicit "undefined" elements with special properties [7]. Our approach follows from one of the main tenets of M, namely that a specification method should allow the system designer to concentrate on the essential things first, without being overwhelmed at once by all the details that the final system will have to take into account.

As the specification is being refined, however, partial functions cannot usually remain partial indefinitely: in implemented systems, one likes all functions to be total, if only out of politeness towards the users of the system.

The extension paragraph (not described any further in this version of the paper) makes it possible to improve a specification containing partial functions by associating with every partial function (attribute or transform) an alternate function, known as its **doppelgänger**, to be used in lieu of the original function for arguments that fall outside its domain.

5 - SYSTEM COMPOSITION AND DECOMPOSITION

As pointed out in section 1.4, it is essential for practical specifications to allow the decomposition of system descriptions into descriptions of subsystems, and of re-using existing specifications when describing new systems.

The modular features of M are based on an analysis of the relationships that may exist between systems. The following relations are of primary importance.

- **1** - *B is a particular case of A*. In other words, anything that is true of *A* is also true of *B* (but some properties may be true of *B* that are not necessarily true of *A*).
- **2** - *B contains an instance of A*. For example, *A* could be the system of "trees", where *B* uses one or more trees.
- **3** - *B is a particular case of a, with some exceptions*. This is like case **1**, except that some of the properties of *A* may not hold for *B*. This is very important in practice, since so many systems are "almost" upward-compatible with existing systems. Thus there must be a way to import elements from a specification while explicitly rejecting some of their properties.

M provides support for these three kinds of interaction in the interface paragraph. To support **1**, we include in the interface paragraph a section of the form:

from *A* **use**

$\alpha; \beta ; ...$

end *A* **use**

In this notation, α, β, etc. denote "syntactic" elements, that is to say, sorts, attributes and/or transforms. The semantic properties (invariants, effects, constraints) of these elements should not be included: they follow automatically.

If, on the other hand, some of these properties are **not** wanted (that is to say, in case **3** above), then they can be excluded explicitly. The **use** section will then contain an **except** clause, as follows:

from *A* **use**

$\alpha; \beta ; ...$

except $\gamma, \delta, ...$

end *A* **use**

where $\gamma, \delta, ...$ refer to invariants (denoted by their tags, e.g. i_3 in our example), effects (denoted as **effect** *transf* **on** *attr*), or constraints (denoted as **constraint on** *f*).

As a notational convenience, it is permitted to have a **use** section of the form

from *A* **use**

all ;

except $\gamma, \delta, ...$

end *A* **use**

making all elements of *A* available to the current system description except those which are explicitly excluded. In this case, the excluded elements (*gamma*, δ,...) may include sorts as well as semantic properties.

Interfaces of type **2** above are expressed within the same notation using a very simple device: a **use** section may include "renamed" clauses, as follows:

```
from A use
        α ;
        β renamed β1 ;
        ...
end A use
```

In this fashion, several instances of the specification for the same system A may be used in the specification for B. For example, assume that we have the specification of a system *LISTS* describing the properties of lists. This specification includes sorts *LIST* and *ELEMENT*; on the former sorts, it has attributes such as ***empty*** and transforms such as ***insert_front***, ***insert_back***, ***append*** etc. Now assume we have a specification which needs lists of integers and lists of reals. Then this specification will have sorts *INTEGER*, *REAL*, *INTEGER_LIST*, and *REAL_LIST*; its interface paragraph will need two **use** clauses, as follows:

```
system S interface
        from LISTS use
                ELEMENT renamed INTEGER ;
                LIST renamed INTEGER_LIST ;
                empty renamed empty_integer_list ;
                insert_front renamed integer_insert_front ;
                insert_back renamed integer_insert_back ;
                append renamed integer_append ;
                - - etc.
        end LISTS use ;

        from LISTS use
                ELEMENT renamed REAL ;
                LIST renamed REAL_LIST ;
                empty renamed empty_real_list ;
                insert_front renamed real_insert_front ;
                insert_back renamed real_insert_back ;
                append renamed real_append ;
                - - etc.
        end LISTS use

end system interface
```

The fundamental rule here is that **no overloading of names** whatsoever is permitted in the LM notation: any conflict must be resolved by renaming as above. As a consequence of this rule, if several properties are given for the same object and they are not logically contradictory, they are considered as cumulative rather than conflicting.

6 - PROVING THE CONSISTENCY OF A SPECIFICATION

6.1 - Overview

The reader may have noted that the process of writing an M specification, as seen so far, is rather open; one may write many things, and not much control is exercised, even though the method uses potentially unsafe features like partial functions.

What justifies this somewhat easy-going approach is that at early stages the most difficult problem is to understand what the system is all about; so the emphasis in the M features seen so far has been on expressive power more than security. One should not prematurely confuse specification with verification.

This cannot go on forever, however: one of the primary aims of specifications, especially formal ones, is to significantly increase the trust that users can put in software systems. So at some point one has to get serious about the consistency of the specification.

Thus we now study the properties that must be proved to make sure that an M specification is consistent. Such properties are of three kinds:

- invariant-transform consistency (transforms preserve invariants);
- constraint consistency (constraints are meaningful);
- constraint-effect consistency (effects are meaningful under the given constraints).

