85/857

Sc N 85/1072 A

Université de Nancy I — UER de Mathématiques

ETAPES
SUR LE CHEMIN
du
GENIE LOGICIEL

UNIVERSITE * BIBLIOTHEQUE-SCIENCES * NANCY

∞∞∞∞∞∞∞∞

Thèse présentée à l'Université de Nancy I

pour l'obtention du grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

(*Mention : Informatique*)

par

Bertrand Meyer

∞∞∞∞∞∞∞∞

Thèse soutenue le 9 septembre 1985
devant la Commission d'Examen,
composée de MM. :

Claude **Pair** (*Président*),

Alain **Bossavit**,
Jean-Pierre **Finance**,
Jean-François **Perrot** (*Rapporteurs*)

Jean-Raymond **Abrial**,
Jean-Claude **Derniame**,
Michel **Galinier**,
Gilles **Kahn** (*Examinateurs*).

TOME 2

☞ *Université de Nancy I* *UER de Mathématiques* ☜

ETAPES
SUR LE CHEMIN
du
GENIE LOGICIEL

∞∞∞∞∞∞∞∞∞

Thèse présentée à l'Université de Nancy I

pour l'obtention du grade de

DOCTEUR ES SCIENCES MATHEMATIQUES

(*Mention : Informatique*)

par

Bertrand Meyer

∞∞∞∞∞∞∞∞∞

Thèse soutenue le 9 septembre 1985
devant la Commission d'Examen,
composée de MM. :

Claude **Pair** (*Président*),

Alain **Bossavit**,
Jean-Pierre **Finance**,
Jean-François **Perrot** (*Rapporteurs*)

Jean-Raymond **Abrial**,
Jean-Claude **Derniame**,
Michel **Galinier**,
Gilles **Kahn** (*Examinateurs*).

TOME 2

[82a]

INTEGRATED INTERACTIVE COMPUTING SYSTEMS
P. Degano and E. Sandewall, editors
North-Holland Publishing Company / © IFIP, 1983

TOWARDS A TWO-DIMENSIONAL PROGRAMMING ENVIRONMENT

Bertrand Meyer
Electricité de France, Direction des Etudes et Recherches
Service Informatique et Mathématiques Appliquée
1 avenue du Général de Gaulle 92141 Clamart
France

ABSTRACT

The use of modern video display terminals for communication with a computer has a profound effect on the nature of the resulting dialogs. Screen-oriented interactive programs require a new set of tools, techniques and methods. We report on studies on these topics performed in a computing environment based on standard commercial hardware. The paper describes some of the tools which we have used and the ones we have designed ; it then discusses the methodological issues involved in designing two-dimensional dialogs, and shows the kind of program modularity which is required in this framework. Object-oriented programming appears to provide the right basis ; we have applied this methodology using the class concept of the Simula 67 language and the associated prefixing mechanism.

1 - INTRODUCTION

Interactive facilities play an ever increasing part in all the application areas of computers. Today, this evolution does not only imply that the traditional "batch" mode of submitting programs to computers yields more and more to conversational execution ; it also impacts the very form of such executions : whereas dialogs on typewriter-like terminals and the first CRT devices would proceed in a "line by line" fashion, current terminal technology makes it possible to use the full contents of a screen as the basic unit of communication with the computer, giving rise to the so-called "full-screen" or "full-page" mode of interaction.

One of the best-known applications of this technique is the preparation of documents on a computer using one of the "full-screen editors" now available on many computer systems, most notably mainframes and word-processing systems. Users of such tools unanimously appreciate their power and ease of use, to the extent that going back to a line-oriented editor is resented as a painful experience. Full-screen facilities also find applications in many other domains ; examples are software development and maintenance aids, application programs designed to be used by non-specialist users under the guidance of successive "menus", business data processing (where many "transactional systems" are being developed) and Computer-Aided Instruction. In these and many other areas, programmers in ever growing numbers would like to be able to provide full-screen dialogs for the execution of their own programs.

The construction of such dialogs implies that the texts to be exchanged between the programs and their users are two-dimensional ; this requirement adds a new set of difficulties to the general problems of conversational programming, which are themselves far from being fully mastered (in particular as regards the human engineering, or ergonomic, aspect of dialogs). This paper studies some of these problems, and describes some of the solutions which have been implemented at the Direction des Etudes et Recherches of Electricité de France (EDF), laying the basis for what may be called a two-dimensional programming environment. The discussion focuses on three of the basic issues of software engineering, as applied to two-dimensional interactive programming : tools, methods and languages. The ergonomics of dialog systems, which is another important topic, is touched upon only briefly.

In some respects, it may be felt that the discussion below lags behind the current "frontier" technology in hardware and software. In particular, we limit ourselves to the manipulation of text objects, even though considerable experience has been gained in recent years in two neighbouring domains, namely graphics systems and Computer-Aided Design, where more complicated visual objects are processed. It is clear, on the other hand, that some research laboratories have developed two-dimensional environments which are more sophisticated than the one described here ; two examples worth noting are the set of tools built around LISP /14, 15/ and the Xerox PARC SMALLTALK system /3/, which utilizes special-purpose terminals and a dedicated operating system.

On the other hand, the tools which are described in this paper do not appear to be so commonly available in the most widely used environments, whether in industry or universities ; neither do the underlying ideas. It is quite interesting in this respect to study two recent papers in the Communications of the ACM on the subject of interactive programming /4, 10/ ; although quite different from one another, they both discuss how successive questions should be asked from users, how mnemonics and keywords should be designed, how errors should be dealt with, etc. ; both implicitly assume that the dialog considered proceeds in a completely sequential, line-by-line fashion, without even considering that there may exist other cases. Much of the discussion in these papers becomes pointless when one goes to a two-dimensional environment.

Furthermore, an important characteristics of the tools described below should be emphasized, namely the fact that they were developed and are being used in a standard "production" environment rather than in a computer science laboratory. The computing center at the Direction des Etudes et Recherches of EDF is based on IBM hardware (3081, 3033, 370-168, 4341, etc.) under the MVS-SP operating system. The time-sharing system is TSO ; full-screen terminals are of the IBM 3270 or compatible series ; most of them are 3278, 3279-2B and 3279-3B models (the latter having seven colors, semi-graphic possibilities and various other options). Most application programs are written in Fortran. This environment (which also includes a Cray-1 and many other computers) is quite representative of many large classical computing centers.

2 - THE CHARACTERISTICS OF TWO-DIMENSIONAL DIALOGS

The usefulness of two-dimensional dialogs stems from the combination of three properties :

- The second dimension as such, which provides the program user with an overview of a full page of text, rather than just a single line ;
- The use of a page as unit of communication with the computer, which allows the user to design first an overall sketch and then look back on his decisions, correct errors, reverse some choices, before he sends a page of information to the system ;
- The default facility, which makes it possible for the program to fill some zones where user response is expected by predetermined values, so that the user will only have to write the answers if they are different from these values, but not if the questions are unneeded in his particular case, or call for the same answer as in the previous use of the system (one of the criticisms heard most frequently from users of non-page-oriented interactive programs is that one must answer a whole bunch of seemingly useless questions every time one starts using the system).

It should be noted here that a good page-mode interactive program should keep a profile of every user, so that the default answer suggested for each question will be the one chosen by the user during the last execution of the program, rather than a fixed value assumed to suit all users.

Below is an example of a full-page dialog. It is extracted from the FORTRAN command procedure in our AL library (see section 3) and shows the first three screens to be filled when running a Fortran program : the user types in the names of the files containing source and object code, the destination of printouts, the compiling options, the libraries used, etc. It is easy to imagine how many successive questions would have to be answered in an equivalent line-by-line dialog ; most answers would be indentical from one use of the procedure to the next. If full-screen is not available, the designer of such a dialog constantly faces the contradictory demands of two categories of users : the sophisticated ones, who would like to use many advanced features and thus request many options, i.e. many questions ; and the more numerous "vulgar" users, who use standard options and want short dialog sessions.

Worth noting is the presence of an option called "same as last time" which allows the user, from then on, to remain entirely silent, and directs the system not to ask any more questions. This option is particularly useful in a repetitive task such as the test of a given module.

```
HELLO BERTRAND
WELCOME TO THE AL FORTRAN EXECUTION SYSTEM

PLEASE CHECK THE APPROPRIATE BOX :

     SAME AS LAST TIME                          ===) / /

     COMPILATION, LINK-EDIT, EXECUTION          ===) / /

     LINK-EDIT, EXECUTION                       ===) / /

     EXECUTION                                  ===) / /
```

```
                    COMPILATION, FORTRAN IV EXTENDED

NAME OF THE FILE CONTAINING SOURCE CODE             ===) tryit.fort(first)

COMPILATION LISTING DESTINATION                     ===) prt
     (TER, PRT, LOC, DMY, SYS=x or file name)

CLASS (only if SYS=C, R, S or U)                    ===)

NAME OF THE FILE FOR OBJECT CODE                    ===) tryit.obj(first)

COMPILER OPTIONS :
                         OPTIMIZATION LEVEL         ===) 2
                         GENERATED CODE LIST        ===) no
```



```
                    --- COMPILATION WAS OK ---

                          LINK-EDIT

NAME OF THE FILE CONTAINING OBJECT CODE ===) tryit.obj(first)

LIBRARIES TO BE INCLUDED
     You may request a library by giving either :
          - a keyword (FORTLIB, GENERALE, IMSL, LINPACK, BENSON, ATELBIB...)
          - the actual name of a file containing the library in load module
            form.

     ===) fortlib

     ===) 'edf.myownlib.load'

     ===) 'edf.peterslib.load'

     ===)

     ===)

     ===)

     ===)

     ===) generale
```

It may be said without overstating the argument that, for the programmer who writes systems having this kind of interaction with their users, the leap from traditional, line-by-line conversational programs to page-oriented ones is as big as the leap from non-interactive "batch" programming to line-oriented interactive programming. The new discipline may (perhaps emphatically) be called "two-dimensional programming" ; the second, vertical dimension introduced by screen dialogs raises many important issues with respect to the methods, techniques and tools of interactivity.

3 - COMMAND PROCEDURES : THE DIALOG MANAGER AND THE AL LIBRARY

The first tool which is available to our users is one which is distributed by the manufacturer. IBM has recently released /8/ a new version of SPF (System Productivity Facility, previously known as Structured Programming Facility), a subsystem of TSO, the basic interactive system under MVS. The main characteristics of SPF, which make it rather nice to use for such functions as text editing or file management, are the following :

- the use of two-dimensional dialogs ;
- the presence of "user profiles" which keep useful information from one interactive session to the next ;
- a particular technique for error processing.

The main improvement brought about by the new version of SPF is the set of functions called the "Dialog Manager" /9/. Thanks to this facility, any programmer writing command procedures in the command language of TSO may use some of the internal tools and techniques of SPF, thus being able to take full advantage of the three properties mentioned above.

The dialog manager may be called through special functions which have been added to the TSO command language. It is not, however, easy to use for novice or occasional users ; neither is it readily interfaced with application programs (in particular those written in Fortran). Its main use in our environment so far has been the implementation of a general-purpose command procedure library, called AL (Atelier Logiciel).

The AL library currently contains some forty procedures which encompass a wide spectrum of tools : access to compilers of the various available languages (Fortran, Cobol, assembly, Algol W, Pascal, Simula 67, Reduce), file manipulation and management, use of specialized programs, access to on-line documentation, etc. Until recently, all were line-oriented conversational procedures, suffering from the drawbacks mentioned above. It is interesting to note that our desire to keep the dialogs simple, and thence to limit the number of available options, had resulted in the proliferation of "customized" versions of the more popular procedures : programmers would copy and modify them, thus hampering our efforts to maintain and improve them.

With the development of two-dimensional versions, these problems have disappeared : we may now afford to include many options, since the user's choices are remembered from one session to the next and he will usually change few of them each time ; no more tedious recoding of the same values is required. During the first use of a procedure, default standard values are pre-filled by the system.

Currently available two-dimensional procedures in AL include Fortran IV (of which the dialog in section 2 was an example), Fortran VS (offering access to the IBM version of Fortran 77), Simula 67, Pascal, Algol W, Cobol, Apothèce (a system for the management of program libraries). The entire library will be progressively adapted.

4 - TOOLS FOR TWO-DIMENSIONAL APPLICATION PROGRAMMING : GESCRAN

Once one has discovered the delights of two-dimensional interactivity, perhaps through the use of SPF and AL, one is often tempted to apply the same techniques to one's own application programs. One available IBM product makes this possible : GDDM (Graphical Data Display Manager /7/), a very powerful tool which also includes semi-graphic facilities. GDDM is also, however, rather complex and heavy, and closely tied to IBM hardware and systems. We thus felt it necessary to design a product which, albeit much less ambitious, would cater for simple uses while remaining rigorous in its definition and more portable.

The result of this effort is a package called Gescran (for "Gestion d'écrans", screen management) /1/. Gescran is a set of Fortran subroutines, designed according to the methodological principles expounded in /13/ ; it allows the programmer to describe and manipulate objects called "screens", to create rectangular "windows" in these screens, to define and change the attributes of these windows (such as associated text, color, brilliance, protection, etc.), and to visualize all or part of a screen on the available terminal. It is important to note that screens and windows are in no way bound to the display hardware : they are purely abstract objects, known to the program solely through a name, which in Fortran is implemented as an integer variable, used internally to contain an address and control flags ; the only operation which may be applied to such a variable is its use as an actual argument in a call to one of the Gescran subroutines. Association with a physical screen occurs only when a visualisation subroutine is called.

Gescran works on the IBM 3270 series of screen terminals, but was designed so as to be adaptable to any terminals offering similar capabilities. The construction and manipulation of the data structures representing screens and their windows are entirely independent from the physical I/O operations.

Among the current developments, we shall mention a study aimed at interfacing Gescran with a graphics package, so that the programmer will have the possibility of describing a Gescran window as graphical and use the graphics package rather than Gescran to manipulate this particular window, provided of course the terminal used provides the corresponding facilities.

5 - COMPUTER-AIDED SCREEN DESIGN : CONSCRAN

An important tool for the efficient use of Gescran, called Conscran, provides a higher-level interface for the design of screens as defined above.

The requirement for Conscran stemmed from a problem which had been met by all Gescran users : before being able to write the sequence of subprogram calls which describes a set of screens and windows, one must design each screen by defining the position of its various windows, the parts they play in the interaction, their contents, color, protection, special features (e.g. blinking, reverse video), etc. Until Conscran became available, the best available technique for this phase was to use a sheet of paper and draw a picture of the screen. Such a medium and method appear rather primitive when compared with the aim pursued.

Conscran relies explicitly on concepts taken from Computer-Aided Design to improve the screen design process. It allows the programmer to perform such design in a two-dimensional interactive fashion : the screens will be "drawn" at the terminal, with all the resulting flexibility ; various designs may be tried, observed, modified. Conscran automatically generates the Fortran subroutine containing the calls to Gescran subroutines which are necessary for the construction of the corresponding screens, thus freeing the programmer from a rather tedious task. Conscran stores the resulting screen designs in a data base, thus allowing for later retrieval and modification. It also generates a paper "map" of the screen, showing the position of the various windows, and a "legend" giving their attributes.

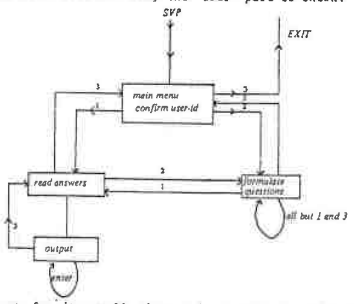
Our current efforts go towards extending Conscran to a system allowing for the design not only of individual screens, but of entire applications as well, using the same underlying principles.

Conscran itself is a two-dimensional interactive program, written in Gescran. Its aim is what may be called "Computer-Aided Screen Design".

6 - THE STRUCTURE OF DIALOG PROGRAMS

Even with the world's best tools, two-dimensional programming raises several difficult issues. One of the most delicate ones is the structure of dialog programs. The behaviour of such programs may usually be quite faithfully modeled by a state transition diagram : one execution of the program will correspond to a path in the associated graph.;

Below is an example of such a graph ; this is one of the applications which we have written with Gescran, the SVP system /5/, which allows users to ask (non-urgent) questions and get answers from the programming assistance service on their terminal. Only the "user" part is shown.



Except for its small size, this example is quite representative of the structure of page-oriented, menu-driven interactive programs. At every step in the execution, associated with one of the states in the diagram, the program outputs a screen ; certain zones are then filled by the user ; after having checked the validity of the answers, the program will perform some action (usually reading or updating a data base). The next step depends on the user's choice, often expressed by his pressing some function key on the terminal. The labels of the edges in the graph correspond to these possible choices.

In a straightforward realization of this scheme, the program for an interactive, menu-driven application will consist of a number of "paragraphs", one per state, each looking somewhat like the following :

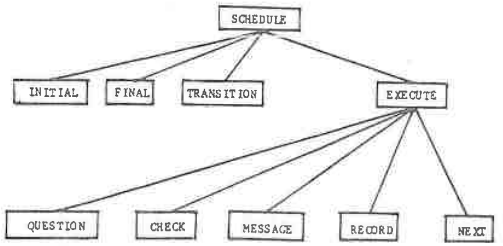
```
state x :
     output screen for state x ;
     repeat
          read user's answers and his choice c for the next step ;
          if error in answer then
               output message
     until no error in answer ;
     record answer ;

     case c in
          c1 : proceed to state x1,
          c2 : proceed to state x2,
          ..............................,
          cn : proceed to state xn
```


Using such a scheme for the actual programming will result in programs with an intricate branching structure, belonging to the well-known "bowl of spaghettis" type. It has been argued /2/ that such a structure should be avoided in the first place, by applying to the state graphs of menu-driven systems such restricting rules as are imposed by modern programming methodology upon the control structures of programs. We think that the analogy is wrong : designing the internal structure of an engineering product such as a program is really not at all the same as designing the external structure of a process involving humans, such as the dialog with a machine. In our opinion, the structural intricacy of the state graph of many interactive systems is an inherent property of these systems, and artificial "structuring" rules are pointless in this domain. The complexity of the graph may stem from various reasons : there may be temporary detours (corresponding e.g. to "help" keys), shortcuts (which were introduced at some point because a user requested, quite legitimately, the possibility to go directly from a certain state to another one, whereas he previously had to backtrack first to the initial menu), and multi-level exits (corresponding to "escape" keys or "quit" commands). Note that these requirements will defeat any effort to implement menu-driven systems by straightforward application of "structured programming" in its naive form.

Some authors have introduced special-purpose control structures to solve this problem ; one example is the language PLAIN /16/, which uses "exceptions" as in Ada, CLU or PL/I. The use of such constructs seems only marginally preferable to that of ordinary jumps.

A much better solution, as it seems to us, is to completely disconnect the description of the overall structure of the dialog, i.e. the traversal of the graph, from the description of what happens at every step, i.e. the operations performed while in a given state. The latter may be treated with ordinary programming constructs ; for the former, the finite automaton, as used in compilation or real-time applications, is a helpful model. It will be quite useful (although not compulsory) to implement the systems in a table-driven fashion, i.e. represent the state transition diagram by a data structure (usually an array) rather than a function subprogram ; using this technique, the changes in the scheduling of states, which are quite common as projects evolve and users request new facilities, will be easy to accomodate.

More precisely, we shall make use of ten program units on three hierarchical levels :



SCHEDULE only defines the traversal of the transition graph ; it knows nothing about the particular screens of a given application, and should be identical for all applications :

```
SCHEDULE :
    var current : STATE, label : CHOICE ;
    current := INITIAL ;
    repeat
        EXECUTE (current, label) ;
        current := TRANSITION (current, label)
    until FINAL (current)
```

TRANSITION is the function which describes the state diagram : TRANSITION (s, l) is the new state reached when leaving state s by the branch labeled l. As mentioned above, TRANSITION may be represented either by a function subprogram or by a two-dimensional array, the latter leading to a more easily adaptable program.

EXECUTE does what is required in a given state : ask the right question, check the answer, perform the necessary action and return the choice c for the next step :

```
EXECUTE (in s : STATE ; out c : CHOICE) :
    var c : CHOICE, a : ANSWER ;
    repeat
        a := QUESTION (s) ;
        correct := CHECK (a, s) ;
        if not correct then
            MESSAGE (a, s)
    until correct ;
    RECORD (a, s) ;
    c := NEXT (a, s)
```

QUESTION, CHECK, MESSAGE, RECORD and NEXT, on the other hand, are application-specific. The call QUESTION (s) will output the screen associated with state s and read the user's answers :

```
QUESTION (in s : STATE) :
    output the screen for state s ;
    read and return the answer a
```

CHECK (a, s) will return true or false depending on whether answer a is acceptable or not in state s ; MESSAGE (a, s) outputs the error message corresponding to answer a in state s, where CHECK (a, s) is false ; RECORD (a, s) records answer a in state s, where CHECK (a, s) is true ; NEXT (a, s) determines from the user's answer a the exit label which was chosen for leaving state s.

It is natural to look for tools which may help in the construction of interactive systems described in the above framework. Some of the "author languages" in Computer-Aided Instruction (CDC's Plato or IBM's IMG for example) pursue similar goals. Can one use the above scheme to build general-purpose tools for helping in the design of interactive, full-screen applications ? As mentioned before, this is our aim in the current extensions to Conscran.

It is soon realized that this scheme cannot reasonably be implemented as presented above if what is sought is a modular, easily extendible system. A simple remark should convince the reader of this impossibility : if procedures such as RECORD, CHECK, MESSAGE, QUESTION or NEXT were to be put in a library, so as to be re-usable for various applications, then a closer look at the above design shows that these procedures must include among their parameters the state (s), but also the precise interactive application to which this state belongs. In other words, any such general-purpose should know about and discriminate amongst all states of all available applications using them ! This is clearly incompatible with any attempt at modularity.

As it is often the case which such problems, a proper solution may be found by going from procedure-oriented to object-oriented programming, i.e. by basing the structure of the program on the main data structures rather than on the functions to be performed. This is the direction that we have taken ; we have been greatly helped in this effort by the availability in our computing center of one of the few generally available modular, object-oriented languages : Simula 67.

7 - USING A MODULAR , OBJECT-ORIENTED LANGUAGE : SIMULA

Simula 67 /6/ appears particularly well-suited for the practical application of the methodological principles introduced above. The main concepts are those which have been emphasized in /12/ : abstract data types, top-down program and data structure design, genericity. Similar techniques could be applied to a descendant of Simula, Smalltalk /3/.

We will only outline part of the system design. In order to implement the above scheme, it is particularly useful to be able to use a structure corresponding to the abstract notion of a "state". The following characteristics are associated with every state s :

- attributes of s : state number, screen to be output when s is reached ;
- operations which may be requested when the system is in state s : QUESTION, CHECK, MESSAGE, RECORD ;
- actions to be performed when s is reached : EXECUTE.

Such characteristics correspond closely to what may be included in the basic program structure of Simula, the class, which is the implementation of an abstract data type : variables representing the attributes of each state, procedures (subprograms) representing the admissible operations, and statements representing the initial actions. One is thus quite naturally led to the design of a class STATE.

A fundamental property of Simula which will be used here is known as class prefixing : a class may be used as "parent" of other classes, which will inherit its characteristics, to which they will add their own refinements. Procedures may be specified at the level of the parent class, their realizations being given in the descendants ; usually these will not be the same in every descendant. Such procedures are declared as virtual in the parent class. Class prefixing and virtual procedures together form one of the best-known systems for the authentic top-down design of both program and data structures. Here they will allow us to define the class STATE with the following structure :

```
class STATE ;
    comment operations :     ;
        virtual :
            ref (answer) procedure QUESTION ;
            boolean procedure CHECK ;
            procedure MESSAGE ;
            procedure RECORD ;
            ref (choice) procedure NEXT ;
        begin
        procedure EXECUTE (c) ; ref (choice) c ;
            begin    boolean correct ;
            correct := false ;
            while not correct do
                begin    ref (answer) a ;
                a := QUESTION ;
                correct := CHECK (a) ;
                if not correct then
                    MESSAGE (a)
                end validation ;
            RECORD (a) ;
            c := NEXT (a)
            end EXECUTE ;
    comment attributes :     ;
            integer screen ;    comment Recall that Gescran uses
                                        integers to denote screens ;
    end STATE
```

Class STATE defines the general properties of a screen. Procedure EXECUTE has now become part of this class ; the same is true for procedures QUESTION, MESSAGE, CHECK, RECORD and NEXT. Note that all these procedures have lost their "STATE" parameter (s in the procedure-oriented version). There is an important difference between EXECUTE and the other five : at the level of class STATE, the latter, while needed, cannot be refined, since their precise implementation may only be known for a given STATE. They are thus defined at the STATE level as "virtual", i.e. only the procedure headings (partial specification) is given. In contrast, procedure EXECUTE is the same for all STATEs ; thus both its heading and body (which uses calls to the five virtuals) may be given at the level of class STATE.

For any given application, there will be a certain number of instances of class STATE, corresponding to the various states of the application. This instantiation concept is readily implemented by the prefixing mechanism :

```
STATE class INITIAL_MENU ; begin ... end ;
STATE class COMPILATION_OPTIONS ; begin ... end ;
etc.
```

The body of each of these subclasses will include the corresponding body for the procedures QUESTION, CHECK, MESSAGE, RECORD and NEXT.

One of the main benefits of this method is that it allows a truly modular construction of interactive applications, the general-purpose and application-dependent parts being programmed separately. All problems pertaining to a certain state (formulation of the question, treatment of errors, recording of answers, etc.) are dealt with in the module (class) for that state, and there only ; on the other hand, the module for a state does not know anything about its connections with the rest of the application's graph. Thus it becomes possible to add or change states, transitions between states etc. without disturbing anything in any module other than the ones associated with the states directly involved in the modification. Apart from its elegance, such a modular, object-oriented programming yields software products on which modifications and extensions are much easier to perform than with programs structured in a more conventional, procedure-oriented fashion.

8 - CONCLUSION

We hope to have shown that the two-dimensional aspect of screen dialogs has important effects on the structure and use of interactive systems. We hope that the ambitious ongoing developments in the area of integrated software environments will take into consideration the key issues which arise in the design of systems for successful communication between man and machine.

BIBLIOGRAPHY

/1/ E. Audin, G. Brisson, B. Meyer : Gescran ; EDF Report, Atelier Logiciel n° 22, December 1980 (version 4, December 1981).

/2/ J.W. Brown : Controlling the Complexity of Menu Networks ; Communications of the ACM, 25, 7, pp. 412-418, July 1982.

/3/ BYTE Magazine : Special issue on SMALLTALK, August 1981.

/4/ B. Dwyer : A User-Friendly Algorithm ; Communications of the ACM, 24, 9, pp. 556-561, September 1981.

/5/ E. de Drouas : Manuel d'Utilisation de SVP ; EDF Report, Atelier Logiciel n° 32, October 1981.

/6/ O. J. Dahl and K. Nygaard : Simula 67 Common Base Language ; Norsk Regnesentral (Norwegian Computing Center), Oslo, 1970.

/7/ IBM : Graphical Data Display Manager - Release 2 ; order no. SC33-0101-1, October 1981.

/8/ IBM : System Productivity Facility for MVS - Program Reference ; order no. SC34-2038-0, December 1980.

/9/ IBM : System Productivity Facility for MVS - Dialog Management Services ; order no. SC34-2036-1, March 1981.

/10/ H. Ledgard, J.A. Whiteside, A. Singer, W. Seymour : The Natural Language of Interactive Systems ; Communications of the ACM, 23, 10, pp. 556-563, October 1980.

/11/ B. Logez, M.-P. Nardy : Conscran, manuel d'utilisation ; EDF report, Atelier logiciel n° 38, 1982.

/12/ B. Meyer : Quelques Concepts des Langages de Programmation modernes, et leur Application à SIMULA 67 ; Bulletin AFCET-GROPLAN n° 9, 1979.

/13/ B. Meyer : Principles of Package Design ; Communications of the ACM, 25, 7, pp. 419-428, July 1982.

/14/ E. Sandewall : Programming in the Interactive Environment : The LISP Experience, ACM Comp. Surv., 10, 1, March 1978, pp. 35-72).

/15/ W. Teitelman : A Display Oriented Programmer's Assistant, in Proc. 5th Int. Jt. Conf. on Artificial Intelligence, Dpt. Comp. Sc., Carnegie-Mellon Univ., Pittsburgh, 1977, pp. 905-915

/16/ T. Wasserman : PLAIN : An Algorithmic Language for Interactive Information Systems ; in Algorithmic Languages, de Bakker and van Vliet (Eds.), North-Holland, 1981, pp. 29-47.

Principles of Package Design

Bertrand Meyer
Electricité de France (EDF)

SUMMARY: Subprogram packages are groups of related subroutines used to extend the available facilities in a programming system. The results of developing several such packages for various applications are presented, with a distinction made between external and internal design criteria—what properties packages should offer to their users and the guidelines designers should follow in order to provide them. An important issue, the design of reusable software, is thus addressed, and the concept of abstract data types proposed as a desirable solution.

1. Introduction

For several years some of us at EDF have been writing software tools of general applicability. The term *Atelier logiciel* (software workshop) has been used to describe our team's activity. The tools which have been constructed and distributed differ widely in their nature and mode of utilization. An important category is that of subprogram packages. A subprogram package is a group of routines which may be called by any program; its purpose is to provide a means of performing tasks in some domain of application which the available programming language does not directly address.

Examples of subroutine packages which we have developed during the past three years include those listed in Figure 1. Working on these packages, we have gained various insights. Our aim here is to convey some of these to other practitioners who may be confronted with similar problems. No breakthrough is claimed; our techniques are mostly standard. We feel, however, that their presentation and a discussion of the software engineering methods used in the design of our packages may be helpful to practicing programmers working in an "industrial" environment.

In Section 2, we describe our environment, a large scientific computing center, and underscore the need for subprogram packages in relation to other kinds of software tools. Section 3 is a detailed discussion of external design criteria, i.e., how packages should appear to the outside world. Section 4 presents our methods for internal design, i.e., implementation to fulfill the requirements of the preceding section; the gist of our approach is that it considers a package the implementation of one or more *abstract data types*. Section 5 concludes with some reflections on the scope of our experience.

Since naming conventions form an important part of our discussion, we have, throughout the text, translated the French words and abbreviations appearing in subprogram names. The package names themselves have been preserved.

2. Why Subprogram Packages

The ideas presented here certainly reflect to some extent the fact that our computing center is geared toward scientific, mostly Fortran programming; and, to a lesser one, that it uses three IBM computers (370–168, 3033, 3081) under MVS, to which a Cray–1 has recently been added.

The first question the reader may ask is why we concentrate on collections of subprograms. Our aim is to extend the range of facilities offered by the existing language. There are at least four other solutions.

(1) convincing users to switch to a better or more powerful language;
(2) writing JCL procedures;
(3) writing conversational procedures;
(4) designing special-purpose preprocessors.

CR Categories and Subject Descriptors: D.2.0 [Software Engineering]: General–*standards*; D.2.2 [Software Engineering]: Tools and Techniques–*modules and interfaces, software libraries, user interfaces*; D.2.7 [Software Engineering]: Distribution and Maintenance–*documentation, extensibility*; D.3.3 [Programming Languages]: Language Constructs–*abstract data types, modules, packages*.
General Terms: Design, Documentation, Languages, Reliability.
Additional Key Words and Phrases: Reusable software, software tool, Fortran.
Author's present address: B. Meyer, Electricité de France (EDF)—Direction des Etudes et Recherches, 1, avenue du Général de Gaulle, 92141 Clamart, France.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
©1982 ACM 0001-0182/82/0700-0419 75¢.

Briefly, we shall discuss why these choices are not always satisfactory.

Solution (1) is certainly the ideal one. However, the sad fact is that most programmers in industry use first-generation languages and are unlikely to try another one. If your aim is to produce tools that will be used, you had best conform to the majority rule. (An even sadder fact, as we shall see in Section 4, is that the tool writer is usually barred from using modern languages because of technical constraints.)

Solutions (2) and (3) (batch or conversational procedures) are adequate for tools intended for "end users", but not for tasks whose execution is initiated by programs.

Solution (4) (preprocessors) may seem attractive but there are many drawbacks involved. One is that it may lead to the proliferation of preprocessors serving various purposes, which will not be, as a rule, mutually compatible. As an example, consider the case of a Fortran programmer who wishes to use the control structures of "structured programming. His programs output results to various graphic devices, and they require that some arrays have dynamic bounds (i.e., the bounds are read on a file before processing begins). Many preprocessors, such as Ratfor [5], are available for the first purpose; others, such as Fortran 3D [11], serve the second one (note, however, that the current release of the latter product uses the subprogram package formula); still others exist for the third requirement. The input languages for these preprocessors will, in general, use wildly different conventions. Their treatment of errors will not be the same. Some of them, in generating Fortran code, will delete comments, while others will recognize comments under a certain predefined syntax as directives. Their combined use will thus be very difficult and, in many cases, impossible.

Preprocessors present another well-known problem. Often simple-minded, they do not provide all the services expected from a well-engineered compiler (cross-references, symbol tables, data flow analysis, useful error messages, source level optimization). They usually have no associated run-time systems, let alone debugging aids. Since they generate code in existing programming languages, they rely on the associated facilities. This makes run-time errors a source of distress: they must be traced back through a program-generated program, which is hardly more readable than the object code produced by a compiler.

Ensorcelé—	free-form input and output
Chronos—	time measurement
Textes—	text manipulation
Axédir—	direct-access file management
Gescran—	full-screen programming
Tri—	internal sorting

Fig. 1. Packages and Their Aims.

Initialization and Termination	
CALL ASKGGE (answer)	May I use full screen? (yes, if answer = 0)
CALL LEAGGE	Leave full-screen mode.
Defining Screens and Creating Windows	
CALL DEFSGE (*ns*)	Define *ns* as the name of a screen.
CALL MXLSGE (*n*)	Set to *n* the maximum number of window lines per screen.
CALL CREWGE (*nw, ns, il, ir, iu, id*)	Create window *nw* in screen *ns* with *il, ir, iu, id* as coordinates.
CALL DELWGE (*nw*)	Delete window *nw*.
CALL BRIWGE (*nw, b*)	Assign brightness *b* to window *nw*.
CALL PROWGE (*nw*)	Make window *nw* protected.
CALL FREWGE (*nw*)	Make window *nw* free (unprotected).
CALL CAPWGE (*nw*)	From now on, convert letters in window *nw* to capitals.
CALL ASIWGE (*nw*)	From now on, leave any character in window *nw* as it stands.
Changing or Examining the Internal Image	
CALL REPWGE (*nw, tabcha*)	Replace contents of window *nw* with *tabcha*.
CALL BLAWGE (*nw*)	Fill window *nw* with blanks.
CALL ASSSGE (*nst, nss*)	Assign value of screen *nss* to screen *nst*.
CALL BLASGE (*ns*)	Fill all unprotected windows of screen *ns* with blanks.
CALL NBCWGE (*nw, n*)	Assign to *nm* the number of changes to window *nw* since the last input operation.
CALL EXAWGE (*nw, tabcha*)	Assign to *tabcha* the current contents of window *nw*.
Input and Output (Affecting the External Image)	
CALL WRISGE (*ns*)	Display screen *ns* on the terminal.
CALL REASGE (*ns*)	Input screen *ns* from the terminal.
Manipulating the Cursor and Function Keys	
CALL POSCGE (*nw, nline, ncol*)	Position the cursor in window *nw*, at position [*nline, ncol*].
CALL EXACGE (*nw, nline, ncol*)	To what position [*nline, ncol*] was the cursor in window *nw*? ([0, 0] if not in window)
CALL EXAKGE (*ns, n*)	Assign to *n* the number of the function key used to send the screen contents.
Typed Input-Output (Interface with Package Ensorcelé)	
CALL UNIOGE (*nw*)	Direct subsequent output to window *nw*.
CALL UNIIGE (*nw*)	Obtain subsequent input from window *nw*.

Fig. 2. Reference Sheet for Gescran.

Additionally, it should be noted that preprocessors only add surface improvements to Fortran. They usually do not provide remedies for this language's intrinsic limitations with regard to data structuring, dynamic allocation, pointer variables, intra and inter-routine type checking, recursion, etc.

Subprogram packages do not suffer from these defects, although, admittedly, they raise other problems which we discuss in the next two sections. To the potential user, they offer a very neat way of enriching the existing programming language with new instructions, implemented as subprogram calls.

3. External Design Criteria

A subprogram package is a collection of mutually related subprograms. Just how they should be "related" to each other will be studied in more detail in Section 4. For the moment, we turn to an important question: How should these subprograms be presented to their potential users? This problem is vital, especially in light of the fact that programmers are often reluctant to invest the effort necessary to learn a new methodology. They will not be lured into using our packages unless some very attractive arguments convince them to do so.

In the following subsections we shall list those desirable qualities which our packages should possess and explain exactly how these design criteria—namely, simplicity; self-restraint; ease of use; homogeneity, safety—were met.

3.1 Overall Simplicity

In the area of simplicity our central thesis was that most programmers would not use a subprogram package if it required constant reliance on a reference manual. Although we did insist that users read part of the manual at least once, our ideal was that they should then be able to employ a package for standard applications without further reference to any written document. In practice, we have not succeeded in reaching this goal completely, but we have nevertheless succeeded in concentrating all the necessary information for normal use of a package on a single page. This we consider a mandatory requirement. For an example, see the reference sheet for the package Gescran as outlined in Figure 2.

The most important aspect of our approach is that we do not try to write complex packages providing a wide range of services and satisfying all users' fantasies. Instead, we concentrate on a careful study of user needs and strive to offer simple and efficient answers to the most important of them. Of course, deciding which issues are the most important is a design decision since often user needs are either unexpressed or, if expressed, require much work to be transformed into realistic specifications.

3.2 Self-Restraint

Our subprograms are called by other programs or subprograms: they are not directly concerned with solving "interesting" problems, but rather with performing general utility tasks. The application program/subprogram package/system hierarchy is pictured in Figure 3; other levels may, of course, exist. We shall refer to the programs which call our subprograms as *application programs*; on the other hand, *users* will be those individuals (or programs) who (which) run application programs. (These terms, especially the latter, are two of the most misused in data processing; we shall strive to use them precisely.)

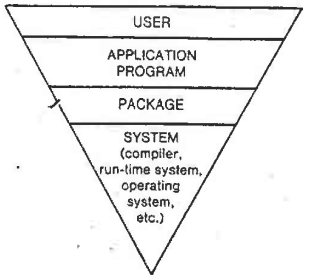


Fig. 3. Hierarchy of Programs and Program Users.

Self-restraint is necessary because there is at least one level, that of an application program, between users and our subprograms. The latter must thus be as invisible to users as possible. This is especially important in connection with errors (Section 3.5).

3.3 Ease of Use

Documentation

Documentation is organized in terms of simplicity, ease of use, and homogeneity. All packages are documented by manuals with the same structure, as shown in Figure 4.

Order of Arguments

One key to ease of use is consistency of design. This criterion becomes even more crucial as new packages are employed and the number of available subprograms grows. It requires that a set of regular, coherent conventions be strictly observed for all distributed products.

Foreword ("How to Use This Manual")
Section 1—Introduction
2—Individual Subprogram Description
3—Restrictions and Caveat
4—Examples
5—Notions on the Implementation
Appendix A—Error Messages
B—List of External Names
C—Portability
D—Performance
E—Control and Data Flow Graph
F—Quick Reference List (last page)

Fig. 4. Structure of the Manuals.

An important area requiring a homogeneous policy is parameter order. In a language environment not providing for key word parameter transmission, actual arguments to any subprogram must be given in a fixed order, which matches that of formal parameters for the subprogram. Package users must thus know this order; such a constraint often becomes a source of annoyance and errors. It is therefore desirable that the package designer adhere to some convention.

For example, in the Textes package, which allows character string manipulation using pseudo-string variables that appear to the compiler as integer variables (see Section 4.6), the syntax of some typical calls would be what is seen in the box below. *itext*, *jtext*, and *ktext* are pseudo-variables, *i* and *j* integers.

The order of arguments decided on here was the following: in assignments the destination should always precede the source. This is consistent with the syntax of most programming languages:

$$A := f(B, C, \ldots)$$

Moreover, since the package's aim is to provide the equivalent of a "string" data type as it exists in, say, PL/I, the chosen order aims to imitate the syntax of languages which do offer operations on this type; e.g., CALL CNCTTX (*itext*, *jtext*, *ktext*) follows the PL/I pattern *itext* = *jtext* || *ktext*.

The rule of consistency in the order of arguments may conflict with other, equally important criteria concerning the homogeneity of design. For example, in the Axédit package for direct-access file management, there is a read routine whose call has the following form

CALL REAFDA (*file-id, target, record-number, error-indicator*)

This conforms to the "destination first" rule, although the file identifier comes before the target for reasons of consistency with the rest of the package. For the write routine, however, we chose the syntax

CALL WRIFDA (*file-id, source, record-number, error-indicator*)

since we thought it would be easy to remember that corresponding arguments occupy exactly the same position in both operations, "target" for read and "source" for write being symmetric. The destination first rule is thus violated by WRIFDA.

3.4 Homogeneity

Number of Arguments (operands and parameters)

The question of arguments involves simplicity as well as homogeneity. Not only should arguments appear in a carefully chosen order, but the number of them should also be small if programmers are to remember the calling sequence.

Of all the subprograms listed in Figure 1, 71 percent have zero, one, or two arguments, and less than 4 percent have more than four (a function result being counted as an argument). The maximum number of arguments is six.

Requiring short argument lists has an immediate consequence: since any means of data transmission between an application program and a package subprogram other than argument passing (such as explicit COMMON block sharing) is banned, every subprogram may perform only a well-defined single task. In our case, this property became another motivation for requiring short argument lists, rather than a consequence of this requirement. It is indeed integral to our design philosophy (see Section 4).

Such an approach has interesting practical consequences which distinguish our packages from many commercially available ones. Let a subprogram, say *f*, be used to implement an operation with a certain number, say *n*, of *operands*. It is often the case that several operating modes are available, described by a certain number, say *m*, of *parameters* or *options*. Quite commonly, *n* is small, but *m* may be large and will grow as users request new refinements.

At this point, the reader may ask for a precise definition of the distinction between parameters and operands. Although the difference is in many cases intuitively obvious, an absolute definition does not exist. Rather, the distinction should be thought of as design decision which the designer bases on the following guidelines:

—The number *n* of operands should remain small.

—The system should be able to set default values for parameters.

—During the package's evolution, as parameters are added (or removed), the specification of operands for any single subprogram must not be changed.

Thus, the distinction between parameters and operands is partly a pledge made by the designer with respect to the future of the package.

There are three ways of specifying a subprogram *f* with both operands and parameters:

CALL CRETTX (*itext*)	(CREate a Text variable) Create a new string variable, of name *itext* (pseudo-declaration).
CALL CNSTTX (*itext*, '*xyz . . .*')	(CoNStant Text) Assign the character string '*xyz . . .*' to the string variable *itext*.
CALL CNCTTX (*itext*, *jtext*, *ktext*)	(CoNCatenate Text) Assign to *itext* the value of *jtext* concatenated to that of *ktext*.
CALL SUBTTX (*itext*, *jtext*, *i*, *l*)	(SUBText) Assign to *itext* the value of the substring of *jtext* starting at position *i*, with *l* characters.

(a) Include all necessary operands and parameters in every subprogram call, as in

CALL $f(opnd_1, \ldots, opnd_n, parm_1, \ldots, parm_m)$.

(b) Include only operands, as in

CALL $f(opnd_1, \ldots, opnd_n)$

and provide other subprograms, one per parameter, to set the values of parameters, in the form

CALL $setval_i(parm_i)$

with the understanding that the *i*th parameter will remain set to the value $parm_i$ until a new call to $setval_i$.

(c) Use a mixed-mode approach, with some parameters included in the calls to *f* and others separately.

Throughout our packages, we adhered to the second approach (b), which we find preferable for two basic reasons:

(1) It allows the package designer to set default values for all parameters, thus freeing the user from providing arguments corresponding to options not of primary concern.

(2) Including parameters in the operation invocation inevitably leads to problems as the package evolves: although operands usually do not change if the initial design is sound, requests for new parameters will appear. We have experienced this phenomenon over and over again. For example, users of Ensorcelé (free-form input and output) requested new facilities for output formatting. To meet their request, we added a "color" parameter to the Gescran subprograms when color displays became available. Had we included parameters in the calls, all the calling programs would have had to be changed, making it very difficult to entice anyone into using our programs afterwards. Thanks to our seemingly drastic policy, we have so far been able to avoid such a situation.

Note that the use of a language allowing subprograms to have both positional and key word arguments (such as Ada) would solve the problems inherent in situation (1), but not (2).

One may object that our technique increases the size and external complexity of packages since there will be one subprogram per parameter per operation. This does not worry us too much because there is not much difference in added complexity between a new subprogram, on the one hand, and a new argument to an existing subprogram, on the other.

Another possible drawback is that application programs will contain many subprogram calls when they require nondefault options. For example, if a user wishes to output a real number *X* in a particular format, using Ensorcelé, the sequence of instructions could be as long as the one listed in the box on this page.

Although such code may seem horrendous to experienced programmers, we find it quite acceptable (and have even come to like it!). It is really very readable since every call has a clearly stated single purpose. Also, remember that parameters remain set until explicitly changed so after initialization, there will usually be fewer calls to the parameter-setting routines (unless, of course, the user program wishes to often change options).

All in all, we feel our strictly functional approach, with a clear distinction between operands and parameters (and between operation and parameter-setting subprograms), is very helpful in the design of coherent, easy-to-use, and simply maintained packages.

CALL SAVPAR	Save the current values of Ensorcelé parameters.
CALL EXPON (5)	Real numbers will be output using the exponent (*E*) format if their absolute values are not in $]10^{-5}, 10^{+5}[$.
CALL BLANKS (3)	Output items will be separated by at least three blanks.
CALL ZONE (9)	Items will be justified to the right in zones of length 9 (or a multiple thereof if they do not fit).
CALL NBRDIG (8)	At least eight significant digits should be printed.
CALL PUTZER	Trailing zeros should be written (default: blanks).
CALL WRIREA (*X*)	Write *X*.
CALL RESPAR	Restore previous parameter values.