The rules given below imply relatively tedious proofs. The need for verifications of the last two kinds should be considered in light of the ease of specification gained by the use of partial functions. As opposed to other specification methods (e.g. the traditional way of dealing with abstract data types, see[7,8]), the M specifier does not have to clutter his system description with special cases for "error elements" associated with each type. He can thus concentrate on the meaningful properties of the specification. The price to pay for this simplicity of expression is the need to check the consistency of the eventual specification (and to correct possible oversights resulting from inadvertently using a function outside its domain). It is expected that this latter process should be strongly supported by tools.

6.2 - Consistency of modular specifications

When defining consistency, we shall be talking in terms of a single, independent specification; for specifications with interface paragraphs, the proofs described below need to be performed on the composite specification resulting from combining the elements of the given specification with all those it uses from other specifications.

Assuming the specification of S refers to T, we thus request as consistency proof for S a proof of the composite specification combining the specification of S with all the elements it uses from T. It would clearly be much preferable to separately prove the consistency of T and the conditional consistency of S. Further investigation is needed on this problem, which is made non-trivial by the versatility of the modular facilities of M.

6.3 - Notations

Consider a sort X. A function f on X (attribute or transform) may be modeled mathematically as a possibly partial function of the form

$$f : X \longrightarrow (Y \longrightarrow Z)$$

where Y is a one-element set if f has no parameters, and Z is the same as X in the case of a transform.

We shall denote by C_f the function defining the constraint on f; that is to say, C_f is of the form

$$C_f : X \longrightarrow (Y \longrightarrow BOOL)$$

where $BOOL$ is the set {*true, false*}. Function f is applicable to x and y if and only if

$$x \bullet C_f\ [y]$$

has value true (note that we apply to C_f the same dissymmetric dot notation used for attributes and transforms).

If f is a total function, then C_f is identically true. Otherwise, f appears in the constraints paragraph with a clause of the form

x in domain f for y iff $\Gamma_f\ [x, y]$

$\Gamma_f\ [x, y]$ must be expressed in terms of some of the attributes of x; in other words, $\Gamma_f\ [x, y]$ is of the form

$$\Gamma_f\ [x \bullet \alpha_1 (y),\ x \bullet \alpha_2 (y),\ \cdots\ x \bullet \alpha_{n_f} (y)]$$

where $\alpha_1, \alpha_2, \cdots \alpha_{n_f}$ are attributes on sort X. Let *Attrib* (C_f) be the set of attributes $\{\alpha_1, \alpha_2, \cdots \alpha_{n_f}\}$, i.e. the set of all attributes on X that take part in the definition of the constraint on f. *Attrib* (C_f) is empty if f is total.

Similarly, if a is an attribute on sort X and t a transform on X that may change a, there will be a line in the effects paragraph of the form

$$x \bullet t\ (y) \bullet a\ (z) = x \bullet E_{t,a}\ [y, z]$$

where $E_{t,a}$ is the function defining the effect of t on a. We denote by *Attrib* $(E_{t,a})$ the set of attributes that appear in the expression for $x \bullet E_{t,a}\ [y, z]$.

Finally, for an invariant of the form $x \bullet I\ (z)$, we write *Attrib* (I) for the set of attributes that appear in I.

6.4 - Invariance properties

The first kind of properties to be checked is that the invariants are preserved by the transforms.

Let t be a transform on a sort X. The set of attributes on X that may be changed by t is given in the transforms paragraph. For each such attribute a, the effect $E_{t,a}$ of t on a is given in the effects paragraph; note that this clause is only valid when the application of the transform and of the attribute is defined.

Denote by *ALLINV* the conjunction of all the invariants involving attributes on sort X. Let I be one of these invariants, involving a (and possibly other attributes on X). I appears in the invariants paragraph under the form

$$x \bullet I\ (z)$$

with implicit universal quantification on x and z.

To say that transform t is consistent with the invariants means that for any such invariant I, whenever an element x satisfies *ALLINV* (thus, in particular, I) and t is applied to x, the resulting element $x \bullet t\ (y)$ satisfies I.

This property is only required to hold when the transform is applicable, i.e. when the constraint C_t holds on x. Hence the first law of consistency:

Invariant-Transform Consistency Rule

For any sort X, any invariant I involving elements of X and any transform t on X, the following must hold:

$$\forall\ x, y, z, (x \bullet C_t\ [y] \ \wedge\ (\forall z', x \bullet I(z'))) \ \Rightarrow\ x \bullet I_{t,a}\ (y,z))$$

where $x \bullet I_{t,a}\ (y,z)$ is the expression obtained by substituting, for every attribute $\alpha \in$ *Attrib* (I), $x \bullet E_{t,\alpha}\ [y,z]$ for $x \bullet \alpha(z)$ in $x \bullet I(z)$.

As an example, let us consider invariant i_3 of the above example specification and prove that it is preserved by transform *make_global*. The invariant is

i_3 : *s* • *globfile* (*s* • *globname* (*f*)) = *f* ;

The property to be proved is:

$\forall$ *s* ∈ *STATE*, *f'* ∈ *FILE*, *g* ∈ *GLOBAL_NAME*,

$(C_{make\_global}[f', g] \ \wedge\ (\forall f \in FILE,\ ALLINV)) \ \Rightarrow\ i'_3$

where i'_3 is i_3 with *s* • *make_global* (*f '*, *g*), as obtained from the effects paragraph, substituted for *s*. In other words, i'_3 is

s' • *globfile* (*s'* • *globname* (*f*)) = *f*

where

s' = *s* • *make_global* (*f'*, *g*)

Let *lhs* be the left-hand side of i'_3. We have

lhs = *s'* • *globfile* (*g'*)

with *g '* = *s* • *make_global* (*f'*, *g*) • *globname* (*f*)). The effect $E_{make\_global,\ globname}$ gives that

g ' = **if** *f* = *f'* **then** *g* **else** *s* • *globname* (*f*) **end if**

Thus, factoring out the conditional expression, we get:

lhs = **if** *f* = *f'* **then** *s'* • *globfile* (*g*) **else** *s'* • *globfile* (*s* • *globname* (*f*)) **end if**

The value obtained in the **then** clause is

s • *make_global* (*f'*, *g*) • *globfile* (*g*)

that is to say *f'*, according to the effect $E_{make\_global,\ globfile}$.

The value obtained in the **else** clause is

s • *make_global* (*f'*, *g*) • *globfile* (*s* • *globname* (*f*))

that is to say, applying $E_{make\_global,\ globfile}$ again:

if *g* = *s* • *globname* (*f*) **then** *f'* **else** *s* • *globfile* (*s* • *globname* (*f*)) **end if**

where the second case is just *f* because of the presence of invariant i_3 in the hypothesis. Thus we get the following expression for the left-hand side *lhs* of i'_3 (which we must prove is equal to *f*):

```
if $f = f'$ then $f'$
else if $g = s\bullet globname(f)$ then $f'$
else $f$
end if
```

The value of this expression is f in the first and third cases. But the condition for the second case, namely $g = s\bullet globname\ (f\,)$, is contradictory with the constraint on $s\bullet make\_global\ (f', g)$, as defined in the constraints paragraph:

not $s\bullet used\_globname\ (g)$

when one takes into account the invariant i_5[4]:

$s\bullet used\_globname\ (s\bullet globname\ (f))$

Thus the value of *lhs* is f in all legal cases, which concludes the proof that transform *make_global* preserves invariant i_3.

Note that as evidenced by this example, it is necessary in general to include the relevant constraints and all the invariants in the hypotheses for invariant preservation proofs.

A proof such as the above one (for just one transform and one invariant!) is not difficult but tedious; supporting tools are obviously required.

6.5 - Constraint consistency

The constraint consistency rule ensures that constraints are meaningful as given in the specification.

The problem here is that the constraint on a transform or attribute may be defined in reference to one or more attributes, some of which may be partial. This is quite clear in the example discussed above: the constraints on attributes *owner* and *key* as well as those on transforms *copy* and *take* refer to *taken*, itself a partial attribute. Thus the problem arises of whether the constraints define anything at all.

This problem is solved by imposing a strict order on constraints.

Constraint Consistency Rule

Consider the relation $\textcircled{<}$ defined as follows: $f \textcircled{<} g$ if and only if the constraint on f refers to g (where f is an attribute or a transform and g an attribute).

Then the relation $\textcircled{<}$ must be acyclic.

This rule (which is indeed satisfied by our example of section 4), must be understood together with the convention defined in section 4.7: in the predicate defining a constraint, any subpredicate involving a partial function is considered **false** outside the domain of that function.

This corresponds to a special logic for dealing with undefinedness, different from the ones examined in [3], with the following truth tables.

[4] It may be worthwhile to mention that we had initially overlooked the need for invariant i_5. It is only when trying to prove the invariance of i_3 that we realized the invariant now called i_5 was required to carry out this proof, as shown here.

The symbol $\perp$ denotes the result of applying a function outside of its domain.

The first two tables (for equality and inequality) apply to a simple flat domain with elements 0, 1, 2, $\perp$ and can be generalized to any flat domain.

The next three tables (for and, or and not) are to be used for constraints involving attributes that return boolean results. Such a boolean-valued attribute, usually total, is often used in connection with a partial attribute, to serve as explicit characteristic function on the domain of the latter: in our example, ***file_exists*** plays this role for ***file_of_name***, ***used_globname*** for ***globfile*** and ***taken***, ***taken*** for ***owner*** and ***key*** (but ***globname*** has a non-boolean attribute, ***mode***, for this purpose).

$=$	*0*	*1*	*2*	$\perp$
0	t	*f*	*f*	*f*
1	*f*	t	*f*	*f*
2	*f*	*f*	t	*f*
$\perp$	*f*	*f*	*f*	*f*

$\neq$	*0*	*1*	*2*	$\perp$
0	*f*	t	t	*f*
1	t	*f*	t	*f*
2	t	t	*f*	*f*
$\perp$	*f*	*f*	*f*	*f*

$\neg$	
t	*f*
f	t
$\perp$	*f*

$\wedge$	t	*f*	$\perp$
t	t	*f*	*f*
f	*f*	*f*	*f*
$\perp$	*f*	*f*	*f*

$\vee$	t	*f*	$\perp$
t	t	t	t
f	t	*f*	*f*
$\perp$	t	*f*	*f*

$\Longrightarrow$	t	*f*	$\perp$
t	t	*f*	*f*
f	t	t	t
$\perp$	t	*f*	*f*

Note that $(a \Longrightarrow b) \equiv (\neg\ a \vee b)$. But De Morgan's laws are not satisfied when undefined elements are taken into account: for example, $\neg \perp \vee \neg\ t = f$, but $\neg\ (\perp\ \wedge\ t) = t$. Even a simple law of boolean algebra such as $\neg\ \neg\ a = a$ does not hold for $\perp$. Also, the basic functions are not strictly monotonic.