External Subprogram Names

An important component of homogeneity as well as the aforementioned criteria of ease of use is how external subprogram names are chosen. This issue is a delicate one (which we had not well understood when we started our work) because of four conflicting requirements:

(1) the desire to provide mnemonic names, as expressive as possible;
(2) the need to avoid possible conflicts with names of subprograms or data segments in the application programs;
(3) the need for a coherent set of naming conventions, which grows with the number of available packages and subprograms (and the size of the programming team);
(4) for subprograms callable from IBM Fortran, the tight 6-character limit.

At the outset, we had, with clarity our goal, concentrated on the first requirement. The reader may have noted names such as BLANKS and ZONE in the Ensorcelé example cited in Section 3.3. Inevitably, this led to conflicts with names chosen by application programmers and we had to adopt a more balanced strategy. All of our current subprograms have 6-character names with the following structure:

—Three letters which are an abbreviation for a "verb" denoting the action to be performed, e.g., REA for read, SET for set;

—One letter indicating the type of object to which the action applies, such as I for integer, C for cursor;

—Two letters which are a code assigned to the package, e.g., GE for Gescran.

Thus, the subprogram positioning the cursor somewhere in Gescran has the name *SETCGE*.

Using this technique, we have been able (with some care) to avoid name clashes. Additionally, the method is simple to explain in the package manuals so the name may be considered mnemonic for the application programmers.

3.5 Safety

Treatment of Errors

An important but difficult issue is that of errors: How should a general-purpose routine react in an error situation?

First, we shall define precisely what an error is in the context of our packages. A subprogram in such a package is intended to complete some *actions* and/or to compute some *values*. An error arises when the subprogram detects that an action cannot be performed or that a requested value does not exist. In either case, it means the subprogram is able to determine the fact that a certain element does not belong to the domain of a certain function (which is part of the subprogram's abstract specification).

The possibility of an error made in writing the subprogram being ruled out, the cause of the error may be either of the following:

—The user has provided illegal arguments to a subprogram.
—Some well-founded request cannot be satisfied because of external conditions (e.g., dynamic memory allocation fails since no more space exists).

What policy should the package writer adopt in regard to such errors? There are two conflicting requirements: safety and self-restraint.

(1) Safety implies that no operation not conforming to the application programmer's intent and, in particular, no modification of the application program's state other than those explicitly provided for in the package's manual should ever be performed. Additionally, the application program must be able to find out about the error and take any corrective action it wishes.

(2) The need for *self-restraint*, on the other hand, stems from the fact that it is very difficult to decide what action to take on the sole basis of what is known to the subprogram (the same situation is experienced by, say, the writer of a lexical analyzer in a compiler). It suggests that the package should be able to make a reasonable correction, without unnecessarily bothering the calling program, let alone causing a system interrupt.

One way to ameliorate the problem of errors is to avoid illegal arguments by enforcing as few restrictions on subprogram calls as possible (which in effect means expanding the specifications to include most "error" cases as peculiar but legal ones). Because of such a policy, we experienced very few error cases in our first packages and were able to adopt a rather haphazard approach to error treatment (see the "error-indicators" in the calls to the Axedir subprograms in Section 3.3).

Recently, we have arrived at the following approach. A small package, called Errare, which is comprised of only three subprograms has been designed:

(1) CALL RECEER (*n*, '*message*'), RECE standing for RECord Error, sets a global error indicator to *n* and outputs the message along with other information, in particular the operating chain (in order to avoid avalanche effects, a shorter text is output whenever *n* is equal to the previous error indicator).

(2) INDEER (0), an integer function with no arguments (a dummy argument is required in Fortran 66), returns the value of the global error indicator (as set by the last call to RECEER; zero if none).

(3) CALL SETUER (*n*), SET Unit, directs subsequent message output performed by RECEER to output unit number *n* (recall that in Fortran, I/O devices are designated by integers between one and 99). If SETUER is not called, error output will be printed on the standard output file.

With these subprograms, a package subprogram takes the following course of action when it detects an error.

—Record the error number and output a message with RECEER.
—If an action was requested, do not do anything.
—If a value should have been computed, then two subcases arise: when a sensible approximation exists, use it as a substitute; otherwise, return a value chosen to be as "out of bounds" as possible (e.g., a negative integer if an address was requested).

This technique seems both self-restrained and safe. It is self-restrained because INDEER is a public function. Thus, if the application programmer wishes to correct errors possibly occurring in a package subprogram, he can do so by testing INDEER after the call; the programmer will thereby remain in full control of all events since the package itself does nothing abnormal except outputting a message. The technique is safe because it guarantees that no illegal action will be performed by the subprogram. On the other hand, if no reasonable value can be computed, the result will be so absurd that it will inevitably lead to program abort shortly after the call unless the application program regains control with INDEER. It is certainly much better to provoke a "negative address" error than to allow the program to work on an erroneous but physically meaningful address.

The use of "abnormal" values, such as negative numbers when an address or array index would have been required, is only possible because of the lack of strong type checking in Fortran. The transposition of this technique to languages with stronger type requirements requires the presence of an *undefined* value in every type. This condition is met by languages like Algol W and Simula 67 in which all programmer-defined types are pointer ones with a special empty value (called *null* or *none*) as one of their elements. No such possibility exists in Pascal or Ada whose record types, for example, do not possess a void value.

One advantage of our method is that the treatment of errors does not interfere with other criteria. In particular, in terms of argument lists, the external specification of package subprograms does not have to be changed. Better general solutions are hard to find, short of an exception facility like those in PL/I or Ada.

3.6 Functions vs. Subroutines

Almost all of our subprograms are subroutines (actions) rather than functions. Using a function may seem preferable in the case of a subprogram returning a single value and having no side-effect; the reader may have wondered while reading about the Textes package (Section 3.3) why we used a subroutine to compute the concatenation of two strings. Indeed, if we want to output the concatenation of *jtext* and *ktext*, we must write what appears in the box above instead of the much more natural

CALL PRNTTX (*fcnttx* (*jtext*, *ktext*))

where *fcnttx* would be a function returning the concatenated string.

We found three objections to using functions.

(1) In many systems, including ours, Fortran functions cannot be called from Cobol programs (whereas subroutines can). Since we do have a few Cobol users, subroutine interfaces must be written anyway.

(2) A function type must be declared in the calling program, except when it is integer or single-precision real and follows the Fortran default rule (which eliminates logical, double precision, and the Fortran 77 character type). This is a source of errors in systems with no checking at link or load time.

(3) An important issue in deciding whether to express the same semantics as $x = f(a, b, \ldots)$ or CALL $f(x, a, b, \ldots)$ is that only the latter construct gives the subprogram writer access to *all* the operands involved, including x, which may be needed in order to make f safer and/or more efficient. Both safety and efficiency were at stake in the choice made for the Textes package. On the one hand, since string operands are integers for the compiler, our subprograms must be able to check whether both sources and target have been correctly pseudo-declared, thus avoiding dangling run-time references. On the other hand, the package uses quite an elaborate memory management algorithm [7] and will save a lot of space when *itext* is the same string variable as *jtext* or when the previous allocation for *itext* is greater than or equal to length (*jtext*) + length (*ktext*).

```
INTEGER itext
.....
CALL CRETTX (itext)        pseudo-declaration of string variable
.....
CALL CNCTTX (itext, jtext, ktext)   assign to itext the concatenated string
                                    output
CALL PRNTTX (itext)
```

In view of these factors, we only use Fortran functions for integer functions giving the value of some attribute of an object. This occurs in the sense of Section 4.2 (that is, an "accessor function" as defined in connection with abstract data types). For example, the length of string *itext* is denoted by *LNGTTX* (*itext*).

4. Internal Design Techniques

4.1 Framework

In the previous section we described our basic aim: to provide our products' potential users (the application programmers) with packages whose external appearance is sound and coherent. The key to success is, of course, that these properties be matched by the stability and consistency of internal design. As Jackson [3] remarked about early attempts to define modular programming, words like "functional integrity" are not very useful as practical design guidelines as long as they remain unsupported by more technical definitions of the methods used. The concept which we have found most fruitful as a design base for sound subroutine packages is abstract data types, a notion now well-established in academic and research circles although practically unheard of by most practicing programmers.

4.2 Abstract Data Types

An abstract data type is the formal definition of a data structure or class of data structures, as characterized by purely functional properties. The definition of an abstract data type T comprises three parts:

— a list of *domain names*, one of which is T;
— a list of *function names* with associated functionalities, i.e., domains of the arguments and results (at least one of these domains must be T for every function); these functions are the abstract representation of the operations available on the type;
— a list of logical *assertions* on these functions, which describe the operations' formal properties.

A definition comprised of these elements is a formal *specification* of the data type.

An *implementation* of an abstract data type is a set of data definitions and subprograms operating on the data defined, such that each datum's type (with the ordinary meaning of the word "type" in programming languages) is associated with one of the domains in the abstract data type's definition. Each subprogram corresponds to one of the functions and satisfies its functionality requirement with respect to input and output arguments. The values of these arguments satisfy the assertions for every call of the subprogram.

Some have argued that a good way, perhaps the best, to construct truly modular programming systems is to organize them as sets of abstract

data type implementations. This claim is supported by practical evidence [13].

It and other reasons explain why we have used abstract data types as the model for our packages. In fact, every one of our packages is the conscious implementation of one or more abstract data types. In particular:

— The Textes package implements the "text" or "string" type with operations like the creation of a constant text, the extraction or modification of the *i*th character, or concatenation.

— The Chronos package implements the "time counter" concept.

— The Axédir package implements the external array type with "initialize," "read," and "write" as operations.

— The Gescran package implements the "page" (or "screen") and "window" abstract data types with operations like "define window in screen," "write into window," or "visualize screen."

It is therefore not surprising that the main design choices we encountered in implementing packages are conveniently expressed in terms of abstract data types. In the following subsections, we study some of the most important, namely: linguistic issues; heirarchical design; static *vs.* dynamic instanciation; information hiding.

4.3 Linguistic Issues

The programming language for writing a package should offer a structure corresponding to the schema just presented. This is indeed the case in many recent languages. Foremost among these, from the practitioner's point of view, are the pioneer, Simula 67 [1, 8], and the youngest, Ada [2]—the former because of its availability on a variety of machines, the latter on account of its intended wide circulation.

These languages, like their relatives (Lis, Clu, Alphard, Euclid, Mesa, Modula), include a program structure ("class" in Simula and "package" in Ada) with three categories of elements: data definitions, subprogram declarations, and statements. Such a structure may be used to implement an abstract data type (or an object of such a type, see Section 4.5); its three components correspond to data representation, operations, and initialization, respectively. Given an instance, A, of a class/package and x as one of its components (subprogram or data element), an external module which is entitled to "use" A may reference x. This is done either with a "dot notation", $A.x$, or directly by its name, x, provided that the external module has "acquired" A in some fashion (**inspect** A in Simula, **use** A in Ada) and there is no name conflict.

This kind of solution is very convenient, both from the package writer and application programmer's point of view. The former designs and implements the package as a single module, separately compilable and verifiable: all the relevant information is concentrated in a single, coherent entity. The application programmer, when requesting a function performed by the module, simply supplies the names of the module and the function.

Unfortunately, it is usually impossible to write subprogram packages in such a language, even if one is available. Although "first-generation" languages like Fortran and Cobol and the assembly languages for most machines are geared toward a very simple, static allocation policy, newer languages (including not only "modular languages" but also PL/I, Algol W, and Pascal) require a much more ambitious memory management scheme, usually with a stack and a heap, the latter being subject to garbage collection. Therefore, even with well-engineered language systems permitting separate compilation and linking with modules written in other languages, the system for the more elaborate language must exercise control at run-time. For most systems, this precludes the use of such a language for writing subprogram packages since the latter must be accessible to any program.

The tool writer is thus placed in a very frustrating situation. He knows the right language(s) in which to write a subprogram, but he remains unable to use it. We, for instance, have a very good Simula system [9] but must resort to Fortran for subprogram packages, with all its drawbacks: no data structure other than the array, no control structure other than the If and Goto, no pointer variables, no dynamically created elements, no parameterized-dimension arrays, no recursion, and, of course, no "class" or "package" structure.

4.4 Hierarchical Design

In order for each element of a package to remain simple and understandable, it is necessary that the package's structure consist of several layers in all but the most trivial cases. For packages seen as implementations of abstract data types, this means such an implementation will use objects belonging to other types, also defined abstractly, i.e., used through their properties rather than representation. Thus, a package is generally implemented as a hierarchy of types. Such a hierarchy is illustrated in Figure 5. Ensorcelé 1 (output) appears as a means for manipulating a *stream* of "printable" objects, which is represented using the concept of unbounded *character string*, itself implemented as a sequence of *lines*.

Out of the many advantages of this approach, two are worth noting. First, it allows the designer to push down all machine- and system-dependent elements to the lowest levels of the hierarchy, thus increasing portability (for example, Gescran was built for the IBM 3270 terminals, but only a few subprograms must be recoded for other similar devices). Second, it lends itself to top-down design, which, as Wirth pointed out [12], should apply to data as well as control.

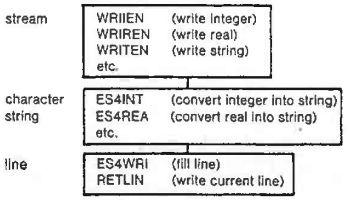


Fig. 5. Hierarchy of Types for Writing (Ensorcelé 1).

4.5 Static vs. Dynamic Allocation

A package implementing an abstract data type may provide one of the following:

(1) one object of the type;
(2) a fixed number of objects of the type;
(3) an unlimited number of objects, within the limits of the available space at execution time.

Solution (1) provides for the implementation of what may be called an "abstract object" rather than a type. It is used, for example, in Ensorcelé which acts on a single stream of objects.

Solution (2) is quite natural in Fortran because of the arrays' static dimensions. For example, one package similar to our Textes in terms of the services offered [10] provides a fixed number of text variables, corresponding to the size of an array in a COMMON block. Of course, this often results in unpleasant repercussions since the limit may appear too large (entailing undue space use) or too small (requiring recompilation of separate versions of the package). We have seldom used this technique; an example is Chronos, which sets an absolute limit of 100 time counters.

Solution (3) comes closest to what is offered in languages providing user-defined nonstatic types. Every object of the type needed in the application program must be explicitly created by it (*new* statement in Simula or Pascal). This is the most powerful solution; its main drawbacks within our framework is that a few non-Fortran (or nonstandard) routines for dynamic memory allocation must be used. Packages like Gescran, Axédir, and Textes provide an unlimited number of instances (character strings in the first, files in the second, "screens" and "windows" in the third).

4.6 Information Hiding

One of the main goals of the abstract data type approach is a clear separation between what is visible to application programmers and what remains private to the package designer. The latter category should include all elements dependent on non-essential hardware, system, implementation, or design peculiarities. We have found two techniques useful in enhancing this property: the careful choice of names and the use of pseudo-variables.

Internal names are chosen so as to seem mnemonic only to team members. Like external names (see section 3.4), they follow a regular pattern and make collisions unlikely.

The notion of a pseudo-variable is more important. In the case of packages offering an unbounded number of type instances, the individual objects must be nameable by the application programs, although Fortran does not offer a declaration other than for standard types. The solution is to declare the objects using names which appear to the compiler as those of integer variables. Actual "declaration" will then be effected by a call to an instantiating subprogram. Such pseudo-variables were used in the Textes example cited in Section 3.3.

Internally, the integer variable will usually contain a pointer to the location assigned to the object and a code allowing package subprograms to check that the variable has not been modified by an illegal operations. Indeed, the only legal kind of operation in which such a pseudo-variable may appear in an application program is parameter transmission. Any other use (e.g., integer addition) is forbidden and will normally be detected in the next call to a subprogram of the package.

This technique seems the best way of adapting abstract data type concept to Fortran: an object is only available through its name and a set of well-delimited operations. The resulting programming style is not, of course, typical of Fortran. In the box below, an example of Gescran programming appears.

5. Conclusion

We believe that the principles expounded upon in this paper may be applied with equal success to widely different kinds of software, and we

```
INTEGER SCREE, WINDO1, WINDO2          Declare pseudo-variables.
.....
CALL DEFSGE (SCREE)                    Pseudo-declaration of SCREE as a
.....                                    screen pseudo-variable.
CALL CREWGE (WINDO1, SCREE, 2, 5, 7, 12)  Pseudo-declaration of WINDO1 and
CALL CREWGE (WINDO2, SCREE, 6, 15, 1, 4)    WINDO2 as windows in screen
                                            SCREE.
.....
CALL REPWGE (WINDO1, string 1)         Initialize contents of windows (RE-
CALL REPWGE (WINDO2, string 2)           Place contents of windows).
.....
CALL BRIWGE (WINDO1, 'B')              Define WINDO1 as bright (BRIlliance
.....                                    of Window).
CALL WRISGE (SCREE)                    Display SCREE.
```

hope that our discussion has shed some light on a key problem in software engineering: how to write reusable software. It should be pointed out, however, that all of our products are conceptually small. This was a deliberate decision on our part since we felt modest-sized team best succeeds with simple, efficient, and reliable programs, rather than large-scale, ambitious ones. Although we feel many of our methods would apply successfully to larger projects, we do recognize that their applicability to, say, a vast numerical library remains to be proved.

Acknowledgments

The work reported on here involved, in particular, E. Audin, G. Brisson, E. de Drouas, and B. Logez. Many others provided advice—most notably, A. Bossavit. At a meeting of the groupe "Génie Logiciel" (Software Enginnering) of AFCET-TTI (French Computer Society), additional useful suggestions were made. The author is also indebted to I. Qualters for many improvements in the style of this paper, and to the reviewers for their comments.

References

1. Dahl, O.J., Myrhaug, B., and Nygaard, K. *Simula 67: Common Base Language.* Rep. S–10, Norsk Regnesentral, Oslo, Norway, 1970. The original description of the first of the modern modular languages. Assumes the Algol 60 report as a prerequisite; an integrated report is currently in preparation.

2. Honeywell, Inc. *The Ada Programming Language—Proposed Standard Document.* U.S. Dept. of Defense, 1980, Washington, D.C. The report on the new U.S. Department of Defense language, designed by a team led by J. Ichbiah.

3. Jackson, M.A. *Principles of Program Design.* Academic Press, London, 1975. Describes a popular program design methodology, based on the idea that a program's structure should be modeled on the structure of the data it manipulates. Prime target: business data processing.

4. Kernighan, B.W., and Plauger, P.J. *Software Tools.* Addison-Wesley, Reading, Mass., 1976. A methodology for constructing composable programs, with a bottom-up presentation of a number of examples.

5. Kernighan, B.W. Ratfor—A preprocessor for rational Fortran. *Software—Practice and Experience* (Oct. 1975). One of the most popular Fortran preprocessors.

6. Meyer, B., and Baudoin, C. *Methodes de Programmation.* Eyrolles, Paris, 1978. A fairly comprehensive survey on programming methodology, programming techniques, basic algorithms, and data structures.

7. Meyer, B. *Un Ramasse-Miettes par Tri.* Rep. Atelier Logiciel 8, EDF—Direction des Etudes et Recherches, Sept. 1978. Describes a particular garbage collection algorithm used in a package for text manipulation.

8. Meyer, B. Sur quelques concepts modernes des langages de programmation et leur Representation en Simula 67. AFCET-GRO-PLAN, Vol. 9, Cargèse, 1979, pp. 331–395. How Simula supports modern programming concepts, such as modularity, genericity, top-down design of both algorithms and data structures, etc.

9. Norsk Regnesentral. *Simula 67 for IBM System/360—User's Guide; Simula 67 for IBM System 360—Programmer's Guide.* Pub. S-24-1 and S-23-1, Oslo, Norway, 1975. Reference for the IBM Simula implementation, a programming environment with desirable features like separate compilation and symbolic debugging.

10. Rose, L.R., and Hellerman, H. Portable character processing in Fortran and fixed character environments. *IEEE Trans. Software Eng. SE-2,* 3 (Sept. 76), 176–185. A package for text manipulation.

11. Saltel, E. *Manuel Fortran 30.* IRIA, Rocquencourt, France, 1978. An extension of Fortran which allows graphic processing.

12. Wirth, N. Program development by stepwise refinement. *Comm. ACM 14,* 4 (April 1971), 221–227. A classic reference on the top-down design of programs. Mentions that the refinement process should apply to data structures as well as the algorithmic part.

13. Woodfield, S.W., Dunmore, H.E., and Shen, V.Y. The effect of modularization and coments on program comprehension. Proc. 5th Internat. Conf. Software Eng., San Diego, Calif., March 1981, pp. 215–223. An experimental study on what factors affect the readability of programs. Some results support the view that abstract data types are a good basis on which to construct modules.

NOTE (août 1985)

Au moment de préparer la version finale de cette thèse, il nous a semblé utile de compléter cet article de 1984, qui décrit le prototype de Cépage réalisé à EDF, par un document plus récent (août 1985), donnant l'état actuel du nouveau produit en cours de développement, et qu'on trouvera à la suite du premier. Le second article (*Cépage : Towards Computer-Aided Design of Software*) contient certain nombre de redites mais aussi des compléments importants.

CEPAGE: UN EDITEUR STRUCTUREL PLEINE PAGE

Bertrand Meyer [+], Jean-Marc Nerson [++]

[+] Computer Science Department, University of California
Santa Barbara, California 93106 (Etats-Unis)
Tél. (805) 961 43 85[1]

[++] CIMSA, 10 Avenue de l'Europe
78140 Vélizy (France)[1]

RESUME[2]

Nous décrivons Cépage, un éditeur de documents structurés conçu pour etre d'emploi agréable sur les terminaux actuels. Cépage se trouve au confluent des travaux sur les éditeurs syntaxiques, du développement des éditeurs pleine page, et des études sur les environnements logiciels avancés. C'est un éditeur universel, dans lequel la description du langage est un simple paramètre ; son interface externe est faite pour les enfants de l'ère vidéo. Cépage constitue un prototype de ce que pourrait etre un éditeur structurel utilisable dans un environnement industriel.

MOTS-CLES

Editeurs structurels, éditeurs syntaxiques, ergonomie des dialogues, communication homme-machine, environnements de programmation, formatage des programmes.

ABSTRACT

This paper describes Cépage, an editor for structured documents, designed for ease of use on modern terminals. Cépage was conceived as the common child of three influences: syntax editors; full-screen editors; and advanced software environments. Cépage is universal, the language description being a mere parameter to the system; its user interface is intended to be acceptable to the children of the video game era. We think Cépage is a prototype of what a structural editor should look like in order to succeed in production environments.

KEYWORDS

Structural editors, Syntax Editors, Human Interfaces, Man-Machine Communication, Programming Environments, Program Formating, Ergonomics.

[1] Travail effectué initialement à: Electricité de France, Direction des Etudes et Recherches, 1 avenue du Général de Gaulle 92141 Clamart (France)

[2] Nous présentons nos excuses au lecteur pour les anomalies de typographie (accents circonflexes) et de bibliographie (mentions en anglais), dues au fait que cet article a été préparé aux Etats-Unis sur un système de traitement de textes mal adapté à la langue française.

CEPAGE: UN EDITEUR STRUCTUREL PLEINE PAGE

Bertrand Meyer
Jean-Marc Nerson

1. LES OBJECTIFS

Cépage est un éditeur structurel (terme que nous préférons à celui de "syntaxique"), dans la conception duquel l'interface humaine a été étudiée avec un soin tout particulier. Il est entièrement paramétrable et peut s'appliquer à tout langage défini par une grammaire: langage de programmation, de spécification, mais aussi langage de description de documents structurés de toute nature (nous appellerons ci-après "documents" les objets que l'éditeur sert à construire).

Cépage s'inscrit dans toute une lignée d'éditeurs structurels développés au cours des dernières années [Allison1983, Donzeau-Gouge1981, Donzeau-Gouge, Habermann1982, Hansen1971, 1981, Teitelbaum1981, Wilander1980, Teitelbaum1981] . Les éditeurs structurels, par opposition aux éditeurs de textes classiques, permettent de manipuler des documents non comme de simples suites de lignes ou de caractères, mais comme des objets structurés, en leur appliquant des opérations définies relativement à leur structure. Parmi les principaux avantages de cette méthode, on peut citer:

- la garantie d'obtenir des documents syntaxiquement corrects;
- la possibilité d'effectuer des transformations éventuellement complexes mais garanties correctes, par exemple des transformations de programmes en vue de leur optimisation ou de leur transport;
- la possibilité de décharger l'utilisateur d'une partie des tâches de routine liées à la nécessité, dans un éditeur de textes classique, de fournir tous les détails de la syntaxe "concrète" des documents;
- la possibilité de traduction automatique d'un cadre syntaxique dans un autre (par exemple dans le cas de conversions entre langages de programmation);
- l'utilisation d'une structure de données normalisée (en général l'arbre syntaxique abstrait) qui peut servir de support à d'autres outils logiciels (cf. par exemple [Schroeder1983]), voire à des environnements de programmation complets ([Habermann1982]).

En dépit de ces qualités, les éditeurs structurels n'ont pas encore gagné droit de cité dans l'industrie. L'une des raisons principales de cette situation est, selon nous, liée à leur interface externe qui, dans la plupart des cas, est de type "ligne à ligne", c'est-à-dire que le dialogue avec l'utilisateur consiste en une suite d'échanges de commandes et de réponses. Or les environnements de programmation disponibles aujourd'hui offrent de plus en plus couramment des éditeurs de texte **pleine page**, tels SPF (sur IBM), Emacs (sur Multics et Vax-Unix) ou Vi (sur Vax-Unix), qui tirent parti des possibilités des terminaux actuels. Parmi les caractéristiques de ces systèmes, on peut citer [Meyer1983a] :

- L'utilisation de l'écran complet, de préférence à la ligne, comme unité de communication entre le système et l'utilisateur, donnant à celui-ci une vision notablement plus large sur le document en cours de construction, et lui permettant donc d'exercer un meilleur contrôle sur l'ensemble du processus d'édition;

- la possibilité, plus facile à fournir que dans un système ligne à ligne, de personnaliser le dialogue en conservant des informations relatives à chaque utilisateur;
- l'utilisation en parallèle, dans certains systèmes, de plusieurs **fenêtres**, permettant à l'utilisateur de posséder à chaque instant plusieurs vues différentes sur le document manipulé;
- enfin, et plus généralement, l'application du principe de "manipulation directe" [Shneiderman1983] , selon lequel on maîtrise mieux un système lorsqu'il fournit à chaque instant une représentation claire et à jour de l'état courant des objets traités.

Le bénéfice de ces différentes propriétés est tel qu'il est à peu près impossible de convaincre un utilisateur d'un éditeur pleine page de revenir à un éditeur ligne à ligne, quelles qu'en soient par ailleurs les qualités. Ceci, selon notre expérience, vaut aussi pour les éditeurs structurels: s'ils sont de type ligne à ligne, ils ne pourront gagner les faveurs des utilisateurs habitués à des systèmes pleine page.

Les objectifs de Cépage découlent des réflexions précédentes. Il s'agissait de combiner les avantages des éditeurs structurels en matière de sûreté et de puissance avec la commodité d'emploi des éditeurs de textes pleine page, en tirant le meilleur parti possible des terminaux modernes.

Le projet Cépage ne se voulait pas un projet de recherche, mais plutôt un transfert de technologie, destiné à rendre industriellement utilisables des idées, celles de l'édition structurelle, qui ont fait l'objet de travaux importants de la part des chercheurs. En fait, nous avons dû, à notre corps défendant, "inventer" un peu plus que nous ne l'avions envisagé initialement.

Les principales sources d'inspiration ont été, pour les éditeurs structurels, Gandalf et (dans une moindre mesure) Mentor et CPS; comme modèle d'interface homme-machine, Smalltalk nous a également influencés.

Selon tout critère objectif, le projet Cépage est un petit projet. La spécification et la conception sont l'oeuvre des deux auteurs de cet article, la réalisation presque exclusivement du second (Cépage inclut un petit éditeur de textes, écrit par N. Triquet). Les premières discussions remontent à la fin de 1982; le projet a véritablement pris corps au début de 1983, avec pour objectif (qui a été respecté) d'obtenir un prototype en état de fonctionnement le 20 décembre 1983. La programmation proprement dite n'a commencé qu'en septembre 1983. Le programme comprend environ 6000 lignes en Pascal; il utilise par ailleurs le progiciel Gescran pour la gestion de l'interface écran [Audin1980] , réalisé dans la même équipe, et qui représente environ 4000 lignes de Fortran 77 (Gescran est un ensemble de sous-programmes permettant de décrire commodément les interactions "plein écran" en ne manipulant que des objets appartenant à quatre types abstraits, appelés *écran*, *fenêtre*, *zone*, *terminal* et accessibles uniquement à travers les primitives du progiciel [Meyer1982] ; il s'appuie sur le progiciel d'entrée et sortie Ensorcelé [Brisson1982, Meyer1981]). Les conditions quelque peu particulières dans lesquelles ce projet a été réalisé expliquent sans doute que ces paramètres ne correspondent guère à ce que l'on pourrait déduire de l'étude des bons auteurs [Boehm1982] .

Il peut être intéressant de noter que l'utilisation partielle de spécifications formelles, fondées sur le langage Z [Abrial1980] puis sur la méthode M [Meyer1984a] , a rendu quelques services.

2. L'UTILISATION DE CEPAGE

2.1. L'écran

L'écran affecté à une session de Cépage est divisé en un certain nombre de **fenetres** (figure 1). Chacune de ces fenetres remplit une fonction précise:

- la fenetre "document" contient une représentation de l'état actuel du document en cours de construction ou de modification; certains des éléments de cette représentation, affichés entre chevrons (par exemple *instruction*), correspondent à des éléments du document qui n'ont pas encore été affinés et sont dits **non-terminaux**;
- la fenetre "texte" est destinée à recevoir les textes non structurés qu'il peut etre nécessaire de fournir à certaines étapes d'une session (par exemple des identificateurs, des commentaires);
- la fenetre "menu" offre à chaque étape la liste des choix disponibles;
- la fenetre "type" donne le type syntaxique des éléments délimités (cf. ci-après);
- des fenetres "réserves" (non présentes sur la figure 1) donnent des informations sur des documents ou éléments de documents autres que le document en cours d'édition; ces fenetres sont utilisées pour changer de document pendant la session ainsi que pour les opérations de copie et de transfert.
- la fenetre "message" sert à afficher les diagnostics.

2.2. Le dialogue

A chaque étape de l'exécution d'une session de Cépage, le système propose à l'utilisateur de choisir entre un certain nombre de possibilités à l'aide d'un menu. Pour utiliser les fonctions de base de Cépage, les menus suffisent; un manuel d'utilisation n'est donc pas nécessaire pour peu que l'on ait compris les concepts principaux du système. Dans la version IBM actuelle, le choix entre les différents éléments du menu s'effectue grace aux touches de fonction du terminal. Sur des terminaux plus évolués, on peut imaginer d'utiliser une souris.

Chaque fois qu'il est nécessaire de désigner un élément du document (par exemple pour indiquer à quel terminal s'applique un affinage, comme sur la figure 1a), on utilise à cet effet le curseur, que l'on positionne sur l'élément en question. C'est la seule facon d'accéder au document (la notion de numéro de ligne, par exemple, est absente). L'utilisation d'un dispositif plus rapide tel que la souris serait particulièrement bienvenue ici.

Quelques fonctions plus avancées exigent l'emploi de commandes; ces commandes sont formées d'un mot unique, et leur existence découle uniquement du nombre limité de touches de fonctions disponibles (12). Cépage n'a donc pas de "langage de commande" au sens classique du terme: toutes les interactions avec le système se font par "pointer-toucher".

En particulier, l'utilisateur construisant avec Cépage un texte de programme, en Pascal par exemple, n'est jamais amené à frapper au clavier des éléments de syntaxe concrète, par exemple des mots-clés tels qu' **if**, **procedure**, **record**, etc. Au lieu de cela, un menu lui permet de choisir entre *conditionnelle*, *déclaration de procédure*, *déclaration de type enregistrement*, etc., et le système produit pour lui la syntaxe correcte (les taches de routine sont l'affaire des ordinateurs, non celles des humains).

Le seul cas où le clavier (hors touches de fonction) est nécessaire est celui où l'utilisateur doit fournir un texte que le système ne pourrait inventer seul, comme un identificateur ou un commentaire. La fenetre "texte" est utilisée à cet effet; le texte y est construit grace à un éditeur de textes (pleine page) inclus dans Cépage.

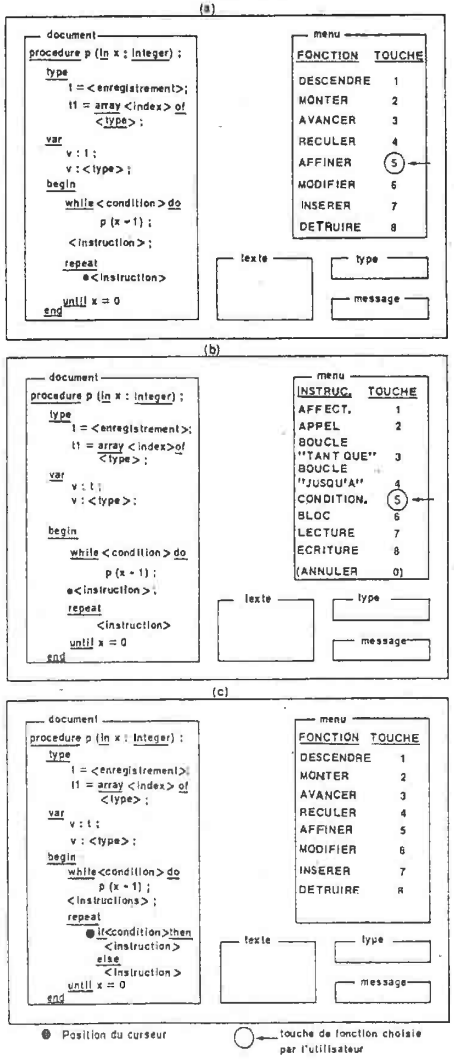


FIGURE 1 - UN AFFINAGE

2.3. Les fonctions de base

Les principales fonctions offertes par Cépage se rattachent aux catégories suivantes.

- **promenade:** parcours du document (montée et descente dans la hiérarchie des entités syntaxiques, avancée et récul dans les listes);

- **construction-modification:** affinage, changement d'un affinage antérieur, insertion et destruction dans une liste;

- **copie-transfert:** reproduction ou déplacement d'un élément de texte (utilisant l'opération de "délimitation": voir ci-après);

- **archivage-restauration:** archivage sur un fichier, sous une forme adéquate, de l'état actuel d'un document en cours d'élaboration, partiellement ou complètement affiné; restauration d'un document précédemment archivé.

- **génération:** production de la forme finale d'un document complètement affiné;

- **controle de session:** choix du document courant, passage d'un document à un autre, définition de bibliothèque etc. (une bibliothèque est un ensemble de documents; on peut au cours d'une session travailler sur plusieurs documents, dont un seul est actif à chaque instant, et passer librement de l'un à l'autre).

2.4. La délimitation

La délimitation (figure 2) est une opération nécessaire pour les fonctions qui exigent de l'utilisateur qu'il définisse un sous-ensemble syntaxique du document: ainsi, pour une copie ou un transfert, il faut délimiter la partie du document à laquelle s'appliquera l'opération. Cette délimitation s'effectue selon les principes de la manipulation directe.

Pour "délimiter", on place le curseur à un emplacement quelconque de l'élément à délimiter, et l'on précise la portée de ce document par une suite de commandes, effectuées grace aux touches de fonction (indiquées sur le menu de délimitation); à chaque étape, le système fait ressortir l'élément délimité par un changement des attributs d'affichage (couleur, affichage en négatif, etc.).

Les commandes de délimitation sont les suivantes:

- englober: inclure dans l'élément délimité la structure syntaxique immédiatement englobante (par exemple, si l'on avait jusque là délimité une instruction, inclure l'ensemble du bloc qui la contient);

- "désenglober": annuler l'effet d'une opération "englober" en revenant au niveau inférieur;

- étendre à gauche: inclure l'élément immédiatement antérieur (cette opération s'applique au cas où l'élément délimité est une sous-liste; les trois opérations complémentaires sont exclure à gauche, étendre à droite, exclure à droite);

- terminer (accepter l'élément actuel); annuler.

2.5. Modification du langage

Cépage est entièrement indépendant du langage; la syntaxe (concrète et abstraite) est un paramètre qui peut etre modifié à volonté. Dans la version actuelle, la description ou la modification du langage se fait de facon assez classique, par l'entrée d'une grammaire. Il est prévu ultérieurement de fournir pour cette opération l'interface du système lui-meme, ce qui revient à dire que l'un des langages pour lesquels Cépage sera défini est un langage de description syntaxique (il est bien conforme aux principes généraux de la conception de Cépage de faire en sorte que l'utilisateur n'ait pas à connaître la syntaxe concrète de ce "langage").

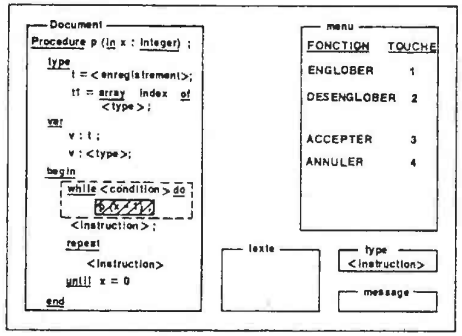


FIGURE 2 - LA DELIMITATION

La portion hachurée a été délimitée (et apparait en négatif ou dans une couleur spéciale). En appuyant sur la touche de fonction 1 ("ENGLOBER"), on délimite l'ensemble de la zone entourée en pointillé.

La modification du langage peut paraître une opération peu utile en pratique, pour autant que Cépage soit fourni avec des descriptions des principaux langages. En fait, la possibilité d'adapter facilement la description du langage à des conditions locales nous paraît une caractéristique vivement souhaitable. Elle permet en particulier de mettre en place des normes de programmation d'une facon plus commode (et plus facile à faire accepter) que par l'utilisation d'outils de controle *a posteriori*. On peut ainsi définir des sous-ensembles d'un langage, des conventions relatives aux commentaires, à la structure des programmes, etc.

3. CEPAGE: LES CHOIX TECHNIQUES

3.1. Les structures de données fondamentales

Au cours d'une session, Cépage travaille (figure 3) à partir de deux structures de données principales:

- la description interne du langage, ou graphe de grammaire;

- la description interne d'un ensemble de documents: foret syntaxique abstraite.

Il est important de noter que ces deux structures de données sont traitées sur un pied d'égalité. C'est ce qui fait de Cépage un système entièrement paramétré par le langage: la description du langage est interprétée répétitivement par le système. Ceci distingue nettement Cépage d'un système tel que Gandalf, paramétrable certes, mais

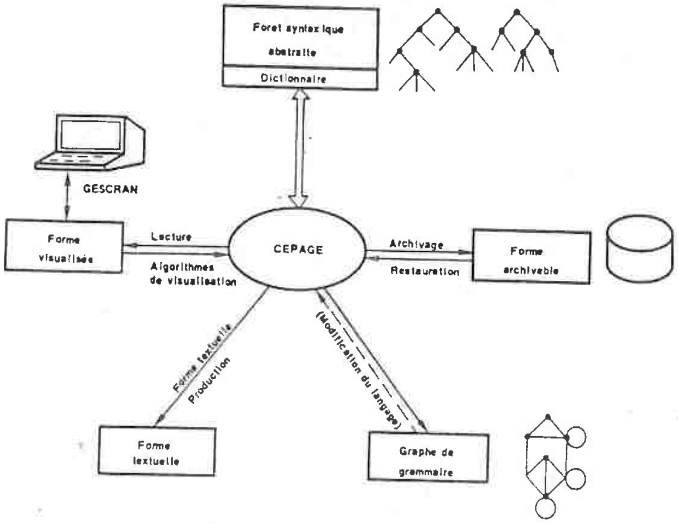


Figure 3 : Les structures de données

dans lequel la description du langage est "compilée", c'est-à-dire dans lequel on doit partir d'une version "noyau" de Gandalf et d'une description d'un langage X (ou Z, ou C) pour obtenir un outil Gandalf-X, ou Gandalf-C, adaptée au langage choisi. La solution adoptée par Cépage offre une plus grande souplesse et explique qu'il soit possible de modifier facilement le langage. En revanche, elle ne permet pas de prendre aussi facilement en compte des actions sémantiques, ce qui est un des buts de Gandalf.

Le **graphe de grammaire** est une structure de données permettant de représenter l'ensemble des propriétés de la grammaire du langage. La **syntaxe abstraite** est utilisée comme base; elle est décrite par un ensemble de **types syntaxiques** et de **productions**. Chaque type syntaxique apparaît à gauche d'une production au plus; ceux qui n'apparaissent à gauche d'aucune production sont dits terminaux. Il y a trois sortes de productions, dites "concaténation", "union" et "liste", illustrées respectivement par les exemples suivants:

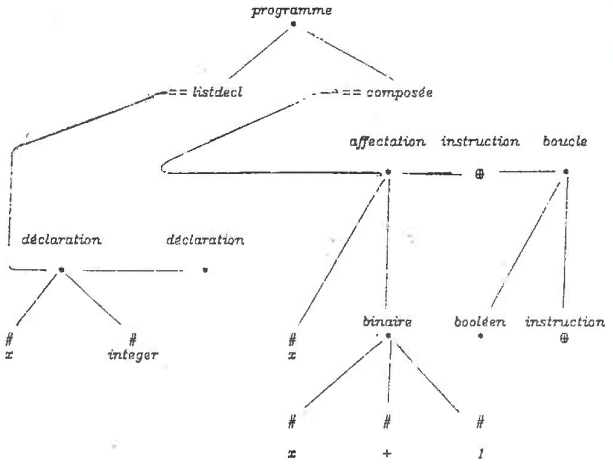
conditionnelle = c: *booléen* ; st1, st2: *instruction*;

instruction = *affectation* | *conditionnelle* | *composée*

composée = *instruction**

La **syntaxe concrète** est obtenue par "décoration" des productions de la syntaxe abstraite; par exemple, à toute production de type liste sont associés un en-tete, un délimiteur et une fin (par exemple **begin**, le point-virgule et **end** dans le cas de *composée*). Le graphe de grammaire regroupe l'ensemble de ces informations.

La **foret syntaxique abstraite** comprend un ensemble d'arbres syntaxiques abstraits, associés chacun à un document ou élément de document en cours d'élaboration.



* : noeud concaténé
== : noeud liste
⊕ : noeud alternatif
: noeud texte

Figure 4: Arbre Syntaxique Abstrait

Les noeuds internes d'un arbre syntaxique abstrait (figure 4) sont de quatre sortes, correspondant aux quatre types de productions:

- les noeuds "concaténés" ont une arité fixé;
- les noeuds "alternatifs" représentent seulement un choix dans une production de type union;
- les noeuds "liste" peuvent avoir un nombre quelconque de fils.

- les noeuds "texte", correspondant à des éléments terminaux affinés par l'utilisateur à l'aide de l'éditeur de textes inclus dans Cépage.

3.2. Autres structures de données

D'autres structures de données complètent les deux précédentes.

Outre les arbres syntaxiques abstraits, trois représentations sont nécessaires pour les documents:

- la forme visualisable, un ensemble d'éléments destinés à etre transmis à Gescran pour affichage sur le terminal à chaque étape de la session;

- la forme archivable, pour préservation et restauration ultérieure de l'état instantané d'un document;

- la forme textuelle, but ultime du processus d'édition.

Par ailleurs, la foret syntaxique abstraite s'accompagne d'un **dictionnaire**, contenant les différents éléments textuels nécessaires (identificateurs, etc.). Les feuilles des arbres syntaxiques contiennent des références au dictionnaire.

3.3. Les algorithmes

Il convient de faire remarquer que les objectifs définis précédemment impliquent **l'absence d'analyse syntaxique** dans Cépage. La construction d'un texte s'effectue par choix successifs, correspondant à la syntaxe abstraite; la syntaxe concrète est construite par le système, qui effectue en réalité l'opération inverse de l'analyse syntaxique, appelée parfois "désanalyse" (*un-parsing*).

On notera que la liberté laissée aux utilisateurs dans la description du langage permet d'établir en pratique un bon compromis entre la facilité d'utilisation et le degré de détail auquel descend le système; par exemple, on peut envisager de considérer *expression* comme un terminal. Une autre technique pour ce type d'entité syntaxique, non mise en oeuvre dans la version actuelle de Cépage, est celle de [Kaiser1982] , intermédiaire entre "analyse" et "déanalyse".

S'il n'y a pas d'analyse syntaxique, un autre type d'algorithmes a posé des problèmes sérieux: la construction de la forme visualisée. Il s'agit de proposer à chaque instant une représentation aussi riche que possible de l'état du document, en tenant compte des limites imposées par la taille physique du terminal.

Avec un éditeur de textes, pleine page ou non, on ne peut en général fournir qu'un extrait du document formé d'une suite contiguë de lignes (certains éditeurs offrent la possibilité d'exclure des groupes de lignes de la partie affichée afin de se concentrer sur les éléments les plus intéressants à un moment donné). Un éditeur structurel doit etre capable de fournir une vue globale du document ou d'une partie de celui-ci, meme s'il ne peut la représenter sur l'écran avec tous ses détails. La solution est l' **élision**: on remplace certains éléments du document par une abréviation - plus précisément, par une simple indication de leur type. Ainsi, une procédure de 2000 lignes pourra etre figurée par la simple indication *"procédure"*; nous appelons ce type d'abréviation **abstraction**. Le second type d'abréviation effectue par Cépage est le **rétrécissement**, qui consiste en une abstraction appliquée à une ou plusieurs sous-listes d'une liste, comme dans-

"231 instructions";

p := *expression*;

"57 instructions"

A chaque étape de la session, le système détermine le foyer sur lequel l'utilisateur semble vouloir concentrer son attention d'après les dernières opérations qu'il a effectuées, et cherche à afficher une vue aussi détaillée que possible d'une portion du

document, de part et d'autre du foyer, déterminant les abstractions et retrecissements nécessaires. Il en déduit la forme visualisable qui est transmise à Gescran pour affichage.

La recherche d'une bonne représentation visualisable s'est révélée une tache d'une difficulté inattendue. Nous avons été surpris par le peu de documents disponibles; si l'on excepte une brève allusion dans [Barstow1984] , la seule référence publiée est à notre connaissance [Mikelsons1981] , qui est difficilement utilisable du fait de son imprécision et des caractéristiques particulières de l'environnement décrit.