The motivation for this seemingly strange logic should be clear. Logical expressions appearing in constraints define the conditions under which a given function, say f, may be applied. Since all the other properties of f (invariants and effects) are meaningless outside of the domain of f, it is essential to know for sure that f is defined when we need it. Thus if the constraint on f involves another partial function, a conservative attitude ("*when in doubt, say no!*") is taken: any condition that is not defined is considered to be false.

6.6 - Constraint-Effect Consistency

The last type of property to check relates to the effects. The effect of a transform t on an attribute a is given under the form

$$x \bullet t\,(y) \bullet a\,(z) = x \bullet E_{t,a}\,[y, z]$$

where t and a may be partial functions, and the right-hand side is an expression that may also involve partial functions. The interpretation given in section 4.6 is that the effect is only applicable when the left-hand side is defined; but then one should make sure that the right-hand side is defined. This is the constraint-effect consistency problem.

Informally, the constraint-effect consistency rule expresses that whenever the constraints of the specification imply that the left-hand side $x \bullet t\,(y) \bullet a\,(z)$ is defined, then they must also imply that the right-hand side $x \bullet E_{t,a}\,[y, z]$ is defined.

In other words, if $x \bullet L_{t,a}\,[y, z]$ is the condition for the left-hand side to be defined, and if $x \bullet R_{t,a}\,[y, z]$ is the condition for the right-hand side to be defined, the constraint consistency rule is that

$$\forall\ x, y, z, \quad x \bullet L_{t,a}[y,z] \implies x \bullet R_{t,a}[y,z]$$

To refine this rule, we must examine more closely the conditions under which each side of the "effect" specification is defined.

The right-hand side, $x \bullet E_{t,a}\,[y, z]$, is usually given by case analysis (as in the example above; recall that the **replace...** form is an abbreviation for a conditional expression):

```
**if** $x \bullet Cond_1\,(y, z)$ **then** $x \bullet Val_1\,(y, z)$
**else if** $x \bullet Cond_2\,(y, z)$ **then** $x \bullet Val_2\,(y, z)$
...
**else if** $x \bullet Cond_{n-1}\,(y, z)$ **then** $x \bullet Val_{n-1}\,(y, z)$
**else** $x \bullet Val_n\,(y, z)$
**end if**
```

The condition for such a conditional expression to be defined is:

```
$x \bullet R_{t,a}\,[y, z] =$
    **if** $x \bullet Cond_1\,(y, z)$ **then** $x \bullet Defined_1\,(y, z)$
    **else if** $x \bullet Cond_2\,(y, z)$ **then** $x \bullet Defined_2\,(y, z)$
    ...
    **else if** $x \bullet Cond_{n-1}\,(y, z)$ **then** $x \bullet Defined_{n-1}\,(y, z)$
    **else** $x \bullet Defined_n\,(y, z)$
    **end if**
```

where $x \bullet Defined_i\,(y, z)$ is the condition for $x \bullet Val_i\,(y, z)$ to be defined, obtained as the conjunction of all the constraints $x \bullet C_\alpha\,[y, z]$ for every attribute $\alpha \in Attrib\,(Val_i)$ occurring in the definition of the i-th alternative. This right-hand side is usually less formidable to determine in practice that the above general form would suggest (an example is given below).

We now examine the left-hand side of the "effects" specification for t and a. This left-hand side is a function composition (of t and a). A basic theorem on partial functions is that, if f and g are two functions and h their composition (in this order), then

$$\textbf{domain}\ (h) = \{a \in \textbf{domain}\ (f) \mid f(a) \in \textbf{domain}\ (g)\}$$

Thus, for the left-hand side to be defined, two conditions must be met:

• The constraint on t, namely $x \bullet C_t\ [y]$;

• The constraint on a, namely C_a, but applied to the result $x' = x \bullet t\ (y)$ of applying the transform. According to the notation introduced in section 6.3, this constraint may be expressed as

$$\Gamma_a\ [x' \bullet \alpha_1(z),\ x' \bullet \alpha_2(z),\ \cdots\ x' \bullet \alpha_{n_f}(z)]$$

This condition applies to x', not x. It can be transformed into a condition on x, however, by using the "effects" defined for t and the attributes in *Attrib* (Γ). The condition will be:

$x \bullet \mathit{derived\_constraint}_{t,a}\ (y, z) =$

$$\Gamma_a\ [x \bullet E_{t,\alpha_1}[y, z],\ x \bullet E_{t,\alpha_2}[y, z],\ \cdots\ x \bullet E_{t,\alpha_{n_f}}[y, z]]$$

The last consistency rule follows from this analysis.

Constraint-Effect Consistency Rule

For any sort X, any transform t on X, any attribute a changed by t, the following must hold:

$$\forall x, y, z,\ \ x \bullet L_{t,a}[y,z] \implies x \bullet R_{t,a}[y,z]$$

where

• $R_{t,a}$ is the condition for $E_{t,a}$ to be defined, and

• $x \bullet L_{t,a}[y,z] \ = \ x \bullet C_{t,a}[y] \ \wedge \ x \bullet \mathit{derived\_constraint}_{t,a}\ (y, z)$

and $x \bullet \mathit{derived\_constraint}_{t,a}\ (y, z)$ is the expression obtained by substituting, for every attribute $\alpha \in \mathit{Attrib}\ (\Gamma_a)$, $x \bullet E_{t,\alpha}[y,z]$ for $x \bullet \alpha(z)$ in the expression Γ_a (...) defining the constraint $x \bullet C_a\ [z]$ on a.