L'abondante littérature sur le formatage des programmes ("prettyprinting", paragraphage) est ici de peu d'utilité; l'hypothèse fondamentale, quoique en général implicite (cf. en particulier [Oppen1980]) est que, si la longueur des lignes est limitée, le nombre de lignes, lui, ne l'est pas. Pour un formatage sur écran, **les colonnes et les lignes sont des ressources sévèrement limitées.**

Nous avons donc été amenés à concevoir des algorithmes spécialisés décrits ailleurs [Meyer1983b, Meyer1984b] , et qui dépassent le cadre de cet article. Ces algorithmes sont linéaires par rapport au nombre de noeuds de l'arbre syntaxique. Il s'agit de l'un des domaines où nous avons dû "inventer".

4. L'AVENIR DE CEPAGE

Comme il a été indique au début de cet article, la version de décembre 1983 est un prototype, comprenant cependant les fonctions essentielles du système. Les actions ci-après sont ensuite prévues.

- Il faudra étudier les réactions des utilisateurs. La conception de Cépage repose sur ce que nous pensons etre une bonne base ergonomique pour des systèmes interactifs, opinion confortée par des études récentes reposant sur de solides bases scientifiques [Card1983] , mais demande, bien entendu, à etre validée expérimentalement.

- Il est également prévu d'adapter le système à d'autres environnements. Cépage a été conçu pour etre portable: le choix de Pascal, de préférence à un langage orienté objets comme Simula 67 (utilisé précédemment avec succès dans la meme équipe pour réaliser des outils interactifs de qualité), était justifié par cet objectif. Il est prévu à court terme d'adapter Cépage à un environnement Unix, à la fois sur Vax et sur une station de travail SUN (à l'université de Californie); le SUN est un poste de programmation à base de 68000, possédant un écran à haute résolution ("bit-map") et une souris. Ce projet est pour nous particulièrement important, car c'est seulement dans des environnements matériels de ce niveau que des outils tels que Cépage pourront, selon nous, tenir toutes leurs promesses; nous espérons que Cépage sera également adapté à d'autres systèmes de ce type (Perq, Apollo, SM 90...).

- Il convient également d'ajouter les principales fonctions absentes du prototype, en particulier l'outil de modification du langage, et préparer des grammaires-Cépage pour les principaux langages utilisés en pratique (le prototype a été testé avec une grammaire d'un langage voisin de Pascal).

A la lumière des premières expériences, nous aurons peut-etre la réponse à quelques-unes des questions qui restent actuellement en suspens, comme celle de l'analyse syntaxique: faudra-t-il, dans une version ultérieure, ajouter un analyseur syntaxique, de facon à permettre de manipuler par Cépage des programmes existants, obtenus par d'autres moyens?

Nous espérons que la mise en service des premières versions confirmera ce que nous pensons etre le grand intéret potentiel du système actuel, et permettra d'en faire un élément essentiel d'un environnement de programmation puissant et ergonomique.

References

Abrial1980.
Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs*, ed. C.A.R. Hoare and R. Perrot, Cambridge University Press, Cambridge (U.K.), 1980.

Allison1983.
R. Allison, "Syntax-Directed Program Editing," *Software, Practice and Experience*, vol. 13, pp. 453-465, April 1983.

Audin1980.
Eugène Audin, Gérard Brisson, Bertrand Meyer, and Françoise Vapné-Ficheux, "Gescran, Manuel de Référence," Atelier Logiciel 22, Electricité de France, 1980. (Fourth Edition, 1984)

Barstow1984.
David R. Barstow, "A Display-Oriented Editor for INTERLISP," in *Interactive Programming Environments*, ed. David R. Barstow, Howard E. Shrobe, Erik Sandewall, pp. 288-299, McGraw-Hill, New York, 1984.

Boehm1982.
Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs (N.J.), 1982.

Brisson1982.
Gérard Brisson, Bertrand Meyer, and Françoise Vapné-Ficheux, "Ensorcelé: Entrées et Sorties Sans Format (2ème partie)," Atelier Logiciel no. 6, Electricité de France, December 1982.

Card1983.
Stuart K. Card, Thomas P. Moran, and Allen Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale (New Jersey), 1983.

Donzeau-Gouge1981.
Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang, "Environnement de Programmation Mentor: Présent et Avenir," in *Actes des Troisièmes Journées Francophones sur l'Informatique*, Genève, 1981.

Donzeau-Gouge.
Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, ed. David R. Barstow, Howard E. Shrobe, Erik Sandewall, pp. 128-140, McGraw-Hill, New York.

Habermann1982.
Nico Habermann and others, *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh (Pa), 1982.

Hansen1971.
Wilfred J. Hansen, "Creation of Hierarchic Text with a Computer Display," ANL-7918, Argonne National Laboratory, Argonne (Ill), 1971 (Also as dissertation, Computer Science Department, Stanford University, June 1971).

1981.J.W. Lewis , "Beyond ALBE/P: Language and Neutral Form," in *Proceedings of the 5th International Conference on Software Engineering*, pp. 422-429, San Diego (Ca.), March 1981.

Kaiser1982.
Gail E. Kaiser and Elaine Kant, "Incremental Expression Parsing for Syntax-Directed Editors," Computer Science Report, Carnegie-Mellon University, October 1982.

Meyer1981.
Bertrand Meyer, "Ensorcelé: Entrées et Sorties Sans Format (1ère partie)," Atelier Logiciel no. 4, Electricité de France, April 1981. (Fourth Edition)

Meyer1982.
Bertrand Meyer, "Principles of Package Design," *Communications of the ACM*, vol. 25, no. 7, pp. 419-428, July 1982.

Meyer1983a.
Bertrand Meyer, "Towards a Two-Dimensional Programming Environment," in *Proceedings of the European Conference on Integrated Computing Systems (ECICS 82), Stresa (Italy), 1-3 September 1982*, ed. Pierpaolo Degano and Erik Sandewall, North-Holland , Amsterdam (The Netherlands), 1983.

Meyer1983b.
Bertrand Meyer and Jean-Marc Nerson, "Showing Programs on a Screen," Internal Report HI/4590-01, Electricité de France, September 1983.

Meyer1984a.
Bertrand Meyer, *A System Description Method*, Workshop on Specification Languages, to appear, Orlando (Fl.), March 1984.

Meyer1984b.
Bertrand Meyer and Jean-Marc Nerson, *Showing Programs on a Screen*, Submitted for Publication, 1984.

Mikelsons1981.
M. Mikelsons, "Prettyprinting in an Interactive Programming Environment," *SIGPLAN Notices*, vol. 16, no. 6, pp. 108-116, June 1981.

Oppen1980.
Derek C. Oppen, "Prettyprinting," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 4, pp. 465-483, October 1980.

Schroeder1983.
Anne Schroeder, "Outils d'Analyse des Programmes sous Mentor," *GLOBULE (AFCET)*, no. 4, 1983.

Shneiderman1983.
Ben Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer (IEEE)*, vol. 16, no. 8, pp. 57-69, August 1983.

Teitelbaum1981.
Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563-573, September 1981.

Wilander1980.
Jerker Wilander, "An Interactive Programming System for Pascal," *BIT*, vol. 20, pp. 163-174, 1980.

CEPAGE:
TOWARDS COMPUTER-AIDED DESIGN OF SOFTWARE

CEPAGE :
Vers la conception de logiciel assistée par ordinateur

Bertrand Meyer

Computer Science Department, University of California
Santa Barbara, California 93106 (USA)
(805) 961-4321

and

Interactive Software Engineering, Inc.
270 Storke Road, suite #7
Goleta, California 93117 (USA)

ABSTRACT

The system described in this paper, Cépage, is a powerful tool for creating programs or other documents in any language with a non-trivial structure. With Cépage, the computer, not the user, generates the proper *syntax* for the documents under construction, and produces on the terminal screen, at every step of the interaction, a clear and consistent *display* of the current state of the the document.

Cépage applies principles of Computer-Aided Design to provide users with *structural* views of programs and other documents, allowing them to look at the program at any chosen level of detail.

The language is a *parameter* for Cépage, so that it is easy to use the sytem to support a new language or a local variant of an existing language.

The system offers facilities not only for *creating* and *modifying* texts, but also for performing systematic *transformations* and, in the case of programs, *checking* and *execution*. It may thus be used towards interactive testing and *rapid prototyping*, more generally, as a basis for an advanced *programming environment* based on the manipulation of structured documents through a sophisticated user interface.

A preliminary version of this paper was presented at the Convention Informatique, in Paris, on September 18, 1985.

Table of Contents

1 - CONTEXT

Many of the tools used to design software are still very primitive when compared to those which have been made available by software engineers to the engineers of other fields. This paper presents a tool whose aim is to provide software designers with facilities similar to what is known in other application areas under the general name of "Computer-Aided Design".

A prototype of the system presented here, called Cépage (English-speaking readers should pronounce its name as *Sea-Page*) was developed at Electricité de France in 1983 [13] using standard mainframe equipment: an IBM 3081, running MVS—TSO—SPF. The version described in this paper is an entirely new development; although based on the same fundamental ideas and on experience with the prototype, it pursues more ambitious aims and is designed as a commercial product. This new product is being manufactured by Interactive Software Engineering, Inc. (in Goleta, California), initially for a Unix environment[1]; plans are under way to port it to other architectures (VAX-VMS, IBM-PC, IBM-MVS, Apple Macintosh)[2].

Cépage will show its best on a bit-mapped display, but scaled-down versions for less expensive terminals are also useful.

The design of Cépage relies on a simple but (we think) powerful idea: to allow **visual** manipulation of **structured** documents in terms of their structure, not just as if they were mere sequences of characters. Typical "structured documents" are programs written in some high-level language; but it is important to add immediately that all documents with non-trivial structures, such as specifications, designs, schedules, technical reports whose structure follows a regular pattern and other standardized documentation are equally good candidates for handling by Cépage.

Since the visual aspect of Cépage is so important to its understanding, we will for the time being defer any theoretical explanation of the tool and rather give a short "**demonstration**" of the system, to help the reader get a feeling for the kind of interaction that goes on with such a tool.

2 - EXTRACTS FROM A SESSION

We are using Cépage on a unspecified display. In this paper, we use various font conventions (roman, italics, boldface) to distinguish the display styles that emphasize the different types of elements; on an actual screen, Cépage relies on the facilities provided by the hardware: fonts on a black-and-white bit-mapped screen, different colors on a color display, various levels of highlighting, etc.

Below is the picture that we might have at a given moment in a Cépage session (figure 1).

Actually figures 1 to 4 do not show the whole screen, but the main window, devoted to the currently active document. The screen contains other windows, for such things as session information, help messages and the catalog of available documents (one may work on several documents at a time and switch back and forth between them).

---

[2] The kernel version (serving as a basis for the others) is developed on a Sumitomo U-station, a 68000-based Unix System V workstation with a color bit-mapped display. The implementation language is Dilars (Design and Implementation LAnguage for Reusable Software), an object-oriented language with multiple inheritance and information hiding, pre-processed into C.

[2] VAX is a trademark of Digital Equipment Corporation, Unix of AT&T Bell Laboratories, Macintosh of Apple Corporation.

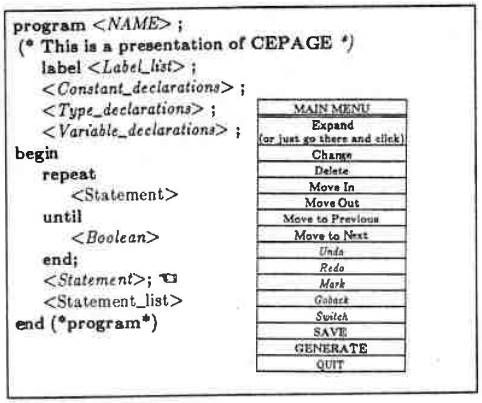


Figure 1: A Cépage Document Window

As you see, we are working on a program in some Pascal-like, slightly Adaish language[3]. out to be slightly Adaish as well.[4]

The most original feature of this program is that it is not complete: it contains not only "true" program elements, appearing in boldface on the figures like **label, until,** the initial comment ("This is a presentation of..."), etc., but also things in italics like *<Statement>*, *<Statement_list>* etc. which represent as yet unexpanded program parts (the reader who remembers his compiler courses will know them as "non-terminals"). They are distinguished from the expanded parts by the angle brackets and by the italics font (or, on a particular terminal, by a different color).

Texts such as the one displayed here are called **Partially Expanded Documents** (we use the word "document" rather than "program" to emphasize again the fact that the underlying language could describe structured objects other than programs). Partially expanded documents are the basic entity that Cépage handles. Of course, eventually a document should be completely expanded, and Cépage can then generate a textual version of the finished product - which, in the case of a program, may be passed on, for instance, to a compiler; other kinds of documents might be handed over to a text formater such as troff on Unix, etc.

There is also an instance of <Statement>, in the **repeat...** loop at the beginning of our program body, that appears in roman font (again, it might be a color), rather than italics. That one does not represent an unexpanded statement: quite to the contrary, the statement has indeed been expanded, and its expansion is so long that, given the size of the window, there is no way to display the details of the statement without losing some of the context (the whole program). The expansion of this statement perhaps contains as many as several hundred lines (which would imply that you are not a programmer of the most modular kind).

We call **abstraction** the process of displaying just the name of a non-terminal type, like <Statement>, to stand for a possibly large part of a document.

[4] So far as we know, Pascal-like and Adaish are not trademarks of anybody yet.

Of course, you may at some point want to see some of the abstracted part. Nothing could be easier: just move the cursor to some position in the *<Statement>* in roman and press a mouse button (or function key, depending on the terminal). Of course, as you go down you will lose some context, which you may see again by moving "out" again, using the corresponding option in the menu.

For the moment, however, we are interested instead in developing our program a little more. We have decided to expand the *<Statement>* that appears just before the **end;** thus we have brought the cursor (represented by the hand on figure 1) to the "window" in which the word *<Statement>* appears. We look and choose the *Expand* option, again using whatever selection medium is available: mouse to point in the menu, function key etc. Actually, as the menu shows, the *Expand* operation is so fundamental that you don't really need to select it explicitly: just moving the cursor to a non-expanded element and pressing a button or function key will trigger the expansion mechanism.

The basic interaction with Cépage is normally done in this fashion: Show and Select (S&S), i.e. indicate a position on the screen and select a function from a menu. S&S is a very effective way of dealing with computers interactively; Shneidermann [18] indicates that many of the interactive systems that are really popular with their users rely on the principle of **direct manipulation** and on the idea that the user should "see what he has got" at every stage of the interactive session. This applies not only to editors but also to systems for Computer-Aided Instruction, Computer-Aided Design (an application area which influenced Cépage significantly, as will be seen below), to video games etc.

The effectiveness of this approach is backed by extensive psychological studies [3]. Of course, the "S & S" principle is at its best when the display and the selection device (mouse, joystick) are adequate.

Once we have said that we wanted to expand a particular statement, something new will appear on the screen. The text of the document does not change, but a new menu pops up, listing the set of possible statements in the language at hand (figure 2).

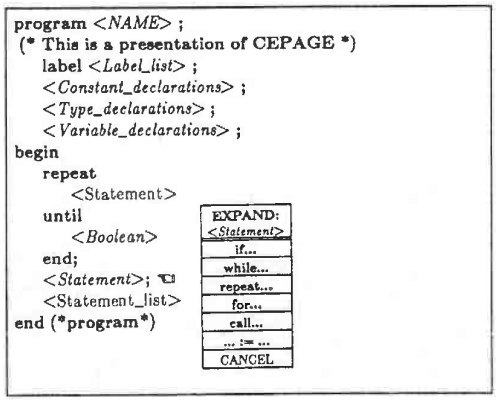


Figure 2: Selecting an expansion in a menu

The new menu allows us to select the type of statement we want. We do not need to type any keywords (e.g. if, etc.); we just select the choice we want in the menu, and the system will take care of generating the proper syntax for us. (However, we may also type the beginning of

the statement if we prefer to work in this fashion, as will be explained below).

Here assume we decide we want a conditional statement and choose the corresponding item in the menu, with any available selection facility. The system generates the resulting structure: figure 3 shows what now appears on the window.

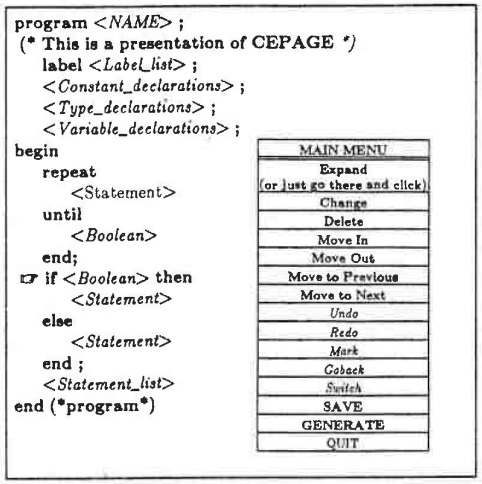


Figure 3: Result of Expansion

The part which previously read <*Statement*> has been replaced with the syntax for a conditional statement.

Note that up to now we haven't used an alphabetic keyboard: the mouse suffices for the manipulations done so far. We do not *have* to use the mouse: we could type phrases, or meaningful beginnings of phrases, if we prefered to. But we may work by S & S if we like. Which solution makes more sense depends on the user's individual taste and on the power of the terminal hardware available.

At some point, for elements such as expressions, it may become tedious to have to describe the structure; one just wants to type in the stuff. For the elements of lowest levels such as identifiers or constants, this is the only possibility anyway since they have no further structure. To enter such elements, one just types them at the place where they appear; they will be immediately parsed by the system.

For example, one may wish to resort to typing when entering the boolean expression of our newly built conditional statement, as shown on figure 4.

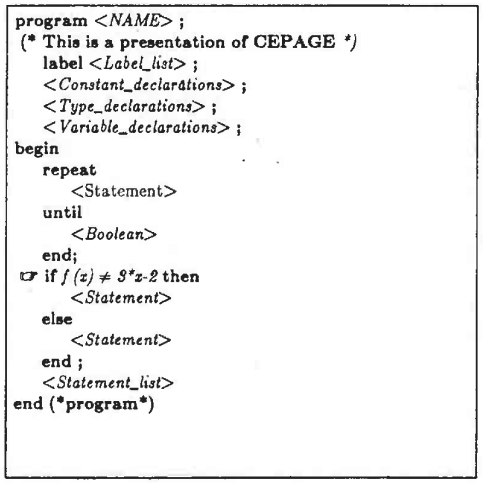
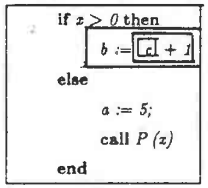


Figure 4: Entering Text

It is important to note that the user has a choice at all levels between menu selection and typing. In the latter case, an added advantage over text editors is that one may type just the beginning of a phrase provided it is long enough to dispel any ambiguity; for example, typing just **while** at a place where a statement is expected is enough for the system to generate the entire pattern for a while loop.

We stop here our little demonstration. Other features resemble those which are generally available in text editors: delete, copy, move, search, replace, "yank" (i.e. put aside for later reuse), etc. There is an important difference, however, since here all such manipulations may only apply to syntactically meaningful parts of the partially expanded document; so if we have, say



and want to apply an operation such as delete, copy etc. to a part of the document containing *c*, then this part may only be one of the boxes above. On the other hand, there is no way to, say, replace "**else call**" by "; **goto**", since neither pattern corresponds to a syntactic entity. This is what is meant by "structural" manipulation.

3 - THE BACKGROUND: STRUCTURAL EDITORS

A tool such as Cépage is known as a "structural editor" (in other terminology for the same concept, "editor" has been used with the qualifiers "structure", "syntax", "syntax-oriented", "language-based", etc.). A structural editor manipulates documents in terms of their underlying structure, not as if they were flat sequences of characters. Many of the basic ideas were contained in Hansen's EMILY system [10]; the best-known structure editors are Mentor, developed at INRIA [4, 5, 6], Gandalf, from Carnegie-Mellon [9] and the Cornell Program Synthesizer [19]. A more recent tool with graphical facilities is Pecan [16, 17].

Structural editors offer several potential benefits:

- they guarantee that only syntactically correct programs are generated;
- they provide a unified basis on which to build complete programming environments, where all tools can rely on a single data structure describing programs (this data structure, as we shall see below, is the **abstract syntax tree**);
- they make it possible to perform possibly complex program transformations in a safe and efficient way;
- they allow automatic translation from one programming language to another;
- they may be used to free the designer from routine tasks like generation of the concrete syntax.

Note that in listing these benefits we have referred to "programs" to emphasize the most immediate applications but, again, it should be noted that other kinds of documents may be handled by structured editors.

Most structural editors have been used so far in academic environments only, which we think is a pity because of their great potential advantages. In our opinion, the main reason for this situation is that structural editors have lagged behind in terms of their **user interfaces**.

In most present programming environments, one or more *full-screen editors* are available. These tools make it possible to take advantage of current video terminals to edit documents in a "direct manipulation" mode; the size of the "window" provided by the system on the document is the size of the available screen, which gives the user a much wider view of the document and better control of the editing process than with traditional "line-by-line" editors.

The advantage of full-screen editors over line-oriented ones is so clear that it is impossible to convince users to go back to the latter once they have experienced the joys of the former. We were particularly aware of this fact after having witnessed in two different cases how how a full-screen editor (IBM's SPF and Vi on Unix, respectively) all but ousted the previous line editors in a matter of months in two different installations; the philosophies of Vi and SPF are remarkably different, but the results were identical. This is all the more significant when one considers the resistance of most users to any kind of change in their software habits - languages, methods or tools.

It was thus clear to us that no structural editor, regardless its other qualities, would become successful in industry if it did not provide *at least* the services of modern full-screen editors. Cépage is an attempt to combine the best of both worlds.

A particular attention was devoted in Cépage to the design of the **display algorithms**: the idea is to provide users with *structural* views of their programs or other documents, instead of just the contiguous extracts offered by text editors. The paradigm here is that of **computer-aided design**: one should be able to hierarchically *traverse* a program in the same way that one explores, say, an electronic system at various levels (system, subsystem, wafer, gate, transistor...); similarly, one wants to see the global structure of a software system, then a little more of a particular module, then one of the procedures of that module, then some of its statements, etc. The display policy will be outlined below.

4 - SYSTEM STRUCTURE

The structure of Cépage is given by Figure 5. A kernel, or "pilot system", works on a set of data structures: grammar graph, abstract syntax forest, display form, visual form, library, external form. The design of Cépage was done according to the "object-oriented" philosophy, which may be roughly summarized as implying that a system should be described by the types of objects it manipulates and their patterns of communication. True to this approach, we shall present the internal structure of Cépage by describing successively each of its main data structures.

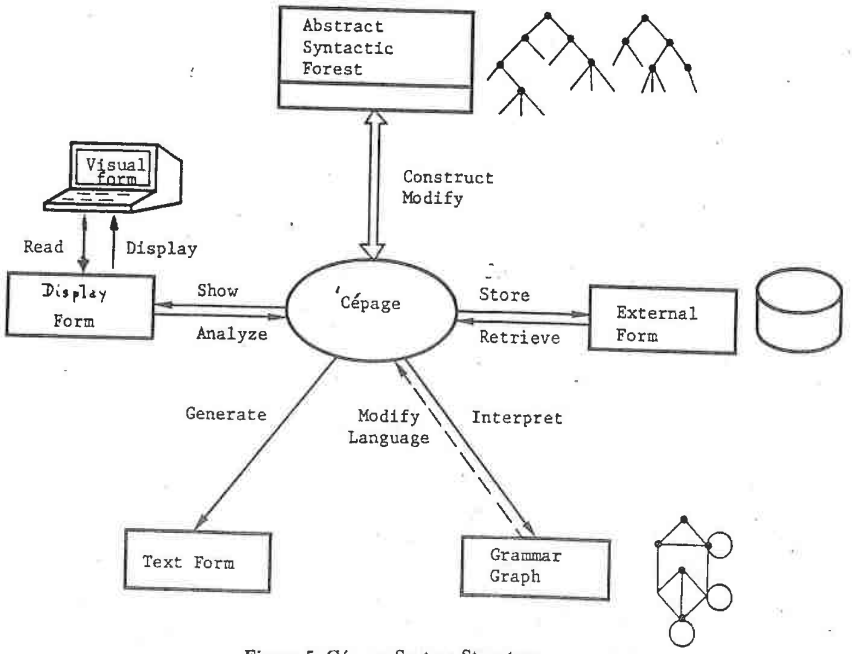


Figure 5: Cépage System Structure

4.1 - The Grammar Graph

The grammar graph data structure is used to describe the language in which the documents are written. A distinctive feature of Cépage is that this data structure is repetitively interpreted by the kernel system, which means that the system as such is completely language-independent; the language is a parameter, easy to modify even at run-time. This flexibility may be used for instance by a software project leader to modify the standard description of the language used, so as to enforce programming standards. For example, the syntax for the modified Pascal used in the above figures includes a compulsory comment at the head of all

programs, procedures and functions (the comment used in the examples was **This is a presentation of CEPAGE**).

In a different application of the same concept, one might wish to extend the syntax of Fortran to include statements such as **while... do** or **repeat... until**, which will be translated on-the-fly into lower-level Fortran equivalents (using *IF*s and *GOTO*s). The idea that the language should be a mere parameter thus allows Cépage to offer a modern interactive alternative to the concept of pre-processor.

The description of a language, as embodied in the grammar graph, is based on the notion of **abstract syntax**. The abstract syntax contains the specification of **syntactic types** covering the various kinds of constructs in the language. A syntactic type is defined by a **production** of the abstract syntax, which is a description of its deep structure, independently of its external representation. Thus the production defining conditional statements, which in classical Backus-Naur Form would be something like

<conditional> ::= **if** *<boolean>* **then** *<statement>* **else** *<statement>* **end**

will just be, in abstract syntax:

conditional = statement ; boolean ; statement

The abstract syntax that we use for describing languages, has three kinds of productions. The one describing *conditional* in the above example is called an aggregate production; it defines the elements of a syntactic type as having a certain number of components ; some components may be defined as optional as e.g. the *label_part* in a Pascal program. An aggregate production is not unlike a Pascal record type definition. A type may also be defined by a **choice** production, giving the list of alternative expansions, as in the following example:

statement = assignment | procedure_call | conditional | loop | compound

The third and last type of production comprises **list** productions. As an example, a compound statement is defined abstractly as consisting of zero, one or more statements; this is expressed as a list production, using the star notation:

*compound = statement**

The abstract syntax of a language is defined by a set of productions of the above three types. A syntactic type is defined by (i.e. appears on the left of) at most one production, so that we can speak of aggregate types, choice types, list types. Syntactic types that do not appear in the left-hand side of a production are called **terminal** types; examples are types like *Identifier, Constant* etc. that have no further meaningful structure.

The grammar graph contains a representation of the abstract syntax of the language, i.e. of the productions defining it. The nodes of the graph correspond to the syntactic types; an appropriate data structure is associated with productions of each kind; for example, the description of an aggregate or choice type will contain a list of pointers to the nodes associated with the types appearing on the right-hand side of the corresponding production.

More information must be present in the grammar graph in order for the system to be able to display readable views of the documents. Such views must be shown in the form familiar to the user, i.e. the concrete rather than abstract syntax. The operation which makes it possible to construct a concrete representation from an abstract one is known as **un-parsing**, since it is the exact opposite of the "parsing" task performed by compilers (and by Cépage, since the user has a choice between entering by menu or by typing the beginning of a meaningful phrase). In order to un-parse a document, the system must know the concrete syntax of the language.

In the grammar graph, the concrete syntax is defined by additions to the abstract productions. For example, the concrete syntax for *conditional*, defined above by a production of the aggregate type, may be included in the grammar graph through a list of elements representing the following sequence:

if *@2* **then** *@1* **else** *@3* **end**

Such a sequence is to be understood as follows. Elements such as **if**, etc., are called **operators** and represent the constituents that appear in the concrete syntax only. Operators may be keywords of the language; they can also be formating marks like *line_break, indent (n)* (meaning "indent right by *n* positions"), *blanks (m)* (meaning "skip *m* blanks"), *tab (p)* (meaning "continue at position *p* of the line" and useful for fixed-format languages like Fortran), etc. Elements of the form @i represent abstract syntactic elements, indexed relative to their position in the right-hand side of the abstract production; thus here *@1* is the first statement, *@2* is the boolean expression, *@3* is the second statement. So the above concrete syntax addition means that to display a conditional statement we display **if**, followed by the boolean expression, followed by **then**, etc.

In this example, the order of the components of a conditional expression is not the same in the concrete and abstract forms. This was done not only for elegance (the order *statement, boolean, statement* is more symmetric than the concrete one), but also to point out that these do not have to be the same. In fact, the notion of "order" of components in the concrete syntax disappears if, as may happen, some components appear more than once in the concrete form; for example, we may wish to automatically include at the end of each procedure a comment reminding the reader of the name of the procedure, so that we will associate with the abstract production

procedure = name ; parameter_list ; body

the following concrete syntax, using the Pascal convention for comments:

procedure *@1 (@2) ; @3* **end procedure** {*@1*}

Associating a concrete syntax with a non-terminal defined by a list production is simple; all we need to record in the grammar graph is three operators: a header *h*, a terminator *t* and a delimiter *d*; a list will be displayed as

h @1 d @2 d @3 ... d @n t

where @i is the concrete representation of the *i*-th element of the list. Thus for a *compound* statement in an Algol-like language, *h* is **begin**, *t* is **end** and *d* is the semicolon.

There is no need to associate concrete syntactic information with nodes of the grammar graph representing choice types such as *statement*.

The grammar graph is thus a powerful structure which makes it possible to describe possibly complex languages in a flexible way. As stated previously, it seems to us very important to allow for easy creation and modification of grammar graphs.

One of the standard languages supported by Cépage is thus **LDL**, a **language description language**. LDL documents are descriptions of grammars by abstract syntax productions and concrete syntactic additions, as seen above.

4.2 - The Abstract Syntactic Forest

To represent partially expanded documents, the system uses a a set of abstract syntax trees, or abstract syntactic forest.

An abstract syntax tree is not unlike the "parse tree" used by compilers, but it corresponds to the abstract syntax of the language rather than to the concrete one; in other words, it contains only essential, structural information, and excludes anything that is only associated with the external representation of documents (i.e. keywords and more generally what we have called "operators" above).

Figure 6 gives an example abstract syntax tree.
The relevant productions of the corresponding abstract syntax are the following (with [A], [C] and [L] standing for aggregate, choice and list production respectively):

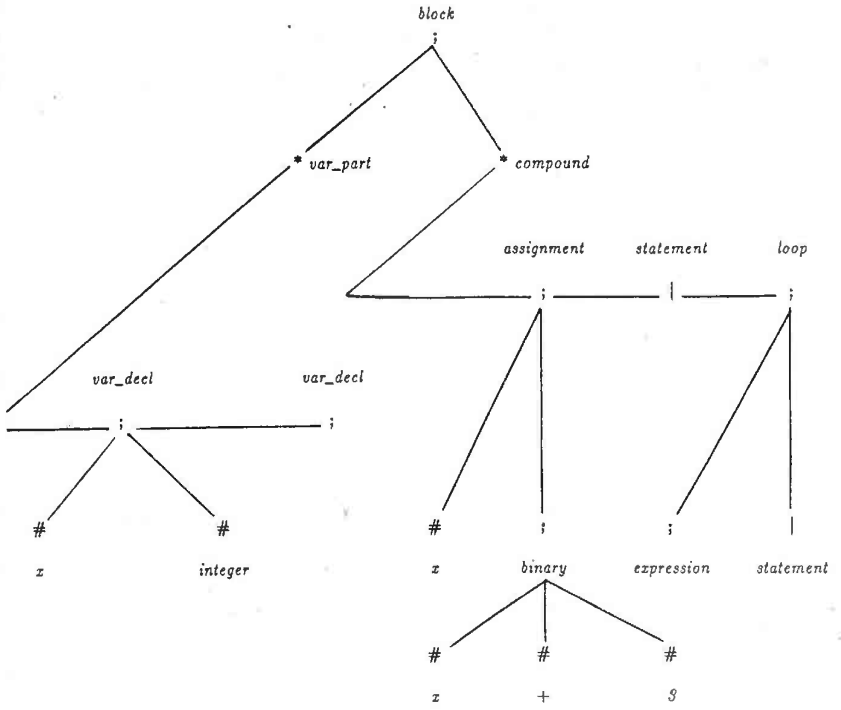


Figure 6: An Abstract Syntax Tree

[A] *block = var_part ; compound*
[L] *var_part = var_decl**
[A] *var_decl = variable_list ; type_description*
[L] *variable_list = variable**
[A] *variable = name*
[A] *compound = statement**
[C] *statement = assignment | conditional | loop | compound*
[A] *assignment = variable ; expression*
[C] *expression = variable | constant | binary*
[A] *binary = expression ; operator ; expression*
[A] *loop = expression ; statement*

The Cépage abstract trees have four kinds of nodes, corresponding to the syntactic type categories: aggregate nodes, choice nodes, list nodes and terminal nodes. In the implementation of an abstract syntax tree, the type of each tree node is known through a pointer to the node of the grammar graph associated with the type.

The leaves of a tree are either:

- nodes corresponding to elements yet to be expanded (like the leaves labeled *var_decl*, *boolean* and *statement* in the tree of figure 6); such nodes may belong to any of the four categories;
- expanded terminal nodes, to which a text has been associated by the user thanks to the text editor.

Note that choice nodes may only appear as leaves, corresponding to the first of these two cases.

We have referred above to the basic data structure as abstract syntactic **forest** rather than tree. The reason is that users will normally manipulate not one but several partially expanded documents or sub-documents simultaneously. Each such element is represented by an abstract tree; their reunion constitutes a forest. At each time, only one element is active; the list of all available elements is contained in a **catalog**. Users may freely add elements to or delete elements from the forest, using the catalog, and go from one element to the others, making them active in turn.

All document manipulations performed in response to user requests are executed by the system as operations on the abstract syntax forest. The available operations are listed in section 5 below.

4.3 - Display form

Although the abstract forest form of the document is best from the system's point of view, users need a clear, concrete representation of the current state of the document.

A full-screen text editor can only show a contiguous excerpt of the document, which makes it very hard for users to keep a global view of the document and the editing process; often, in applications such as program design, users end up going constantly back and forth from one end of the document to the other. Some text editors (e.g. SPF) provide an *elision* mechanism that makes it possible to mask temporarily certain lines of the text in the output, but this feature only yields a marginal improvement in the ergonomics of document preparation.

With a structural editor, it should be possible to do much better. The view offered should itself be structural: users should be provided both with the details of the particular local part of the document in which they are concentrating their attention at any given time, but also with the relevant structural context, for example the enclosing structures in a block-structured

language.

The display mechanism used by Cépage is entirely automatic: after each operation requested by the user, the system will determine the best possible view of the text that it can present to the user, and display it.

The display algorithm, one of the main contributions of Cépage, is rather complex and described in another paper [14]. We shall only mention here some of the main problems involved.

The display task is close to what is known as **pretty-printing**, i.e. printing of a program in a form suggestive of its syntactic structure. However, most published discussions of pretty-printing (see e.g. [15]) are of little interest for an interactive structural editor because they apply only to the case of paper output, for which it is assumed that the output width is fixed but there is no constraint on the number of lines. With a screen editor, both lines and columns are scarce resources: we want to find a representation of a partially expanded document in the limited space available in a given window.

Since the concrete texts may be of arbitrary length, there will in general only be solutions if we allow abbreviating the parts of a document that are the least relevant at display time. Such abbreviation was called **holophrasting** by Hansen [10]. It occurs in Cépage in two different ways:

- We may perform **abstraction** by replacing a possibly complex substructure by its syntactic type name. This was done with the <Statement> in roman on figures 1 to 4.
- In a long list, we may perform **collapsing** by replacing a certain number of list elements, say 35 statements in a *<Statement_list>*, by the mere mention

  <35 Statements>

The aim of the Cépage display algorithm is thus to un-parse the abstract syntax tree into a concrete form that will fit into the available window area, performing abstraction and collapsing as necessary.

Technically, the algorithm produces a list of rectangular *windows* containing the text of the various parts of the document; these windows are handed down to a screen management package, called Screenpack, which takes care of the physical display. Screenpack works on abstract objects called "windows", characterized by attributes which may be modified by Screenpack's primitives.

An interesting possibility is for users to attach comments to nodes; there is a special **explain** display mode, in which the information displayed for an abstracted node is not just a syntactic type indication such as <Statement> (not very informative), but rather the comment attached to the node, if there is one. Note that this feature supports both top-down and bottom-up design: in the former case, the comments will normally be written before the nodes are expanded, in the latter the nodes will be expanded first.

One may imagine systems for displaying programs that go beyond the facilities offered by Cépage and offer true **graphical** views of programs. Research on such tools was recently described in a special issue of IEEE *Computer* [8]. As mentioned previously, the Pecan system [16, 17] also offers graphical views of programs. This line of research is obviously important, and future versions of Cépage may include graphical views. We have, however, included textual views only in both the Cépage prototype and the current industrial version, for four reasons: first, designers are used to manipulating programs and other software-related documents as texts and, regardless of the usefulness of pictures for explanatory purposes, text remains the ultimate basis on which to determine what a program really does, what a specification really means etc.; second, the problem of providing consistent views at a variable level of detail, with zooming and un-zooming capabilities, seemed to us at least as important as the inclusion of graphical facilities; third, graphic programming is still at the research stage, as the articles in [8] clearly show, and we are interested in producing a practical product for today's software professionals in industry; finally, the variable-level display problem seemed difficult enough with text, as we learned by solving it for Cépage [14]: so it was reasonable to first limit ourselves to textual views before we went to graphical representations.

4.4 - The Library

The library is an external data structure that makes it possible to store and retrieve partial designs. Thus an editing session may be interrupted at any time and re-started later. The library is organized as a database, where documents may be retrieved by name.

Technically, abstract syntax trees are stored in the library in an extended Polish form. Trees, however, are not the only thing to store: since the language description is entirely parameterizable, a suitable external form must also be found for storing and restoring grammar graphs, and care must be taken to ensure that each tree is stored together with a reference to the appropriate grammar graph; an abstract syntax tree without a grammar graph is as meaningless as a dinner without cheese (*un repas sans fromage est comme une belle à qui il manque un oeil* [2]).

4.5 - The External Form

When a document is ready (completely expanded), the system must be able to generate a text form suitable for handling by other tools. This is done by simply using the standard display algorithm, with its output directed to a file or printer and its parameters set up in such a way that the number of available lines is considered infinite and no abstraction or collapsing may occur.

5 - THE FUNCTIONS OF CEPAGE

To allow the reader to get a better grasp of the whole scope of Cépage, we now give a systematic list of the functions that are or will be supported by the current version of Cépage.

5.1 - Moving around

A basic set of functions makes it possible to move around a document, by climbing along the corresponding abstract syntax tree:

- up (to parent)
- down (to i-th child)
- left (to sibling)
- right (to sibling)
- existence tests: is there a parent, a left sibling, a right sibling? How many children?

The names above refer to the abstract syntax forest. We have been very careful, however, to make Cépage usable by non-sophisticated users who do not necessarily know about trees and forests; thus the names of the options in the basic menu (see figure 1 above) are not *up, down, left* and *right*, but (respectively) *Move out, Move in, Move to previous* and *Move to next*, expressing the move in user's terms rather than system terms.

5.2 - Marking

The marking commands make it possible to take note of positions in the document while moving around, and to come back to them later.

There are three such commands: **Mark** marks the current position in the document; **Back** returns to the most recent position to which one has not yet returned; **Forth** cancels the effect of the most recent *Back* command that has not yet been canceled in this fashion.[5]

[5] The effect of commands such as *Back* and *Forth* is rather awkward to explain in natural language. The same applies to *Undo* and *Redo* (see below). We have written formal specifications of these commands, which el-

5.3 - Expansion

The expansion function makes it possible to expand a previously unexpanded node of the abstract syntax tree. It is executed according to the information contained in the grammar graph. For an aggregate or list node, no user input is needed; for a choice node, the user must make a selection between the various possibilities (see 5.7 and 5.8 below); for a terminal node, he must enter the text to be attached to the node.

5.4 - Cancel/Modify

Canceling a expansion puts an expanded node back into the unexpanded state. In the case of a choice node, the *Modify* function allows the user to make a new selection; as much as possible of the initial expansion will be carried over to the new one (for example, when transforming an **if ... then ... else ...** statement into a **while** loop, the boolean expression and the **then** part of the conditional statement will be transferred to the loop).

5.5 - Comment/Explain

The **Comment** command attaches a comment to a node (expanded or not). The **Explain** command changes the display mode so that comments will be displayed with particular emphasis.

5.6 - Search/Replace

Search and *Replace* correspond to traditional editor functions. In Cépage, however, the search pattern and (in the *Replace* case) the replacement are structured elements similar to the document being edited: the editor is called recursively to enable the user to define them (in special windows).

5.7 - Selection

The selection facility allows the user to make a choice among a set of predefined possibilities and allows the system to determine which item was selected. The way in which the list of choices is displayed and the user makes his selection (pointing with a mouse in a menu, pressing a function key, typing an ordinary key etc.) depends on the terminal hardware.

5.8 - Parsing

The parsing function makes it possible to read a text typed in by the user and to build the corresponding syntactic structure (subtree of the abstract syntax tree). The text can be incomplete: the parsing method used in Cépage allows the system to fill in the missing parts if the text typed is incomplete but unambiguous.

5.9 - Undo/Redo

Undo makes it possible to back up to previous states of the editing session by canceling the effects of previously issued commands. **Redo** cancels such a cancellation.

---

iminate any potential ambiguity.

5.10 - Record/Replay

The record/replay facility makes it possible to archive the succession of commands issued during an editing session and to replicate them. It thus allows recovering from a system crash.

5.11 - Catalog management/Copy

Catalog management keeps several documents during an editing session, one of which is the "active" document, the others constituting the "catalog"; this function allows the user to select an element of the catalog as the new active document, to copy part of the active document into a new entry of the catalog, or to copy an element of the catalog onto an unexpanded node of the active document.

5.12 - Delimit

The delimiting function enables the user to define a part of a document, to be used as parameter for a function such as cancel, copy etc. Since the "moving around" functions are particularly simple to invoke, delimiting is mainly useful for selecting sublists.

5.13 - Save/Restore

The save/restore function copies documents (which may be partially or totally expanded) from memory to files and back, using an appropriate external representation.

5.14 - Library management

Library management maintains databases of (partially or totally expanded) documents, stored under the external representation mentioned above.

5.15 - Generation

The generation function creates textual versions of totally expanded documents.

5.16 - Language description

The language description function makes it possible to translate descriptions of languages to be supported by Cépage into their internal representations (grammar graphs). The descriptions must be expressed in a language called LDL (Language Description Language), not further described in this paper.

5.17 - Interactive language description and modification

The interactive language description and modification function is similar to the previous one but uses Cépage itself to enter and modify LDL descriptions (grammars). It thus relies on a grammar graph obtained by applying the previous function to the description of LDL in LDL. This function provides for incremental language modification, i.e. makes it possible to construct and modify a grammar graph in a stepwise fashion, as the corresponding LDL description is being developed and updated.

5.18 - Semantic checking

The semantic checking function makes it possible to perform verifications on documents; it is only applicable if the language description includes the definition of the corresponding semantic constraints.

5.19 - Execution

The execution function makes it possible to execute the active document, considered as an executable program. It is only applicable if the language description includes dynamic semantics for each operand type. A partially expanded document may be executed: when execution reaches an unexpanded element, the user is interactively asked to provide the results of the execution. This facility is a first step towards making Cépage into a tool for rapid prototyping and program testing.

5.20 - Display

The display function displays an abstract syntax tree or subtree in a given window area, finding the best representation it can (a detailed description of the display algorithms for Cépage may be found in the paper [14]).

5.21 - Library of primitives

The library of primitives is a set of procedures which enable outside programs to access all of the above Cépage functions and the Cépage data structures. By making these Cépage "internals" accessible to other software tools, it is planned that Cépage will be used as the kernel of a more complete software environment, in which tools of various kinds (e.g. for static program analysis, complexity analysis, program transformation, testing, text processing, etc.) will be able to take advantage of the basic data structures and functions provided by Cépage.

6 - THE NEXT STEP: PATTERN-BASED INTERACTIVE PROGRAM GENERATION

We have emphasized three aspects of Cépage:

- editor, i.e. system for creating and modifying documents at the source language level;
- program development system, with facilities for program checking, testing, and rapid prototyping;
- basis for a programming environment.

These are the short term goals. To conclude with a more futuristic view, we will now present a more remote but very promising application of this system towards solving the problem of **software reusability**. We may call the Cépage solution *pattern-based interactive program generation*.

Most of the software being written today is of a repetitive nature: there exist a small number of basic program patterns (counting, searching, sorting, comparing, exchanging, assigning, creating...) on which programmers compose endless variations. Most of this work is done at the lowest reasonable level, that of common languages; the use of shared, standard components is not, despite a few exceptions such as libraries of numerical software, commonplace. This situation stems in part from the fact that each new situation may be slightly different from the ones encountered previously. For example, even though most search routines share a general organization (go to the beginning of the table, loop until either the required element has been found or the subset of the table in which it may appear has been exhausted, report "found" or "absent"), the representation details will considerably vary from one case to the next.

It is not easy to construct software components that provide a suitable answer to the problem of reusability. Consider the simple problem of providing the users of a computing center with a tool for sorting arrays. Assume that the algorithm is chosen to be, say, Quicksort, which is well explained in computer science textbooks, so that one does not have to worry about

this aspect of the question. A particular problem instance is characterized by how elements will be compared and how they will be exchanged. The solutions open to a software toolsmith are the following:

- A - Provide procedures for the most frequently occurring cases: e.g. increasing and decreasing sort of integer, real, etc. arrays.
- B - Provide a single procedure (or operating system command) with many parameters or options.
- C - Provide a single procedure with two procedure parameters, corresponding to the comparison criterion and the exchange mechanism.
- D - Have a sorting procedure "skeleton" and manually create a tailor-made version for each user who requests it, filling in his particular sorting criterion and exchange mechanism, with the help of a text editor.
- E - As D, but use a macro-processor to generate the various versions.