As an example of the application of this rule, let us prove the constraint-effect consistency of *owner* with respect to *take* in the above specification. Their effect clause may be expressed as:

```
s • take (g, f, u, k) • owner (g') =
        if g = g' then u
        else s • owner (g') end if
```

(Recall that the **replace...** form is just an abbreviation).

The condition $s \bullet R_{take,owner}\ [g, f, u, k, g']$ under which the right-hand side is defined follows from the constraint on attribute *owner*:

$$s \bullet R_{take,owner}\ [g, f, u, k, g'] = (g' \neq g \implies s \bullet \mathit{taken}\ (g'))$$

The condition $s \bullet L_{take,owner}\ [g, f, u, k, g']$ under which the left-hand side is defined is of the form

$s \bullet L_{take,\,owner}\ [g,\ f,\ u,\ k,\ g']=$

$\quad s \bullet C_{take}\ [g,\ f,\ u,\ k]$ **and**

$\quad s \bullet L'_{take,\ owner}\ [g,\ f,\ u,\ k,\ g']$

The first operand of the **and** is the condition under which *take* is applicable, namely:

$s \bullet C_{take}\ [g,\ f,\ u,\ k] =$

$\quad (s \bullet \mathit{globfile}\ (g) \bullet \mathit{ftype} = f \bullet \mathit{ftype}$ **and not** $(s \bullet \mathit{taken}\ (s \bullet \mathit{globname}\ (f)))$

$\quad$ **and not** $s \bullet \mathit{taken}\ (g))$

The second operand is the condition under which *owner* is applicable to the result of *take*, namely, given $s' = s \bullet take\ (g,\ f,\ u,\ k)$:

$s \bullet L'_{take,\ owner}\ [g,\ f,\ u,\ k,\ g'] = s' \bullet \mathit{taken}\ (g')$

To expand this condition, we apply the effect $E_{take,\,taken}$ of *take* on *taken*, namely

$s \bullet take\ (g,\ f,\ u,\ k) \bullet \mathit{taken} =$ **replace** $s \bullet \mathit{taken}$ **at** g **with** *true*

and obtain:

$s \bullet L'_{take,\ owner}\ (g,\ f,\ u,\ k,\ g') =$ **if** $g' = g$ **then** *true* **else** $s \bullet \mathit{taken}\ (g')$ **end if**

It follows from this form that the validity of $s \bullet R_{take,\,owner}\ [g,\ f,\ u,\ k,\ g']$ is implied by $s \bullet L'_{take,\,owner}\ [g,\ f,\ u,\ k,\ g']$, and thus by $s \bullet L_{take,\,owner}\ [g,\ f,\ u,\ k,\ g']$ as well.

7 - FROM SPECIFICATION TO DESIGN AND IMPLEMENTATION[5]

Once the specification paragraphs have been completed, it is possible to remain in the same framework when going on to the next stages, design and implementation.

The relative difficulty of producing a complete specification (especially if the consistency proofs are performed seriously) pays off at this point. As should be clear from the outline given below, the existence of an adequate M specification provides strong guidance and help during the design and implementation process.

As in the previous section, we consider the combined specification possibly resulting from merging several descriptions.

7.1 - Design

The first non-specification paragraph is the design paragraph. By "design", we mean here "architecture": the aim of this paragraph is to express the design decisions leading to a decomposition of the software into modules.

Starting from the M specification, such a decision is very easy to express. The whole description is based on the sorts; thus it suffices to distribute the sorts among modules. The functions (attributes, transforms) will automatically follow since each has been attached to one and only one.

Thus a typical design paragraph will have the form:

```
system S design

    module MODULE_1 sorts
        A ;
        B ;   - - etc. (names of sorts of the system)
    end MODULE_1 sorts ;

    module MODULE_2 sorts
        C ;
        D ;
        E ;   - - etc.
    end MODULE_2 sorts ;

    module MODULE_3 sorts
        F ;   - - etc.
    end MODULE_3 sorts ;

end system design ;
```

This decomposition embodies the designer's architectural choices. Note that in a pure object-oriented decomposition à la Simula or Smalltalk there will be exactly one sort per

[5] This section benefited from suggestions by Mike Mansur. It is still in tentative form.

module. In general, however, the designer has some leeway in the assignment of sorts to modules. The main criterion is to minimize the amount of intermodule communication.

7.2 - Imports

In order to evidence such communication, an imports paragraph may be written, that spells out for every module the elements needed from other modules.

We will again rely on an example. Assume the above decomposition: *MODULE_1* is responsible for sorts *A* and *B*, *MODULE_2* for *C*, *D* and *E*, and *MODULE_3* for *F*. Assume that the attributes paragraph for the system defines attributes *attr1*, *attr2* and and *attr3* on *A*, *attr4* on *B*, *attr5* on *C* and *attr6* on *E*, as follows:

```
system S attributes

     on A attributes
          attr1 : B total ; - -  whether "total" or "partial" doesn't matter for this discussion
          attr2 (E) : D total ;
          attr3 : C total ;
          ..............................
     end A  attributes ;
     on B attributes
          attr4 : A total ;
          ..............................
     end B  attributes ;
     on C attributes
          attr5 : D total ;
          ..............................
     end C  attributes ;
     on E attributes
          attr6 : F total ;
          ..............................
     end E  attributes ;
     ...........................................

end system attributes ;
```

MODULE_1 is in charge of sorts *A* and *B*, thus of their attributes *attr1*, *attr2*, *attr3* and *attr4*; because of the second and third, it needs access to sorts *C*, *E* and *D*, managed by *MODULE_2*. Access to a sort does not necessarily mean access to the functions on that sort; the most restricted kind of access just implies the ability to name elements of the sort as arguments or results of a function (e.g. here elements of sorts *E* and *D* in connection with *attr2*). This type of access is not unlike using a "**limited private**" type from another module in the programming language Ada.

Access to another module's sorts is not, however, the only type of intermodule communication that will be required once we consider not only the attributes but also the invariants, transforms and effects. Assume for example a transform *transf* on A, as follows:

```
system S transforms

        ................................

        on A transforms
              transf (C) total change attr1, attr2 ;
              ..............................................
        end A transforms ;

end system transforms ;
```

The effects of transform *transf* on attributes *attr1* and *attr2* are described in the effects paragraph:

```
system S effects
        declare a : A, c : C, e : E, ... ;
        .......................................................
        a • transf (c) • attr1 = E_{transf, attr1} [a, c] ;
        a • transf (c) • attr2 (e) = E_{transf, attr2} [a, c, e] ;
        .......................................................
end system effects ;
```

To define these effects, the expressions $E_{transf, attr1}$ and $E_{transf, attr2}$ may need to refer to attributes of objects c and, in the latter case, e, for example *attr5* and *attr6*. *MODULE_1* is in charge of *transf*, a transform on sort A, and is thus responsible for its effects as well. In terms of information flow, this means that *MODULE_1* must have access not only to sorts C,D,E and F, but also to attributes *attr5* and *attr6*.

In the same fashion, the invariants pertaining to a certain sort may involve other sorts and attributes and thus imply inter-module communication.

The imports paragraph is used to describe these access requirements. In the example, it will have the form:

```
system S imports
        on MODULE_1 imports
              from MODULE_2 use C, D, E, attr5, ..... ;
              from MODULE_3 use F, attr6, ..... ;
              ..........................................
        end M1 imports

        ..........................................
end system imports ;
```

There is no new information in the imports paragraph: it is a combined consequence of the design paragraph and of the previous specification paragraphs. Thus the TM tools should be able to synthesize the "imports". In the absence of such tools, however, it may be useful to write this paragraph by hand since it gives interesting information on the structure of the software.

7.3 - Implementation

The next step is to go to implementation[6]. Here the method may help in several ways.

The first application is the representation of data structures. A possible policy is to represent every sort by the cartesian product of its simple attributes; in terms of the discussion in section 1.1, this means going from an implicit to an explicit definition once the list of attributes is frozen. (This list may result from combining several specifications if the modular facilities of M have been used).

Take for example the *POINT* definition of section 1.1, with attributes rephrased here in the LM notation:

```
system POINTS attributes

    on POINT attributes
        x : REAL total ;
        y : REAL total ;
        z : REAL total ;
        speed : VECTOR total ;
    end POINT  attributes ;

    ......................................

end system attributes ;
```

If we decide that this list of attributes is complete, then we can proceed to the implementation of *POINT*s as records:

```
type POINT =
    record
        x, y, z : real;
        speed : VECTOR
    end
```

If the structure of the simple attribute definitions is directly or indirectly recursive (e.g. there is an attribute on A with values in B, and an attribute on B with values in A), then pointers must be used. Thus the implementation paragraph will contain sections of the following form, assuming the above example (where *attr1* and *attr3* are attributes on A, yielding results in sorts B and C respectively, with recursion in the first case):

[6] The step called here "implementation" will result in a program which one may want to write in a language ("PDL" or "pseudocode") different from the programming language used for the final coding. In such a case what we call "implementation" is really the software lifecycle step known as "detailed design"; a straightforward translation step is needed to produce the executable program.

```
system S implementation

      implement A as record ;
      implement attr3 as C field ;
      implement attr1 as B pointer ;
      - - .... more (see below)

end system implementation ;
```

Implementation clauses may also be written for non-simple attributes (those with arguments) and for transforms. Let us first study the latter case. Transforms will be implemented as procedures with side-effects on their first parameters, corresponding to the sort "on" which each transform is defined. Here the specification provides guidance in the form of a precise pre-and post-condition. Assume as previously a transform ***transf*** on sort *A*, defined in the corresponding paragraph as

transf (C) **total change** *attr1, attr2 ;*

The implementation part will then contain a clause of the form

```
implement transf as
      procedure
            (a : in out A ;
            c : in C)
      pre
            $Constr_{transf}$ and $INV_{A,transf}$
      post
            $Eff_{A,transf}$ and $INV_{A,transf}$
```

In this notation, $Constr_{transf}$, $INV_{A,transf}$ *and* $Eff_{A,transf}$ are boolean-valued expressions (predicates) involving *a* and *c*: $Constr_{transf}$ *is deduced from the constraint on transf* in the constraints paragraph; $INV_{A,transf}$ is the conjunction of all the invariants that involve one or more of the attributes on *A* that may be changed by the transform (***attr1*** and ***attr2*** in our example); and $Eff_{A,transf}$ is the conjunction of the relevant effects as defined in the effects paragraph.

Thus the specification yields a very strict framework for building the various procedures involved: the role of each procedure is precisely defined as a precondition-postcondition pair. All that remains to be done is to write a procedure body that will satisfy this pair (of course this may still require significant work).