None of these solutions solves the reusability problem satisfactorily. Solution A is too partial; in many cases, the users will want to sort an array of pointers, leaving the elements themselves in place, or use only part of the elements as keys, etc., so it is unlikely that many actual cases will be covered by the library routines. In solution B, the options may cover a larger number of cases, but the tool will require coding many options, thus using a reference manual, a cumbersome and error-prone process. Solution C will work but with great inefficiency, since the procedures passed as parameters will be called repeatedly in the inner loops of the sorting program; the overhead, which is typically a factor of 10, will be unacceptable in many cases. Solution D, using an editor to generate tailor-made versions, is tedious and error-prone. Solution E implies learning the conventions of the particular macro-processor on hand, which may be at odds with those of the programming language used, even if they were designed by the same group[6]; furthermore, macro-processing is not interactive: the user must first provide actual arguments in the adequate formalism, then wait for the macro-processor to generate a text for inclusion in his program.

Structural editing may provide a better solution. A simple idea is to apply the notion of abstract syntax. A sorting program may be defined as belonging to the following syntactic type:

sorting = comparison ; exchange

One can thus envision a simple extension of the editing process in which the user interactively describes the comparison criterion and exchange mechanism to be used in a particular instance, and the system generates the appropriate sorting procedure by the same expansion mechanism (abstract to concrete) which was used to automatically produce the displayable form

if *c* **then** *A* **else** *B* **end**

from the description of the language, the user providing only *c*, *A* and *B*. The only difference is that the amount of text generated by the system will be proportionally larger in the case of program generation.

We believe that such interactive, pattern-directed program generation is possible in the Cépage framework as presented above. The basic mechanisms are already present; in particular, since the language is a modifiable parameter, it is possible to extend the basic constructs such as *conditional, loop* etc. with **libraries** of program patterns such as *search, sort,* or even *payroll,* etc. Such patterns will be defined in the same way as basic language constructs: by their abstract and concrete syntax.

The idea of program patterns is close to the concept of "plans" used in the Programmer's Apprentice project [20]. We think, however, that reusable, parameterizable program modules

[6] For example, on Unix, the macro-processor embedded in the C compiler [11] and the M4 macro-processor [12] have different conventions regarding parentheses, commas, reserved words etc.

can be implemented in the Cépage framework without recourse to the Artificial Intelligence techniques used in the Programmer's Apprentice.

7 - ACKNOWLEDGEMENTS

Cépage would not have existed if we had not read about the pioneer structural editors: EMILY, Mentor, Gandalf, the Cornell Program Synthesizer. It is a pleasure to acknowledge their influence and, more specifically, that of several discussions with Gilles Kahn, of the lectures by Nico Habermann at the EDF-CEA-INRIA Summer School in Le-Bréau-en-Yvelines (July 1982), and of discussions with David Notkin and Tim Teitelbaum at the Simula workshop on Programming Environments in Lund, Sweden (February 1983).

Outside the domain of structural editors, we have also benefited from the ideas of the Programmer's Apprentice system [20], from object-oriented design and programming as pioneered by Simula 67 [1], and from the general approach to interaction embodied in the Smalltalk system [7].

Most of the ideas come from the systems mentioned. We think our main contributions are in the user interface, in the design of the display algorithm [14], in the idea that the language should be an interpreted, easy to modify parameter, and, more generally, in our ambition to take structural editing out of the laboratory and make its exciting potential available to practicing programmers in ordinary industrial environments.

References

1. Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard, *Simula Begin*, Studentliteratur and Auerbach Publishers, 1973.
2. Brillat-Savarin, quoted by Gustave Flaubert in *Le Dictionnaire des Idées Reçues*, Lemerre, Paris, 1881.
3. Stuart K. Card, Thomas P. Moran, and Allen Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale (New Jersey), 1983.
4. Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang, "Environnement de Programmation Mentor: Présent et Avenir," in *Actes des Troisièmes Journées Francophones sur l'Informatique*, Genève, 1981.
5. Véronique Donzeau-Gouge, Gilles Kahn, Bernard Lang, and Bertrand Mélese, "Documents Structure and Modularity in Mentor," *SIGPLAN Notices (Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, 23-25 April 1984, Ed. Peter Henderson)*, vol. 19, no. 5, pp. 141-148, May 1984. (This issue is also Software Engineering Notes, vol. 9, no. 3)
6. Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, ed. David R. Barstow, Howard E. Shrobe, Erik Sandewall, pp. 128-140, McGraw-Hill, New York, 1984.
7. Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading (Massachusets), 1983.
8. Robert B. Grafton and Tadao Ichikawa (Editors), *Graphical Programming Techniques*, Special issue of IEEE *Computer*, Vol. 18, no. 8, August 1985.
9. Nico Habermann et al., *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh (Pennsylvania), 1982.
10. Wilfred J. Hansen, "Creation of Hierarchic Text with a Computer Display," ANL-7818, Argonne National Laboratory, Argonne (Ill), 1971. (Also as dissertation, Computer Science Department, Stanford University, June 1971)
11. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
12. Brian W. Kernighan, *The M4 Macro-Processor*, Bell Laboratories, 1978.
13. Bertrand Meyer and Jean-Marc Nerson, "Cépage : Un Editeur structurel Pleine Page," in *Second Colloque de Génie Logiciel (Second Conference on Software Engineering)*, pp. 153-158, AFCET, Nice (France), 1984. English translation: "CEPAGE, a full-screen structured editor" in *Software Engineering: Practice and Experience*, North Oxford Academic, Oxford, 1984, pp. 60-65.
14. Bertrand Meyer, Jean-Marc Nerson, and Soon Hae Ko, "Showing Programs on a Screen," *Science of Computer Programming*, vol. 5, no. 2, pp. 111-142, 1985.
15. Derek C. Oppen, "Prettyprinting," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 4, pp. 465-483, October 1980.
16. Steven P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," in *Proceedings of Seventh International Conference on Software Engineering*, pp. 324-333, Orlando (Florida), March 26-29, 1984.
17. Steven P. Reiss, "Graphical Program Development with PECAN Program Development System," *SIGPLAN Notices (Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, 23-25 April 1984, Ed. Peter Henderson)*, vol. 19, no. 5, pp. 30-41, May 1984. (This issue is also Software Engineering Notes, vol. 9, no. 3)

18. Ben Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer (IEEE)*, vol. 16, no. 8, pp. 57-69, August 1983.

19. Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563-573, September 1981.

20. Richard C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing," in *Interactive Programming Environments*, ed. David R. Barstow, Howard E. Shrobe, Erik Sandewall, pp. 464-486, McGraw-Hill, New York, 1984. Originally in IEEE Transactions on Software Engineering, SE-8:1, January 1982

AN APPLICATION OF PROGRAM TRANSFORMATION TO SUPERCOMPUTER PROGRAMMING

Alain Bossavit

(Electricité de France, Direction des Etudes et Recherches[1])

Bertrand Meyer

(University of California, Santa Barbara[2])

ABSTRACT

We show how a sequence of systematic program transformations may be used to derive an efficient, vectorizable program (to be used on vector computers such as the Cray machines) from an initial version which is mathematically simple but recursive and very inefficient.

The example chosen is that of cyclic reduction. We start with a description of the algorithm which follows directly from a mathematical analysis of the problem and is expressed in terms of the operations of the "vector machine", specified as an abstract data type; we end up with an Ada package.

We discuss the advantages and limitations of Ada with respect to vector programming and raise some issues concerning the use of program transformations in software design methodology.

A first version of this paper was presented as an invited talk at the Second VAPP Conference (Vectors and Parallel Processors in Computational Science), Oxford (Great Britain), August 20-24, 1984. The present version will be published in a special issue of *Computer Physics Communications* devoted to the proceedings of that conference.

[1] EDF-DER Service IMA, 1 avenue du Général de Gaulle 92141 Clamart (France)
(1) 765 41 40.

[2] Department of Computer Science, University of California Santa Barbara 93106 (USA)
(805) 961-4321
...lucbvax!ucsbcs!bpm (uucp-usenet);
bpm@ucsb (Arpanet+CSnet)
(on leave from EDF)

Table of Contents

1. BACKGROUND

In previous work, we have investigated the application of modern software engineering techniques to the design of vector programs (e.g. [15,5,6,7] etc.). Our general approach has been to investigate supercomputer programming not as a set of recipes designed to yield maximum performance on some or other specific machine architecture, but rather as a systematic design activity, in which the concern for efficiency must not offset other important software qualities such as correctness, reliability, extensibility, portability and others.

Techniques which can be applied towards this goal include assertion-guided stepwise program development [10] and the use of abstract data types for the specification of "virtual vector machines" as models of actual vector processing hardware.

This paper continues our previous efforts by studying the application of another well-known program construction method, program transformation, to the development of an efficient vector program corresponding to an important algorithmic concept, cyclic reduction. We start from a correct but very inefficient program, obtained as a straightforward implementation of the basic mathematical idea and expressed in terms of high-level operations of the abstract "vector machine"; we then perform a series of transformations, each aimed at removing some of the inefficiency while preserving the semantics of the program. The final version, for which we offer an Ada implementation, is an efficient, readily vectorizable program.

2. THE TOTAL REDUCTION PROBLEM

2.1. Statement of the Problem

Consider a set S with a binary operation, written $\oplus$, which gives S the structure of a monoid, i.e. $\oplus$ is associative and has a zero element, written 0. Note that $\oplus$ is *not* required to be commutative. Elements of S will be called **scalars.**[3]

We define $V = VECTOR\ [S]$, the set of finite sequences of elements of S. An element v of V, called a **vector**, is of the form

$$v = \langle v_1, v_2, \ldots v_n \rangle$$

where $v_i \in S$ for $i = 1, 2, \ldots n$. The number of elements of a vector v is written $|v|$.

We define the **shift** operation

$$\tau : V \longrightarrow V$$

such that

$$\tau(\langle v_1, v_2, \ldots v_n \rangle) = \langle 0, v_1, v_2, \ldots v_n \rangle$$

The total reduction problem is, given a vector $a \in V$, to find another vector $x \in V$ such that

$$x = a \oplus \tau x \qquad /1/$$

which can also be written, in scalar terms:

$$\begin{cases} x_1 = a_1 \\ x_i = a_i \oplus x_{i-1} \end{cases}$$

or equivalently:

$$x_i = a_i \oplus a_{i-1} \oplus a_{i-2} \oplus \ldots \oplus a_1$$

for $i = 1, 2, \ldots |a|$.

[3] This use of the word "scalar" does not quite conform to standard mathematical usage, but is common in discussions of vector programming.

2.2. Applications

The total reduction problem, as defined by /1/ above, has several applications. The most obvious ones are the sum of the elements of a_i obtained by taking ordinary addition for Θ, and linear recurrences, which may be written as

$$\begin{vmatrix} x_i \\ 1 \end{vmatrix} = \begin{vmatrix} a_i & b_i \\ 0 & 1 \end{vmatrix} \Theta \begin{vmatrix} x_{i-1} \\ 1 \end{vmatrix}$$

which is an instance of the total reduction problem obtained by taking for Θ the product of 2×2 matrices.

But some classes of non-linear recurrences fall into the same model, a straightforward generalization is

$$x_i = \frac{a_i * x_{i-1} + b_i}{c_i * x_{i-1} + d_i}$$

which can be put into the form of /1/ by again taking for Θ the product of 2×2 matrices and writing the equation as

$$x_i = u_i / v_i$$

where

$$\begin{vmatrix} u_i \\ v_i \end{vmatrix} = \begin{vmatrix} a_i & b_i \\ c_i & d_i \end{vmatrix} \Theta \begin{vmatrix} u_{i-1} \\ v_{i-1} \end{vmatrix}$$

A useful particular case where this is applicable is **Cholesky factorization**: consider a symmetric matrix with diagonal

$$<d_1, d_2, \ldots d_n>$$

and subdiagonal

$$<s_1, s_2, \ldots s_{n-1}>.$$

The recurrence to be solved for Cholesky factorization is

$$\begin{cases} b_{i-1}^2 + a_i^2 = d_i \\ b_i * a_i = s_i \end{cases}$$

i.e. by eliminating b_i:

$$a_i^2 = d_i - \frac{s_{i-1}^2}{a_{i-1}^2}$$

which is a problem of the above form if we take $x_i = a_i^2$.

3. THE VECTOR MACHINE

3.1. Vector operations

Equation /1/ does not seem to lend itself naturally to efficient solution on vector processors such as the Cray-1 or Cray-XMP, which favor the execution of "extension" operations [15, 5]. Roughly speaking, extension operations are those which can be executed in parallel on all the elements of a vector (or more generally, in the case of the Cray machines, on whole vector slices). A typical extension operation is the addition of two vectors, element by element.[1]

[1] It should be noted that on the Cray machines or on the CDC Cyber 205 vector operations are not actually performed on all elements in parallel, but rather use pipelining. For most practical purposes, however, pipelining may be considered as a form of parallelism.

Such operations on vectors may be executed by vector hardware much more efficiently than by just applying repetitively their non-vector, or "scalar" counterparts. More precisely, a scalar operation which takes time S when applied to one element will take time

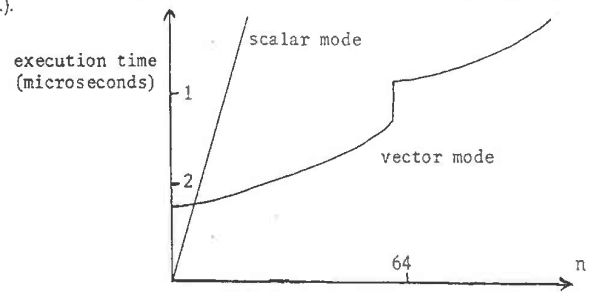
$$t_{scal}(n) = n * S$$

when applied to a vector of n elements. A true vector operation, when applied to this vector, will take a time approximately equal to

$$t_{vect}(n) = U + n * V$$

where U is the start-up time and V is the asymptotic unit vector time. On a vector machine, of course, V must be significantly less than S.

The performance of vector addition in both scalar and vector mode on the Cray-1 is illustrated by the diagram below. Vector mode becomes better than scalar mode for vector lengths $n > U / (S-V)$. The non-linearity of actual vector processing time, which is apparent on the figure, is due to the fact that the Cray processes vectors by slices of maximum length 64, hence the discontinuity at $n = 64$ (and also 128, 192 etc.).



The performance of an operation executed in vector mode may be characterized by two parameters [13]:

- the asymptotic vector speedup $p = \frac{S}{V}$
- the "half-performance length" $n_{\frac{1}{2}} = \frac{U}{V}$, defined as the value of n for which the per-element performance is half the asymptotic one, i.e. $\frac{U + n * V}{n} = 2 * V$; this parameter gives an idea of the minimum length for which the benefits of vector mode offset the penalty incurred for short vectors because of the startup time.

On a Cray-1, depending on the operation, p varies between 7 and 10 and $n_{\frac{1}{2}}$ between 20 and 30.

Only those parts of a program which conform to certain rules may be executed in vector mode and thus achieve high performance. For Fortran programs on the Cray-1, the rules are the following [15]:

- 1. only "DO" loops are "vectorizable";
- 2. these loops may only contain "primitive" operations such as assignment and arithmetic or boolean operations (no jumps, etc.);
- 3. the data elements accessed during successive loop iterations must be regularly spaced in memory, i.e. array indexes must be linear functions of the loop index;
- 4. no "backward dependency", in which a statement updates an array value $a(i)$ and uses a previous value of the same array, $a(i-p)$ (for some $p > 0$), is permitted;
- 5. no "cross dependency", in which an array value may be updated by one statement of the loop and used by another, is permitted.

In the last two cases, vectorization is inhibited by the compiler not because the hardware could not carry out the computation in vector mode, but because the vector semantics of the program may be different from the standard (sequential) semantics implied by Fortran and other common languages. If, on the other hand, one feels certain that the dependency is only apparent, for instance if the element updated in a loop with index i is $a(2*i+1)$ and the value used is that of $a(2*i)$ (so that the array slices updated and used are in fact disjoint), then one may force vectorization; the Cray Fortran compiler will accept a special directive, *IVDEP*, to that effect.

The above rather stringent rules seem to preclude the vectorization of many simple algorithms; for example, the formula which we have given for total reduction, i.e. /1/ above, clearly implies repeated backward dependencies.

In order to obtain vectorizable versions of this and other algorithms, more perspective is needed on the "vector machine" and the operations it may perform.

3.2. An Abstract Model

Rather than studying at the scalar (e.g. Fortran) level what can be vectorized and what cannot, it is preferable to provide a formal model of the machine at the appropriate level of abstraction. Here we consider a vector computer as a virtual machine associated with an abstract data type, type $V = VECTOR\ [S]$, and capable of performing a certain number of operations.

There is in fact probably no such thing as *the* vector machine, but rather various models adapted to various applications. We thus tailor our specification to the problem at hand. Rather than giving a complete formal description of the abstract data type "vector", we concentrate on some useful operations and their essential properties.

Operation	Type	Notation	Properties
Zero Vector	V	0	All elements *zero*
Length	$V \longrightarrow Integer$	$\|v\|$	
Access to Elements	$V \times Integer \longrightarrow S$	v_i	
Extension of a Scalar Operation $\oplus$	$V \times V \longrightarrow V$	$v \oplus w$	let $z = v \oplus w$: $\|z\| = min(\|v\|,\|w\|)$; $z_i = v_i \oplus w_i$ ($i \in 1..\|z\|$)
Shift	$V \longrightarrow V$	τv	$\|\tau v\| = \|v\| + 1$; $(\tau v)_i :=$ v_{i-1} for $i > 1$, 0 for $i = 1$
Odd Part	$V \longrightarrow V$	Ov	$Ov_i = v_{2i-1}$
Even Part	$V \longrightarrow V$	Ev	$Ev_i = v_{2i}$
Merge into Odd and Even Parts	$V \times V \longrightarrow V$	*alternate* (v, w)	Let $z =$ *alternate* (v, w): $Oz = v$; $Ez = w$

On a vector computer such as the Cray-1, all the operations in the above table (except for "length" and "access to element" which require constant time) are "extension operations" which can be executed in vector mode. It should be noted, however, that some vector computer architectures may be more restrictive: the CDC Cyber 205, for instance, requires array elements to be contiguous, not just equally spaced, so that operations such as "odd part", "even part" and "merge" do not qualify.

The above list of operations is by no means exhaustive; more complete lists may be found in e.g. [6,7]. It should also be noted that for some applications it may be useful to introduce operations extracting other "slices" than just the odd and even parts. The operations given here will suffice, however, for our purposes.

Among the abstract properties of these operations which are particularly interesting are the following (for any vectors $v, w \in V$):

$$\begin{cases} E\tau v = Ov & /i/ \\ O\tau v = \tau Ev & /ii/ \\ O(v \oplus w) = Ov \oplus Ow & /iii/ \\ E(v \oplus w) = Ev \oplus Ew & /iv/ \\ \tau(v \oplus w) = \tau v \oplus \tau w & /v/ \end{cases}$$

4. CYCLIC REDUCTION

The above properties, expressed at the vector rather than scalar level, provide the key to an efficient solution of the total reduction problem /1/ by a vector algorithm. The idea to be applied here is a very fruitful heuristics, using the concept of recursion and close to techniques such as "red-black ordering" which can be applied to the development of several efficient vector algorithms.

In the "total reduction" equation

$$x = a \oplus \tau x \qquad /1/$$

let us try to reduce the problem size by a factor of 2 by applying operators O and E (odd and even parts) to both sides, yielding:

$$Ox = O(a \oplus \tau x)$$

$$Ex = E(a \oplus \tau x)$$

i.e. by applying properties /i/ to /iv/:

$$Ox = Oa \oplus \tau Ex \qquad /2/$$

$$Ex = Ea \oplus Ox \qquad /3/$$

The interesting fact here is that by substituting the value of Ex, as obtained from /3/, into /2/, and using the associativity of $\oplus$ combined with property /v/ above, we obtain a new equality:

$$Ox = (Oa \oplus \tau Ea) \oplus \tau Ox \qquad /4/$$

which is a new instance of the total reduction problem, applied to the new vector variable Ox, a being replaced by $Oa \oplus \tau Ea$. This new instance uses vectors of approximately half the size of the original ones.

We thus have the essential ingredients for an efficient recursive algorithm, known as **cyclic reduction**:

- for vectors of length 0 or 1, the result x will be just a :
- For larger vectors, we apply the algorithm recursively, using formula /4/, to obtain Ox ; formula /3/ then yields Ex ;
- we obtain x by merging these two vectors (*alternate* operator).

5. PROGRAM DEVELOPMENT

5.1. First Procedural Version

The first version of the procedure is a direct translation of the basic mathematical definition. We use an Ada-like notation.

```
procedure total_reduction_1 (a : in VECTOR ; z : out VECTOR)
    var oddpart, evenpart : VECTOR
begin
    if |a| ≤ 1 then
        z := a
    else - - |a| > 1
        total_reduction_1 (Oa ⊕ τEa, oddpart) ;
        evenpart := Ea ⊕ oddpart ;
        z := alternate (oddpart, evenpart)
    end if
end procedure - - total_reduction_1
```

The above version is correct but grossly inefficient for several reasons:

- the procedure is recursive;
- it has local vector variables (*oddpart* and *evenpart*) which must be allocated anew for each recursive instance of the procedure;
- it uses two parameters, an input a and an output z, whereas in practice one usually prefers to work on a single vector, which is initially the input and will gradually be "transformed" so as to become the output (the initial value being saved if necessary).

We shall get rid of these sources of inefficiency through a stepwise process. To make the successive program transformations clearer, we underline in each version the elements which have been changed from the previous version.

5.2. Removing Extra Variables

Our first transformation is a straightforward one, which gets us a little closer to our aim of working on a single object (z): we note that it is harmless to begin the procedure by the assignment $z := a$ in all cases, not just when $|a| \leq 1$ (in the other case, this assignment will be overridden by the assignments to the odd and even parts of z).

```
procedure total_reduction_2 (a : in VECTOR ; z : out VECTOR)
    var oddpart, evenpart : VECTOR
begin
    z := a ;
    if |a| > 1 then
        total_reduction_2 (Oa ⊕ τEa, oddpart) ;
        evenpart := Ea ⊕ oddpart ;
        z := alternate (oddpart, evenpart)
    end if
end procedure - - total_reduction_2
```


The next simplification is to get rid of the local variables *oddpart* and *evenpart* by extending the notation a little: we now allow assigning vector values directly to the slices Oz and Ez of a vector z. For example, to change the even part of z to y, we shall just write

$Ez := y$

instead of

$z := alternate\ (Oz, y)$

With this new notation, the procedure can be simplified as follows:

```
procedure total_reduction_3 (a : in VECTOR ; z : out VECTOR)
begin
    z := a ;
    if |a| > 1 then
        total_reduction_3 (Oa ⊕ τEa, Oz ) ;
        Ez := Ea ⊕ Oz ;
    end if
end procedure - - total_reduction_3
```

The next obvious step towards the goal of working with only one vector variable is to replace all occurrences of a with z after the initial assignment $z := a$. We have to be very careful here: in the procedure resulting from such a transformation, the same vector z will be used as both an **in** and **out** actual parameter of the recursive call. It should be noted that Hoare's specification of the semantics of recursive procedures [12] specifically excludes this case.

The replacement will be correct, however, if for the time being we assume a copy mechanism for parameter passing. In other words we take **in** to mean "parameter passed by value", i.e. copied upon each procedure call into a variable local to the procedure instance; and we take **out** to mean "parameter passed by result", i.e. copied back, on procedure return, from the local variable. To avoid any confusion resulting from the fact that we are using an Ada-like notation, it should be noted that this mode of parameter passing is *not* the normal Ada mechanism for **in** and **out** parameters.

```
procedure total_reduction_4 (a : in VECTOR ; z : out VECTOR)
begin
    z := a ;
    if |z| > 1 then
        total_reduction_4 (O z ⊕ τE z , Oz) ;
        Ez := E z ⊕ Oz ;
    end if
end procedure - - total_reduction_4
```

5.3. Isolating the Recursion

It is useful now to separate the procedure into two parts: one which uses the initial vector a and one which does not. To this effect, we transform the procedure into a set of two mutually recursive procedures, only the first of which depends on a; the second one, called *internal_part*$_5$, has only z as a parameter, of mode **in out**. Again, this is correct only if we assume a copy mechanism for parameter passing, i.e. an **in out** parameter is copied to (at call time) and from (at return time) a variable local to the procedure instance.

```
procedure total_reduction5 (a : in VECTOR ; z : out VECTOR)
begin
    z := a ;
    internal_part5 (z)
end procedure - - total_reduction5

procedure internal_part5 (z : in out VECTOR)
begin
    if |z| > 1 then
        total_reduction5 (Oz ⊕ τEz, Oz) ;
        Ez := Ez ⊕ Oz ;
    end if
end procedure - - internal_part5
```

We can now isolate the recursion by expanding the call to *total_reduction* in *internal_part*. The effect of this call is to assign the value of the first parameter to the second and to call *internal_part* recursively. By carrying out this expansion, we get rid of the mutual recursion introduced in the previous step: in the new version, only *internal_part* will be (directly) recursive; *total_reduction* remains useful for initialization only.

```
procedure total_reduction6 (a : in VECTOR ; z : out VECTOR)
begin
    z := a ;
    internal_part6 (z) ;
end procedure - - total_reduction6

procedure internal_part6 (z : in out VECTOR)
begin
    if |z| > 1 then
        Oz := Oz ⊕ τ Ez ;
        internal_part6 (Oz) ;
        Ez := Ez ⊕ Oz ;
    end if
end procedure - - internal_part6
```


5.4. Introducing an Integer Parameter

The remarkable feature of the recursive scheme which we have obtained is that the recursive call now has a single and simple actual parameter, Oz, where the formal parameter was z. Thus the sequence of actual parameters in successive recursive calls, starting with the initial call from *total_reduction*$_6$, will be

$$z = a,\ Oz,\ O^2 z,\ \ldots,\ O^m z,$$

where $O^k z\ (k \geq 0)$ is the k-th iterate of O. The value of the exponent for the innermost call is

$$m = 1 + \lfloor \log (|a| - 1) \rfloor$$

(here and in the sequel, logarithms are in base two; for any real number x, $\lfloor x \rfloor$ denotes the floor of x, i.e. the greatest integer n such that $n \leq x$).

This remark suggests a new version in which the explicit parameter to the recursive part is not z itself any more, but k, the number of times operator O must be iterated. Of course all instances of the recursive procedure must be able to work on z; thus we make z a variable global to the recursive procedure. To this end we make procedure *internal_part* local to the non-recursive procedure *total_reduction*.

```
procedure total_reduction7 (a : in VECTOR ; z : out VECTOR)
    var m : NATURAL - - i.e. non-negative integer ;
    procedure internal_part7 (k : in NATURAL)
        - - local to total_reduction7
    begin
        if k ≤ m then
            O^k z := O^k z ⊕ τ E O^(k-1) z ;
            internal_part7 (k+1) ;
            E O^(k-1) z := E O^(k-1) z ⊕ O^k z ;
        end if
    end procedure - - internal_part7
begin - - total_reduction7
    z := a ;
    m := 1 + ⌊log (|a| - 1)⌋ ;
    internal_part7 (1) ; - - initial parameter is one
end procedure - - total_reduction7
```

5.5. Removing the Recursion

These procedures can be further simplified. The body of procedure *internal_part*$_7$ is of the form

if $k \leq m$ **then**

U_k ;

internal_part$_7$ *(k+1)* ;

D_k

end if

where U_k is the statement

$O^k x := O^k x \oplus \tau\, EO^{k-1} x$

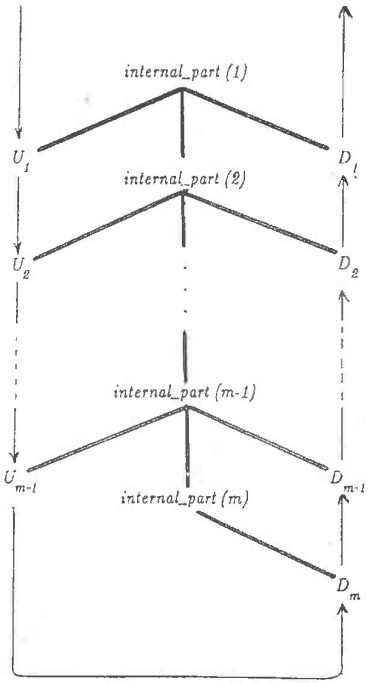
and D_k is the statement

$EO^{k-1} x := EO^{k-1} x \oplus O^k x$

Thus the execution of the successive recursive calls amounts to a traversal of the following tree in the order indicated by the dotted line, i.e. the successive execution of

$U_1, U_2, \ldots U_{m-1}, D_m, D_{m-1}, \ldots D_2, D_1$.

where $m = \lfloor 1+\log(|a|-1) \rfloor$. Note that there is one more instance of D_k than of U_k since U_m is a null statement.



Thus no recursion is needed after all: the body of procedure *total_réduction*$_7$ may be readily represented by

up ; down

where *up* and *down* are two simple loops:

- - *up:*

for $k := 1$ **to** $m-1$ **do**

U_k

end for ;

- - *down:*

for $k := m$ **downto** 1 **do**

D_k

end for ;

(the mnemonics used for the loops reflect the fact that the index k goes up in the first loop and down in the second one).

It is particularly interesting to note that, although the recursion initially seemed quite necessary, it has been completely removed. The above version is truly non-recursive in that it does not seem to contain any hidden recursive feature, for example a stack lurking in the guise of an integer representing an array of binary values as in some iterative implementations (see e.g. [14]) of the Tower of Hanoi, Quicksort, the Deutsch-Schorre-Waite tree traversal algorithm etc.

6. A SCALAR, VECTORIZABLE VERSION

6.1. The Program

It is useful to write U_k and D_k in a form which is closer to how they would be expressed in an ordinary (scalar) programming language, but still easily amenable to automatic vectorization. We define

slice (low, high, step)

where *low, high* and *step* are integers such that $low \leq high$ and $step > 0$, as the set of all integers of the form

$low + k * step$

which fall into the range *low .. high*. Then U_k and D_k can be written as follows:

- - U_k *(i.e.* $O^k x := O^k x \oplus \tau\, EO^{k-1} x$*)* :

forall *i* **in** *slice* $(1 + 2^k, |a|, 2^k)$ **do**

$x[i] := x[i] \oplus x[i - 2^{k-1}]$

end forall

- - D_k *(i.e.* $EO^{k-1} x := EO^{k-1} x \oplus O^k x$*)* :

forall *i* **in** *slice* $(1 + 2^{k-1}, |a|, 2^k)$ **do**

$x[i] := x[i] \oplus x[i - 2^{k-1}]$

end forall

We have used the notation **forall** ... in ... to emphasize the fact that the above are parallel loops: on a vector processor, all the vector operations corresponding to an instance of U_k or D_k can be performed simultaneously.

Note that the backward dependencies in these loops are only "apparent" in the sense of section 3.1 : since both loops are low-level translations of vector operations (U_k and D_k, kept as comments in the above code), the expected interpretation is the vector one (which anyway turns out to be identical to the sequential loop semantics in this case). Thus if a conservative vectorizer such as the Cray Fortran Translator inhibits vectorization of these loops because of the apparent dependencies, the programmer should override the inhibition.

Below is a non-recursive version of *total_reduction* which integrates the various improvements achieved so far. This version would be readily vectorizable by any simple vectorizer (such as CFT, the Cray Fortran Translator, on the Cray-1). A further simplification is obtained by using variables *step* and *half_step*, corresponding to 2^k and 2^{k-1} respectively, in lieu of k.

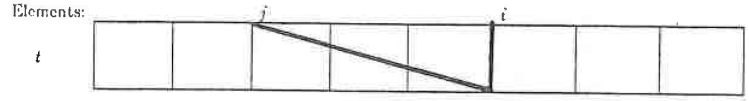
```
procedure total_reduction_9 (a : in VECTOR ; x : out VECTOR)
    var step, half_step : NATURAL ;
        size : NATURAL ; - - size will stand for |a|
begin
    size := |a| ;
    forall i in slice (1, size, 1) do
        x [i] := a [i] ;
    end forall ;
    step := 2 ; half_step := 1 ; - - This corresponds to k := 1
    while step < size do - - U_k
        forall i in slice (1 + step, size, step) do
            x [i] := x [i] ⊕ x [i - half_step]
        end forall ;
        half_step := step ; step := 2 * step
    end while ;
        - - here {1 ≤ half_step < size ≤ step = 2*half_step}
    while step > 1 do - - D_k
        forall i in slice (1 + half_step, size, step) do
            x [i] := x [i] ⊕ x [i - half_step]
        end forall ;
        step := half_step ; half_step := half_step /2
    end while
end procedure - - total_reduction_9
```


6.2. A Timing Diagram

The diagram below may be helpful in visualizing the operations performed on x during an execution of the procedure. It applies to the case $|a| = 9$. The elements are represented horizontally; the vertical axis represents time. Execution of the operation

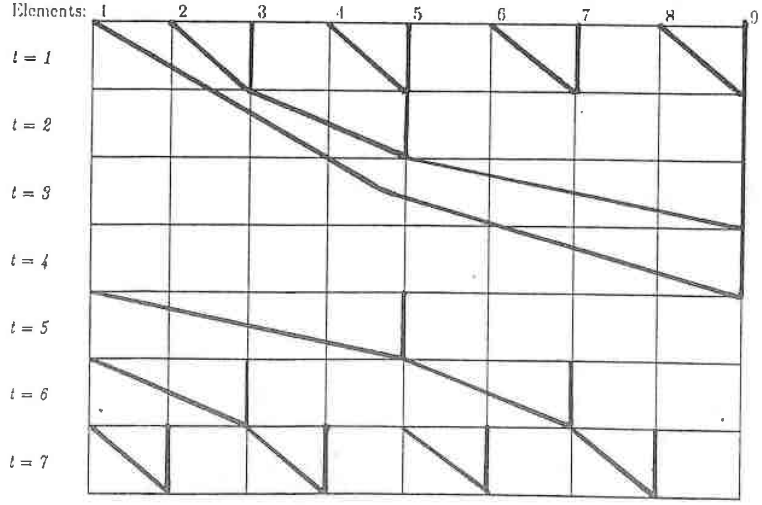
$x[i] := x[i] \oplus x[j]$

at time t is pictured as



The two main loops ("*up*" and "*down*") appear clearly on the diagram: the first one is executed in steps 1 to 3, the second one in steps 4 to 7.

It is interesting to note that this diagram follows directly from the non-recursive version of the procedure; it can also be deduced from the initial recursive version (by expanding the call graph), but the deduction is much more difficult.



Note that there is a minor possibility for extra parallelism, between steps 4 and 5, that our development method has not captured.

The time needed for total reduction of a vector a using cyclic reduction on the Cray is approximatively

$t_{CYCL} = 2*(r-1)*U + (2*(n-1) - r)*V$

where $r = \lfloor \log(|a|) \rfloor$. This time should be compared to $t_{SCAL} = (n-1) * S$ for the trivial algorithm (constants U, V and S were introduced in section 3.1). For the Cray, the cutoff point at which cyclic reduction becomes more efficient is approximately $|a| = 40$.

7. AN ADA VERSION

Below is an implementation of the algorithm as an Ada function, embedded in a generic package. The following points are worth noting:

- the generic mechanism of Ada provides a way to write the package so that it can be applied to various cases; the same generic package can have many instances depending on what the type *SCALAR* and the "+" operation, which corresponds to the operation written $\oplus$ above, are chosen to be: for instance the type *INTEGER* and integer addition, a matrix type and matrix multiplication, etc.

- The Ada generic mechanism is flexible but strictly syntactical: the language provides no way to specify that the actual generic parameters must have predefined semantic properties, for instance that "+" must be associative. A language such as LPG (Language for Generic Programming, [4]) makes it possible to impose such conditions on generic parameters.

- Procedure *ADD_TO_VECTOR* is the one which performs the vector operations (corresponding to U_k and D_k as defined above). These operations must be expressed in scalar form, using loops (**for ... in ... loop ... end loop**). Thus on a vector computer an Ada program such as this one will require the intervention of a vectorizer, similar to those which exist for Fortran (e.g. CFT on the Cray-1), in order to take advantage of the vector computation facilities of the hardware.

- The loop in procedure *ADD_TO_VECTOR* seems to involve a backwards dependency. However, this is only an apparent dependency, as defined in section 3.1, since the loop updates s and uses s - *offset*, but these two slices are disjoint whenever *offset* $\neq$ *s.step*, which is the case for the two calls to *ADD_TO_VECTOR* in the package. This implies, however, that a vectorizing Ada compiler would still have to provide some kind of "vectorize at any risk" directive similar to Cray Fortran's *IVDEP*.

The fact that vector programmers should still resort to such low-level and error-prone techniques in Ada is all the more disappointing that Ada comes close to providing adequate notations for true vector programming: it has vector operations such as vector assignment (used below in the initializing statement $x := a$ of function *TOTAL_REDUCTION*) and the notion of slice; however, an Ada slice must be a contiguous subarray, whereas the slices which we need here are not contiguous, which is why we must use loops.

On the other hand, a language such as Actus [16], explicitly designed for use on vector computers, readily allows for non-contiguous slices, but lacks the generic facility of Ada.

```
generic
    type SCALAR is private ;
    with function "+" (X, Y : SCALAR) return SCALAR is <> ;
package CYCLIC_REDUCTION is
    type VECTOR is array (NATURAL range <>) of SCALAR ;
    function TOTAL_REDUCTION (a : VECTOR) return VECTOR ;
private
    type SLICE is record low, high, step : NATURAL end;
end CYCLIC_REDUCTION ;
```



```
package body CYCLIC_REDUCTION is
    procedure ADD_TO_VECTOR (x : in out VECTOR ;
                             s : in SLICE
                             offset : in NATURAL)
        -- x (s) := x (s) + x (s - offset)
    is
        bottom : constant NATURAL := s.low ;
        top : constant NATURAL := s.high ;
        stride : constant NATURAL := s.step ;
        last : constant NATURAL := (top - bottom) / stride ;
    begin
        for i in 0..last do
            x (bottom + i*stride) := x (bottom + i*stride) + x (bottom + i*stride - offset)
        end for ;
    end ADD_TO_VECTOR ;

    function TOTAL_REDUCTION (a : VECTOR) return VECTOR is
        initial : constant NATURAL := a'FIRST ;
        final : constant NATURAL := a'LAST ;
        size : constant NATURAL := initial - final + 1 ;
        x : VECTOR := a ;
        step : NATURAL := 2 ; half_step : NATURAL := 1 ;
    begin
        UP :
            while step < size loop
                ADD_TO_VECTOR (x, (initial + step, final, step), half_step) ;
                half_step := step ; step := 2 * step ;
            end loop UP ;
                -- here {1 ≤ half_step < size ≤ step = 2*half_step}
        DOWN :
            while step > 1 loop
                ADD_TO_VECTOR (x, (initial + half_step, final, step), half_step) ;
                step := half_step ; half_step := half_step / 2 ;
            end loop DOWN ;
        return x ;
    end TOTAL_REDUCTION ;
end CYCLIC_REDUCTION ;
```

8. CONCLUSION

Transformational programming has been advocated by several authors, e.g. [1, 2, 9, 3, 8], whereas other researchers in software design methodology prefer a more direct approach to the synthesis of programs from specifications [10, 11]. Although we do not wish to enter this debate here, the derivations obtained in this paper may bring some interesting elements.

Even though the sequence of transformations needed to produce the final program may seem overly long and complex, we do not know of any other rigorous way to derive that program. We would be interested to learn of a more direct argument, if there is one.

On the other hand, it is not clear to us whether any of the existing program transformations systems (where the term "system" is taken to denote coherent sets of tools and/or methods) may indeed support the transformations described here.

In any case, we feel that the development presented here is another example of the need for applying systematic techniques to the design of vector programs. Effective supercomputer programming requires a wide range of modern software engineering techniques; program transformation may be one of them.

Acknowledgement

We are grateful to Alan Wilson for the useful comments he made as a referee for this paper.

References

1. Jacques Arsac, "Syntactic Source to Source Transformation and Program Manipulation," *Communications of the ACM*, vol. 22, no. 1, pp. 43-54, January 1979.
2. Robert Balzer, Neil Goldman, and David Wile, "On the Transformational Implementation Approach to Programming," in *Proceedings Second International Conference on Software Engineering*, pp. 223-231, 1976.
3. F.L. Bauer, M. Broy, W. Dosch, R. Gnatz, F. Geiselbrechtinger, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nicki, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner, *The Munich Project CIP*, Technische Universität München, Munich (Germany), December 1983.
4. Didier Bert, "Manuel de Référence du Langage LPG, Version 1.2," Rapport R-408, IFIAG, IMAG Institute (Grenoble University), Grenoble, December 1983.
5. Alain Bossavit and Bertrand Meyer, "The Design of Vector Programs," in *Algorithmic Languages*, ed. Jaco de Bakker and R.P. van Vliet, pp. 99-114, North-Holland Publishing Company, Amsterdam (The Netherlands), 1981.
6. Alain Bossavit, "The Vector Machine: An Approach to Unsophisticated Programming of Linear Algebra Algorithms on Vector Computers," in *Proceedings of IFIP TC2 WG 2.5 (Numerical Software) Working Conference on PDE Software: Modules, Interfaces and Systems*, Söderkoping (Sweden), August 1983.
7. Alain Bossavit, "Programming Discipline on Vector Computers: 'Vectors' as a Datatype and Vector Algorithms," in *Proceedings of Conference on The Use of Supercomputers in Theoretical Science*, Antwerpen (Belgium), July 30 - August 1, 1984.
8. James M. Boyle and Monagur N. Muralidharan, "Program Reusability through Program Transformation," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 574-588, September 1981.
9. John Darlington and Rod M. Burstall, "A System which Automatically Improves Programs," *Acta Informatica*, vol. 6, pp. 41-60, 1976.
10. Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs (New-Jersey), 1976.
11. David Gries, *The Science of Programming*, Springer-Verlag, Berlin, 1981.
12. C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," in *Symposium on the Semantics of Programming Languages, Lecture Notes in Mathematics*, ed. Erwin Engeler, vol. 188, pp. 103-116, Springer-Verlag, Berlin, 1971.
13. C.W. Hockney and Christopher R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol (Great Britain), 1981.
14. Bertrand Meyer and Claude Baudoin, *Methodes de Programmation*, Eyrolles, Paris, 1978.
15. Bertrand Meyer, "Un Calculateur Vectoriel: Le Cray-1 et sa Programmation (Version 2)," Atelier Logiciel no. 24, HI-34552/01, Electricité de France, June 4, 1980.
16. Ron Perrott, "A Language for Vector and Array Processors," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 2, pp. 177-195, October 1979.

On Formalism in Specifications

Bertrand Meyer, University of California, Santa Barbara

A critique of a natural-language specification, followed by presentation of a mathematical alternative, demonstrates the weakness of natural language and the strength of formalism in requirements specifications.

Specification is the software life-cycle phase concerned with precise definition of the tasks to be performed by the system. Although software engineering textbooks emphasize its necessity, the specification phase is often overlooked in practice. Or, more precisely, it is confused with either the preceding phase, definition of system objectives, or the following phase, design. In the first case, considered here in particular, a natural-language *requirements document* is deemed sufficient to proceed to system design—without further specification activity.

This article emphasizes the drawbacks of such an informal approach and shows the usefulness of formal specifications. To avoid possible misunderstanding, however, let's clarify one point at the outset: We in no way advocate formal specifications as a *replacement* for natural-language requirements; rather, we view them as a *complement* to natural-language descriptions and, as will be illustrated by an example, as an aid in *improving* the quality of natural-language specifications.

Readers already convinced of the benefits of formal specifications might find in this article some useful arguments to reinforce their viewpoint. Readers not sharing this view will, we hope, find some interesting ideas to ponder.

The seven sins of the specifier

The study of requirements documents, as they are routinely produced in industry, yields recurring patterns of

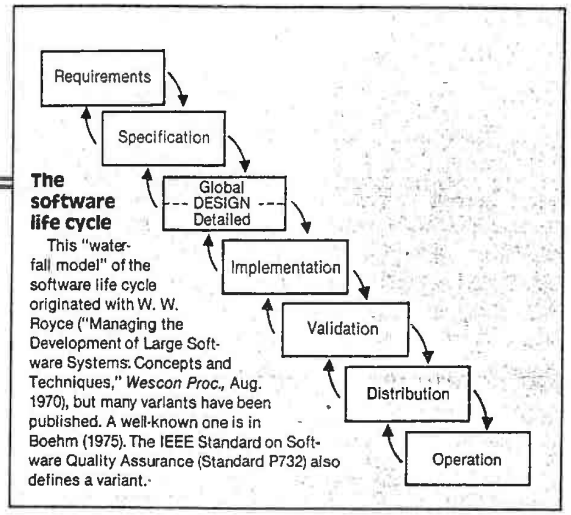
0740-7459/85/0001/0006$01.00 © 1985 IEEE

deficiencies. Table 1 lists seven classes of deficiencies that we have found to be both common and particularly damaging to the quality of requirements.

The classification is interesting for two reasons. First, by showing the pitfalls of natural-language requirements documents, it gives some weight to the thesis that formal specifications are needed as an intermediate step between requirements and design. Second, since natural-language requirements are necessary whether or not one accepts the thesis that they should be complemented with formal specifications, it provides writers of such requirements with a checklist of common mistakes. Writers of most kinds of software documentation (user manuals, programming language manuals, etc.) should find this list useful; we'll demonstrate its use through an example that exhibits all the defects except the last one.

A requirements document

The reader is invited to study, in light of the previous list, some of the software documentation available to him. We could do the same here and discuss actual requirements documents, taken from industrial software projects, as we did in a previous version of this article.[1] But such a discussion is not entirely satisfactory; the reader may feel that the examples chosen are not representative. Also, one sometimes hears the remark that nothing is inherently wrong with natural-language specifications. All one has to do, the argument continues, is to be careful when writing them or hire people with good writing skills. Although well-written requirements are obviously preferable to poorly written ones, we doubt that they solve the problem. In our view, natural-language descriptions of any significant system, even ones of good quality, exhibit deficiencies that make them unacceptable for rigorous software development.



The software life cycle

This "waterfall model" of the software life cycle originated with W. W. Royce ("Managing the Development of Large Software Systems: Concepts and Techniques," *Wescon Proc.*, Aug. 1970), but many variants have been published. A well-known one is in Boehm (1975). The IEEE Standard on Software Quality Assurance (Standard P732) also defines a variant.