Non-simple attributes (those with arguments) and will usually be implemented as "functions" in programming language terminology, i.e. value-returning procedures with no side-effects. Thus for attribute ***attr2*** on *A* in the above example, i.e.

attr2 (E) : D ;

the implementation paragraph will contain

```
implement attr2 as
      function (a : in A ; e : in E) return D
pre
      $Constr_{attr2}$ and $INV_E$ and $INV_{A,attr2}$
post
      $INV_D$
```

where *INV_E* is the conjunction of all invariants on sort E and *INV_D* is the conjunction of all invariants on sort D, which the value returned by the procedure must satisfy.

8 - ON USING THE M METHOD

We now give some general guidelines that should be helpful to writers of M specifications[7], based in part on the experience gained through the modest (but non-zero) number of non-trivial system specifications that have been written up to now in M.

8.1 - General form of a specification

As any (non-solitary) worker on formal specifications knows, any such specification usually seem crystal clear to the person who has written it, and hopelessly obscure to anyone else. Readability should thus be a basic concern. This is all the more important with a new method such as M: many readers of a specification can be expected to have trouble both with the notation and with the object domain of the specification (the real system being described).

The LM comment convention is the Ada one (a comment begins with two consecutive hyphens and extends over the rest of the line). Comments are useful for explaining local details of a specification; they usually do not suffice, however, to make a complete specification really understandable.

Thus when preparing a specification for human readers (as opposed to automatic analysis tools, referred above as TM), it is in generally advisable to present it as an **article**, with French language explanations[8] forming the bulk of the presentation and the formal material appearing as inserts. Such inserts may be boxed, as in section 4 above; another acceptable solution is to write formal elements on odd-numbered pages, with even-numbered pages serving as a running commentary in natural language. The natural language text should serve both to comment on the object domain (the system being described) and to explain how the M specification deals with it.

8.2 - Incremental description

When trying to describe a system, either new or existing, one is often overwhelmed by the amount of detail to be taken into account. The advice here is not to panic, but to focus on the basic features first (like those that could be included in a beginner's manual for the system at hand), then add more and more features in an incremental fashion. The modular features of the M method should help to make this a smooth process.

If you are specifying an existing system that you know well, you should design the overall structure of the specification beforehand. In other words, you should plan the specification as a set of "systems" in the M sense, each corresponding to a level of abstraction in the description of the real system being modeled. It is usually a good idea in this case to start out the specification by writing the **interface** paragraphs of the successive M systems.

Sometimes, the informal documentation associated with a system may already distinguish between levels of abstraction, thus easing the task of structuring the M specification. This is the case with such examples as the 7-layered ISO model for open interconnection of computer systems, the ACM Core graphics library, etc.

[7] Some of these rules obviously apply to other specifications methods as well.

[8] English may in some cases be acceptable, as evidenced by this paper.

8.3 - Completeness

A question often heard about specifications is "When do we know we have written everything of significance?". There is obviously no general answer to this question, since completeness of a specification could only be defined with respect to a formal list of the system's functions, and that is precisely what the specification is about.

Some guidelines can be given, however. For example, although it is hard be sure that no attribute or transform has been omitted, the method implies checking each transform and each attribute on a given sort to determine whether the transform may change the value of the attribute: this is an incentive to perform a systematic review of possible combinations. In particular, one may see if there is any attribute not changed by any transform (not necessarily an error, especially at an early stage in the specification process, but still definitely something to look at).

The method also guarantees that once a transform has been declared to change an attribute, the corresponding effect has to be included.

Finally, although one cannot guarantee that all relevant invariants have been included, performing some of the proofs associated with the method will often reveal missing invariants. This is part of our next topic, proofs.

8.4 - Proofs

The ability to prove properties of the specification is an essential feature of formal methods. We have presented in section 6 the required proofs in M. Performing all these proofs by hand is difficult. In the absence of adequate tools, it is still recommended to do as many proofs as possible; this process reveals much about the system and will more often than not lead to the discovery of errors or missing elements, as was the case with the example discussed in this presentation (see the footnote in section 6.4).

8.5 - Attributes versus Transforms

The reader may have noted that the definitions given of attributes and transforms are not exclusive. We defined an attribute on a sort X as being, mathematically, a function

$$f : X \times U_1 \times U_2 \times \cdots \times U_m \longrightarrow Y$$

whereas a transform on the same sort is a function

$$f : X \times V_1 \times V_2 \cdots \times V_n \longrightarrow X$$

Nothing in the first definition precludes Y from being the same sort as X, so that any transform may also be described as an attribute (the reverse, however, is not true in general). Thus one may hesitate in some cases.

There is, however, a strong criterion for transforms which should help dispel the hesitation in any particular case. A function may only be defined as a transform on X if one is able to list precisely the attributes of X that this transform may change; although the exact way in which each of these attributes is affected will only be given later (in the effects paragraph), one must still be prepared to spell it out in full detail. If such a complete specification of the function's effect cannot be given, then the function is an attribute, not a transform.

9 - FURTHER WORK

As will be clear from this paper, there remains a lot of work to do on M. We list below our main current focuses of attention.

9.1 - Concrete Syntax

The concrete syntax of the LM notation clearly needs some polishing. The language must be defined more extensively. Some syntactic sugar is needed; for example, it should be possible to define functions (attributes or transforms) with an infix syntax. Also, it may be useful to define functional and relational operators (composition, transitive closure and the like) to avoid the current restriction to low-level expressions of first-order predicate calculus in invariants, effects and constraints.

So far we have steadfastly resisted the temptation to add nice but non-essential syntactic features, and plan to do so until the dust has settled on the fundamentals concepts.