Table 1.
The seven sins of the specifier.

Noise:	The presence in the text of an element that does not carry information relevant to any feature of the problem. Variants: *redundancy; remorse.*
Silence:	The existence of a feature of the problem that is not covered by any element of the text.
Overspecification:	The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.
Contradiction:	The presence in the text of two or more elements that define a feature of the system in an incompatible way.
Ambiguity:	The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.
Forward reference:	The presence in the text of an element that uses features of the problem not defined until later in the text.
Wishful thinking:	The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

To support this view, we have chosen a single example, which, although openly academic in nature, is especially suitable because it was explicitly and carefully designed to be a "good" natural-language specification. This example is the specification of a well-known text-processing problem. The problem first appeared in a 1969 paper by Peter Naur where it was described as reproduced here in Figure 1.

Naur's paper was on a method for program construction and program proving; thus, the problem statement in Figure 1 was accompanied by a program and by a proof that the program indeed satisfied the requirements.

The problem appeared again in a paper by Goodenough and Gerhart, which had two successive versions. Both versions included a criticism of Naur's original specification.

Goodenough and Gerhart's work was on program testing. To explain why a paper on program testing included a criticism of Naur's text, it is necessary to review the methodological dispute surrounding the very concept of testing. Some researchers dismiss testing as a method for validating software because a test can cover only a fraction of significant cases. In the words of E. W. Dijkstra,[2] "Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." Thus, in the view of such critics, testing is futile; the only acceptable way to validate a program is to prove its correctness mathematically.

Since Goodenough and Gerhart were discussing test data selection methods, they felt compelled to refute this a priori objection to any research on testing. They dealt with it by showing significant errors in programs whose "proofs" had been published. Among the examples was Naur's program, in which they found seven errors—some minor, some serious.

Goodenough and Gerhart found seven errors—some minor, some serious—in Naur's program.

Our purpose here is not to enter the testing-versus-proving controversy. The Naur-Goodenough/Gerhart problem is interesting, however, because it exhibits in a particularly clear fashion some of the difficulties associated with natural-language specifications. Goodenough and Gerhart mention that the trouble with Naur's paper was partly due to inadequate specification; since their paper proposed a replacement for Naur's program, they gave a corrected specification. This specification was prepared with particular care and was changed as the paper was rewritten. Apparently somebody criticized the initial version, since the last version contains the following footnote:

> Making these specifications precise is difficult and is an excellent example of the specification task. The specifications here should be compared with those in our original paper.

Thus, when we examine the final specification, it is only fair to consider it not as an imperfect document written under the schedule constraints usually imposed on software projects in industry, but as the second version of a carefully thought-out text, describing what is really a toy problem, unplagued by any of the numerous special considerations that often obscure real-life problems. If a natural-language specification of a programming problem has ever been written with care, this is it. Yet, as we shall see, it is not without its own shadows.

Figure 2 (see p. 11) gives Goodenough and Gerhart's final specification, which should be read carefully at this point. For the remainder of this article, numbers in parentheses—for example, (21)—refer to lines of text as numbered in Figure 2.

Analysis of the specification

The first thing one notices in looking at Goodenough and Gerhart's specification is its length: about four times that of Naur's original by a simple character count. Clearly, the authors went to great pains to leave nothing out and to eliminate all ambiguity. As we shall see, this overzealous effort actually introduced problems. In any case, such length seems inappropriate

Rococo interior with fashionable pair dancing; engraving by Gravelot, 1770. The Bettmann Archive

for specifying a problem that, after all, looks fairly simple to the unprejudiced observer.

Before embarking on a more detailed analysis of this text, we should emphasize that the aim of the game is not to criticize this particular paper; the official subject matter of Goodenough and Gerhart's work was testing, not specification, and the prescription period has expired anyway. We take the paper as an example because it provides a particularly compact basis for the study of common mistakes.

Noise. "Noise" elements are identified by solid underlines in Figure 2. Noise is not necessarily a bad thing in itself; in fact, it can play the same role as comments in programs. Often, however, noise elements actually obscure the text. When first encountering such an element, the reader thinks it brings new information, but upon closer examination, he realizes that the element only repeats known information in new terms. The reader must thus ask himself nonessential questions, which divert attention from the truly difficult aspects of the problem.

Here, a fraction of a second is needed to realize that a "nonempty sequence" of characters (8) is the same thing as "one or more" characters (9). These two expressions appear within a line of each other; the authors' aim was, presumably, to avoid a repetition. One is indeed taught in elementary writing courses that repetitions should be avoided, and no doubt this is a good rule as far as literary writing is concerned. In a technical document, however, the rule to observe is exactly the opposite—namely, the same concept should always be denoted by the same words, lest the reader be confused.

Given a text consisting of words separated by BLANKS or by NL (new line) characters, convert it to a line-by-line form in accordance with the following rules:

(1) line breaks must be made only where the given text has BLANK or NL;
(2) each line is filled as far as possible, as long as
(3) no line will contain more than MAXPOS characters.

Figure 1. Naur's original statement of a well-known text-processing problem.

References on the Naur-Goodenough/Gerhart problem

Original reference, Naur:

Peter Naur, "Programming by Action Clusters," *BIT*, Vol. 9, No. 3, 1969, pp. 250-258.

First version, Goodenough and Gerhart:

John B. Goodenough and Susan Gerhart, "Towards a Theory of Test Data Selection," *Proc. Third Int'l Conf. Reliable Software*, Los Angeles, 1975, pp. 493-510. Also published in *IEEE Trans. Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 156-173.

Revised version, Goodenough and Gerhart:

John B. Goodenough and Susan Gerhart, "Towards a Theory of Test: Data Selection Criteria," in *Current Trends in Programming Methodology, Vol. 2*, Raymond T. Yeh, ed., Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 44-79.

Another paper that uses the same problem as an example:

Glenford J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," *Comm. ACM*, Vol. 21, No. 9, Sept. 1978, pp. 760-768.

An interesting variant of noise is **remorse**, a restriction to the description of a certain specification element made not where the element is *defined* but where it is *used*, as if the specifier suddenly regretted his initial definition. An example here is "the output text, if any" (20). Up to this point, the specification freely used the notion of output text (12,17); nowhere was there any hint that such a text might not exist. If the reader wondered about this problem, the specification did not provide an answer. Now, suddenly, when the discussion is focusing on something else, the reader is "reminded" that there might be no such thing as an output text, but no precise criterion is given as to when there is and when there isn't.

Another instance of remorse is the late definition of the "line" concept (24), to which we will return. We will meet again the tendency to say too much, which generates noise, as a source of contradiction and ambiguity.

Silence. In spite of all his efforts, the specifier often leaves, along with overdocumented elements, undefined features. Commonly, these features are fairly obvious to a community of application specialists, who are close to the initial customers, but they will be more obscure to those outside this circle. An example is the concept of "line," which is not really defined except in a parenthetical bit of remorse toward the end of the text (24), where it is described as a sequence of characters "between successive NL characters." (By the way, are those characters part of the line?)

An interesting point here is the cultural background necessary to understand this concept. In ASCII-oriented environments, "New Line" is a character; thus, people working on ASCII environments (DEC machines, for example) will probably understand easily the specification's basic hypothesis —namely, that NL is treated as an ordinary character upon input but triggers a carriage return upon output. These concepts are foreign, however, to somebody working in an EBCDIC environment, especially on IBM OS systems, on which files are made up of a sequence of "records" (corresponding, for example, to lines), each made up of a sequence of characters. A person coming from such an environment would not have written the above specification and will probably have trouble understanding it.

Besides, the late definition of line is plainly wrong. It applies only to lines that are neither at the very beginning nor at the very end of the text. In both these cases, a line is not "between successive NL characters" but between the beginning of the file and an NL, or between an NL and the end of the file—that is, between an NL and an ET. If we accept the authors' definition, the first and last lines of the output may be of arbitrary length; in fact, an output containing *no NL at all* is acceptable regardless of its length, since it does not have lines according to the definition given! This is obviously absurd and not what the authors had in mind, but the use of natural language leads naturally to such slips of the pen.

Another interesting silence concerns the variable Alarm. Line 16 specifies that this variable should be set to TRUE in case of an error, but nothing is said about what happens to it in other cases. The answer is obvious, of course; but the matter can only be brushed aside as minor by programmers who have never run into a bug due to an uninitialized variable. . .

It must be pointed out that Goodenough and Gerhart corrected a notable silence in Naur's original description. Naur's text does not explain what should be done with consecutive groups of more than one break character; this is one of the seven errors analyzed in Goodenough and Gerhart's paper. Their specification corrects it by requiring that such groups be reduced to a single break character in the output. Although something had to be done about the problem, note that this solution is, to some extent, obtained at the expense of simplicity. Eliminating redundant break characters and dividing a text into lines are two unrelated problems; merging them into a single specification complicates the whole affair.

It is probably better to deal with these two requirements separately, and this is what we do in the formal specification given below. Some of the current trends in programming methodology emphasize this approach—most notably under the influence of the Unix programming environment,

Ball at the home of a German baron; engraving circa 1750. The Bettmann Archive

which, at least in principle, favors tools that are simple and composable rather than large and multipurpose.

Contradictions. There is another problem with the concept of line. Given a type *t*, one should distinguish between the types **seq**[*t*], whose elements are finite sequences of objects of type *t*, and **seq** [**seq** [*t*]], whose elements are sequences of sequences of objects of type *t*. Such a confusion can be found in Figure 2, where we are first told (1) that the input is a "stream," or sequence, of characters and later (10) that it "can be viewed" as a sequence of words and breaks. As any Lisp programmer knows, the sequences

<a b a c c a>
[sequence of objects]

and

<<a> <b a> <c c a>>
[sequence of sequences of objects]

are not the same. Note that the same problem with respect to the output is redeemed only by ambiguity; the type of the output is not clear:

- Is it **seq** [*CHAR*] as (21-22) seems to imply?
- Is it **seq** [*WORD*]—that is, **seq** [**seq** [*CHAR*]]—as (12-13) indicates?
- Or is it even **seq** [*LINE*] —that is, **seq**[**seq**[**seq**[*CHAR*]]]—if we consider a line as a sequence of words and breaks?

Thus, a sentence that at first appears to be only noise (9-11) yields a contradiction within a few lines (13-14): "The program's output should be the same sequence of words as in the input." This last comment is remarkable since *neither the input nor the output* is a sequence of words. Worse yet, even if we parse the input into a sequence of words, this sequence is not sufficient to determine the output—one also needs two binary informations: whether there is a leading and/or a trailing break.

1 The program's input is a stream of characters whose end is
2 signaled with a special end-of-text character, ET. There is exactly
3 one ET character in each input stream. Characters are classified
4 as
5 • break characters—BL (blank) and NL (new line),
6 • nonbreak characters—all others except ET,
7 • the end-of-text indicator—ET.
8 A *word* is a nonempty sequence of nonbreak characters. A
9 *break* is a sequence of one or more break characters. Thus, the
10 input can be viewed as a sequence of words separated by breaks,
11 with possibly leading and trailing breaks, and ending with ET.
12 The program's output should be the same sequence of words
13 as in the input, with the exception that an oversize word (i.e., a
14 word containing more than MAXPOS characters, where MAXPOS
15 is a positive integer) should cause an error exit from the program
16 (i.e., a variable, Alarm, should have the value TRUE). Up to the
17 point of an error, the program's output should have the following
18 properties:
19 1. A new line should start only between words and at the be-
20 ginning of the output text, if any.
21 2. A break in the input is reduced to a single break character in
22 in the output.
23 3. As many words as possible should be placed on each line
24 (i.e., between successive NL characters).
25 4. No line may contain more than MAXPOS characters (words
26 and BLs).

Figure 2. Goodenough and Gerhart's final specification of the original problem statement in Figure 1. Analysis of this text, overprinted in blue, is according to the following key:

Noise	——	**Ambiguity**	▱
Remorse	= = = = =	**Overspecification**	.....
Contradiction	⊃	**Forward reference**	═══

Formalism

The same sentence (9-11), in its overzealous effort to leave no stone unturned, ends up introducing another contradiction. An unbiased reader would be puzzled. How can the input "end with [the character] ET" (11) and at the same time have a "trailing break" (11)? "Trailing," precisely, means "at the end"! What's the last character if there is a "trailing" break: ET or a break character?

A more experienced reader, such as a programmer, will have no difficulty resolving this contradiction; his experience will tell him that "end" markers follow "trailing" characters. But this reliance on intuition and knowledge of the application domain can be particularly damaging when transposed to large requirements documents, which will be handed down to a group of system designers and implementors of diverse backgrounds and abilities.

Overspecification. Overspecification in requirements can be annoyingly close to silence. The reader is told too much about the *solution* while he is desperately trying to grasp the *problem* and figure out—by himself—features not covered by the text. Overspecification is typically, although certainly not exclusively, found in requirements documents written by programmers. Psychologically, this is understandable. An implementation-level concept is good, concrete, technical stuff, whereas true requirements deal with much less tangible material. To a computer specialist, a stack is easier to visualize than, say, the flow of information in a company or the needs of a radar operator. Thus, many specifiers have a natural tendency to cling to programming concepts. There is a price to pay for this: Implementation decisions taken too early may turn out to be wrong, and important problem features can be overlooked.

The example text contains an overspecification right from the first sentence: the notion of the end-of-text character ET. The only reason for the presence of this notion is Goodenough and Gerhart's desire to correct Naur's original program. Input-output facilities of the version of Algol 60 used by Naur (and, for fairness, by Goodenough and Gerhart) do not provide for end-of-file detection when reading, so one must assume the presence of a special character at the end of the file to make up for this deficiency. But ET is an implementation detail and should not be included in an abstract specification. Conceptually, the input is a finite sequence of characters; it should be transformed into an output that is a sequence of lines or, depending on the interpretation chosen, a sequence of characters. It is a programmer's vice to insist that finite sequences be specially marked at the end.

Why does the ET character receive such emphasis in Goodenough and Gerhart's specification? The reason is one of the errors in Naur's original program, which would go into an infinite loop unless the input was incorrect (that is, contained an oversize word). Upon closer examination, however, a case can be made for Naur's solution (without the other errors, of course). It is not so unrealistic to consider the required program as a potentially infinite process, which takes characters as input and produces lines as output, working somewhat like a device handler (for instance one that drives a printer) in an operating system. Such an interpretation should, of course, be clearly described in the specification, which was not the case with Naur's text. That decision would be less arbitrary than the one taken by Goodenough and Gerhart: their inclusion of ET changes the data structure at the specification level to accommodate the programming language used at the implementation stage.

The unacceptability of the change is further evidenced by the fact that the output does not satisfy the requirement on the input. Is it realistic to expect an existing file to be terminated by an explicit marker? If it is, the output produced by the program should satisfy that condition; however, examination of the specification, which is not completely clear on this matter, and, as a final criterion, of the proposed program, shows that ET will *not* be passed on to the output file. Assume that we want to write another program, for, say, right-justifying the text, that will take Goodenough and Gerhart's output (in "pipe" mode à la

Dancing the minuet in the open air, copper engraving by Charles Eisen. The Bettmann Archive

Unix). In designing that program, we will not be able to make the same assumption on its input. Thus, the overspecification has opened the way to serious inconsistencies.

Another overspecification in the text is the concept of "error exit" (16), which causes a "variable," Alarm, to have the value TRUE. Clearly, the notion of a variable belongs to the world of programs, not specifications. This piece of overspecification would have been less shocking if the problem had been defined as the task of writing a *procedure,* with Alarm as one of its parameters, or as one of the "exceptions" (in the sense of Clu or Ada) it might raise. A variable is internal to the program unit to which it belongs, whereas the specification of a parameter or an exception can be given relative to the environment of that unit.

The problem of the Alarm variable is less innocuous than it seems. One reason for shock at meeting the reference to this variable in a sequential reading of the text is that the definition of the error case (the one in which there is an oversize word) looks like overspecification until one sees the *last* sentence (25-26), 10 lines down, which gives the basic line-size constraint, MAXPOS. The world is really standing upside down here. Clearly, the constraint on word size is a consequence of the constraint on line size, and the definition of the error case cannot be understood until the latter constraint has been introduced.

We see here one of the major deficiencies plaguing requirements documents of more significant size: early inclusion of detailed descriptions of error handling, interwoven with descriptions of normal cases, which are usually much simpler. Here the matter is even worse; error processing is described before the reader has had a chance to recognize the problem—that is, before gaining an understanding of normal processing. Failure to clearly separate normal cases from erroneous ones makes the document much harder to understand.

	1 2 3 4 5 6 7 8 9 10
1	U N I X I S A
2	T R A D E M A R K
3	O F B E L L
4	L A B O R A T O R I E S
	1 2 3 4 5 6 7 8 9 10

Figure 3. Output requirement (MAXPOS = 10).

Mathematically, a program that performs an input-to-output transformation often corresponds to the implementation of a partial function, which is not defined for some arguments of the input domain. Error processing then consists in "completing" the function with alternate results, such as error messages, for those arguments. This completion should not be confused with the definition of the function in its normal cases. Here, as we'll see later in a formal specification, failure to accommodate words larger than MAXPOS is a consequence of the requirements for normal processing, which can be *proved,* as a theorem, from the definition of the function.

Ambiguities. Error processing raises an ambiguity in the example text (Figure 3). The requirement that the output text satisfy properties 1 to 4 "up to the point of an error" is susceptible to at least two interpretations.

The text says that up to (and presumably including) the point of the error, the program's output should correspond to the input. But where is the "point of the error" in Figure 3? Is it [line 4, column 10], last acceptable letter, or [3, 7], end of the last acceptable word? Nothing in the text allows the reader to decide between these two interpretations.

Another interesting ambiguity is connected with the basic constraint on acceptable solutions (23): "As many words as possible should be placed on each line." If we have, say, MAXPOS = 10 and the input text

WHO WHAT WHEN

there are two equally correct two-line solutions (WHAT may be on either the first or second line). This ambiguity may be acceptable since neither solution appears superior to the other; the specification as such is nondeterministic. We suspect (perhaps wrongly) that this nondeterminism was not intentional and that there was an implicit overspecification in the authors' minds: they considered it obvious that the input would be processed sequentially, so any ambiguity, as in the example above, would be solved by placing as many words as possible on the earlier line (giving line WHO WHAT followed by line WHEN). In this interpretation, property 3 (23-24) actually means, "As many words as possible should be placed on each line as *it is encountered in the sequential construction of the output.*" If this is the case, the specification should state it precisely.

Another potential source of ambiguity is the use of imprecise or poorly defined terms—for example, the use of "stream" (1) rather than the more standard "sequence." The expression "error exit" (15), stemming from the overspecification seen above, is ambiguous, and the reader is not comforted by the explanation that follows it ("i.e., a variable, Alarm, should have the value TRUE"); the notion of assigning a value to a variable does not by itself imply the idea of an "exit," which also means that the program stops in some fashion. We have seen that the concept of "line" is not well defined (24). Also note that the expression "new line" is to be parsed as a single entity (the *new line* character) in its first appearance (5) and as separate words ("a new *line* should start. . .") in its second (19).

Forward references. In a requirements document, not all forward references are bad. Some, corresponding to a top-down presentation of the concepts ("the notion of . . . will be studied in detail in section . . ."), might even be considered good practice, provided there are not too many. But *implicit* forward references (that is, uses of a concept that come before the proper definition of the concept, without particular warning to the reader) can present much more of a problem. They make a document extremely hard to read, especially in the absence of the technical apparatus (index, glossary, etc.) that should be a part of all requirements specifications and other software documents.

Here, of course, the text is very short, so the annoyance caused by forward references is nowhere near what it can be with full-size documents. Note, however, that ET is used three times (2, 3, 6) before it is defined (7), that the notion of line, defined not quite satisfactorily (24), has been used earlier (19-20), and that MAXPOS is used just before its definition (14).

So what? In dissecting Goodenough and Gerhart's specification, we identified a significant number of problems in a text that may seem innocuous to a superficial observer. Not all the problems were equally serious, and the reader may have felt that we were a bit pedantic at times. We submit, however, that one must be pedantic in dealing with such matters. Inconsistencies, ambiguities, and the like may not warrant the gallows when the problem is to split up a sequence of characters into lines. But keep in mind how the above defects transpose to more serious matters—a nuclear reactor control system, a missile guidance system, or even just a payroll program. The computer that executes the code resulting from a faulty specification is more pedantic than any human referee could ever be.

Thus, we should consider Goodenough and Gerhart's specification not only as an object of study in itself but also, and more importantly, as a microcosm for conveniently observing deficiencies typical of more meaningful requirements documents. Al-

Two people doing the minuet; copper engraving by Nilsson. The Bettmann Archive

though the text was written with great care, we have witnessed how the authors, who started out to improve upon Naur's terse but simple text, sentence after sentence became a little more entangled in their own rosary of caveats. This says a lot about why interminable manuals occupy so much shelf space in programmers' offices and computer rooms.

In our opinion, the situation can be significantly improved by a reasoned use of more formal specifications. But again, let's emphasize that such specifications are a complement to natural language documents, not a replacement. In fact, we'll show how a detour through formal specification may eventually lead to a better English description. This and other benefits of formal approaches more than compensate for the effort needed to write and understand methematical notations.

We will now introduce such notations, which will allow us to give a formal specification of the Naur-Good-enough/Gerhart problem.

Elements for a formal specification

Many formal specification languages have been designed in recent years (see box). Choosing one of these languages would force the reader to learn its particular notation and would obscure the essential fact—namely, that their underlying concepts are, for the most part, well-known mathematical notions like sets, functions, relations, and sequences. We thus prefer to use a more-or-less standard mathematical notation. The style of exposition will be similar to that found in mathematical texts; translation to a specific formal specification language should not be hard, provided the language supports the relevant concepts.

References on formal specification

Many formal specification languages have been designed in recent years. A few are listed here, without any claim to exhaustivity.

Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs*, R. McNaughten and R.C. McKeag, eds., Cambridge University Press, 1980.

Rod M. Burstall and Joe A. Goguen, "Putting Theories Together to Make Specifications," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence*, Cambridge, Mass., 1977, pp. 1045-1058.

Cliff B., Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, N.J., 1980

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, Calif., June 1980.

David R. Musser, "Abstract Data Type Specification in the AFFIRM System," *IEEE Trans. Software Engineering*, Vol. SE-6, No. 1, Jan. 1980, pp. 24-32.

L. Robinson and Olivier Roubine, *Special Reference Manual*, Stanford Research Institute, 1980.

Overview. Perhaps the only difficult part of the Naur-Goodenough/Gerhart problem is that the processing to be performed on the text involves three aspects: reducing breaks to a single break character, making sure no line has more than MAXPOS characters, and filling lines as much as possible. If these three requirements are separated, things become much simpler. Consequently, we will define the problem formally by considering two simple binary relations, called *short_breaks* and *limited_length*, and a function called *FEWEST_LINES*. (Throughout the discussion of the formal specification, the reader may wish to refer to Figure 4 for a picture of the overall structure of the relations and functions involved.)

Relation *short_breaks* holds between two sequences of characters a and b if and only if b is identical to a, except that breaks in a (i.e., successive break characters) have been reduced to single break characters in b.

Relation *limited_length* holds between two sequences of characters b and c if and only if c is a "limited length version" of b: that is, no line in c has length greater than MAXPOS, and c is identical to b except that some blanks may have been replaced with new lines and/or some new lines with blanks.

By applying these two relations successively, we associate with any sequence of characters a all sequences of characters that are "made of the same words," separated only by single breaks, and fit on lines no longer than MAXPOS. Given such a set of sequences, say, *SSC*, then *FEWEST_LINES* (*SSC*) is the subset of *SSC* containing those sequences that consist of a minimum number of lines and thus are acceptable outputs for the program.

We'll now define these notions formally, but a few simple conventions are needed first.

Basic form of the specification. As a general convention, we use uppercase for sets and for functions whose results are sets and lowercase for other functions, elements of sets (except for MAXPOS, which we write in uppercase as in the original specification), sequences, and relations.

The program to be written is the implementation of a function

$$sol: INPUT \rightarrow OUTPUT$$

where *INPUT* and *OUTPUT* are the sets of possible inputs and outputs, which we will describe below as sets of sequences. Function *sol* must satisfy certain constraints, which it is the role of the specification to express.

As noted above, there may be more than one correct output for a given input; in other words, a truly general specification of the problem should be nondeterministic. We will represent this fact by defining a binary relation between sets *INPUT* and *OUTPUT*. We call *goal* this binary relation; then a function *sol* will be a correct solution if and only if the following two conditions are satisfied (readers who are not so sure about functions and relations are referred to the refresher in the adjacent box):

- function *sol* is defined wherever relation *goal* is defined—that is, *sol* (*i*) exists for any *i* in the domain of *goal*;
- for any *i* for which *goal* is defined, then *sol* (*i*) yields a "solution" to *goal*—that is, *goal* (*i*, *sol* (*i*)) holds.

This definition is expressed in mathematical notation by writing that *sol* is an acceptable function if and only if

$$\forall i \in \mathbf{dom}\ (goal),\quad i \in \mathbf{dom}(sol) \text{ and } goal\ (i, sol\ (i))$$

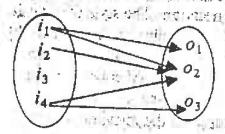
where **dom** (*sol*) is the domain of function *sol*. Note that there may be some inputs for which there is no acceptable solution (those not in the domain of *goal*), so *sol* may be a partial function. Also, in more concise notation, the above property can simply be

A reminder on functions and relations

Consider two sets—for example, *INPUT* and *OUTPUT*. A **binary relation** between these two sets is a set of pairs

$$\{<i_1, o_1>, <i_2, o_2>, \ldots\},$$

where each i_k belongs to set *INPUT* and each o_k belongs to set *OUTPUT*. Such a relation is represented pictorially at right. If *goal* is a relation, then we write *goal* (*i*, *o*) to express that the pair $<i, o>$ belongs to the relation.



A relation.

The domain of such a relation, written **dom** (*goal*), is the subset of *INPUT* containing only those elements *i* such that *goal* (*i*, *o*) holds for **at least** one element *o* in *OUTPUT*. Thus, in the example pictured, i_1, i_2, and i_4, but not i_3, belong to the domain of the relation.

A function is a relation *f* such that for any *i* there is **at most** one *o* for which *f* (*i*, *o*) holds; if *o* exists, then one may write $o = f(i)$. The relation pictured above is not a function, since i_1, for instance, has two buddies o_1 and o_2. Note that the domain of a function is made of those elements of *INPUT* for which there is **exactly** one corresponding element in *OUTPUT*.

Le Bal Paré: Typical Louis XVI court scene of the 18th century. The Bettmann Archive

expressed by writing that the domain of *sol* is at least as large as the domain of *goal*, and that *sol* is **included** in *goal* (both being defined as sets of pairs):

dom (*goal*) ⊂ **dom** (*sol*)
and *sol* ⊂ *goal*

This way of presenting a specification is of very general applicability for programs performing input-to-output transformations. Such a program may be viewed as the implementation of a certain function (*sol*) which must ensure that a certain relation (*goal*) is satisfied between its argument and its result; in mathematical terms, the function is included in (is a subset of) the relation. To specify the problem is to define the relation; to construct the program is to find an implementable function *sol* satisfying the above conditions.[3]

Characters and sequences. The principal set of interest in our problem is the set of characters, which we denote by *CHAR*. The only property of *CHAR* that matters here is that *CHAR* contains two elements of particular interest, *blank* and *new_line*. We call *BREAK_CHAR* the subset of *CHAR* consisting of these two elements:

BREAK_CHAR ≡ {*blank, new_line*}

The basic concept in this problem is that of sequence. If X is a set, we denote by **seq** [X] the set whose elements are finite sequences of elements of X. Such a sequence is written, for example, as

$<a, b, a, c, c, d>$

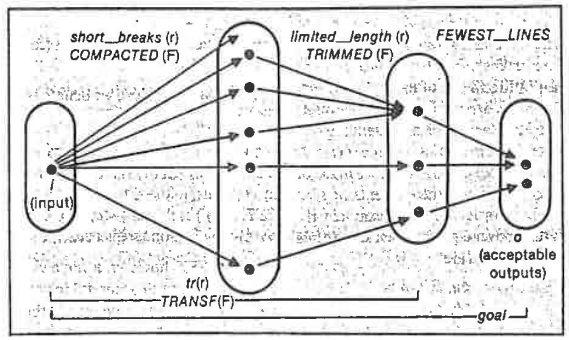


Figure 4. Overall structure of the specification: (r) indicates a relation, (F) a function.

Basic set and logic notations

The definitions marked (*) introduce predicates, that is, expressions which may have value "true" or "false."

{*a, b, c, . . .*}: the set made up of elements *a, b, c, . . .*

$x \in A$: x is an element of A (*).

$x \notin A$: x is not an element of A (*).

$A \subset B$: A is a subset of B (all elements of A are elements of B) (*).

$\{x \in A \mid P(x)\}$: The (possibly empty) subset of A made up of those elements x which satisfy property P.

$\forall x \in A, P(x)$: All elements x of A, if any, satisfy property P (or: no element of A violates P); holds in particular whenever A is empty (*).

$\exists x \in A, P(x)$: There is at least one element x in A which satisfies property P; may only hold if A is nonempty (*).

$a \Rightarrow b$: a implies b.

$a..b$: the integer interval containing all the integers i such that $a \le i \le b$; empty if $a > b$. This notation is borrowed from Pascal.

The symbol ≡ means "is defined as."

and has a length that is a nonnegative integer; thus, *length* is a function from **seq** [X] to the set of natural numbers. Elements are numbered starting at 1; the i-th element of a sequence s (for $1 \le i \le length(s)$) is written $s(i)$. A **subsequence** of s is a sequence made of zero or more of the elements of s, in the same order as in s; for example, if s is the above sequence, then some of its subsequences are

$<a, b, c, d>$
$<b, c, c>$

On the other hand, $<b, d, c>$ is not a subsequence of s because the original order of its elements in s is not preserved.

The set of subsequences of s will be written *SUBSEQUENCES* (s).

The concept of sequences is well known, and we rely on the reader's understanding here. A formal definition of sequences and of the above notions is given in the box on the adjacent page.

Minima and maxima. If X is a set, and f is a function from X to the set of natural numbers,

MIN_SET (X, f)

denotes the subset of X consisting of the elements for which the value of f is minimum. For example, if X is the following set, containing four sequences

$X \equiv \{<a, c, b, a>, <a, b>, <b, a, b>, <c, c>\}$

and f is the *length* function on sequences, then *MIN_SET* (X, f) will be the set consisting of the shortest of these sequences, namely, the second and last.

In the same fashion, we denote by

MAX_SET (X, f)

the subset of X consisting of the elements for which the value of f is maximum; thus, in the above case, *MAX_SET* (X, f) is the set $\{<a, c, b, a>\}$, containing just one sequence.

MAX_SET, however, is not always defined; we have to be careful to apply it only to sets X which are finite; otherwise, there might be no maximum value for f. Note that the results of *MIN_SET* and *MAX_SET* are a subset of X rather than a single element, since there may be more than one element with minimum or maximum f value. These subsets are nonempty if and only if X is nonempty.

We will also need a way to denote the minimum and maximum elements of a set of natural numbers *SN*. They will be written, in the usual fashion, *min* (*SN*) and *max* (*SN*). Thus, if *SN* is the set

$SN \equiv \{341, 7, 3, 654\}$

then *min* (*SN*) is 3 and *max* (*SN*) is 654. Note that *min* and *max*, contrary to *MIN_SET* and *MAX_SET*, yield a natural number, not a set. Also in contrast to *MIN_SET* and *MAX_SET*, which are defined for empty sets (they yield an empty result), both *min* and *max* are defined only if the set *SN* is not empty; *max* further requires that *SN* be finite. It is essential to check for these conditions whenever using these functions.

Input and output sets. In the problem at hand, the input is a sequence of characters; we choose to describe the output as a sequence of characters as well. Thus, we define the two sets:

INPUT ≡ **seq** [*CHAR*]
OUTPUT ≡ **seq** [*CHAR*]

Note that, as mentioned above, another interpretation could have defined the set of possible outputs as **seq** [*LINE*], with *LINE* itself being defined as **seq** [*CHAR*] (or possibly **seq** [*WORD*] with *WORD* ≡ **seq** [*CHAR*], plus information on leading and trailing breaks).

We will now define the relations *short_breaks* and *limited_length* and the function *FEWEST_LINES*.

The formal specification

Short breaks. Let a be a sequence of characters. We define *SINGLE_BREAKS* (a) as the set of subsequences of a such that no two consecutive characters are break characters:

SINGLE_BREAKS (a) ≡
$\{s \in SUBSEQUENCE\ (a) \mid$
$\forall i \in 2..length\ (s),$
$s(i-1) \in BREAK\_CHAR$
$\Rightarrow s(i) \notin BREAK\_CHAR\}$

Note that we use the Pascal notation, *a. .b,* to denote the (possibly empty) set of integers i such that $a \le i \le b$.

Next, we define *COMPACTED* (a) as the subset of *SINGLE_BREAKS* (a) containing those sequences of maximum length:

COMPACTED (a) ≡ *MAX_SET* (*SINGLE_BREAKS* (a), *length*)

As stated above, *MAX_SET* (*X, f*) may be be undefined if *X* is an infinite set. This cannot occur here, however, since *SINGLE_BREAKS* (*a*) is a subset of *SUBSEQUENCES* (*a*) which, for any sequence of characters *a*, is finite.

Note that any sequence *b* in *COMPACTED* (*a*) must have retained from *a* all nonbreak characters (if such a character had been omitted, it could be inserted into *b* and yield a longer element of *SINGLE_BREAKS* (*a*)), and has a single break character where *a* had one or more consecutive break characters.

Thus, the relation *short_breaks* (*a, b*), which holds between *a* and *b* if and only if *a* and *b* are made of the same sequences of words and breaks but the breaks in *b* consist of a single break character, can be expressed simply by

short_breaks (*a, b*) ≡
  b ∈ *COMPACTED* (*a*)

Limited length. The relation *limited_length* (*b, c*) holds between sequences *b* and *c* if and only if

- *c* is the same sequence as *b*, except that it may have a *new_line* wherever *b* has a *blank*, or conversely; and
- the maximum line length of *c*, defined as the maximum number of consecutive characters none of which is a *new_line*, is less than or equal to MAXPOS.

This is expressed more precisely as follows:

limited_length (*b, c*) ≡
  c ∈ *TRIMMED* (*b*)

A definition of sequences

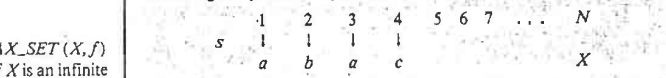
The following presentation is based on the formal specification of sequences given in the Z reference manual.[11]

N will denote the set of natural numbers.

Definition:

seq [*X*], the set of finite sequences of elements of *X*, is defined as the set of partial functions from *N* to *X* whose domains are intervals of the form 1..*n* for some natural number *n*.

So a sequence is defined as a partial function; for example, the sequence *s* = <*a, b, a, c*> is the function defined for arguments 1, 2, 3, and 4 only, and whose value is *a* for 1 and 3, *b* for 2, and *c* for 4. The following is a pictorial representation of *s*:



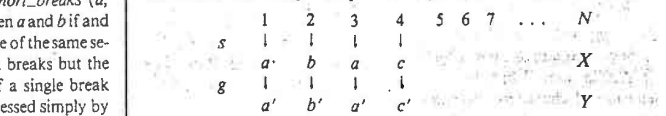
Note that the above definition allows *n*=0 (empty interval, thus empty function – that is, empty sequence) and that it justifies the notation *s*(*i*) for the *i*th element of sequence *s* (which is the result of applying function *s* to element *i*).

The **length** of a sequence is defined as the largest integer for which the associated partial function is defined (i.e., *n* in the above definition).

Now let *s* be a sequence of elements of *X* and *g* be a (total) function from *X* to some set *Y*. The composition

g∘*s*

is a partial function from the set of natural numbers to *Y*, which has the same domain as *s*; thus, it is a sequence of elements of *Y*, with the same length as *s*. This sequence is obtained from *s* by applying *g* to all the elements of *s*. Again, a picture may help (we set *g*(*a*)=*a'*, etc.):



Now take for *X* the set *N* of natural numbers. A **sorted sequence** of natural numbers is an element *s* of seq [*N*] such that

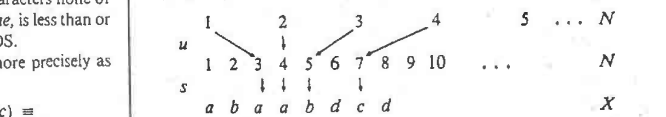
∀*i* ∈ 2 . . *length* (*s*), *s*(*i*−1) ≤ *s*(*i*)

With this definition, it becomes easy to formally define the notion of **subsequence** used in the text.

Definition:

Let *s* be an element of seq [*X*] for some set *X*. A subsequence of *s* is a sequence of the form *s*∘*u* where *u* is a sorted sequence of natural numbers.

The following picture shows how <*a a b c*> is obtained as a subsequence of <*a b a a b d c d*> using the above definition. The sorted sequence *u* of natural numbers used here is <3 4 5 7>; <1 3 5 7> or <1 4 5 7> would also work.



Formalism

where

TRIMMED (*b*) ≡
  {*s* ∈ *EQUIVALENT* (*b*) |
  max_line_length (*s*) ≤ MAXPOS}

EQUIVALENT (*b*) ≡
  {*s* ∈ **seq**[*CHAR*] |
    length (*s*) = *length* (*b*) **and**
    (∀ *i* ∈ 1..*length* (*b*),
      s(*i*) ≠ *b*(*i*) ⇒
        s(*i*) ∈ *BREAK_CHAR* **and**
        b(*i*) ∈ *BREAK_CHAR*) }

max_line_length (*s*) ≡
  max ({*j*−*i* |
    0≤*i*≤*j*≤*length* (*s*) **and**
    (∀ *k* ∈ *i*+1. .*j*,
      s(*k*) ≠ *new_line*) })

A few explanations may help in understanding these definitions. If *s* is a sequence of characters, *max_line_length* (*s*) is the maximum length of a line in *s*, expressed as the maximum number of consecutive characters, none of which is a new line. In other words, it is the maximum value of *j*−*i* such that *s*(*k*) is not a new line for any *k* in the interval *i*+**1**. .*j*. (We will have more to say about this definition below.) *EQUIVALENT* (*b*) is the set of sequences that are "equivalent" to sequence *b* in the sense of being identical to *b*, except that *new_line* characters may be substituted for *blank* characters or vice versa. Finally, *TRIMMED* (*b*) is the set of sequences which are "equivalent" to *b* and have a maximum line length less than or equal to MAXPOS.

Fewest lines. Let *SSC* be a set of sequences of characters. These sequences can be interpreted as consisting of lines separated by *new_line* characters. We define the set *FEWEST_LINES* (*SSC*) as the subset of *SSC* consisting of those sequences that have as few lines as possible:

FEWEST_LINES (*SSC*) ≡
  MIN_SET (*SSC,*
    number_of_new_lines)

where the function *number_of_new_lines* is defined by:

number_of_new_lines (*s*) ≡
  card ({*i* ∈ 1..*length* (*s*) |
    s(*i*) = *new_line*})

and **card** (*X*), defined for any finite set *X*, is the number of elements (cardinal) of *X*.

The basic relation. The above definitions allow us to define the basic relation of the problem, relation *goal*, precisely. Relation *goal* (*i,o*) holds between input *i* and output *o*, both of which are sequences of characters, if and only if

o ∈ *FEWEST_LINES* (*TRANSF* (*i*))

TRANSF (*i*) is the set of sequences related to *i* by the composition of the two relations *short_breaks* and *limited_length:*

TRANSF (*i*) ≡ {*s* ∈ seq [*CHAR*] |
    tr (*i, s*) }

with

tr ≡ *limited_length*∘*short_breaks*

The dot operator denotes the composition of relations (see box). A look at Figure 4 may help explain the role of the various functions and relations in the above specification.

Existence of solutions. Once we have a formal specification, what can we do with it? Relying on the specification as a basis for the next stages of the software life cycle—program design and implementation (e.g., translating ∀s into loops) is the most obvious use. However, we'd like to emphasize two others. One use, studied in the next section, is as a starting point for better natural-language requirements. The other, to which we now turn, is querying the specification to learn as much as possible about properties of the problem and valid solutions.

What can the given specification teach us about the Naur-Goodenough /Gerhart problem and its solution? First, let's determine when solutions do exist. It is trivial to prove that, given a sequence of characters *a*, there is always at least one sequence *b* such that relation *short_breaks* (*a, b*) holds. Given *b*, however, the necessary and sufficient condition for the existence of at least one sequence *c* such that *limited_length* (*b, c*) holds is that *b* contains no word (i.e., contiguous subsequence of non-break characters) of length greater than MAXPOS. This follows from the definitions of *TRIMMED* and *max_line_length* used in the definition of *limited_length*. Thus, the domain of definition of the relation *tr*, which is also the domain of the function *TRANSF* and thus of the relation *goal*, is the set of input texts containing no word longer than MAX-

"The Compleat Figure of the Minuet," an engraving from George Bickham's *An Easy Introduction to Dancing*, shows the basic spatial shapes used in the minuet, 1738.
From the library of Christena L. Schlundt, University of California, Riverside

POS. This can be formulated as a theorem:

dom (*goal*) =
  {*s* ∈ **seq** [*CHAR*] |
    ∀*i* ∈ 1..*length*(*s*) − MAXPOS,
    ∃*j* ∈ *i*. .*i*+MAXPOS,
      s(*j*) ∈ *BREAK_CHAR*}

The property expressed by this theorem is that the domain of relation *goal* consists of sequences such that, if a character *c* is followed by MAXPOS other characters, at least one character among *c* and the other characters must be a break.

An important problem, not addressed here, is how the specification deals with erroneous cases—that is, with inputs not in the domain of the *goal* relation—like sequences with oversize words. Clearly, a robust and complete specification should include (along with *goal*) another relation, say, *exceptional_goal*, whose domain is *INPUT*−**dom** (*goal*) (set difference); this relation would complement *goal* by defining alternative results (usually some kind of error message) for erroneous inputs. Formal specification of erroneous cases falls beyond the scope of this article, but a discussion of the problem and precise definitions of terms such as "error," "failure," and "exception" can be found in a paper by Cristian.[4]

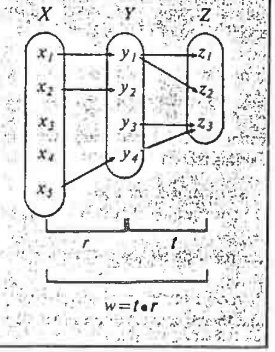
Discussion. What we have obtained is an abstract specification—this is, a mathematical description of the problem. It would be difficult to criticize this specification as being oriented toward a particular implementation: if

Composition of relations

Let *r* and *t* be two relations; *r* is from *X* to *Y* and *t* is from *Y* to *Z* (see figure).

The composition of these two relations, written *t*•*r* (note the order), is the relation *w* between sets *X* and *Z* such that *w* (*x*, *z*) holds if and only if there is (at least) one element *y* in *Y* such that both *r* (*x*, *y*) and *t* (*x*, *y*) hold.

Thus, in the example illustrated, *w* holds for the pairs $<x_1, z_1>$, $<x_1, z_2>$, and $<x_5, z_3>$ (and for these pairs only).



Formalism

followed to the letter, the specification would lead to a program that (as illustrated in Figure 4) would first generate all possible distributions of the input over lines of length less than or equal to MAXPOS and then search the resulting list for solutions with minimum number of *new_line* characters—not a very efficient implementation!

An element that does seem to point toward a particular implementation technique is the composition of relations *short_breaks* and *limited_length*, which seems to imply a two-step process (first remove break characters, then cut into lines). A first design could indeed use a two-step solution. The steps could then be merged using coroutine-like concepts, such as the Unix notion of pipe or the "program inversion" idea of Jackson's program design method.[5]

We chose to model the problem's object and operations with very simple mathematical notions (sets, relations, functions, sequences). Because of the specific nature of this problem, another approach would have been to rely on a more advanced theory, such as the theory of regular languages. As emphasized below, a realistic specification system should permit reuse of existing theories.[6]

Starting from the above definition, the specification should of course be refined, taking into account the physical form of the data structure (including, for example, the end-of-file marker) and the particular response that should be given by the program in case of erroneous input.

Conclusion

Although natural language is the ideal notation for most aspects of human communication, from love letters to introductory programming language manuals, there are cases[7] where it is not appropriate. Software specifications, for example, require more rigorous formalism.

The use of formal notation does not, however, preclude that of natural language. In fact, mathematical specification of a problem usually leads to a better natural-language description. This is because formal notations naturally lead the specifier to raise some questions that might have remained unasked, and thus unanswered, in an informal approach.

Mathematical definition. Formal specifications help expose ambiguities and contradictions because they force the specifier to describe features of the problem precisely and rigorously. The problem studied in this article contains many examples of this. For example, let us try to redefine the function *max_line_length* using the definition of "line" taken from Goodenough and Gerhart's specification (line 24: "between successive NL characters"). Writing this definition mathematically, we obtain something like

max_line_length (*s*) ≡
  max ({*line_length* (*s*, *i*) |
    1 ≤ *i* ≤ *length* (*s*) and
    s(*i*) = *new_line*})

where *line_length* (*s*, *i*), the length of the line beginning after the *new_line* at position *i* in sequence *s*, may be defined as a minimum:

line_length (*s*, *i*) ≡
  min({*k* |
    0 ≤ *k* < *length* (*s* − *i*) **and**
    s(*i*+*k*+1) = *new_line*})

However, as mentioned above, the maximum or minimum of a set of natural numbers is defined if and only if this set is nonempty and, in the maximum case, finite; so using mathematical notation prompts us to check for these conditions. Finiteness presents no problem, but we see immediately that the set whose maximum is sought in the definition of *max_line_length* will be empty if the sequence *s* does not contain any *new_line* character. Even if it contains one, *line_length* (*s*, *i*), itself a minimum, will not be defined if there is no other *new_line* further in the sequence. This prompts us to look for a better definition.