It should be noted that the M method is, to a certain extent, independent from the particular notation (LM) presented in this paper. Other choices of specification languages could still be compatible with the basic principles of M.

9.2 - Completeness of the notation

More important than syntactic extensions is the problem of whether all the facilities needed to describe actual systems are present - in some form.

One construct, not used in the example of this paper, is most likely to be needed: a constructor of the form

some x **in** X **where** $x \bullet E$ **end**

where X is a sort and E a boolean-valued expression, possibly involving attributes. Such an expression denotes an element of the sort satisfying the given condition. Note that in accordance with the "implicitness" essential to the M approach, one only specifies those properties of x that are needed.

An important problem that needs further theoretical investigation is the intrinsic power of the basic M semantic device: spelling out the effects of every transform on every attribute it may change. There may be a need for more partial characterizations of a transform's effect (by properties resembling invariants, but involving transforms as well as attributes).

9.3 - Initialization

The formalism lacks a notion of system initialization. In particular, the consistency proofs (section 6) should include not only invariance proofs as given, but also proofs that the "initial" elements (those given in the **has** clauses of the sort definitions) satisfy the invariants. There is probably a need for an "initial" paragraph, describing properties of these initial objects; properties such as the invariants called j_1 to j_5 in the invariants paragraph of our example (section 4.4), which are quite different in nature from the other invariants, would belong there.

9.4 - Errors and partial functions

We think that partial functions are the right mathematical tool for dealing with computations that may not always produce a normal result. However, the treatment of abnormal cases and the notion of doppelgänger function must be clarified.

9.5 - Tools

An essential aspect in making M (and other formal methods) practical is the need for tools. We hope to be able to base an support system for M (TM) on two sets of software engineering tools currently being developed:

- **Cépage**, a general-purpose screen-oriented structural editor [11], which is easily adaptable to any new language, whether a programming language or a specification language like LM;
- the **Software Knowledge Base**, a system for configuration and project management, which keeps track of the entities in a software project (called "atoms"), the relations between these entities [13], and the constraints that must be satisfied by atoms and relations.

9.6 - Theoretical Basis

More theoretical work is clearly needed. The position of "M theoretician-in-residence" is open.

References

1. Jean-Raymond Abrial, "The Specification Language Z: Syntax and "Semantics"," *Oxford University Computing Laboratory, Programming Research Group*, Oxford, April 1980.
2. Mack W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 60-68, January 1977.
3. H. Barringer, J. H. Cheng, and Cliff B. Jones, "A Logic Covering Undefinedness in Program Proofs," *Acta Informatica*, vol. 21, no. 3, pp. 251-269, October 1984.
4. Rod M. Burstall and Joe A. Goguen, "Putting Theories Together to Make Specifications," in *Proceedings of 5th International Joint Conference on Artificial Intelligence*, pp. 1045-1058, Cambridge (Mass.), 1977.
5. Rod M. Burstall and Joe A. Goguen, "The Semantics of Clear, a Specification Language," in *Proceedings of Advanced Course on Abstract Software Specifications*, pp. 292-332, Springer Lecture Notes on Computer Science, 86, Copenhagen (Denmark), 1980.
6. Rod M. Burstall and Joe A. Goguen, "An Informal Introduction to Specifications using Clear," in *The Correctness Problem in Computer Science*, ed. R. S. Boyer and JJ. S. Moore, pp. 185-213, Springer-Verlag, New York, 1981.
7. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," in *Current Trends in Programming Methodology, Volume 4*, ed. Raymond T. Yeh, pp. 80-149, Prentice-Hall, Englewood Cliffs (New Jersey), 1978.
8. John V. Guttag and Jim J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, pp. 27-52, 1978.
9. Cliff B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs (New-Jersey), 1980.
10. R. Locasso, John Scheid, Val Schorre, and Paul R. Eggert, "The Ina Jo Specification Language Reference Manual," Technical Report TM-(L)-/6021/001/00, System Development Corporation, Santa Monica (Ca.), June 1980.
11. Bertrand Meyer and Jean-Marc Nerson, "A Visual and Structural Editor," Technical Report TRCS84-03, Computer Science Department, University of California, Santa Barbara, March 1984.
12. Bertrand Meyer, "Modularity," Course Notes for CS 272, University of California, Computer Science Department, January 1985. (A chapter of a planned book on "Applied Programming Methodology").
13. Bertrand Meyer, "The Software Knowledge Base," in *8th International Conference on Software Engineering*, London, August 1985. To appear.
14. Carroll Morgan and Bernard Sufrin, "Specification of the UNIX File System," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 2, pp. 128-142, March 1984.
15. David R. Musser, "Abstract Data Type Specification in the AFFIRM system," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pp. 24-32, January 1980.
16. L. Robinson and Olivier Roubine, *Special Reference Manual*, Stanford Research Institute, 1980.
17. D.T. Sannella, "A Set-Theoretic Semantics for Clear," *Acta Informatica*, vol. 21, no. 5, pp. 443-472, December 1984.
18. Bernard Sufrin, "Formal Specification: Notation and Examples," in *Tools and Notations for Program Construction*, ed. D. Neel, pp. 27-53, Cambridge University Press, 1982.
19. Bernard Sufrin, "Formal Specification of a Display-Oriented Text Editor," *Science of Computer Programming*, vol. 1, no. 2, May 1982.

20. Daniel Teichroew and Ernest A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 16-33, January 1977.