A fairly natural reaction at this point is to see that we really don't need to define the concept of "line," only that of **maximum line length.** Once we have noticed this, it's easy to come up with a correct definition: *the maximum number of consecutive characters, none of which is a new line.* This is the definition that was given above:

max_line_length (*s*) ≡
  max ({*j* − *i* |
    0 ≤ *i* ≤ *j* ≤ *length*(*s*) **and**
    (∀ *k* ∈ *i*+1. .*j*,
      s(*k*) ≠ *new_line*)})

Note that we have been careful to apply *max* to a set that always contains at least one value (zero, obtained for

"The Compleat Figure of the Minuet," an engraving from George Bickham's *An Easy Introduction to Dancing*, shows the basic spatial shapes used in the minuet; 1738.
From the library of Christena L. Schlundt, University of California, Riverside

POS. This can be formulated as a theorem:

dom (*goal*) =
  {*s* ∈ **seq** [*CHAR*] |
    ∀*i* ∈ 1..*length*(*s*) − MAXPOS,
    ∃*j* ∈ *i*..*i*+MAXPOS,
      s(*j*) ∈ *BREAK_CHAR*}

The property expressed by this theorem is that the domain of relation *goal* consists of sequences such that, if a character *c* is followed by MAXPOS other characters, at least one character among *c* and the other characters must be a break.

An important problem, not addressed here, is how the specification deals with erroneous cases—that is, with inputs not in the domain of the *goal* relation—like sequences with oversize words. Clearly, a robust and complete specification should include (along with *goal*) another relation, say, *exceptional_goal*, whose domain is *INPUT* − **dom** (*goal*) (set difference); this relation would complement *goal* by defining alternative results (usually some kind of error message) for erroneous inputs. Formal specification of erroneous cases falls beyond the scope of this article, but a discussion of the problem and precise definitions of terms such as "error," "failure," and "exception" can be found in a paper by Cristian.[4]

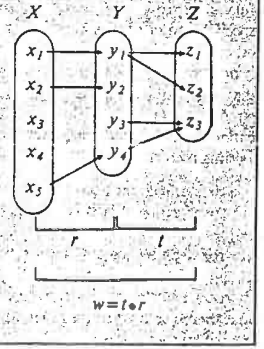
Discussion. What we have obtained is an abstract specification—this is, a mathematical description of the problem. It would be difficult to criticize this specification as being oriented toward a particular implementation: if

Composition of relations

Let *r* and *t* be two relations; *r* is from *X* to *Y* and *t* is from *Y* to *Z* (see figure).

The composition of these two relations, written *t*•*r* (note the order), is the relation *w* between sets *X* and *Z* such that *w* (*x*, *z*) holds if and only if there is (at least) one element *y* in *Y* such that both *r* (*x*, *y*) and *t* (*x*, *y*) hold.

Thus, in the example illustrated, *w* holds for the pairs $<x_1, z_1>$, $<x_1, z_2>$, and $<x_5, z_3>$ (and for these pairs only).



$i = j = 0$), even if s is an empty sequence (see box).

Natural language definition. Once such a mathematical definition has been produced, it may in return influence the natural language definition. In this example, the formal definition suggests that we should refrain from trying to define the concept of "a line in the text" which, although intuitively clear, is slightly tricky when one attempts to specify it precisely, as Goodenough and Gerhart's text shows. Instead, we should focus on the notion of "maximum line length," which is always defined, even for a text consisting of *new_line* characters only. Once we have obtained the specification of *max_line_length*, we can build on it and include it in the English problem definition a sentence such as

> The maximum number of consecutive characters, none of which is a *new_line*, should not exceed MAXPOS.

This sentence, a direct translation from the formal definition, is not, admittedly, of the most gracious sytle; but it is easy to remove the double negation, yielding

> Any consecutive MAXPOS+1 characters should include a *new_line*.

The main advantage of natural language texts is their understandability. One should concentrate on this asset rather than trying to use natural language for precision and rigor, qualities for which it is hopelessly inadequate. Understandability is seri-

The reasoning behind formal specifications: the example of max_line_length

How does one obtain a formal expression such as the one defining *max_line_length*? Let's analyze the different steps involved.

We want to express the fact that *max_line_length* (*s*) is the maximum length of a line in *s*. A definition that avoids the pitfalls mentioned in the analysis of Goodenough and Gerhart's text is, informally, "the maximum number of consecutive characters, none of which is a new line."

To translate this definition into a formal description, we have to express the notion of a contiguous subsequence of *s* that does not contain a *new_line*. A contiguous subsequence can be given by its end indices, say, *i* and *j*. The sequence comprising the elements between indices *i* and *j* will have length $j-i+1$; if it is to yield a line length, then $s(k)$ should be a character other than *new_line* for any *k* between *i* and *j*, inclusive. Thus, a first try might yield

max_line_length (*s*) ≡ *max* (*LINE_LENGTHS*)

where the set *LINE_LENGTHS* is defined as

LINE_LENGTHS ≡ {$j-i+1$ | $1 \le i \le j \le$ *length* (*s*) **and** (∀*k* ∈ *i*..*j*, *s*(*k*) ≠ *new_line*) }

But beware! One should only apply *max* to nonempty sets. With the above convention, we can end up with *LINE_LENGTHS* being empty if *s* is an empty sequence or all its characters are *new_line*; in either case, no *i*, *j* pair satisfies the condition. Now, if we write a program for the Naur-Goodenough/Gerhart problem and put in into a library, sooner or later someone will apply it to a sequence that is empty or entirely made of *new_line* characters, so we had better deal with these cases in a clean fashion.

The culprit is the condition $i \le j$, which prevents us from finding a satisfactory *i* and *j* in the borderline cases mentioned. The problem disappears, however, if we replace this condition by $i-1 \le j$. Then, for a sequence having only *new_line* characters or no character at all, the set *LINE_LENGTHS* will contain one element, 0, obtained for $i=1$ and $j=0$. For these values, the interval *i*..*j* is empty; thus, the ∀... clause is true. (Remember that a property of the form ∀*x* ∈ *E*, *P* (*x*) is always true when the set *E* is empty, regardless of what property *P* is.) Thus, we obtain the following replacement:

LINE_LENGTHS ≡ {$j-i+1$ | $0 \le i-1 \le j \le$ *length* (*s*) **and** (∀*k* ∈ *i*..*j*, *s*(*k*) ≠ *new_line*) }

(The first condition has been written $0 \le i-1$ instead of $1 \le i$.)

We have chosen to simplify slightly the writing of this condition by a change of variable (use *i* for $i-1$, thus eliminating +1 and −1 terms):

LINE_LENGTHS ≡ {$j-i$ | $0 \le i \le j \le$ *length* (*s*) **and** (∀*k* ∈ $i+1$..*j*, *s*(*k*) ≠ *new_line*) }

This new version is defined in all cases.

It should be noted that this kind of analysis, which at first sight might seem quite remote from programmers' concerns, is in fact closely connected to typical patterns of reasoning about programs. Anyone who has tried to debug a loop that sometimes goes one iteration too few or too many, or works improperly for empty inputs or other borderline cases, will recognize the line followed in the above discussion. It is our contention, however, that such analysis is better performed at the specification level, dealing with simple and well-defined mathematical concepts, than at program debugging time, when the issues are obscured by many irrelevant details, implementation-dependent features, and idiosyncrasies of programming languages.

followed to the letter, the specification would lead to a program that (as illustrated in Figure 4) would first generate all possible distributions of the input over lines of length less than or equal to MAXPOS and then search the resulting list for solutions with minimum number of *new_line* characters—not a very efficient implementation!

An element that does seem to point toward a particular implementation technique is the composition of relations *short_breaks* and *limited_length*, which seems to imply a two-step process (first remove break characters, then cut into lines). A first design could indeed use a two-step solution. The steps could then be merged using coroutine-like concepts, such as the Unix notion of pipe or the "program inversion" idea of Jackson's program design method.[5]

We chose to model the problem's object and operations with very simple mathematical notions (sets, relations, functions, sequences). Because of the specific nature of this problem, another approach would have been to rely on a more advanced theory, such as the theory of regular languages. As emphasized below, a realistic specification system should permit reuse of existing theories.[6]

Starting from the above definition, the specification should of course be refined, taking into account the physical form of the data structure (including, for example, the end-of-file marker) and the particular response that should be given by the program in case of erroneous input.

Conclusion

Although natural language is the ideal notation for most aspects of human communication, from love letters to introductory programming language manuals, there are cases[7] where it is not appropriate. Software specifications, for example, require more rigorous formalism.

The use of formal notation does not, however, preclude that of natural language. In fact, mathematical specification of a problem usually leads to a better natural-language description. This is because formal notations naturally lead the specifier to raise some questions that might have remained unasked, and thus unanswered, in an informal approach.

Mathematical definition. Formal specifications help expose ambiguities and contradictions because they force the specifier to describe features of the problem precisely and rigorously. The problem studied in this article contains many examples of this. For example, let us try to redefine the function *max_line_length* using the definition of "line" taken from Goodenough and Gerhart's specification (line 24: "between successive NL characters"). Writing this definition mathematically, we obtain something like

$$max\_line\_length\ (s) \equiv max\ (\{line\_length\ (s, i) \mid 1 \le i \le length\ (s)\ \textbf{and}\ s(i) = new\_line\})$$

where *line_length* (s, i), the length of the line beginning after the *new_line* at position i in sequence s, may be defined as a minimum:

$$line\_length\ (s, i) \equiv min(\{k \mid 0 \le k < length\ (s-i)\ \textbf{and}\ s(i+k+1) = new\_line\})$$

However, as mentioned above, the maximum or minimum of a set of natural numbers is defined if and only if this set is nonempty and, in the maximum case, finite; so using mathematical notation prompts us to check for these conditions. Finiteness presents no problem, but we see immediately that the set whose maximum is sought in the definition of *max_line_length* will be empty if the sequence s does not contain any *new_line* character. Even if it contains one, *line_length* (s, i), itself a minimum, will not be defined if there is no other *new_line* further in the sequence. This prompts us to look for a better definition.

A fairly natural reaction at this point is to see that we really don't need to define the concept of "line," only that of **maximum line length.** Once we have noticed this, it's easy to come up with a correct definition: *the maximum number of consecutive characters, none of which is a new line.* This is the definition that was given above:

$$max\_line\_length\ (s) \equiv max\ (\{j-i \mid 0 \le i \le j \le length\ (s)\ \textbf{and}\ (\forall k \in i+1..j,\ s(k) \ne new\_line)\})$$

Note that we have been careful to apply *max* to a set that always contains at least one value (zero, obtained for $i = j = 0$), even if s is an empty sequence (see box).

Natural language definition. Once such a mathematical definition has been produced, it may in return influence the natural language definition. In this example, the formal definition suggests that we should refrain from trying to define the concept of "a line in the text" which, although intuitively clear, is slightly tricky when one attempts to specify it precisely, as Goodenough and Gerhart's text shows. Instead, we should focus on the notion of "maximum line length," which is always defined, even for a text consisting of *new_line* characters only. Once we have obtained the specification of *max_line_length*, we can build on it and include it in the English problem definition a sentence such as

> The maximum number of consecutive characters, none of which is a *new_line*, should not exceed MAXPOS.

This sentence, a direct translation from the formal definition, is not, admittedly, of the most gracious sytle; but it is easy to remove the double negation, yielding

> Any consecutive MAXPOS + 1 characters should include a *new_line*.

The main advantage of natural language texts is their understandability. One should concentrate on this asset rather than trying to use natural language for precision and rigor, qualities for which it is hopelessly inadequate. Understandability is seri-

The reasoning behind formal specifications: the example of max_line_length

How does one obtain a formal expression such as the one defining *max_line_length*? Let's analyze the different steps involved.

We want to express the fact that *max_line_length* (s) is the maximum length of a line in s. A definition that avoids the pitfalls mentioned in the analysis of Goodenough and Gerhart's text is, informally, "the maximum number of consecutive characters, none of which is a new line."

To translate this definition into a formal description, we have to express the notion of a contiguous subsequence of s that does not contain a *new_line*. A contiguous subsequence can be given by its end indices, say, i and j. The sequence comprising the elements between indices i and j will have length $j-i+1$; if it is to yield a line length, then $s(k)$ should be a character other than *new_line* for any k between i and j, inclusive. Thus, a first try might yield

$$max\_line\_length\ (s) \equiv max\ (LINE\_LENGTHS)$$

where the set *LINE_LENGTHS* is defined as

$$LINE\_LENGTHS \equiv \{j-i+1 \mid 1 \le i \le j \le length\ (s)\ \textbf{and}\ (\forall k \in i..j,\ s(k) \ne new\_line)\}$$

But beware! One should only apply *max* to nonempty sets. With the above convention, we can end up with *LINE_LENGTHS* being empty if s is an empty sequence or all its characters are *new_line*; in either case, no i, j pair satisfies the condition. Now, if we write a program for the Naur-Goodenough/Gerhart problem and put in into a library, sooner or later someone will apply it to a sequence that is empty or entirely made of *new_line* characters, so we had better deal with these cases in a clean fashion.

The culprit is the condition $i \le j$, which prevents us from finding a satisfactory i and j in the borderline cases mentioned. The problem disappears, however, if we replace this condition by $i-1 \le j$. Then, for a sequence having only *new_line* characters or no character at all, the set *LINE_LENGTHS* will contain one element, 0; obtained for $i=1$ and $j=0$. For these values, the interval $i..j$ is empty; thus, the $\forall \ldots$ clause is true. (Remember that a property of the form $\forall x \in E,\ P\ (x)$ is always true when the set E is empty, regardless of what property P is.) Thus, we obtain the following replacement:

$$LINE\_LENGTHS \equiv \{j-i+1 \mid 0 \le i-1 \le j \le length\ (s)\ \textbf{and}\ (\forall k \in i..j,\ s(k) \ne new\_line)\}$$

(The first condition has been written $0 \le i-1$ instead of $1 \le i$.)

We have chosen to simplify slightly the writing of this condition by a change of variable (use i for $i-1$, thus eliminating $+1$ and -1 terms):

$$LINE\_LENGTHS \equiv \{j-i \mid 0 \le i \le j \le length\ (s)\ \textbf{and}\ (\forall k \in i+1..j,\ s(k) \ne new\_line)\}$$

This new version is defined in all cases.

It should be noted that this kind of analysis, which at first sight might seem quite remote from programmers' concerns, is in fact closely connected to typical patterns of reasoning about programs. Anyone who has tried to debug a loop that sometimes goes one iteration too few or too many, or works improperly for empty inputs or other borderline cases, will recognize the line followed in the above discussion. It is our contention, however, that such analysis is better performed at the specification level, dealing with simple and well-defined mathematical concepts, than at program debugging time, when the issues are obscured by many irrelevant details, implementation-dependent features, and idiosyncrasies of programming languages.

[85b]

SHOWING PROGRAMS ON A SCREEN

Bertrand Meyer[1]

Computer Science Department, University of California
Santa Barbara, California 93106 (USA)

Jean-Marc Nerson[1]

CIMSA, 10 avenue de l'Europe
78140 Vélizy (France)

Soon Hae Ko

Computer Science Department, University of California
Santa Barbara, California 93106 (USA)

ABSTRACT

We present a strategy and algorithms for displaying a meaningful view of structured objects such as programs on a screen of limited size. The methods introduced here are language-independent; they were developed for the implementation of Cépage, a structural editor making full use of modern display technology. The algorithms are linear with respect to the number of nodes in the syntax tree.

We use a formal model of the screen allocation, the "calculus of windows", which makes it possible to reason about the display process at a proper level of abstraction. A systematic approach was followed, in which a number of "invariants" and "attributes" were defined before the actual construction of the algorithms and data structures, and served as a basis for their development; the paper describes the methodology used and includes a semi-formal correctness proof of the main algorithm, which involves mutually recursive procedures.

This paper appears in *Science of Computer Programming*, Vol. 5, no. 2, pages 111-142, 1985.

[1]Work begun at Electricité de France, Direction des Etudes et Recherches, 1 avenue du Général de Gaulle 92141 CLAMART (France)

Table of Contents

SHOWING PROGRAMS ON A SCREEN

Bertrand Meyer[1]

Computer Science Department, University of California
Santa Barbara, California 93106 (USA)

Jean-Marc Nerson[1]

CIMSA, 10 avenue de l'Europe
78140 Vélizy (France)

Soon Hae Ko

Computer Science Department, University of California
Santa Barbara, California 93106 (USA)

1 - INTRODUCTION

1.1 - The Need for Structural Views of Software Objects

One of the basic ideas which are making their way into advanced programming environments is that software engineering tools should be able to deal with the various objects they have to handle - programs, design documents, specifications, test data, schedules, maintenance reports, user manuals, etc. - in terms of their structure, not just as if they were mere sequences of characters. This is all the more important that these objects are often quite complex. It is only through the application of this idea that one can lay the foundations for true *Computer-Aided Design* of software.

Tools which manipulate objects through their structure make it possible, at least in principle, to perform very sophisticated operations, affecting entire sub-structures. There is, however, an important problem to be solved before such operations can be made usable in a safe, practical and efficient way: if the tools know about object structure, then so should the *users*. This calls for providing users, at each step of the process, with a proper representation of the objects being acted upon.

Thus in a good software development system the users should "see" the structure of the objects as clearly as possible; this will allow them to traverse the structure quickly, performing "zooming" and "un-zooming" operations as they go along, moving and copying sub-structures, etc.

The problem of providing users with a good structural view of the objects at hand also exists in engineering CAD-CAM, where it is addressed through the use of powerful graphics facilities. In software, although graphical representations may be envisioned, most objects are essentially texts; but in many cases (notably, though certainly not exclusively, when dealing with programs) these texts may have a deep or even intricate structure. It is thus essential to find adequate structural views of these texts, even on character (non-graphic) terminals. This paper presents a solution to this problem.

[1] Work begun at Electricité de France, Direction des Etudes et Recherches, 1 avenue du Général de Gaulle 92141 CLAMART (France)

The basic issue it addresses may be summarized as follows: given a structured document and a screen of finite size, can one find a representation of this text which will fit on the screen while providing the terminal user with a clear view of the text's structure?

In other words, the problem is to find the best possible *mapping* of an abstract structure (that of the document being edited) to a physical area (the screen). A strategy and algorithms will be described.

To avoid any confusion, we shall use the word *document* to denote the structured objects which are to be displayed, reserving the word *text* for external representations built from characters.

The ideas presented here have so far been applied to the display of program texts - hence the title of this paper. They may however be useful for other kinds of documents with a sufficiently rich structure.

1.2 - Relation to Previous Work

In the case when the documents are programs, the problem studied here is of course close to what is known as **pretty-printing**, i.e., printing program texts in a suitable way, using indentation to exhibit their structure. Unfortunately, methods used for pretty-printing on paper are of little use for interactive screen editors: a universal, albeit implicit, assumption in descriptions of pretty-printers (see, e.g. Oppen [18]) is that, whereas the width of the page is fixed, lines are an essentially infinite resource. With a screen, *both* lines and columns are limited resources.

Apart from a very terse hint at the techniques used for INTERLISP in [3], the only published algorithms we know for the problem addressed here are those of Mikelsons [17]; although we were able to gain some fruitful ideas from this work, it could not be applied directly, both because of differing assumptions (the environment described in Mikelsons' paper has quite specific constraints) and because much of Mikelsons' method relies on a procedure (called Measure in [17]) which is not described precisely.

1.3 - Methodological Background

The algorithms presented here were developed in a systematic fashion, using a semi-formal approach in which a set of *invariants* played a fundamental role in defining the purpose of the algorithms and establishing their correctness. Similarly, for the main data structures, abstract *attributes* were defined before representation issues were considered. Invariants and attributes will be presented in section 6.

After our initial implementation was completed, we worked out a simple formal model of the basic objects involved in the display process. We call this small theory the **calculus of windows**; its discovery led to significant improvements in the algorithms and data structures.

1.4 - Structure of the Paper

The rest of the paper is organized as follows. In the next section, we explain the context in which this work was carried out (the development of a parameterizable, visual and structural editor). Section 3 introduces the basic display strategy used. Section 4 gives a first sketch of the display process. Section 5 introduces the "calculus of windows" which serves as a useful mathematical model. Section 6 introduces the attributes and invariants. The basic algorithm is given in section 7. Section 8 contains a semi-formal proof of correctness of this algorithm, followed by an analysis of its efficiency. Section 9 outlines the other important algorithm (for list nodes) and is followed by a conclusion discussing the usage of the system, the problems encountered, and the methodological issues involved.

2 - CONTEXT: THE CEPAGE EDITOR

2.1 - Overview

The system for which these methods were developed is Cépage [15, 16], a *parameterized* editor which is both *structural* and *visual*.

Structural editors (also called "structure", "structured", "syntax-oriented", "language-based" editors), such as Mentor [6], Gandalf [8] or the Cornell Program Synthesizer [20], were the first tools which applied the idea that a program text may be operated upon in terms of its structure, not as a flat sequence of characters. Many such tools have been developed by researchers in the past few years; structural editors, however, have not yet been widely accepted by industry. We feel that this is in part due to the insufficient quality of the user interface in the first prototypes.

On the other hand, a "visual" or "full-screen" editor such as Vi [12], Emacs [19] or SPF [11] provides the user with a good instantaneous view of the document being edited by devoting the whole video screen to a display of part of the document; this relatively large "window" on the document gives the user better control over the editing process than he may enjoy with the more traditional line-by-line text editors.

The design of Cépage resulted from the belief that a powerful yet usable editor should be both structural and visual. Such a decision (discussed in detail elsewhere [15, 16]) has important consequences on the user interface. A good visual editor should make the best possible use of modern display technology (within the constraints imposed by portability concerns). If the editor is also structural, this view should rely on the structure of the document; in particular, the system should show the **hierarchical** context of the current focus of interest (e.g. the enclosing blocks in a block-structured language), whereas non structural full-screen editors may only display a contiguous, **linear** excerpt of the document.

We now briefly introduce the characteristics of Cépage which are relevant for this study.

2.2 - Language independence

Cépage is entirely language-independent. The language, i.e. the description of the structure of the documents to be edited, is a parameter of the system. This parameter is **interpreted**, i.e. it is represented by a data structure, the **grammar graph**, which is used by the editor along with the structure of the document being edited (abstract syntax tree).

2.3 - Abstract syntax

The most important part of a language description (grammar graph) is a representation of the **abstract syntax** of the language. Such an abstract syntax consists of a set of **syntactic types** and a set of **productions**.

The productions may be of three different kinds: aggregate, choice or list. These three categories are illustrated by the example abstract syntax given on figure 1, where they are distinguished by the labels [A], [C], [L] respectively; this example is the syntax of a fairly realistic subset of Pascal and should be self-explanatory (elements which appear in square brackets on the right-hand side of aggregate productions denote optional components; e.g. a *procedure*, as defined in production 17, may or may not have a *procedure_parameter_list*).

1 [A] *program = name ; program_parameter_list ; block*
2 [L] *parameter_list = variable**
3 [L] *block = [label_part] ; [constant_part] ; [type_part] ; [var_part] ; [procedure_part] ; compound*
4 [L] *label_part = label**
5 [A] *label = constant*
6 [L] *constant_part = constant_decl**
7 [A] *constant_decl = name ; constant*
8 [L] *type_part = type_decl**
9 [A] *type_decl = name ; type_description*
10 [C] *type_description = record | name*
11 [A] *record = var_part*
12 [L] *var_part = var_decl**
13 [A] *var_decl = variable_list ; type_description*
14 [L] *variable_list = variable**
15 [A] *variable = name*
16 [L] *procedure_part = procedure_decl**
17 [A] *procedure_decl = name ; [procedure_parameter_list] ; block*
18 [A] *procedure_parameter_list = var_part*
19 [L] *compound = statement**
20 [C] *statement = assignment | conditional | loop | compound*
21 [A] *assignment = variable ; expression*
22 [C] *expression = variable | constant | binary*
23 [A] *binary = expression ; operator ; expression*
24 [A] *conditional = statement ; expression ; statement*
25 [A] *loop = expression ; statement*

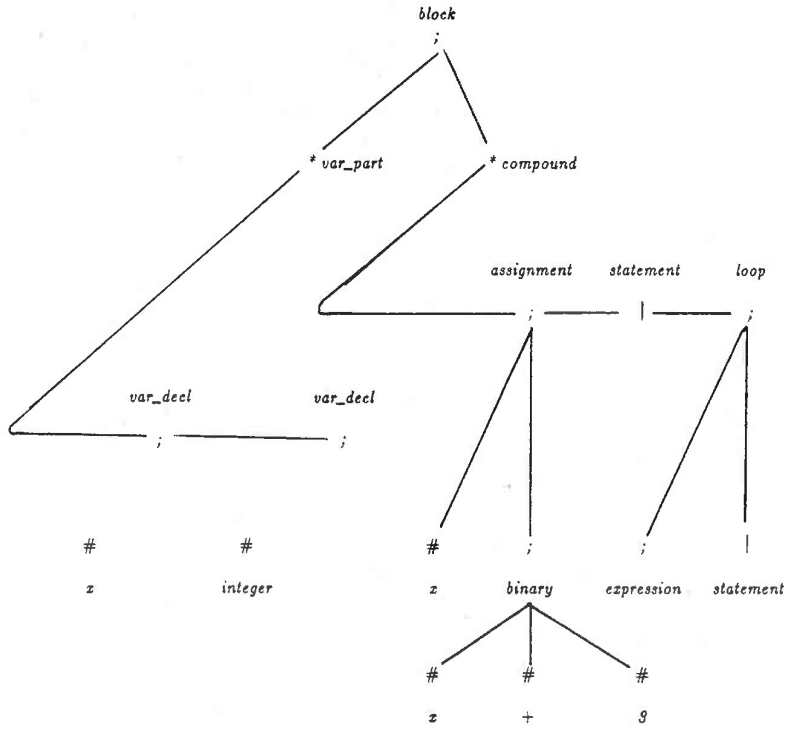
Figure 1: An Abstract Syntax

In such an abstract syntax, a syntactic type may appear on the left of at most one production. Those which do appear on the left of a production are called non-terminals; those which do not are called terminals (here *name, constant, operator*).

The documents handled by the editor conform to an abstract syntax such as this one; they may be **partially refined documents** containing non-terminals yet to be expanded.

A partially or totally refined document will be represented by an **abstract syntax tree** having four kinds of nodes: "aggregate", "choice", "list" and "terminal" nodes, corresponding to the four categories of syntactic types in the abstract grammar. *Choice* nodes may only appear as leaves in the tree representing a partially refined document: they correspond to elements which have not yet been refined, e.g. a *statement* for which the user has not yet decided between *assignment, conditional, loop* and *compound* (when this choice is made, the *statement* node will be replaced with an aggregate node in the first three cases and a list node in the last).

Figure 2 gives an example of such an abstract syntax tree, representing a partially refined program document in the syntax of Figure 1.



; : aggregate node
* : list node
| : choice node
: terminal node

Figure 2: An Abstract Syntax Tree

2.4 - Program Display, Concrete Syntax and Tree Decoration

To display the current state of a document, the system needs to know the **concrete syntax** of the language. The concrete syntax can be given as a set of additions to the productions of the abstract syntax, containing the information necessary to construct the external ("concrete") form of the expansion for each production. Two kinds of elements are needed for the addition of concrete syntax to an abstract production:

- references to constituents of the right-hand side of the abstract production, called **operands** (e.g. the concrete syntax of a *loop* will contain an *expression* and a *statement* as operands);

- elements which only appear in the concrete form, like programming language keywords (**if, repeat** and the like) or, for languages with a strange concrete syntax like Fortran, formating marks such as new line, tab positioning etc. These concrete elements are called **operators**.

The concrete syntax information added to an abstract production consists of the following:

- For a list production, three operators: a header, a terminator and a delimiter, e.g. **begin, end** and the semicolon, respectively, for *compound* (production 19 on figure 1);

- For an aggregate production, a sequence of operands and operators, as in the following concrete syntax for *conditional* (production 24 of figure 1):

if *%2* **then** *%1* **else** *%3* **end if**

where *%i* denotes the *i*-th operand on the right-hand side of the production (i.e. in this case a conditional statement will appear as **if** *exp* **then** *stat1* **else** *stat2* **end if**);

- Choice productions do not imply any addition.

When the program must be displayed, the abstract syntax tree is "decorated" with concrete syntax information, in the form of new leaves associated with operators. The resulting decorated tree may be called a *concrete syntax tree* and resembles the "parse tree" used in compilers[1]. Note that all internal nodes of the concrete tree are operand nodes; its leaves represent operators, terminal operands or unrefined non-terminal operands.

2.5 - User Interface

The abstract syntax serves as a guide to the editing process, which operates by cursor movement and menu selection.

For example, Figure 3 shows a partially refined program obtained at some stage, using the syntax of figure 1. Assume that the user has moved the cursor to a position marked <*statement*> (indicated by *** on the figure). Only a statement may eventually appear at this position. By choosing the "refine" option in the current menu, the user requests refinement of this statement. A new menu will then appear, listing the possible choices for *statement* in this language, i.e. *compound, assignment, loop, conditional* according to the given grammar. Assume the user chooses *conditional* from this menu; the screen will then be updated to reflect this choice, with the proper syntax for the conditional statement inserted at the appropriate place (Figure 4).

If the user requests refinement of an entity whose syntactic type corresponds to a terminal in the grammar (e.g. *name*), then a new frame appears on the screen, on which text for the terminal may be entered using a simple full-screen text editor included in Cépage.

[1] In Cépage, the concrete tree is never physically constructed; the concrete representation is generated from the abstract tree and information contained in the grammar graph. It is conceptually useful, however, to think of the concrete tree as if it actually existed.


```
program <name>
  (<program_parameter_list>) ;
    [label_part]
    [constant_part]
    [type_part]
    var
        x, y : integer ;
        a, b : <type_description>
        [var_part] ;
    procedure pr1
      [procedure_parameter_list] ;
        [label_part]
        [constant_part]
        [type_part]
        [var_part]
    begin
        <compound>
    end procedure - - pr1
begin      - - Main program
    x := 3 ; y := <expression> ;
  ***<statement>;
    while <expression> do
        y := y + 1 ; <statement>
    end while
end program
```

Figure 3: A Partially Refined Document
(*** indicates the cursor position)

```
program <name>
  (<program_parameter_list>) ;
    [label_part]
    [constant_part]
    [type_part]
    var
        x, y : integer ;
        a, b : <type_description>
        [var_part] ;
    procedure pr1
      [procedure_parameter_list] ;
        [label_part]
        [constant_part]
        [type_part]
        [var_part]
    begin
        <compound>
    end procedure ;      - - pr1
begin      - - Main program
    x := 3 ; y := <expression> ;
  ***if <expression> then
        <statement>
    else
        <statement>
    end if;
    while <expression> do
        y := y + 1 ; <statement>
    end while
end program
```

Figure 4: A Refinement

3 - DISPLAY STRATEGY

3.1 - Overview

The previous section gives a rough idea of how Cépage works internally and interacts with the user (for more details, see [15] or [16]). We shall from now on concentrate on the main problem addressed in this paper: how, in such a framework, is it possible to ensure at each stage that the display presents the user with a good picture of his document?

The strategies used for pretty-printing are, to some extent, a matter of taste. We have found, however, that by sticking to some simple and reasonable principles pretty-printing can be made fully automatic, which relieves the users from prescribing any specific options.

3.2 - Four Principles

The fundamental idea is that the way a document appears on the screen will be determined by its underlying abstract syntax. This may be stated more precisely through the following principle:

> **Principle 1**: The purpose of a display algorithm is to show a picture of the *concrete text* of the program which is as reminiscent as possible of its *abstract syntactic structure*.

The main technique for achieving Principle 1 is **indentation**. Indenting part of a document is a way to highlight it and thus to draw attention upon the fact that it constitutes an entity. In view of the preceding discussion, it appears that the only subtexts which should be candidates for indentation are the concrete representations of meaningful entities in the abstract syntax, i.e. **operands**. Since on the other hand one should certainly not force all operand texts to appear indented, we obtain the following second principle:

> **Principle 2**: The concrete text corresponding to the expansion of an operand should either:
>
> - a : appear on a single line with some preceding and/or following text;
> - b : be indented (alone) on one or more lines.

Note that principle 2 is recursive, i.e. it applies to all operands which will appear in the expansion of an operand.

The following representations of the same program fragment all conform to principle 2:

if *c* **then** *st1* **else** *st2* **end if**

if *c*
then *st1*
else *st2*
end if

if
    c
then
    st1
else
    st2
end if

if *c* **then**
    st1
else *st2* **end if**

etc. The principles above may be applied either to screen display or paper pretty-printing. In the former case, further rules must be obeyed in order to make the best possible use of the limited available space. To economize on space, we thus add the following principle:

> **Principle 3**: When applying principle 2, rule a should be chosen rather than b whenever both are applicable.

Even so, however, there will usually not be enough space on the screen to represent any but very short programs. The technique to be used in such cases is called **ellipsis** or **holophrasting** [9]. We apply it by representing some possibly large subtrees with just the name of their syntactic types; aggregate and list nodes are treated in a different way.

An aggregate node may simply be replaced by the name of its syntactic type, in angle brackets; e.g. a complex conditional statement may be displayed as just *<conditional>* if there is not enough space to show more. This we call **abstraction**. Abstraction is also applicable to a list node; if, however, there is a little more room (although not enough to show all list elements), we may try **collapsing**, which is abstraction applied to one or more sublists, each of which will be replaced by *<n t>* where *t* is the name of the syntactic type of the elements and *n* the number of elements in the sublist. For example, if we cannot show a whole compound statement but have enough space to show the beginning and end, we might get (with an abstraction on the third line):

begin
while *c* ≠ *0* **do**
    p := *<expression>*
end while;
<23 statements> ;
a := *b* ; *c* := *d+1* ;
e := *f*
end

This can be expressed by the following, last principle:

> **Principle 4**: The concrete text for an operand may be replaced by the name of the operand's syntactic type; the concrete texts of one or more non-contiguous sublists of a list node may each be replaced by the number and syntactic type of their elements.

The specification of the display algorithms used in Cépage is based on principles 1 to 4. The algorithms will try to make the best possible use of the available display space by applying principle 4 only when they cannot think any better.

3.3 - Efficiency Requirements

Efficiency is an important criterion for an algorithm which, as in the case of Cépage, must be used interactively to display adequate pictures of a document. It was easy to foresee that, if one was looking for the optimal solution to the display problem, one would run into combinatorial algorithms, which seemed unacceptable. We felt necessary to try to find an algorithm with time complexity O (N), where N is the number of nodes in the concrete subtree to be displayed.

4 - OVERVIEW OF THE DISPLAY PROCESS

The basic loop of the system may be described as follows:

decode user request ;
perform the corresponding manipulation on the abstract syntax tree;
update the display to reflect changes to the document

The last statement of this loop is the one of interest here. What the updated display will represent is a certain subtree of the abstract syntax tree; we call the root of this subtree the current **focus**. To *update the display*, the system first determines the new focus a from the user's request, and then issues the procedure call

Show (s, a⟷)

where *Show* is the basic display procedure and s is an abstract description of the available screen.[2]

The *Show (s, a⟷)* operation is performed, at least conceptually, in five stages:

Decorate (a⟷);
-- builds the concrete syntax tree for a;

Measure (a⟷);
-- computes for each node of the subtree of root a
-- the size of the area which its representation would require
-- in the absence of any space limitations and further formating

Fit (area (s), a⟷) ;
-- formats the subtree of root a so that it will fit in s.
-- *area (s)* is the rectangular area associated with s,
-- described as a "window" (see below).

Buildtext (a, s, window_table⟵);
-- interprets the formated tree to build a table
-- of displayable text-filled windows.

Display (window_table)
-- effectively displays the result, using screen management routines
-- which, for Cépage, come from a screen package called Gescran [2]

The effect of the first three calls (to *Decorate*, *Measure* and *Fit*) is to add information to the abstract syntax tree, in order to transform it gradually into a form from which the fourth procedure (*Buildtext*) may build a screen image, which procedure *Display* will actually output.

[2] For readability, we write actual argument lists in such a way that arguments x which may be modified by the procedure are clearly marked: x⟷ if the corresponding formal argument is of mode **in out** and x⟵ if it is **out**.

Decorate adds concrete syntax; *Measure* determines the space associated with the representation of every subtree in the absence of any formating, assuming a screen of infinite height and width; *Fit* transforms the representation so that it will fit in the given screen area.

As regards the problem of screen-oriented formating of structured documents, the key stages in the process are procedures *Measure* and especially *Fit*. We shall thus concentrate on them in the sequel.

5 - A CALCULUS OF WINDOWS

5.1 - Purpose

The aim of the display algorithms is to associate with each node of the syntax tree a rectangular "window" of text of the appropriate size. To understand how this is done, it is useful to define a set of operations which apply to these windows. This has led us to define a "calculus" of windows.

This calculus is a small mathematical theory; as pointed out by a referee, it resembles what in programming is called the specification of an abstract data type. A complete definition of "window" as an abstract data type was not deemed necessary, however, since properties of windows are readily expressed in terms of properties of integers and booleans. On the other hand, the development of the calculus, which only occurred after we completed our first implementation, strongly suggests that the ideas should be carried over to the program level, i.e. that the program should contain an implementation of the calculus in the same fashion that it might contain the implementation of an abstract data type, especially with a language offering direct support for such concepts, like Simula 67, Smalltalk, Ada, etc. This will be done in our next implementation.

5.2 - Basic Definitions

Any window w is characterized (regardless of its contents) by attributes *w.height*, *w.width*, *w.line_break_before* and *w.line_break_after*. The height and width are integers; they are either both positive or both zero. Attributes *w.line_break_before* and *w.line_break_after* are boolean and indicate whether the window must be preceded and/or followed by a new line.

To denote a window of height h and width w, we write

$\boxed{h \mid w}$

where $h \mid w$ may be preceded of followed by ^ to indicate line breaks.

The special windows of height and width 0 are:

- the **empty window**, $\boxed{0 \mid 0}$, written $\boxed{\ }$;
- the **line break windows**: $\boxed{0 \mid 0^{\wedge}}$, $\boxed{{}^{\wedge}0 \mid 0}$ and $\boxed{{}^{\wedge}0 \mid 0^{\wedge}}$; the last one will be written simply as $\boxed{{}^{\wedge}}$.

5.3 - Order Relation

There is a partial order relation on windows, which we write $\leqslant$, defined as follows:

$x \leqslant y$ iff
(x.height $\leq$ *y.height)* and *(x.width* $\leq$ *y.width)* and
(y.height $= 1 \Rightarrow$
(x.line_break_before $\Rightarrow$ *y.line_break_before)* and
(x.line_break_after $\Rightarrow$ *y.line_break_after))*

The inverse relation is written $\geqslant$. Clearly, $\boxed{\ }$ is a minimum element for $\leqslant$.

5.4 - Concatenation

Another important operation is the concatenation of windows, written $\oplus$. Intuitively, concatenating two windows means displaying one after the other; if possible (see principle 3 above), they will be concatenated on the same line; otherwise, the second window will be displayed below the first. More precisely:

Let $z = x \oplus y$;
if $x = \boxed{\ }$, then $z = y$;
if $y = \boxed{\ }$, then $z = x$;
otherwise:
z.line_break_before = x.line_break_before ;
z.line_break_after = y.line_break_after ;

if z.height = y.height = 1 and not *z.line_break_after*
and not *y.line_break_before* then
z.height = 1;
z.width = z.width + y.width + 1
else
z.height = z.height + y.height;
z.width = max (z.width, y.width).

Note that windows of height greater than 1 will be separated by line breaks regardless of the values of their line break attributes. The *+1* term for the width in the first alternative accounts for intervening blanks.

Concatenation has $\boxed{\ }$ as zero element and further satisfies the following properties:

$x \leqslant x \oplus y$;
$y \leqslant x \oplus y$;
$x \leqslant z$ and *(z has no line breaks* or $x.height \geq 1$) $\Rightarrow$
$(x \oplus y \leqslant z \oplus y)$ and $(y \oplus x \leqslant y \oplus z)$

5.5 - Multiplication

From concatenation, we can define multiplication of a window by a non-negative integer:

$i \circledast w =$ if $i = 0$ then $\boxed{\ }$ else $(i-1) \circledast w \oplus w$

Multiplication satisfies

$i \leq j \Rightarrow i \circledast w \leqslant j \circledast w$

but is not distributive over concatenation, as the following counter-example shows:

$2 \circledast (\boxed{1}\boxed{1} \oplus \boxed{1}\boxed{1}^{*}) = 2 \circledast \boxed{1}\boxed{2}^{*} = \boxed{2}\boxed{2}^{*}$, but
$2 \circledast \boxed{1}\boxed{1} \oplus 2 \circledast \boxed{1}\boxed{1}^{*} = \boxed{1}\boxed{2} \oplus \boxed{2}\boxed{1}^{*} = \boxed{2}\boxed{2}^{*}$

5.6 - Division by an Integer; Fairness and Consistent Allocation Theorems

It turns out that multiplication is less useful for application to the display algorithms than **division of a window by an integer** $i \geq 1$, which we write $w \oslash i$ and define as follows. If w is one of the special windows (empty window, line break windows), then $w \oslash i = w$. Otherwise if $i \leq w.height$, then given

$h = \lfloor w.height / i \rfloor$

the result of the division is

$w \oslash i = \boxed{h}\boxed{w.width}$

with the same line break attributes as w. If $i > w.height$, then given

$c = \lceil i / w.height \rceil$

and

$d = \lfloor (w.width - c + 1) / c \rfloor$

then

$w \oslash i =$ if $d = 0$ then $\boxed{0}\boxed{0}$ *else* $\boxed{i}\boxed{d}$

with, in both cases, the same line break attributes as w.

Division satisfies

$w \oslash i \leqslant w$

and

$(i \circledast w) \oslash i = w$

but $i \circledast (w \oslash i)$ is not necessarily equal to w, nor even $\leqslant$ *to* w. It may informally be said to be "no greater" than w, however, in the sense that i windows of size $w \oslash i$ will fit (i.e. can be concatenated with some intermediate line breaks) in the area of w. This property will make division useful for allocating space to various parts of a document on the basis of their relative importance (see procedure *Split_a_line* in section 7.3 below, and the algorithm for list nodes in section 9).

More precisely, assume two elements compete for space in a window w. Each element has an integer weight, or "share" (shares are described below in section 6.5); assume the sum of all shares is γ and the two elements have shares α and β, with $\alpha + \beta \leq \gamma$. The policy used by the algorithms below, when distributing space to elements on the basis of their shares, is to allocate to the two elements windows $w \oslash (\lceil \gamma / \alpha \rceil)$ and $w \oslash (\lceil \gamma / \beta \rceil)$ respectively. That such an allocation is consistent is expressed by the following theorem, whose proof, although not hard, is tedious and thus not included:

Consistent Allocation Theorem: Let α, β, γ be positive integers such that $\alpha + \beta \leq \gamma$. Let w be a window. Let $w_\alpha = w \oslash (\lceil \gamma / \alpha \rceil)$ and $w_\beta = w \oslash (\lceil \gamma / \beta \rceil)$. Then $w_\alpha \oplus w_\beta \leqslant w$ or $w_\alpha \oplus \boxed{\ }^{*} \oplus w_\beta \leqslant w$.

That such an allocation is also "fair", i.e. obeys the order implied by the shares, is expressed by the following theorem:

Fairness Theorem: Let i, j be positive integers such that $i \geq j$. Let w be a window. Let $w_i = w \oslash i$ and $w_j = w \oslash j$. Then $w_i \leqslant w_j$.

5.7 - Division by a Window

Another kind of division operation is the division of a window by another window, written $w \oslash w'$. If w and w' are windows such that $w' \leqslant w$ and w' is not empty, then $w \oslash w'$ is an integer, defined (using the previously defined division operation) as:

$w \oslash w' = \max(\{i > 0 \mid w \oslash i \geqslant w'\})$

Since this definition involves the maximum of a set of integers, we must check that this set is always finite and non-empty (a necessary and sufficient condition for the existence of the maximum). This indeed the case since, whenever $w' \leqslant w$ and w' is not empty, then $(w \oslash 1) \geqslant w'$ (so that the value 1 is always a member of the set) and $w \oslash i = \boxed{\ }$ for sufficiently large i (so that the set is finite).

By definition, $w \oslash (w \oslash w') \geqslant w'$ and $w \oslash (w \oslash w' + 1) \leqslant w'$.

It may be noted that the definition of the two division operations is consistent with the usual integer division in the following sense: if n and j are non-negative integers such that $n \geq j$, then it is not hard to prove that:

$$\lfloor n / j \rfloor = max (\{ i > 0 \mid \lfloor n / i \rfloor \geq j \})$$

5.8 - Subtraction

Lastly, subtraction, written $z = w \ominus w'$ and defined for $w' \trianglelefteq w$, is such that, if $w.height = 1$, then

$z =$ $\boxed{1 \mid w.width - w'.width - 1}$

and otherwise

$z =$ $\boxed{w.height - w'.height \mid w.width}$

with the same line break attributes as w in both cases. Note that $w \ominus w' \trianglelefteq w$, but $(w \ominus w') \oplus w'$ is not necessarily equal to w, nor even $\trianglelefteq$ to w.

6 - ATTRIBUTES, PRECONDITIONS AND INVARIANTS

6.1 - "Name" Attribute

For the algorithms to be able to perform abstraction and collapsing, it is necessary to associate to every node of the decorated tree some string representing its name. More precisely, we will assume that, associated with any node n in the decorated tree, there is an text attribute, which we shall write *n.name*, which represents a displayable name attached to the node and is determined in the following way:

- If n is a leaf associated with an operator, then *n.name* is the character string making up the associated element of the concrete syntax (keyword, delimiter, etc.), e.g. "**begin**", ":=", etc.;
- If n is an internal node, i.e. an operand associated with a non-terminal in the abstract syntax, then *n.name* is the character string making up the name of its syntactic type (e.g. "*statement*");
- If n is a leaf operand (i.e. a leaf whose syntactic type is terminal), then *n.name* is as in the previous case if the node has not been refined (e.g. "*variable*", etc.); if it has been refined into a character string, then *n.name* is that character string.

6.2 - Minimum Space

We assume the existence of an integer constant *MINSPACE* such that, for any node n, its name *n.name* can be written, possibly truncated, using *MINSPACE* characters without too much loss of information (*MINSPACE* = 10 to 14 seems reasonable). We assume that *Show (s, a⟷)* is always called with screen s having at least one line and *MINSPACE* columns; thus the procedure will always succeed while conforming to the principles above, although it may do so in a very degenerate way, by displaying <*atype*>, where <*atype*> is the name of a's syntactic type (e.g. <*program*>), truncated to *MINSPACE* characters.

We call *MINWINDOW* the minimum window which may be associated with a node:

MINWINDOW = $\boxed{1 \mid MINSPACE}$

6.3 - "Window"

The role of the display algorithm is to associate with every node n a window, which will be denoted by *n.window* and will be used to display the text associated with the node. The attributes of this window will be written *n.height, n.width, n.line_break_before, n.line_break_after* (as abbreviations for *n.window.height*, etc.).

The *n.window* attributes of all nodes n of the subtree which has the focus as its root must eventually be such that the representation of this subtree fits in the given screen s. Initially, however, this will usually not be the case; procedure *Fit* must thus modify the windows associated with the nodes until the subtree fits. Throughout this process, it is necessary to make sure that the *n.window* attributes of all nodes n of the subtree are meaningful and consistent; in other words, they must be such that, given a screen of sufficient size, a representation for the subtree could be produced in which each node would be assigned a window of size *n.window*. This very important property, written *Representable (n)*, must be initially ensured for each node by procedure *Measure*, and maintained by procedure *Fit* throughout the space allocation process (however, we will see in section 8.5 that this restriction may be relaxed in some cases).

The property *Representable (n)* may be defined more rigorously as follows:

property *Representable (a : NODE)*

For all nodes n in the subtree of root a:

a. If n is a leaf not representing a line break, then
$n.window =$ $\boxed{1 \mid length (n.name)}$

b. If n is a leaf representing a line break, then
$n.window =$ $\boxed{\hat{}}$

c. If n is an interior node, then
$$n.window = \sum_{c \in children(n)} c.window$$

where the sum refers to the $\oplus$ operation.

It is important to note that this property **does not by itself involve the given screen** s: a may be "*Representable*" even though it does not fit in s. *Representable (a)* just means that the position information associated with a and all its descendants is correct and consistent, but not necessarily that it is compatible with the space available on any particular screen.

6.4 - "Processed" Attribute

We further assume that every node has a boolean attribute *n.processed* which will have value **true** if and only if n has been visited by *Fit* (*n.processed* initialized to **false** for all nodes). This attribute plays no role in the algorithm itself but is introduced as an auxiliary variable [7] which will help us check that the algorithm is linear in the number of nodes of the subtree to be displayed. *Fit* will be built so as to maintain the following invariant for all nodes n:

[IP] *n.processed* ⇒ n belongs to a line of length $\leq w.width$

where w is the window assigned to the parent node of n by the algorithm.

6.5 - "Share" Attribute

Every node except the root has an integer attribute *n.share* which is assigned by the editor and represents its importance **relative to its siblings.** Space will be allocated to the children of a node on the basis of this share. For example, in a list, the editor might decide to assign the largest shares to the leading and trailing elements, so that even if collapsing occurs the user may see some of the beginning and end of the list.

6.6 - "Indented" attribute

The fact that an operand *d* needs to be indented from the immediately enclosing context is represented by a boolean attribute *d.indented*.

7 - THE BASIC DISPLAY ALGORITHM

7.1 - Initializing the Tree With Dimension Information

As mentioned above, the task of procedure *Measure* is to ensure that *Representable (n)* is satisfied for all nodes *n*. This is performed by a postorder traversal of the tree:

```
procedure Measure (in out a: NODE)
    if
        a is a leaf other than line break  →  a.window := [1 | length(a.name)]
      [] a is a line break  →  a.window := [ ]
      [] a is an interior node  →
            a.window := [ ] ;
            for all c in children (a) do
                Measure (c⟷);
                a.window := a.window ⊕ c.window
            end for
    end if
end procedure
```

Note that when *Measure* is applied initially, it will always result in *a.height* = 1 if there are no built-in line breaks in the concrete syntax of the language. For languages such as Fortran whose concrete syntax includes built-in line breaks, a modification to *Measure* may be useful (see section 8.5 below).

7.2 - Outline of the Display Loop

The principle of the algorithm for Fit is as follows. *Measure (a⟷)* has resulted in a state such that *Representable (a)* is satisfied; i.e. attribute *n.window* is correct for all nodes *n* in the subtree of *a*. In general, however, *a.window* will be too wide for the available window *w*, whereas *w* may have more lines; if this is the case, we may try to trade width for height. The task of *Fit (w, a⟷)* is thus to add line breaks, set *indent* attributes, abstract operands and/or collapse sublists until *a.width* becomes lesser than or equal to *w.width*, with *a.height* remaining no greater than *w.height*. The decision process which is repeated by the algorithm is summarized below for the case of aggregate (non-list) nodes.

a.width / *a.height*	≤ *w.width*	> *w.width*
< *w.height*	**Success**	Try adding a line break.
= *w.height*		**Failure**: abstract *a* (i.e. replace it by its name)
> *w.height*		

7.3 - The Algorithm for Aggregate Nodes

Fit is recursive. The call

Fit (w, a⟷)

must ensure the postcondition *a.window* ⋚ *w* . We give below the algorithm for the case when *a* is an aggregate node.

```
procedure Fit (in w: WINDOW;
               in out a: NODE) :
    - - Recursive precondition: w ≳ MINWINDOW
    - - Recursive invariant: Representable (a)
    - - Recursive postcondition: a.window ⋚ w

    var success, failure: BOOLEAN;

    success := false ; failure := false ;
    repeat
        if
            a.window ⋚ w  →
                success := true
          [] a.height > w.height
             or (a.height = w.height and not a.window ⋚ w)  →
                failure := true
          [] a.height < w.height and a.width > w.width  →
                Split_a_line (w, a⟷, failure⟵);  - - see below
        end if
    until
        success or failure
    end loop ;

    if
        success  →  skip
      [] failure  →  a.window := [1 | min (length (a.name), MINSPACE)]
    end if;
    a.processed := true
end procedure
```

The procedure *Split_a_line (w, a⟷)* is detailed below. It uses an integer function *line_length* which, when applied to a sequence *z* of operators and/or operands (not containing any line break or other formating mark), yields the number of characters of its representation, including provision for separating blanks:

$$line\_length\ (z) = (\sum_{o \in z} o.width) + m\text{-}1$$

(*m* being the number of elements of *z*). *Split_a_line* also uses the constant *INDENT* whose value is the number of blanks used for every indentation step.

In the description of *Split_a_line*, *d* and *f* stand for operands; *x*, *y*, *z* stand for (possibly empty) sequences of operands and/or operators.

Split_a_line may be expressed as follows:


```
procedure Split_a_line ( in w: WINDOW;
                         in out a : NODE,
                         out failure : BOOLEAN)
    var w', w1 : WINDOW ;
        d: NODE ;
        remaining_shares : INTEGER ;
    failure := false ;
    Consider a as a sequence of lines ;
        - - By hypothesis, a.width > w.width, so there is at least one line L
        - - such that line_length (L) > w.width;
    if
        there is at least one line of length greater than w.width which does not end with an
        operand ⟶
            Let L be such a line ;
                - - L will be cut, after an operand if it has one
            if
                L has only operators ⟶
                        - - (degenerate case: oversize line with operators only)
                    insert a line break somewhere in L
              [] L has at least one operand d ⟶
                    L is of the form xdy, y not empty ;
                        - - L will be cut after d
                    d.line_break_after := true
            end if ;
            a.height := a.height + 1 ; update a.width
      [] all lines of length greater than w.width end with an operand ⟶
            Let L be such a line ;
            L is of the form xd, d operand ;
                - - If possible, indent d
                - - First compute the window w' which d may claim for indentation
            remaining_shares := Σ (f ∈ operand children of a on oversize lines) f.share ;
            w1 := a.window ; w1.width := w.width;
            w' := (w ⊖ w1) ⓘ (⌈remaining_shares / d.share⌉) ;
            w'.width := max (1, w'.width - INDENT) ;
                - - Can d be indented ?
            if
                w' ⊵ MINWINDOW ⟶
                        - - There is enough room to indent d
                    Fit (w', d⟷) ;
                    d.indent := true ; d.line_break_before := true ;
                    a.height := a.height+d.height ; update a.width;
              [] not w' ⊵ MINWINDOW ⟶
                        - - There is not enough room to indent d, but perhaps it
                        - - may fit on a line with z, possibly in abstracted form
                    w' := ⌈ 1 ⌊ max (0, w.width - line_length(z) - 1) ⌋ ⌉
                    if
                        w' ⊵ MINWINDOW ⟶ Fit (w', d⟷)
                      [] not w' ⊵ MINWINDOW ⟶ failure := true
                    end if
            end if
        end if
    end if
end procedure
```

8 - CORRECTNESS, EFFICIENCY AND IMPROVEMENTS

Although a complete formal proof of correctness has not been performed for this program, it is interesting to note the following properties, which make it possible to check that it performs its intended task and to assess its performance.

8.1 - Partial Correctness

The task which must be performed by *Fit* may be characterized by the following pair of assertions:

Precondition:

Representable (a) **and**

[P] $w \geqslant$ *MINWINDOW*

Postcondition:

Representable (a) **and**

[Q] *a.window* $\leqslant$ w

The procedures *Fit* and *Split_a_line* being mutually recursive, a proof of their properties requires a proof of the corresponding properties of their bodies, in which the properties of the calls may be assumed [10]. It is in this sense that we have used above the expressions "recursive precondition" for [P], "recursive postcondition" for [Q] and "recursive invariant" for *Representable (a)* which appears in both the precondition and the postcondition.

Let us check that if *Fit* and *Split_a_line* are recursively assumed to satisfy the properties mentioned, then their bodies also satisfy them. We first check that the actual parameters to the internal calls to *Fit* satisfy [P]; then that *Fit* ensures postcondition [Q]; finally, that it maintains invariant *Representable (a)*.

The fact that both recursive calls to *Fit* from *Split_a_line* satisfy [P] is readily checked: both are part of **if** statements, executed under conditions written precisely to be equivalent to [P].

Let us now check that *Fit* ensures [Q]. The body of *Fit* is a **repeat ... until** statement which terminates when either *success* or *failure* becomes true; clearly these two cases are disjoint. In the first, the condition [Q] is satisfied since this condition is exactly the test for *success*; in the second, *Fit* will devote to a a window no greater than $\boxed{1 \mid MINSPACE}$, i. e. *MINWINDOW*, a solution which satisfies [Q] since *MINWINDOW* $\leqslant w$ from the precondition [P].

We now check that *Fit* maintains the invariant *Representable (a)*. The only place where this property could be rendered invalid is the call to *Split_a_line*, so we must check the body of this procedure. This body is an **if** statement. In the first alternative, a line break is added and *a.height* is consequently incremented by one; *a.width* is also updated so as to maintain the invariant (more details on the statement *update a.width* will be given below). In the second alternative, two cases arise:

- if indentation is possible, the call to *Fit* may be recursively assumed to ensure that d is adjusted to a window $\boxed{lines \mid w.width\text{-}INDENT}$. Attributes *a.height* and *a.width* are then updated to take into account the space which has been allocated to d, so as to re-establish the validity of *Representable (a)*.

- if indentation is impossible, the algorithm distinguishes between two subcases: in the first, zd can be squeezed on a single line, so that an oversize line is transformed into a line of length at most *w.width*; in the other case, *Split_a_line* reports failure to *Fit*, which, as we have seen, takes then a correct decision.

Since we have written the conditional statements using Dijkstra's non-deterministic **if** construct [4,5], we must also make sure that at least one guard is satisfied whenever any such statement is executed. Here only two conditional statements are not trivially equivalent to simple **if ... then ... else ...** statements:

- The outer **if** statement of *Split_a_line* is correct if and only if there is at least one oversize line in the expansion of a; this property is the precondition of the procedure and is indeed ensured by the call to *Split_a_line* in *Fit*.

- The conditional statement in the loop of *Fit* uses guards which have been designed to cover all possible cases.

The **if ... end if** notation was not used for its non-determinism, but because it makes clear under exactly what condition each branch of a condititional is executed. The reader may have noted that there is non-determinism of another kind in the algorithm, since *Fit* may have to select an oversize line (in two instances in the program text) or an operand on such a line (one instance) in a way which has been left unspecified.

8.2 - Termination

Procedure *Fit* is indirectly recursive and contains a loop; both of these features might lead to non-termination. To prove that any correct call to this procedure terminates, we will first check that the mutual recursion between *Fit* and *Split_a_line* may not lead to non-termination, and then that the loop in *Fit* always terminates.

Termination of the mutual recursion results from the fact that each call to *Fit* in *Split_a_line* has as its first argument one of the children of a, the node of the tree which is the second actual argument in the call to *Split_a_line*. In other words, the variant of this recursive scheme is

h - *depth (a)*

where h is the height of the syntax tree and *depth (a)* is the depth of a in that same tree.

To prove the termination of the loop in *Fit*, including the call to *Split_a_line*, we show that this loop has the following quantity as variant:

$v = NRL + NOL + NF$

where

NRL = *w.height-a.height* (*N*umber of *R*emaining *L*ines),

NOL = number of lines of length greater than *w.width* (*N*umber of *O*versize *L*ines),

$NF = 1$ if *failure* is false, 0 if true.

Indeed, at least one of the three terms of the sum v decreases whenever *Split_a_line* is executed:

- NF is incremented in the last alternative of the last innermost **if** statement.

- NOL is decremented in the first alternative of that same statement, which transforms an oversize line zd into a line of length at most *w.width*.

- In all other cases, *a.height* is decremented by at least 1, without ever becoming greater than *w.height*. The only non-trivial case is the one in which indentation is performed (characterized by the guard $w' \geqslant$ *MINWINDOW*). In this case, *a.height* is increased by the value of *d.height* after the recursive call *Fit (w', d⟷)*. The recursive postcondition [Q] ensures that, after this call, *d.window* $\leqslant w'$. It follows from the properties of the division of a window by an integer ($w \oslash i \leqslant w$, see section 5.5), that $w' \leqslant w \ominus a$.

In each of these cases the terms of v which do not decrease remain unchanged. Thus v is a variant for the loop.

8.3 - Quality of the Result

The correctness criteria defined by the above precondition and postcondition require that a correct representation be found for any a; they give no clue, however, as to how "good" a representation must be, so that a solution which would just abstract a in all cases would be considered correct. Evidence to the claim that the above solution is (much) better is given by the Consistent Allocation Theorem (section 5.5), which guarantees that a conservative policy is used for the allocation of windows to indented operands: when space is allocated to d using division of the available window by $\lceil$*remaining_shares / d.share*$\rceil$, it is a consequence of that theorem that no selfish operand d may use the whole window for itself if there remain windows to which space has not been allocated. It is a consequence of the Fairness theorem that the allocation process will observe, at least for indented operands, the hierarchies implied by shares: no "second-rate" operand will get more space than a "VIP".

8.4 - Efficiency

As mentioned earlier, a basic aim was to obtain an algorithm of complexity $O(N)$, N being the number of nodes of the concrete subtree displayed. We now prove that the above algorithm meets this requirement.

Let us first verify that *Fit*, although recursive, is never called more than once for the same node of the tree. This follows from the fact that throughout the execution of any call to *Split_a_line (w, a⟷)* the following property consistently holds for all children d of a:

[IP] *d.processed* $\Rightarrow$ d belongs to a line of length $\leq$ *w.width*.

Property [IP] is proved by noting that:

- whenever *d.processed* is **false**, which we assume to be initially the case for all nodes d, [IP] is trivially satisfied;

- during the execution of a call *Split_a_line (w, a⟷)*, *d.processed* can only be set to **true** for operands d which are children of a; for both recursive calls, the postcondition [Q] of *Fit* implies that, upon return from *Fit*, any such d belongs to a line of length less than or equal to *w.width*.

Since *Fit* may only be called recursively by *Split_a_line* for children d of the argument a which belong to oversize lines, it follows from property [IP] that *Fit* can never be called for d such that *d.processed* is **true**, and thus that it is called at most once for every operand.

This property is, however, not sufficient to prove that the algorithm is linear in the number of nodes: the algorithm uses twice the statement *update a.width* which may seem to imply that a traversal of all the lines of a is required every time a split (cut or indentation) is performed, thus leading potentially to combinatorial explosion.

A simple data structure representation technique solves this problem. From the properties of the $\oplus$ operation, *a.width* is the maximum length of the lines of a. Thus when an oversize line is split into one or more shorter lines, *a.width* will only change if the oversize line being split is longer than all remaining oversize lines in the expansion of a.

We may thus represent the set of oversize lines of a as a sequential list, in such a way that the last element to be considered is the longest (it is not necessary that the list be otherwise sorted). When a line is split, it will produce either one or two shorter oversize lines:

- in the "cut" case, a line xdy is split into lines xd and y;

- in the "indent" case, xd is split into line x, which may still be oversize, and operand d which is passed to *Fit* and thus will **not** be oversize once indented.

If the line which is split is not the unique element of the list of oversize lines of a, it is not necessary to *update a.width*; the only constraint to be observed in this case is that the new oversize lines may be inserted anywhere but at the end of the list, which is occupied by the longest remaining oversize line. When, on the other hand, a unique oversize line is split, we must make

sure that, if two oversize lines are created, the longer comes last in the list; this takes constant time. So the penalty on the overall process is at most linear in the total number of nodes.

The last operation which might endanger the linearity of the algorithm is the computation of *total_shares*, the sum of the shares of all remaining indentation candidates among the children of a, before each tentative indentation. Clearly, *total_shares* can be initially computed by *Measure* and updated every time the fate of one of a's operand children is decided.

Other optimizations are possible. In particular:

- The editor may try to minimize the amount of work performed by the display algorithm by initially pruning of the syntax tree so as to *a priori* eliminate those subtrees which stand no chance whatsoever of being displayed.

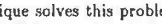
- In the basic loop of the system (see section 4), procedure *Show* will be called after every change resulting from an operation requested by the user. Such a change may involve only a small part of the tree and/or the screen. If this is the case, the editor should only call the display procedure *Show (s, a⟷)* on arguments s and a which denote part of the screen and the tree, respectively. This policy is commendable not only for program efficiency, but also from the human engineering point of view: it improves the user interface by avoiding drastic redisplay of the document and redistribution of its various components over the screen every time a local change is made. Internally, it can be implemented by replacing the *n.processed* attribute by an integer attribute giving the historical index of the last modification (Mikelsons [17] uses a similar scheme).

8.5 - Dealing with Built-in Line Breaks

One aspect of the above algorithm may seem annoying: displaying a long document in a language whose concrete syntax includes compulsory line breaks may result in a degenerate (abstracted) form; this is because *Fit* quits (by *failure*) if the number of lines associated with the focus node, as computed by *Measure*, is initially too large - a condition which will frequently occur for, say, long FORTRAN programs. In many cases, however, a better solution could be found than just abstracting at the uppermost level.

This problem is all the more serious that, when describing a language for Cépage, users may be tempted to add compulsory line breaks in the concrete syntax even for free-format languages, for instance by requiring that Pascal procedures be followed by a break, even though such an explicit addition is unnecessary in view of the algorithm above.

Fortunately, there is a nice solution to this problem. *Fit* does not need to be modified; we just adapt *Measure* so that, in the postorder accumulation of windows which it performs, it transforms non-linear windows into very long linear ones. Let us define the special window *WIDE_WINDOW* as

WIDE_WINDOW = 

where ∞ is a large enough integer (for our purposes, *s.width* + *1*, where *s.width* is the width of the available screen, is a good enough approximation of infinity). Now defining for any window w:

squeezed (w) = if *w.height* = *1* then w else *WIDE_WINDOW*

we just replace *c.window*, in the **for** statement of the last branch of procedure *Measure*, by *squeezed (c.window)*.

This modification solves the problem of built-in line breaks. Any child of a whose expansion initially extends over more than one line will be considered by *Split_a_line* as constituting an oversize line by itself and will thus be the object of a recursive call to *Fit* if its share permits. When *Fit* and *Split_a_line* are first applied to a, only first-level line breaks (those which are part of the concrete syntax for the production defining a, not those attached to its descendants) will be considered.

9 - OUTLINE OF THE ALGORITHM FOR LISTS

The algorithm used for list nodes is a natural extension of the above one for aggregate nodes. It is sketched below.

We consider a list node as equivalent to an aggregate node with one to three children: list header (if present), list body and list tail (if present). The previous algorithm is applied to this structure; only the body has to be subjected to the special treatment described below. We also assume that each element of the list body, except the last, includes the following delimiter.

A new integer attribute is introduced for nodes which are list elements: *d.collapsed* has value 0 if *d* is not is the first element in a collapsed sublist; otherwise, its value is the number of elements in the collapsed sublist beginning with *d*. The head of the last collapsed sublist encountered so far is represented in the following procedure by variable *start*. The fundamental property of the algorithm is expressed by the loop invariant; note that the validity of this invariant results from the Consistent Allocation Theorem. The Fairness Theorem implies that space is allocated to the various elements in a manner which is compatible with their relative importance, as expressed by their "share" attributes.


```
procedure Fit_list_body ( in w: WINDOW;
                          in out a : NODE      - - a represents a list body)
    var collapsing, too_selfish : BOOLEAN ;
      w' : WINDOW ;
      total_shares, remaining_shares, sublist_shares, maximum_ratio : INTEGER ;
      grouped : INTEGER ; start : NODE ;

  collapsing := false ;
  total_shares := Σ (d ∈ children(a)) d.share ;
  remaining_shares := total_shares ;
  maximum_ratio := w ⊘ MINWINDOW ;

  for all children d of a do
  {loop invariant:
              the space allocated so far does not exceed w and
              if d is not the last child of a, then
                  w ⊘ (⌈total_shares / remaining_shares⌉) ⊵ MINWINDOW}
      remaining_shares := remaining_shares - d.share ;
      too_selfish := d is not the last child of a and
                  (⌈total_shares / remaining_shares⌉) > maximum_ratio ;
      if
          not collapsing ⟶
              w' := w ⊘ (⌈total_shares / d.share⌉) ;
              if
                  w' ⊵ MINWINDOW and not too_selfish ⟶
                      Fit (w', d⟷) ;
                      grouped := 0
                [] not w' ⊵ MINWINDOW or too_selfish ⟶
                      collapsing := true ;
                      sublist_shares := d.share ;
                      start := d ; grouped := 1
              end if
        [] collapsing ⟶
              sublist_shares := sublist_shares + d.share ;
              w' := w ⊘ (⌈total_shares / sublist_shares⌉) ;
              if
                  not w' ⊘ 2 ⊵ MINWINDOW or too_selfish
                      ⟶ grouped := grouped+1
                [] w' ⊘ 2 ⊵ MINWINDOW and not too_selfish ⟶
                      start.collapsed := grouped ;
                      Fit (MINWINDOW, start⟷);
                      Fit (w' ⊖ MINWINDOW, d⟷)
                      collapsing := false ; grouped := 0
              end if
      end if
  end do ;

  if
      not collapsing ⟶ skip
    [] collapsing ⟶
          start.collapsed := grouped ;
          Fit (MINWINDOW, start)
  end if
end procedure
```

10 - PRAGMATICS AND CONCLUSION

10.1 - Usage

The first version of the Cépage system was implemented on IBM 3081 hardware, under MVS/TSO. It uses IBM 3279 terminals, and takes advantage of the seven colors and various special effects (reverse video, etc.) available on these terminals to display a clear picture of the document. For example, a syntactic type name like *<statement>* is displayed in a different color depending on whether it represents a non-refined operand or an operand which has been refined but must be abstracted for lack of space.

The system is currently being rewritten and expanded for Vax-Unix and especially for SUN workstations, which provide a nice environment for such software (high-resolution screen, availability of various type fonts, mouse, etc.).

The IBM version uses the display algorithm described in this paper (with some minor differences for list nodes). We have found the results to be up to our expectations; the algorithm displays what we would like it to. We feel that this is a strong case for the fully automatic pretty-printing strategy which we adopted when designing Cépage: the display policy is determined by the system solely from the grammar, using universal rules (of course, the designer of a grammar may add provisions corresponding to special formating requirements).

10.2 - Focus Management

The only serious problem which appeared in actual usage of the display algorithm was connected not with the algorithm itself, but with the way it is used by the rest of the system.

The display procedures are called by the editor with two arguments: a screen area *s* and a focus *f*. Sometimes the focus chosen was not the best possible one. This is because the focus was determined rather conservatively by the editor, so as to be close or identical to the user's logical "focus of interest" (e.g., if the user requests a refinement, the node being refined). In some cases this results in the display not providing enough context. In principle, the solution is simple: choose a focus higher in the tree. The reason a more conservative policy was used was the fear that, in some cases, the user might get stuck by being unable to force the display algorithm to show details he needs to see (e.g. an operand which he wants to refine or explore but whose father in the tree always gets abstracted).

The solution which is currently being implemented relies on the following two techniques:

- Using more boldly the possibility of assigning widely differing "shares" to the various nodes (section 6.5), so that a node can become a "VIP", even if it is far down from the focus, by receiving a high share;

- Dividing the display process into two phases; the first (see section 4) calls procedures *Decorate, Measure, Fit*; the second, *Buildtext* and *Display*. The editor will perform the first phase using an "optimistic" focus, high in the tree. Before performing the second phase, i.e. the actual display, it will test whether any "VIP" node has been abstracted and, if so, will go down to a more conservative focus. Of course, care should be taken to adopt a strategy which will result in the target being hit on the first try most of the time.

10.3 - Implementation

Cépage has been implemented in Pascal; the resulting program includes about 6200 lines, of which about one third are devoted to the display algorithms sketched here. There is also about 4000 lines of supporting software, essentially the Gescran package for screen management [2] and associated tools, mostly in Fortran 77.

We have not performed precise time measurements on the algorithm; in practical usage, the real-time response to all requests was immediate, with no observable delay.

10.4 - On Methodology

One of the essential driving forces in the design of the algorithm described here has been a constant concern for simplicity. We hope this goal has been reached.

The algorithm was conceived as the system was still at the specification and global design stage and described in a first version of this paper [13], written long before any code was produced. The only new concept added since then is the formalization of the "calculus of windows" (section 5) which occurred to us as we were writing this second version. The various invariants and abstract properties were there from the beginning, and we feel that they helped us significantly in getting the design and the code right.

The "calculus" was initially added just for explanatory purposes, but took more importance as we were improving this paper and in fact made it possible to find the solution to the problem of languages with built-in line breaks (section 8.5). This problem had not been evidenced by the first implementation, which we only tested on free-format languages, and when we first discovered it we feared it might require complete re-design of the algorithm; it was thus a relief to find that the simple solution of section 8.5 is obtained by a minor change to procedure *Measure* and fits well into the overall picture. The calculus also allowed us to simplify and improve the algorithm for lists (section 9).

The approach followed for the design of this algorithm might be called "the poor man's formal specifications"; it entails using semi-formal assertions and invariants for the design of algorithms; for the design of the system as a whole, we also used pieces of formal specification, using elements first from the Z specification language [1], then from the M Method [14], which emphasizes modular descriptions (since the first implementation was completed, an almost complete formal specification has been written in M and will serve as a basis for the next implementation).

When describing our approach as "semi-formal", we mean "as formal as one needs to be to get the job done well"; the aim is to obtain the best possible cost-benefit ratio, where the cost is the effort put into specification and global design, and the benefit is quality of the resulting software and speed of implementation (the detailed design and coding of Cépage were performed by one of us, JMN, in ten weeks).

We think that such a moderately formal approach is representative of what can be achieved today, without undue effort, in applying modern software engineering techniques to the design of realistic software.

Acknowledgement

We are grateful to the Editor-in-Chief of SCP for his advice on the paper, which turned out to have a beneficial effect on the algorithm as well, and to Helmut Partsch for his useful role as a referee.

References

1. Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs*, ed. R. McNaughten and R.C. McKeag, Cambridge University Press, 1980.
2. Eugène Audin, Gérard Brisson, Bertrand Meyer, and Françoise Vapné-Ficheux, "Gescran, Manuel de Référence (Reference Manual for a Screen Handling Package)," Atelier Logiciel 22, Electricité de France, Clamart (France), 1980.
3. David R. Barstow, "A Display-Oriented Editor for INTERLISP," in *Interactive Programming Environments*, ed. David R. Barstow, Howard E. Shrobe, Erik Sandewall, pp. 288-299, McGraw-Hill, New York, 1984.
4. Edsger W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *Communications of the ACM*, vol. 18, no. 8, August 1975. Also in R.T. Yeh (ed.), *Current Trends in Programming Methodology*, Volume 1, Prentice-Hall, 1976, pp. 233-242, and in D. Gries (ed.), *Programming Methodology*, Springer-Verlag, 1978, pp. 166-175.
5. Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs (New-Jersey), 1976.
6. Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, ed. David R. Barstow, Howard E. Shrobe, Erik Sandewall, pp. 128-140, McGraw-Hill, New York, 1984.
7. David Gries and Susan Owicki, "Verifying the of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, vol. 19, pp. 279-285, 1976.
8. Nico Habermann et al., *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh (Pennsylvania), 1982.
9. Wilfred J. Hansen, "Creation of Hierarchic Text with a Computer Display," ANL-7818, Argonne National Laboratory, Argonne (Ill), 1971. (Also as dissertation, Computer Science Department, Stanford University, June 1971)
10. C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," in *Symposium on the Semantics of Programming Languages, Lecture Notes in Mathematics*, ed. Erwin Engeler, vol. 188, pp. 103-116, Springer-Verlag, Berlin, 1971.
11. IBM, "System Productivity Facility for MVS - Program Reference," SC34-2038-0, IBM, December 1980.
12. William Joy, *An Introduction to Display Editing with Vi*, Computer Science Division, Department of Electrical Engineering and Computer Science, UC Berkeley.
13. Bertrand Meyer and Jean-Marc Nerson, "Showing Programs on a Screen," Internal Report HI/4590-01, Electricité de France, September 1983.
14. Bertrand Meyer, "A System Description Method," in *International Workshop on Models and Languages for Software Specification and design*, ed. Robert G. Babb II and Ali Mili, pp. 42-46, Orlando (Fl.), March 1984. (also more detailed internal report available from the author).
15. Bertrand Meyer and Jean-Marc Nerson, "Cépage : Un Editeur structurel Pleine Page," in *Second Colloque de Génie Logiciel (Second Conference on Software Engineering)*, pp. 153-158, AFCET, Nice (France), 1984. English translation: "CEPAGE, a full-screen structured editor" in *Software Engineering: Practice and Experience*, North Oxford Academic, Oxford, 1984, pp. 60-65.
16. 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.

17. M. Mikelsons, "Prettyprinting in an Interactive Programming Environment," *SIGPLAN Notices*, vol. 16, no. 6, pp. 108-116, June 1981.
18. Derek C. Oppen, "Prettyprinting," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 4, pp. 465-483, October 1980.
19. Richard M. Stallman, "EMACS: The Extendible, Customizable, Self-Documenting Editor," in *Interactive Programming Environments*, ed. David R. Barstow, Howard E. Shrobe, Erik Sandewall, pp. 300-325, McGraw-Hill, New York, 1984.
20. Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563-573, September 1981.

CONTROL STRUCTURES:

FUNDAMENTALS

(Chapter 3)

Bertrand Meyer

This is a first draft of Chapter 3 of a book in preparation. The working title of the book is **Applied Programming Methodology**.

The book follows the spirit of **Méthodes de Programmation**, which I co-authored with Claude Baudoin (from Schlumberger); this text was published in 1978 by Eyrolles in Paris. The present work is not, however, a translation of the former one; shortly after publication of the French book, we did consider translating it into English, but for various reasons this project was delayed and it soon became clear that an entirely new design was needed. Claude did not wish to participate in such an endeavor; what follows is thus my sole responsibility.

The projected audience of the book includes practitioners (engineers, programmers, etc.) who are looking for a readable survey on modern programming concepts, as well as students, for whom it is intended as a textbook to be used in connection with courses on programming methodology, programming languages, programming techniques or software reliability.

The book uses several programming languages as a means to exemplify the programming concepts discussed and to deepen their analysis. The languages studied include Fortran, Pascal, Simula 67, Ada, Modula, Lisp and, to a lesser extent, PI/I, Cobol, Algol W, Smalltalk and APL.

The tentative plan of the book is as follows.

Chapter 1: The challenge of software engineering
A short introduction recalling the basic problems of software engineering, summarizing the current state of the art, and describing the "two schools" of software engineering.

Chapter 2: The structure and role of programming languages
A description of the structure of programming languages, introducing the basic issues in language design and discussing the role of languages in programming.

Chapter 3: Control structures: Fundamentals (This Chapter).
An introduction to the basic control structures of sequential programming, using from the outset a systematic, semi-formal approach. Includes a discussion of specification-directed program construction.

Chapter 4: Control structures: Techniques
All elaboration on the concepts introduced in the previous Chapter: variants of the basic patterns; control stuctures as implemented in various languages; technical problems associated with procedures.

Chapter 5: Data structures and their description
An introduction to the practical use of abstract data structure descriptions. Emphasizes hierarchical definition of types and reuse of previously written descriptions (through mechanisms of enrichment and restriction derived from those of Simula, Z and Clear). Offers three levels for the description of data structures: implicit (i.e. by one or more abstract data types), constructive, physical.

Chapter 6: Modularity
A discussion of some of the main requirements for modular programming and of existing techniques. Presents a comprehensive definition of modularity through a set of "criteria", "principles" and "keywords", and shows how modular designs can be obtained. Emphasizes the object-oriented approach and its implementation in such languages as Simula, Smalltalk, Ada and Modula.

Chapter 7: Recursion and Functional Programming
An introduction to the "other culture" of programming, with hints for the practitioner as to how to use its concepts.

Inclusion of the next two Chapters is still a matter of discussion.

Chapter 8: Some fundamental data structures
A systematic presentation of some of the most useful data structures, from specification to implementation, the latter including coding examples in various programming languages.

Chapter 9: Some fundamental algorithms
A systematic presentation of some important algorithms, chosen both for their methodological interest, elegance and practical usefulness.

... The second [precept I devised for myself] was to divide each of the difficulties which I would examine into as many parcels as it would be possible and required to solve it better.

The third was to drive my thoughts in due order, beginning with these objects most simple and easiest to know, and climbing little by little, so to speak by degrees, up to the knowledge of the most composite ones: And assuming even an order between those which do not naturally precede one another.

Descartes, *Discourse on the Method* (1637)

3.1. - CRITERIA FOR CONTROL STRUCTURES

In the previous Chapter we have introduced the basic duality of programming: control vs. data. The present Chapter is a study of the first term in this opposition.

The question at hand is simple: how should we organize the operations of a program?

More precisely, we are looking for a set of mechanisms which will make it possible to construct complete programs by various combinations of the basic statements studied in the previous Chapter, such as assignment, input, output, etc. Such mechanisms, allowing the programmer to prescribe the execution-time sequencing of these basic statements, are called **control structures**.

In the search for a good set of control structures, we shall be guided by four basic criteria: simplicity, clarity, hierarchy and provability.

- **Simplicity**. We are looking for a small set of constructs, easy to understand and remember, yet capable of describing any useful arrangement of operations.

- **Clarity**. A program has a finite text; the object which this text describes is a very complex one, consisting of all the possible runs of the program, that is to say all the possible sequences of executions of its operations (depending upon the input data); this usually includes many repetitions (loops). In other words, the program text is a static description of a set of highly dynamic phenomena. It is crucial that the constructs used for this description give to the reader the clearest possible image of these phenomena.

- **Hierarchy**. It is well-known (at least since Descartes' *Discours de la Méthode*) that a complex object, be it natural (as the system of plants on earth) or human-made (as a mathematical theory) may be understood and mastered only through decomposition. Programs are no exception and the constructs we are looking for should lend themselves to the process of combination and decomposition.

- **Provability**. If "reliable software" is to be more than a catch phrase, one should only write programs whose precise behavior can be unambiguously and easily predicted. This is true of the basic statements described in the previous Chapter: we can rest assured that execution of the assignment $y:=x-1$ will lead to a state where $y \geq 0$ if an only if x was initially greater than or equal to 1 (at least assuming perfect arithmetic, no overflow, etc.). It should be possible to draw the same kind of conclusions for actual programs built with control structures. Note that we shall be less interested in **proving** that programs are "correct" after they have been written than in **constructing** programs in such a way that this correctness becomes obvious.

3.2. - SPECIFICATION OF ACTIONS

3.2.1. - Assertions

By insisting on provability we have hit upon one of the basic problems of software engineering: if we want to be able to check that our programs are correct, we must be able to express what they are supposed to do. The task of defining the function of a programmed system is known as specification; we mentioned in Chapter 1 the importance of this step, and now is our first opportunity to study it concretely.

How can we specify the effect of programming constructs, such as basic statements or control structures? In other words, quoting the title of a classical paper in this field (by R.W. Floyd), how can we *assign meaning to programs*? One popular answer is based on the notion of **assertion**. An assertion is a property involving the objects of a program, such as

f is sorted
$x > 0$
C is busy
$gcd(x,y) = gcd(a,b)$
m is definite positive

(assuming i.e. that f is a file, x,y,a,b are integers, C is a communication line, m is a matrix).

3.2.2. - Annotated programs

To express the meaning of a program statement, we may give the relationship between assertions which are true before and after execution of this statement. For example, if the assertion $x > 1$ is true at some point and the statement $y{:=}x{-}1$ is executed, then the assertion $y > 0$ will be true subsequently.

We shall express such facts by writing the assertions in the programs themselves, as special comments delineated by braces:

{...*assertion*...}

In this way our program will be **annotated** by arguments (assertions) pertaining to show their correctness. For instance, we shall write the annotated program

{**given** $x > 1$} $y := x\text{-}1$ {**then** $y > 0$}

to express the above property $x > 0$, assumed to be true before execution of the statement $y{:=}x{-}1$, is called the **precondition** of this statement in the case considered here; $y > 0$ is the corresponding **postcondition**.

In order to fully understand an annotated program, it is important to see clearly which properties are assumed to be satisfied before its execution begins, and which one will be guaranteed to hold after its execution; hence the keywords **given** (for preconditions) and **then** (for postconditions) in our notation for assertions.

3.2.3. - Strongest postcondition, weakest precondition

In the example above, $y > 0$ is not the only postcondition we may attach to $y{:=}x{-}1$ given the precondition $x > 1$; we may as well write the correct annotated program elements

{**given** $x > 1$} $y := x\text{-}1$ {$y > -1$}
{**given** $x > 1$} $y := x\text{-}1$ {$y \neq 0$}

etc. It is clear, however, that they are not as interesting: the postcondition $y > 0$ embodies the maximum information we may assert to be true after execution of $y{:=}x{-}1$, starting in a state where $x > 1$. Assertion $y > 0$ is thus called the **strongest postcondition** of the statement with respect to the given precondition.

Similarly, if we take $x > 2$, $x = 7$ etc. as preconditions in this example, with $y > 0$ as postcondition, we still get correct annotated programs. However, $x > 1$ is the least constraining precondition which will ensure $y > 0$ as postcondition after execution of $y{:=}x{-}1$. It is thus called the **weakest precondition** of this statement with respect to the given postcondition.

An important property of annotated programs is that, starting from a correct annotated program, we may always replace the precondition with a stronger one and/or the postcondition with a weaker one, and still have a correct annotated program. That is, if

{**given** P} A {**then** Q}

is a correct annotated program, and if P' and Q' are such that $P' \Rightarrow P$ (P' implies P) and $Q \Rightarrow Q'$, then the following are correct annotated programs:

{**given** P'} A {**then** Q}
{**given** P} A {**then** Q'}
{**given** P'} A {**then** Q'}

This property means that weakest precondition-strongest postcondition pairs contain more information than other pre-post pairs and are thus to be preferred. Once available, they make it possible to deduce many other pre-post pairs.

3.2.4. - Specifying statements

To specify the meaning of any statement A, we will use the above principles: i.e. we will try to characterize statement A by a pair of assertions, P, Q, so that we may write

{**given** P} A {**then** Q}

where P is the weakest precondition of A with respect to Q and Q is the strongest postcondition of A with respect to P; i.e. Q will be true after execution of A if and only if P was true before.

But in the example above the chosen precondition and postcondition, even though they form a weakest-strongest pair, are not general enough yet: other weakest-strongest pairs seem just as informative, e.g.

{**given** $x \leq 0$} $y := x - 1$ {**then** $y \leq -1$}

or

{**given** $x = 7$} $y := -1$ {**then** $y = 6$}, etc.

The rule should thus involve not just a pre-post assertion pair, but a whole class of such pairs. Here the most general property we may express about the assignment $y := x\text{-}1$ is that whatever condition we choose on y will be true after execution of this statement if and only if it was true of $x{-}1$ before.

In other words, for **any assertion** P, we may write

{**given** P_y^{x-1}} $\quad y \leq x - 1 \quad \{P\}$

where P_y^{x-1} (read "P with $x{-}1$ for y") means "assertion P, with $x{-}1$ substituted for every occurrence of y". For instance, if P is the assertion $y > 0$, then P_y^{x-1} is $x{-}1 > 0$; if P is $x{+}y > 0$, $P_{y_{x-1}}$ is $x{+}x{-}1 > 0$ (an assertion which, because of the properties of numbers, is equivalent to $2 \times x > 1$); etc. Thus, the rule for the assignment $y{:=}x{-}1$, as evidenced by the previous examples, is that one gets the precondition from the postcondition by substituting $x{-}1$ for y. Note that this mechanism gives a weakest precondition-strongest postcondition pair.

This rule is a particular case of the general rule for assignment, namely: for any statement of the form $v{:=}e$, where v is a variable and e an expression, and for any assertion P, we may write

{**given** P_v^e} $\quad v{:=}e \quad$ {**then** P}

where P_e^v means "P, with every occurrence of v being replaced by e".

Note that this holds whether or not the variable v appears in the expression e. The reader is invited to apply this rule to the following annotated program examples, then check the results with his intuition about the effects of the statements.

{**given** $x > 0$} $x := x + 1$ {**then** $x > 1$}

{**given** $x < 0$} $x := x + y$ {**then** $x < 2 \times y$}

{**given** $|x| \neq |y|$ } $x := \frac{1}{x^2-y^2+1}$ {**then** $x \neq 1$}

{**given** *first non-blank character in file f is alphabetic*}

x:= *first non-blank character of f*

{x *is alphabetic*}

{**given** ?} $x := x^2 - y^2$ {**then** $y = \frac{x}{2}$} (find weakest precondition)

{**given** $x > y$} $x := x^2 - y^2$ {**then** ?} (find strongest postcondition)

A very important point regarding the rule for assignment is that it works from right to left: the precondition is obtained from the postcondition (through a substitution of variables), rather than the opposite. This has important consequences on the way program correctness is checked in this method (see section 3.2).

At this point, the reader may be little puzzled. What is the purpose of all this game with assertions? What have we gained? So far, very little. Most of the properties of statements which we have obtained are trivial. Yet we have established a framework which will enable us to assert more and more interesting facts about programs. By annotating programs, we further our goal of obtaining static descriptions of dynamic phenomena: while a statement is an **event**, an assertion is a **property**; the former is dynamic, the latter static, as a mathematical theorem which expresses that certain objects satisfy certain conditions.

Thus there is an important conceptual difference between the statement x:=y and the assertion x=y. The former is an instruction which is to be carried out by a computing system; the latter is a property which may or not be satisfied by variables x and y. This is why the pre- and postconditions associated with a statement, or rather the relationship between them, may be said to give the meaning of that statement.

Any statement may thus be viewed as an **assertion transformer**; this is also true of any program, as characterized by its precondition (hypothesis on the input data) and postcondition (conclusion, i.e. requirements on the computed results). Explicit writing of these assertions will help remind us that a program is not just a sequence of computer codes put together haphazardly, but the model of a process designed to solve a definite problem. Here are some examples of how programs may be specified in this fashion:

{**given** *file f consists of records* $f_1,...f_n$, *and every record r has an integer key* $k(r)$}

SORT

{**then** *g consists of records* $g_1,...g_n$ *which are a permutation of* $f_1,...f_n$
and $k(g)_1 \leq \cdots \leq k(g_n)$}

{**given** *A is a non-singular* (n,n) *matrix; b is an n-vector;* $\epsilon > 0$}

LINEAR-SOLVER

{**then** *x is an n-vector* **and** $\| Ax-b \| \leq \epsilon$}

{**given** *J is a finite sequence of user requests*}

OPERATING_SYSTEM

{**then** *all requests in J have been correctly processed*}

As it is clear from these examples, we are not very firm about the language in which assertions are expressed. Whenever adequate, we shall use predicate calculus (i.e. logical formulae with symbols like **and**, **or**, **for all**, **for some**, etc.); in many cases, however, we shall sacrifice rigor for ease of expression and understanding, and use English phrases.

Having now introduced the framework which will enable us to reason about control structures, we are ready to study the basic set of structures which will be used in the rest of this book.

3.3. - FOUR STRATEGIES

Control structures are tools for constructing programs, that is, for solving problems.

When faced with a problem to solve, one may adopt several strategies. Four of the most common ones are summed up in figure 3.1.

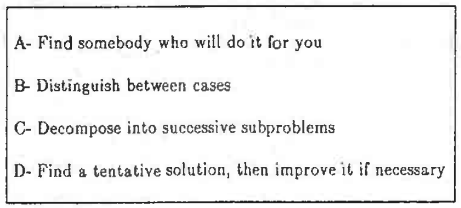


Figure 3.1: Four ways to solve a problem

Most people use these four strategies daily. In programming, they give rise to four fundamental control structures.

- A gives the procedure **call**
- B gives the **conditional**
- C gives the **sequence**
- D gives the **loop**

We now study these four structuring mechanisms, in this order.

3.4. - PROCEDURES

3.4.1. - Definition

A procedure is used whenever there is a need to refer to a certain action, the effect of which is well defined, without giving the details of how it is to be carried out.

A procedure is defined by a procedure **declaration**. It may be used through a statement termed a procedure **call**.

A procedure declaration comprises two parts: a **specification**, which describes the effect of the action associated with the procedure; and an **implementation**, which is a (possibly complex) statement performing this action.

A procedure call is a statement referring to a procedure specification; the effect of this statement, occurring in any program unit which has access to the procedure, is to execute the implementation of the procedure.

The fundamental property of procedures is that calls only make explicit references to the specification, not to the implementation. This means that the procedure is known to the outside world by its effect, not by the way it achieves this effect. Thus a procedure is the **abstraction** of a (possibly complex) statement.

In general, the action associated with the procedure will compute a certain number of objects, called **outputs**, using a certain number of objects, called **inputs**. The inputs may be different in different calls to the same procedure; these calls will then in general yield different outputs.

The specification of a procedure thus comprises three elements:

- 1- The name of the procedure;
- 2- A set of requirements for the inputs to be given to the procedure in any call;
- 3- The properties of the outputs computed by the procedure in any call.

In simple cases, the effect of the action associated with the procedure call is entirely defined by (2) and (3) above. (For more complicated cases, we shall introduce "inputs-outputs" below).

The implementation of the procedure will be a statement whose effect is to compute outputs as required by (3), using inputs satisfying the requirements of (2).

The call to the procedure will include a reference to its name, a set of objects satisfying the requirements of (2), to be used as inputs, and a set of objects to be used as outputs; execution of the call will ensure that they satisfy properties in (3).

The inputs and outputs used in any particular call are called the **actual arguments** (or **actual parameters**) for this call.

3.4.2. - Use of procedures; top-down, bottom-up

Procedures give a way of referencing a possibly complex course of actions by a single name. They play a fundamental part in the construction of programs; they may be used in two manners:

- A- In the design of a program, it frequently happens that the need for an action satisfying certain requirements is recognized, but one does not want to spell out the details immediately. In such a case, a sensible thing to do is to write a procedure specification corresponding to these requirements, and use a procedure call in lieu of the required action. The implementation of the procedure will be written only later, after the specification. This technique allows one to concentrate on *what* an action results in rather than on *how* it does it: it uses procedures as a mechanism of **abstraction**.
- B - When a certain problem which recurs frequently has been solved once, it is good practice to record the solution as a procedure, with a well-documented relationship between input and output, so that it may be used again whenever a similar problem arises later. In this case, a complete specification may be written after the implementation. Procedures used in this fashion add new elements to the set of available operations, as offered by the programming language, operating system and underlying hardware. This use of procedures may thus be characterized as a mechanism of

extension.

These two uses of the procedure concept correspond to two basic approaches to program design: the so-called *top-down* and *bottom-up* strategies, respectively. A convenient way to describe them is to use a diagram (fig. 3.2) which is ordered in *levels of abstraction*, the highest one being that of the problems to be solved, the lowest one being that of the machine in terms of whose operations the solution will have to be expressed. Note that this machine is usually **not** the physical machine, but the **virtual machine** for which programmers actually write programs. It is defined by the set of possibilities offered by the combination of a programming language and accessible features of the hardware and operating system (e.g. the "Fortran - MVS" machine, the "C + Unix" machine, the "Lisp + Lisp machine" machine, etc.).

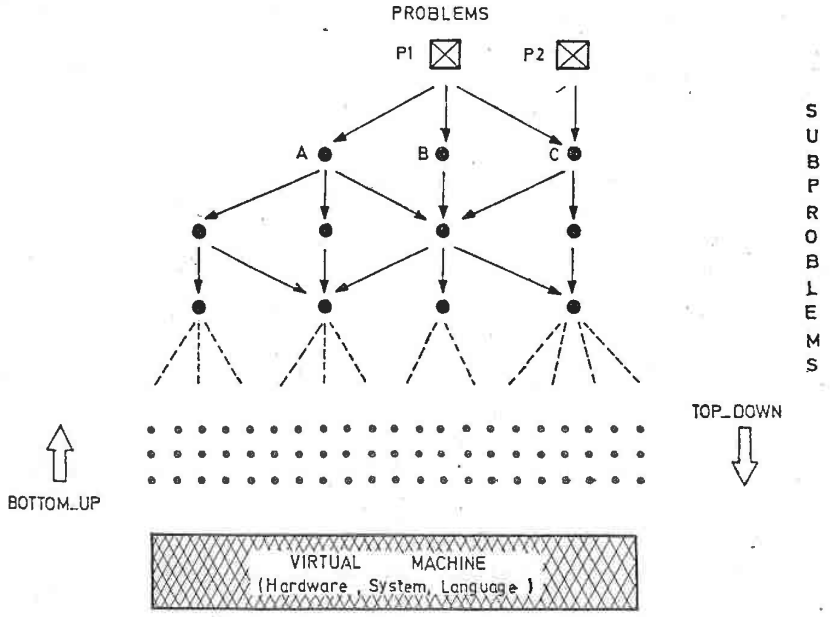


Fig. 3.2 - Top-down, bottom-up

In the top-down method, one starts from the specification of the problems to be solved and expresses their solutions in terms of the solutions (yet to be worked out) of a small number of simpler problems (A,B,C, on figure 3.2). The same process is then applied to these new problems, and repeated until one is left with problems so simple that they may be solved by applying operations of the virtual machine, e.g. programming language statements. The method is also called, for obvious reasons, **programming by stepwise refinements**. It will only work if it indeed leads to the "bottom", that is to say, to a state where everything is expressible in terms of the operations of the virtual machine: every refinement of a problem should yield one or more new problems which are actually simpler, i.e. belong to a lower level of abstraction.

The bottom-up method, on the other hand, builds on solid ground by starting from what already exists - the virtual machine and previously written program elements - and combining existing elements to yield more and more complex ones. It will only work if care is taken to ensure that these elements are indeed composable, and that the "top" (a solution to the problems at hand) is indeed reached.

Both of the above schemes actually characterize general approaches to program design rather than precise strategies which can be used as recipes. It would be foolish to dismiss either one.[1] The top-down approach is a very rational way to start working on a problem and decompose it into simpler ones; it is very efficient as a general design guideline. We will however discover, while studying modularity (Chapter 6) that its application meets some severe limitations. The bottom-up method, on the other hand, favors the very important criterion of reusability of software, one of the key issues in software engineering.

Any reasonable design methodology should embody both a top-down aspect (which will put the emphasis on the *new* aspects of the *problem* to be solved every time the methodology is applied) and a bottom-up one (which will emphasize re-use of *existing* hardware and software *tools*).

In both cases, there is a need to name and specify solutions to subproblems; this can be done with procedures.

3.4.3. - Notation for procedures: specification and implementation parts

We will use for procedures a notation which distinguishes between the "specification" part of each procedure and the "implementation" part.

The specification part describes the arguments that the procedure expects and those which it will return, as follows:

procedure *P* **specification**
  in $x_1 : t_1, x_2 : t_2, \cdots, x_m : t_m$
    {**assume** *PRE*} - - *see below*
  out $y_1 : t_1', y_2 : t_2' \cdots, y_n : t_n'$
    {**ensure** *POST*} - - *see below*
end procedure specification - - *P*

The names $x_1, x_2, \cdots x_m$, $y_1, y_2, \cdots y_n$ represent objects or values that will be passed across between the procedure and other program units. They are called the **formal arguments** of the procedure. Formal arguments stand for actual arguments which will be passed to the procedure in any particular call. Note that each formal argument has a specified type, and that any corresponding actual argument should conform to this type.

The implementation part describes statements that fulfill the purpose of the procedure as described in the specification part:

procedure *P* $(x_1, x_2, ..., x_m, y_1 \leftarrow, y_2 \leftarrow, \cdots, y_n \leftarrow)$ **implementation**
  declarations ;
  statements
end procedure implementation - - *P*

[1] Enthusiasts of the top-down method sometimes refer to Descartes as having shown the way. The quotation at the beginning of this Chapter (a famous excerpt from the preface to the *Discours de la Méthode*) shows very clearly that Descartes' approach to problem solving included both a top-down and a bottom-up component. It is interesting in this respect to compare his second and third "precepts".

As shown here, we repeat the formal arguments, in parentheses, in the implementation part, as a reminder to readers of the procedure implementation. We include only their names, however, and do not repeat their types. We mark the **out** arguments with an arrow, ←, to remind the reader that the procedure must assign a value to the corresponding actual arguments.

In some cases, we may wish to give the specification and implementation together; we shall then merge the two notations as follows (without repeating the formal arguments in the implementation part):

procedure *P* **specification**

in $x_1 : t_1, x_2 : t_2, \cdots x_m : t_m$
{**ensure** *PRE*} - - *see below*

out $y_1 : t_1', y_2 : t_2', \cdots y_m : t_m'$
{**assume** *POST*} - - *see below*

implementation

declarations ;

statements

end procedure - - *P*

3.4.4. Notation for procedures: assertions

In accordance with the discussion at the beginning of this Chapter (3.2), we would like the specification part to give not only the list of all **in** and **out** formal arguments together with their types, but also an abstract description of the procedure's effect. Keeping in line with the above presentation, we will characterize the procedure's effect by assertions: a precondition, which will appear after the list of formal input arguments, preceded by the keyword **assume**, and a postcondition, which will appear after the list of formal output arguments, preceded by the keyword **ensure**. So the complete form of a procedure specification is as follows:

procedure *P* **specification**

in $x_1 : t_1, x_2 : t_2, ..., x_m : t_m$
{**assume** *PRE* $(x_1, x_2, ..., x_m)$};

out $y_1 : t_1', y_2 : t_2', ..., y_n : t_n'$
{**ensure** *POST* $(y_1, y_2, ..., y_n, x_1, x_2, ..., x_m)$}

end procedure specification - - *P*

A procedure specified in this fashion, where *PRE* (...) and *POST* (...) are assertions, will yield, when called with input arguments satisfying *PRE*, output arguments satisfying *POST*.

Note that *PRE* may only depend on the input values, but *POST* will in general involve both output and input arguments, since it is the procedure's purpose to compute the former from the latter.

For example, we could specify in the following way a procedure which computes the maximum value contained in a file of real numbers:

procedure *File_maximum* **specification**

in *f* : **file of** *REAL*
{**assume** *f is non-empty*}

out *x* : *REAL*
{**ensure** $x = a$ **for some** a **in** f **and** $x \geq b$ **for all** b **in** f}

end procedure specification - - *File_maximum*

Note that by requiring precise **assume** and **ensure** clauses, and giving reasonably meaningful names to procedures, we do not need the header comment usually required by good programming practice for each procedure (e.g. "compute the maximum of file *f*, assumed to be non-empty, and return it via *x*").

The implementation of a procedure specified in this manner must be a statement *IMP*, such that the following annotated program fragment is correct:

{**given** *PRE* $(x_1, x_2, ..., x_m)$}

IMP

{**then** *POST* $(y_2, y_2, ..., y_n, x_1, x_2, ..., x_m)$}

In the implementation of the procedure (*IMP*), the names of the input arguments $(x_1, x_2, ..., x_m)$ will be considered to denote **constants** of the appropriate types $(t_1, t_2, ..., t_m)$: their values may be used, **but not changed**. The output arguments $(y_1, y_2, ..., y_n)$ will be considered as **variables** of the appropriate types $(t_1', t_2', ..., t_n')$, uninitialized upon activation of the procedure.

Note that the requirement that **in** arguments be left unchanged implies that the precondition expressed in the **assume** clause still holds when execution of the procedure terminates (this will not be true any more when we introduce **in out** parameters in 3.4.5 below). This requirement also eliminates trivially wrong implementations, such as realizing *file_maximum* above by assigning 0 to x and assigning to f the value of a file with a single zero element.

Here is an example of a procedure specification and implementation (the example is trivial because we have not yet introduced the other control structures necessary to describe interesting computations):

procedure *Compute_account_balance* **specification**

in *credit, debit: REAL*
{**assume** *credit* $\geq$ *debit*};

out *balance: REAL*
{**ensure** *balance* = *credit-debit*}

implementation

{**given** *credit* $\geq$ *debit*}

balance := *credit-debit*

{**then** *balance* = *credit-debit* **and** *balance* ≥ 0}

end procedure - - *Compute_account_balance*

To denote a procedure **call**, we will follow the Algol tradition by just writing the name of the procedure (without any special "call" verb as in Fortran, Cobol or PL/I) followed by a list of actual arguments, i.e. objects of the calling program which correspond one by one to the formal arguments, as follows:

$P(a_1, a_2, \ldots, a_m, b_1\leftarrow, b_2\leftarrow, \ldots, b_n\leftarrow)$

Every actual argument must be of the same type as the associated formal argument.

The details of argument transmission are studied in Chapter 4. At this point, it suffices to note that actual arguments corresponding to **out** formal arguments will be assigned a value by the procedure; to make sure the reader of the program is aware of this important feature of the procedure call, we put an arrow (denoting assignment) after the name of any such actual argument (e.g. $b_1\leftarrow$). Note that any program object used as **out** actual argument must be assignable; thus it must be a variable or array element etc., but not a constant or an expression.

A program calling the two example procedures specified above could have the form:

```
variables g : file of REAL,
          y : REAL,
          b, c, d : REAL ;
read g; - - the file read should be non-empty
File_maximum (g, y←);
. . . . . ;
c := .... ;
d := .... ;
Compute_account_balance (c, d, b←)
```

3.4.5. - The case of in out arguments; snapshots

For some procedures, we need arguments of mode **in out**, i.e. arguments which stand for objects of the calling program whose value may be used **and** changed by the procedure. In theory, the **in out** mode is superfluous, since an argument of this mode may always be replaced by two arguments of the same type, one **in** and one **out**. When, however, it is known that after every call of the procedure the value of the in actual argument will not be needed any more, then merging the two arguments into one will save space in the calling program (and perhaps in the procedure as well, depending upon the passing mechanism which is used: see Chapter 4).

In the same way that our notation draws the reader's attention on **out** parameters in parameter lists (both in calls and implementation parts) by drawing a simple arrow after the argument name (e.g. $a\leftarrow$), we will signal **in out** parameters by a double arrow (e.g. $a\leftrightarrow$).

Arguments of mode **in out** raise some difficulties with respect to the specification of procedures. We will include their list between those for **in** and **out** arguments. Their values will usually be needed for reference in both the expected precondition (**assume**) and the guaranteed postcondition (**ensure**), so that we shall write the former after the **in out** list and the latter after the **out** list, as follows:

procedure *Q* **specification**

in $x_1 : t_2,\ x_2 : t_2\ ,\ldots,\ x_m : t_m$;
in out $z_1 : t_1,\ z_2 : t_2\ ,\ldots,\ z_p : t_p$
{**assume** *PRE* $(x_1, x_2, \ldots, x_m, z_1, z_2, \ldots, z_p)$} ;

out $y_1 : t_1', y_2 : t_2', \ldots, y_n : t_n'$
{**ensure** *POST* $(y_1, y_2, \ldots, y_n, x_1, x_2, \ldots, x_m, z_1 z_2, \ldots, z_p)$}

end procedure specification - - *Q*

One should note, however, that the names of the **in out** arguments $(z_1, z_2, \ldots, z_p)$ as they appear in the **ensure** clause refer to the final values of the corresponding objects (i.e. the values which the procedure has assigned to them when it terminates its execution), whereas the same names denote the initial values in the **assume** clause. Usually, one will want to characterize the new values in terms of the old ones, so one should have some way of recording the latter.

For example, if we are specifying a procedure which writes an element at the end of a file, this file will be denoted by an **in out** parameter; the specification should say that the new file has the same elements as the old one, plus one at the end. So, in order to be able to write the postcondition, we need a way of denoting both the initial and final states of the file.

To be able to express such requirements, we introduce the notion of **snapshot**. A snapshot is a name associated with the value which a certain object of a program (e.g., a variable, an argument, etc.) takes at a certain point in the execution of this program.

A value becomes associated with a snapshot in a certain assertion; to denote this association, we will use the assignment notation in the assertion:

{ - *assertion*... ; $s := v$}

where v is an object of the program, and s is the snapshot. This will be called a **snapshot assignment**. Note, however, that a snapshot is **not** equivalent to a variable; it is in fact a much simpler object. We enforce the following rule, which seriously restricts the manipulations which may performed on snapshots:

Snapshot Rule
Let *sn* be a snapshot. There can be at most one snapshot assignment to *sn* in the program unit in which *sn* appears.

Thus a snapshot may only be used to record the value of a certain program variable or expression at one point of the program, so that further assertions may refer to that value. None of the games permitted with variables, like changing their values, passing them as actual parameters to a procedure, etc., is applicable to snapshots.

Snapshots will allow us to specify procedures with **in out** arguments, as in the following example specifications:

procedure *Round_to_next_power_of_two* **specification**
in out x : *INTEGER*
{**assume** $x \geq 0$; $x_0 := x$};

{**ensure** $x < 2 \times x_0 \leq 2 \times x$ **and** $x = 2^n$ **for some** n **in INTEGER**}
end procedure specification - - *Round_to_next_power_of_two*

procedure *Sort_file* **specification**
(**in out** f : **file of** T
{**assume** .... ; $f_0 := f$}

{**ensure** f *is a permutation of* f_0 **and** f *is sorted*}
end procedure specification - - *Sort_file*

It will also be convenient to use snapshots and snapshot assignment in assertions other than **assume** and **ensure** clauses (provided there is only one snapshot assignment per snapshot). Procedure $SORT_0$, for instance, is a correct implementation of *SORT* if and only if the following is correct:

{**given** $f_0 := f$}
$SORT_0$
{**then** f *is a permutation of* f_0
and f *is sorted*}

Snapshots are only needed in connection with **in out** parameters. An alternative to using them is to have a slightly more complicated notion of assertions, so that postconditions involve not only the current values of program objects, but also their initial ones. This is the solution used in [Jones 80]. It is mathematically more elegant than the use of snapshots, which are a kind of hybrid concept, halfway between the notions of variable in mathematics and programming. This other solution results, however, in assertions becoming longer (since objects will appear twice); moreover, it does not combine well with the idea of annotated program, which we find very useful.

The reader should be warned that the difficulties encountered in dealing with procedures with **in out** arguments do not just stem from technical problems connected with our (or another) notation for procedure specification, but have much deeper roots. They reflect the inherent complication of this notion, having to do with the more general problem of **side effects** (see Chapter 4). In **functional**

programming (see Chapter 7), every procedure is the implementation of a function in the mathematical sense of this term, with only **in** and **out** arguments. One thus avoids all the problems encountered with **in out** arguments.

The main reason for having **in out** arguments is one of efficiency: in most programming environments, we cannot accept that every execution of, say, a "write" operation relative to a file create a new version of the file, or that any array be re-allocated each time it is sorted.

3.5. - CONDITIONALS

We now turn to our second basic structure, the conditional.

As announced in 3.3, conditional statements, or just conditionals, correspond to the problem-solving technique of reasoning by cases. It is useful when a problem to be solved, say P, is the "union" of a certain number of problems, say $P_1, P_2, \ldots, P_n$, in the sense that any instance of P is also an instance of one (or more) P_i. Moreover, each P_i should be easier to solve than P, and there should be a simple way of finding, for any instance of P, what particular P_i it belongs to. In practice, this means that we will need to know, for every i, a computable condition c_i which is true if and only if a given instance of P is also an instance of P_i.

Solving a problem conditionally thus consists in decomposing it into simpler subproblems and constructing solutions to each of these subproblems. The program obtained in this way will, for any instance of the problem, determine which subcase holds, and apply the solution of the corresponding subproblem (figure 3.3).

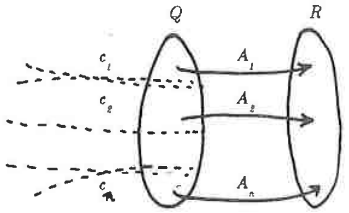


Figure 3.3: Solving a problem conditionally.

More precisely, assume problem P is characterized by precondition Q and postcondition R. A solution to this problem should be a statement S such that the following is a correct program:

{given Q**}** S **{then** R**}**

We will have a conditional solution to P if we know some conditions $c_1, c_2, \ldots c_n$, and some statements $S_1, S_2, \ldots S_n$ such that the following two conditions are satisfied:

- **A**- At least one of the c_i conditions holds whenever Q holds; i.e.:

  $Q \Rightarrow c_1$ **or** c_2 **or ...** **or** c_n

- **B**- Each S_i provides a solution to the subproblem of P corresponding to the case where c_i holds initially (on top of Q); i.e. the following n annotated programs are correct:

  {given *(*Q **and** c_1*)***}** S_1 **{then** R**}**
  {given *(*Q **and** c_2*)***}** S_2 **{then** R**}**
  .....
  {given *(*Q **and** c_n*)***}** S_n **{then** R**}**

If these requirements are met, we can construct a program solution to P, which will work in the following way: in any particular instance of the problem, find a c_i which is true (there must be at least one); then apply the corresponding S_i. The resulting program will be written:

{given Q}
if
$c_1 \longrightarrow S_1$ []
$c_2 \longrightarrow S_2$ []
..... []
$c_n \longrightarrow S_n$
end if
{**then** R}

In this **if** ... **end if** notation for conditional statements, borrowed from Dijkstra[2], the symbol [] serves as a separator between the various branches of the conditional statement; within each branch, the arrow $\longrightarrow$ separates the condition c_i from the associated statement S_i. Each c_i is called the **guard** of the corresponding statement S_i since it controls the condition under which this statement will be executed.

It is important to note that the correctness of this program will only be guaranteed if both conditions **A** and **B** above hold.

A few examples follow. The reader is invited to check their correctness, using the rule for assignment (3.2.3) wherever appropriate.

procedure *Absolute_value* **specification**
in x : *REAL*;
out a : *REAL*
{**ensure** $a = |x|$}
implementation
if
$x \geq 0 \longrightarrow a := x$ []
$x \leq 0 \longrightarrow a := -x$
end if
{**then** $a = |x|$}
end procedure - - *Absolute_value*

[2] We use **end if** rather than Dijkstra's **fi** (which is **if** spelled backwards).

procedure *Maximum_of_three* **specification**
in x_1, x_2, x_3 : *REAL*,
out y : *REAL*
{**ensure** $y = max\ (x_1, x_2, x_3)$}
implementation
if
$x_1 \geq x_2$ **and** $x_1 \geq x_3 \longrightarrow y := x_1$ []
$x_2 \geq x_1$ **and** $x_2 \geq x_3 \longrightarrow y := x_2$ []
$x_3 \geq x_1$ **and** $x_3 \geq x_2 \longrightarrow y := x_3$
end if
{**then** $y \geq x_1$ **and** $y \geq x_2$ **and** $y \geq x_3$}
end procedure - - *Maximum_of_three*

The next example assumes the existence of three procedures, *process_A, process_B, process_C,* which will correctly process a card of type *A, B* or *C* respectively, the type of a card being given by its first character (the reader who finds cards a bit out of fashion may replace the word "card" with "user's request" or "mouse selection in a menu").

procedure *Process_card* **specification**
in *card* : **array** *[1..80]* **of** *CHARACTER*
{**assume** *card[1]* = *'A'* **or** *card[1]* = *'B'* **or** *card[1]* = *'C'*};
out r : R
{**ensure** r *is the result of correctly processing card*}
implementation
if
card[1] = *'A'* $\longrightarrow$ *process_A (card, r←)* []
card[1] = *'B'* $\longrightarrow$ *process_B (card, r←)* []
card[1] = *'C'* $\longrightarrow$ *process_C (card, r←)*
end if
end procedure - - *Process_card*

Note that the conditional construct in the last example is correct only because the **assume** clause of the procedure specification guarantees that property a in the definition above will hold (at least one of the guards is true whenever this statement is executed). An important feature of the conditional as we have introduced it is that a program containing a conditional statement

if
$c_1 \longrightarrow S_1$ []
$c_2 \longrightarrow S_2$ []
..... []
$c_n \longrightarrow S_n$
end if

is incorrect if it may attempt to execute this statement in a state where none of the c_i guards is true. Such an execution is impossible; if one really wants to imagine what "happens" when it is attempted, one may think of it as raising an error (as when attempting to compute a/b where b is zero), or proceeding

indefinitely without ever yielding control back, let alone producing any result.

Another interesting (albeit surprising at first) feature of this construct is that the guards c_i are not required to be mutually exclusive; if more than one is true simultaneously, the rule is that one of the corresponding S_i statements will be executed (and only one), but there is no way to tell which from the program text (in particular, it does not have to be the first a_i such that c_i is true in the order 1, 2, ...n in which the alternatives are listed. This rule follows from the fact that the n alternatives of the **if...end if** play a symmetric role in the notation, and should thus be treated on a par at execution time. There are other justifications, which will be discussed in Chapter 4.

These two conventions on the **if** statement (run-time error if no guard is satisfied, non-deterministic choice if more than one is satisfied) contrast with what is found in conditional constructs of most programming languages. Their motivations should be clear: provability and reliability. This discussion will be pursued in Chapter 4 when we look at the familiar **if ... then ... else ...** construct found in many languages.

3.6. - SEQUENCING

Sequencing is also a problem-solving technique which entails decomposing a problem into subproblems. Here, however, a solution to the initial problem will be obtained by sequentially composing solutions to the subproblems, rather than by choosing one of them as with the conditional.

Sequencing will be appropriate to solve a problem P whenever there exists problems $P_1, P_2, \cdots, P_n$ such that each of them appears easier to solve than P, and applying successively solutions to $P_1, P_2, \cdots, P_n$ yields a solution to P.

More precisely, let P be characterized by precondition Q and post-condition R; to solve P means to find a statement A such that $\{Q\}\ A\ \{R\}$ is a correct annotated program.

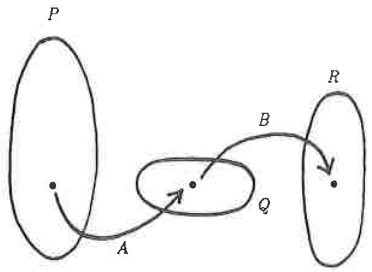


Figure 3.4: Solving a problem by sequencing

A solution to P by sequencing will be obtained if we know n statements $S_1, S_2, ..., S_n$ ($n \geq 1$) and $n + 1$ assertions $S_0, S_1, S_2, \cdots, S_n$, such that $S_0 = Q$, $S_n = R$, and the n annotated programs of the following form are correct (see figure 3.4):

$$\{\textbf{given } S_{i-1}\}\ S_i\ \{\textbf{then } S_i\} \quad (i = 1, 2, \cdots n)$$

The solution to P by sequencing will then be a program whose execution consists in executing statements $S_1, S_2, ..., S_n$ successively, in this order. We will write the resulting program using the semicolon separator introduced by Algol 60:

$$\{\textbf{given } Q\} \quad S_1\ ;\ S_2\ ;\ \cdots\ ;\ S_n \quad \{\textbf{then } R\}$$

If necessary, we will explicitly introduce intermediate assertions:

given Q

S_1 ;
{then A_1**}**

S_2 ;
{then A_2**}**

.....

S_{n-1} ;
{then A_{n-1}**}**

S_n
{then R**}**

A few examples follow.

procedure *Rectangle* **specification**

in *width, height: REAL;*

out *perimeter, area: REAL*
{ensure *perimeter* = 2 × *(height + width)* **and** *area* = *height* × *width***}**

implementation

perimeter := 2 × *(height + width);*
{then *perimeter* = 2 × *(height + width)***}**

area := *height* × *width*
{then *perimeter* = 2 × *(height + width)* **and** *area* = *height* × *width***}**

end procedure - - *Rectangle*

procedure *Compiler* **specification**

in *sp: SOURCE_PROGRAM*
{assume *correct (sp)***};**

out *oc: OBJECT_CODE*
{ensure *oc is a translation of s***}**

implementation

variables *ts : TOKEN_SEQUENCE,*
as : ABSTRACT_SYNTAX_TREE
st : SYMBOL_TABLE;

{given *correct(sp)***}**
LEXICAL_ANALYSIS (st, ts ←);
{then *l is the sequence of tokens from s***}**

SYNTACTIC_ANALYSIS (ts, as ←, st ←);
{then *a is the abstract syntax tree corresponding to l,*
*and st the associated symbol table***}**

CODE_GENERATION (as, st, oc ←)
{then *oc is a translation of [a, st]***}**

end procedure - - *Compiler*

The next example, from numerical analysis, is a classical method of solving systems of linear equations. This example is typical of the way a problem may be solved by reducing it to a sequence of simpler

subproblems. The program below assumes the existence of two procedures: *Triangular-solver*, which solves a linear equation $a \times x = b$ where the regular matrix a is lower-triangular (which makes the task much easier than in the general case), and *Choleski*, which, given a square matrix a, computes two lower triangular matrices a_1 and a_2 such that $a_1 \times a_2 = a$ (this is called "Choleski factorization").

procedure *Linear_solver*
in *a*: **array** *[1..n, 1..n,]* **of** *REAL,*
b: **array** *[1..n]* **of** *REAL*
{assume *a is non-singular***};**

out *x*: **array** *[1..n]* **of** *REAL*
{ensure $a \times x = b$**}**

implementation

variables a_1, a_2: **array** *[1..n, 1..n]* **of** *REAL,*
y: **array** *[1..n]* **of** *REAL;*

{given *a is non-singular***}**
Choleski (a, a_1←, a_2←*);*
{then $a_1 \times a_2 = a$ **and** a_1 *and* a_2 *are lower triangular, non-singular***}**

triangular-solver (a_1*, b, y*←*);*
{then $a_1 \times y = b$ **and** a_2 *is lower-triangular, non-singular,* **and** $a_1 \times a_2 = a$**}**

triangular-solver (a_2*, y, x*←*)*
{then $a_2 \times x = y$ **and** $a_1 \times y = b$ **and** $a_1 \times a_2 = a$**}**
- - *Thus* $a_1 \times a \times x = b$*, giving* $a \times x = b$*.*

end procedure - - *Linear_solver*

3.7. - LOOPS

3.7.1. - Overview

Our fourth and last basic structure, the loop, is an application to programming of the fundamental technique of solving problems by **approximation.** Approximation is a very natural way to proceed when one does not see a direct way to solve a problem: make a guess; then look at the result; maybe you were lucky enough to hit on a solution; if not, try to improve your current estimate, and repeat until you are satisfied with it.

Let us assume again that the problem to be solved is characterized by precondition Q and postcondition R. Q is our hypothesis, or initial assumption, and R is our goal, describing the state that we hope to reach. We are looking for a statement S such that the following is correct:

{given Q**}** S **{then** R**}**

In order to find a solution S in the form of a loop, we need five ingredients:

- two conditions (boolean expressions), the **invariant** I and the **exit condition** E;
- two statements, the **initialization** B and the **transition** T;
- an integer expression, the **variant** V.

These elements must have the following three properties.

- 1 - The invariant I and exit condition E must provide a decomposition of the postcondition R, in the sense that

  $R = I$ **and** E

- 2 - The "initialization" statement, B, must ensure, if executed in a state where the precondition Q is satisfied, that the invariant I becomes true and the variant V (an integer expression involving some objects of the program) becomes non-negative; in other words, the following should be correct:

  {given Q**}** B **{then** I **and** $(V \geq 0)$**}**

- 3 - The "transition" statement, T, when executed in a state where the invariant I holds and the variant V is non-negative but the exit condition E does **not** hold, should yield a state where I is still satisfied (hence the name "invariant"), and V is still non-negative but has decreased (hence the name "variant"). This property may be expressed by the fact that the following annotated program fragment, which uses a snapshot for the initial value of V, is correct:

  {given I **and** *(***not** E*)* **and** $(V > 0)$**;** $V_0 := V$**}**

  T

  {then I **and** $(V_0 > V \geq 0)$**}**

If we have found I, E, V, T and B with these properties, then we can construct a program solution to our problem, which will work in the following way (see figure 3.5): starting from a state where Q is satisfied, execute B, thus yielding a state where I is satisfied. Since $R = I$ and E, we may say that I is an "approximation" of the postcondition R. If E also holds, then the initial "guess" was correct. If not, the program will try to improve it by executing T; this, always performed under falsity of E, will not invalidate I. The process will be repeated until E holds, thus R.

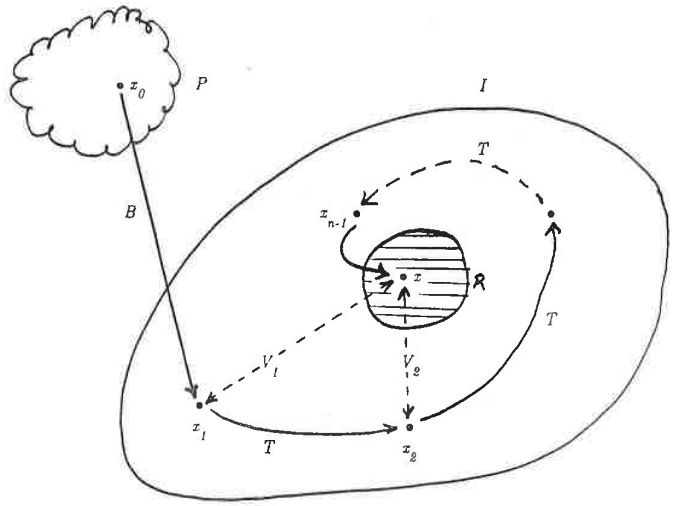


Figure 3.5 - Solving a problem with a loop

The termination of this process is guaranteed by the properties of the variant V: since V is non-negative after the execution of B, and every execution of T (if any at all) will decrease it if started in a state where E is not satisfied, this cannot go on forever (remember that V is integer-valued and T leaves it non-negative). There **must** be a time when E becomes true after a finite number, say n, of executions of T. Note that n may be zero and is less than or equal to the value V has after the execution of B. This initial value thus provides an upper bound on the efficiency of the algorithm used. For this reason, when choosing among several possible variants, we will evidence the smallest one.

The process is very similar to that of computing an approximate solution to a numerical problem. Figure 3.5 is a symbolic illustration. In the "problem space", assertions are identified to subspaces (sets of points which satisfy them). We start from subspace Q and want to reach R, which is part of the larger subspace I; it is the part of I where E holds. B first takes us to some starting position in I. We will then use T to get closer and closer to R, without ever leaving I (the "convergence" region). The variant V may be thought of as the distance to R from any point in I.

Of course, the main difference with mathematical approximation methods is that in programming loops must reach their limits. In mathematical analysis, they usually don't: their variants take real, rather than integer, values, and thus may become infinitely small. For the practical implementation of numerical algorithms, a common practice which brings together the mathematical method and the computational rule is to take $V = \left\lfloor \frac{d}{\epsilon} \right\rfloor$, where d is a bound on the distance to the exact solution, and ϵ, the tolerance, a positive number (for any real number x, $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x).

3.7.2. - Notation

We now introduce a notation for the above process which includes all its "ingredients". We shall write the loop as follows:

```
        {given Q}
from
    B
keep
    I
decrease
    v
until
    g
loop
    T
end loop
        {then R}
```

We illustrate this notation by a few examples before discussing it in more detail.

```
procedure Search_file specification
    in   x: T, - - T is any type
             f: file of T;
    out  isthere: BOOLEAN
             {ensure (isthere ⟺ x appears in f)}
implementation
    var current: T;
    from
        open(f); isthere := false
    keep
        isthere is true if and only if an element equal to x has been read
    decrease
        n - r
        where n = number of elements in f,
              r = number of elements read in f
    until isthere or eof (f) loop
        read (f) current;
        isthere := (current = x)
    end loop
        {then
            (isthere and (current = x)) or
            ((not isthere) and f does not contain any element equal to x)}
end procedure - - Search_file
```



```
procedure Search_ordered_file specification
    in   x: T, - - T is any ordered type
             f: file of T
                 {assume f is sorted};
    out  isthere: BOOLEAN
                 {ensure isthere ⟺ x appears in f}
implementation
    variables current: T,
        one_non_less: BOOLEAN;
            {given f is sorted}
    from
        open(f);
        one_non_less := false
    keep
        one_non_less is true if and only if exactly one element has been read
        which is greater than or equal to x
    decrease
            l - r + 1
        where   l = number of elements of f less than x,
            r = number of elements read in f
                - - Note that the variant of Search-file  is also correct here,
                - - but this one gives some evidence of the efficiency improvement
                - - resulting from the use of the fact that f is sorted.
    until eof(f) or one_non_less loop
        read(f) current;
        one_non_less := (current ≥ x);
    end loop ;
            {then (one_non_less and the first element ≥ x has been read) or
                (not one_non_less) and f contains no element ≥ x)}
    isthere := one_non_less and then (current = x)
end procedure - - Search_ordered_file
```

```
procedure Smallest_integer_with_more_than_6_digit_factorial
    out n: INTEGER
        {ensure n! > 10^6 and (m! ≤ 10^6 for all m in 0..n-1)}
implementation
    var f: INTEGER;
    from
        n := 0; f := 1
    keep
        f = n! and (m! ≤ 10^6 for all m in 0..n-1)
    decrease
        max (0, 10^6 - f)
    until f > 10^6 loop
        n := n + 1; f := f × n
    end loop
        {then f = n! and (m! ≤ 10^6 for all m in 0..n-1) and f > 10^6}
end procedure - - Smallest_integer_with_more_than_6_digit_factorial
```



```
procedure Compute_gcd specification
    in a, b: INTEGER
        {assume a > 0 and b > 0};
    out g: INTEGER
        {ensure g is the greatest common divisor (gcd) of a and b}
implementation
    variables x, y: INTEGER;
    from
        x := a; y := b
    keep
        gcd (x,y) = gcd (a,b)
    decrease
        max (x, y)
    until
        x = y
    loop
            {given x ≠ y and gcd (x,y) = gcd (a,b)}
        if
            x > y  ⟶  x := x - y []
            y < x  ⟶  y := y - x
        end if
                {then x ≥ 0 and y ≥ 0 and gcd (x,y) = gcd (a,b)}
                    - - Note the importance of the assertion x ≠ y
                    - - in the correctness of the if statement.
        end loop
        {then gcd (x,y) = gcd (a,b) and x = y}
                - - Thus gcd (a,b) = x = y
    g := x
end procedure - - Compute_gcd
```

3.7.3. - Discussion

With these examples in mind, we may ponder a little on the benefits and drawbacks of the notation introduced above.

The reader will have noted the differences with the notations of common programming languages. One minor point is the use of

until *exit* **loop** *action*

instead of the more familiar

while *continuation* **loop** *action*

as found in Algol, Pascal, PL/I etc. (the former is found, however in Bliss, Lis, and other languages). The two constructs are equivalent in principle, with *continuation* = **not** *exit_condition*. Choice between the two is mostly a matter of taste. The **while** form makes it immediately clear that *continuation* will be the precondition of *action*; we prefer the *until* form because it emphasizes the exit condition which is true

after execution of the loop, as part of the loop postcondition *exit_condition* **and** *invariant*.[3]

More important is the explicit inclusion of the variant (**decrease** clause) and invariant (**keep** clause) in the program text. This convention may seem a bit puzzling at first sight, since it makes the notation for loops longer, and puts expression of facts about the program on the same step as the actual instructions ("**from** *B*", "**loop** *T*") and test (**until** *exit_condition*) which this program performs. Yet inclusion of these clauses is useful as a means to enhance clarity of programs. Clarity here is taken as including the property that programs **contain their own proof**, so that their correctness (or lack thereof!) can be checked without having to build a complete argument from scratch.

> Although it is certainly possible to use a more traditional notation for loops and include the variant and invariant as comments, we have found that making them a compulsory component of the notation for loops forces one to think about the variant and invariant associated with any particular loop; this, in our experience, gives the programmer a better understanding of the loops he is writing, and enhances the readability of programs (the reader can easily determine for himself the validity of this claim by assessing whether the systematic use of this convention does or does not improve the understandability of the programs in this book and make their correctness easier to check).

It is more surprising that the "**from** *B*" clause is also novel if we compare the above notation with existing programming languages. **Every loop should have an initialization;** the reason for this, as we have seen, is that the initial truth of the invariant should be established before any evaluation of the test (*E*), let alone execution of the loop body (*T*) may take place. It thus seems sensible to include the initialization as a syntactically required component of the loop construct. Indeed, omission of loop initialization is a common programming mistake, which may be hard to detect because the invariant may happen every now and then to hold initially, just by accident.

The only cases for which explicit initialization is not needed are loops with **true** (the condition which always holds) as invariant, usually not very interesting, and loops written at the beginning of the body of another loop, whose invariant implies theirs. We will meet such an example in the next section (and omit the **from** clause). This is, however, a rather special case.

> One of the reasons why initialization has not become a syntactic part of **while** or **until** loops in programming languages may be that the first exposure to loops of many people has been the indexed loop (**for** or *DO*), studied in the next Chapter, which carries its initialization with itself, but in an implicit way. The other reason is that this syntactic extension is not strictly necessary since the initialization may always be written as an independent set of statements before the loop (using sequencing). But it seems better to include in the loop those statements which really belong in it.

[3] Readers familiar with the **repeat...until...** construct found in Pascal, C and other languages should not be misled: such constructs denote a loop whose body is always executed at least once, the test being performed at the end, as opposed to the **while** loop which performs it on entry. Our **from...until...loop...** construct is like a **while** loop in this respect: the only difference with the **while** loop is that one writes the exit condition rather than the continuation condition.

3.8. - ASSERTION-GUIDED PROGRAM CONSTRUCTION

3.8.1. - Overview

In the presentation of four basic control structures in the previous sections, an important part was played by the assertions (pre - and postcondition) associated with each kind of statement. Our motivation for this was to make sure that every programming construct (describing potential events) has a mathematical explanation (in terms of properties). In an "annotated program", the assertions are just as important as the statements which they accompany and justify.

We may give an even more fundamental role to assertions, and consider program statements as deduced, in a certain way, from their pre- and postconditions. This approach may be called assertion-guided program construction; the idea is to try to build the program by working on the specification itself. The important idea here is not that we start from the specification (after all, any program should be built in order to fulfill some specification), but rather that we consider the specification as transformable material of its own, and try to construct the program by first working on this material, transforming it in various ways so as to obtain a form which will more easily yield a program solution.

Such an approach obviously implies that specifications should be expressed in a sufficiently formal way, so that they can be manipulated systematically.

Before we describe some of the techniques which may be used to construct a program by working on its specifying assertions, we should warn the reader about the limitations of this approach: what this section introduces is not a set of recipes for constructing programs "automatically". There is no miracle powder to replace reflection and invention in designing programs. The concepts illustrated below are useful as aids in **understanding** and **explaining** existing programs, and in **guiding** the search for new ones.

3.8.2. - Embedding

We shall concentrate on the systematic derivation of loop algorithms, since they require the most invention on the part of the programmer.

When introducing loops, we defined the loop invariant as an assertion *I* such that the goal (postcondition) *R* may be expressed as

I **and** *g*

where *E* is the exit condition. The invariant may thus be described as a "weakened version of the goal": it is weak enough so that it will be easy enough to ensure it initially; but it is strong enough to yield the goal when combined with teh exit condition. Loop construction strategies are thus strategies for weakening the conclusion in a fruitful way.

This process may be visualized in the following way (see figure 3.6). When looking for a solution to a programming problem, we are trying to find one or more objects satisfying the goal condition in a certain "solution space". A loop solution, using an invariant which is a weakened form of the goal, may be seen as the result of embedding the solution space (whose "characteristic function" is the goal) in a larger one, corresponding to the invariant.

Such an embedding has the property that it is easier to find a starting point in the larger space, but of course there is no guarantee that we will hit the actual solution space right away. The loop solution uses a transformation that, starting from an element in the larger space, will find a new one which closer to the solution space. Figure 3.6 shows a pictorial representation of this strategy.

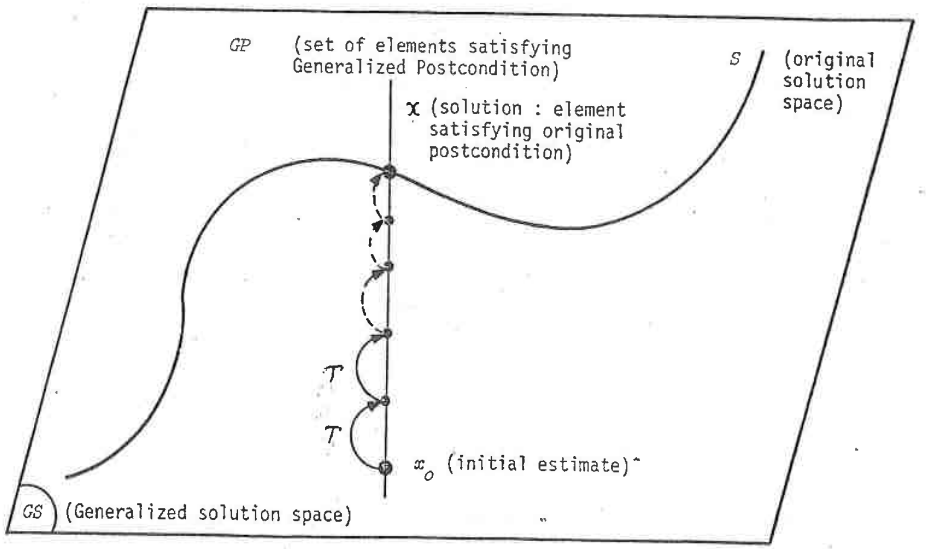


Figure 3.6 : Embedding

Figure 3.6: Embedding

Let S be the original solution space, P the set of elements satisfying the postcondition in S (i.e. the set of solutions), GS the generalized solution space and GP the generalized postcondition (which will serve as invariant). The program resulting from the embedding strategy will have the following form:

variable *s: GS;*
from
    s := "*an element satisfying GP*"
keep
    GP
decrease
    "*distance to S*"
until *s in S* **loop**
    bring s closer to S, preserving GP
end loop

{**then** *GP* **and** *s* **in** *S*} {*i.e. P*}

The remainder of this section will be devoted to the study of some variants of the embedding strategy. We will focus on two of them:

- constant relaxation;
- uncoupling.

3.8.3. - Constant relaxation: Non-ordered table searching

Our first example will illustrate the "constant relaxation" strategy. Assume we have an array of elements of any type T, and an element x of the same type, and we want to determine whether x is equal to any of the elements in t (an easy enough problem, but more difficult ones will come). The specification of the corresponding procedure may be written:

Deciding that the result of the search will be a boolean variable *isthere*, we write down the postcondition as

$$isthere \iff (x = t[i] \text{ for some } i \text{ in } 1..n)$$

(where $a \iff b$ means "either a of b are both true, or a and b are both false).

The specification of the procedure *Find* which we are trying to construct is thus as follows. The dimension n of the array is assumed to be a constant, its value is irrelevant.

procedure *Find* **specification**
    in t : **array** $[1..n]$ **of** T ;
        $x : T$;
    out *isthere* : *BOOLEAN*
        {**ensure** *isthere* $\iff$ $(x = t[i]$ **for some** i **in** $1..n)$}
end procedure specification - - *Find*

The idea of assertion-guided program construction is that we look at the specification (the postcondition) and see how it can be refined to yield a solution (program). If we are looking for a loop solution, the refinement will take the form of a weakening.

Here the postcondition is

$$isthere \iff (x = t[i] \text{ for some } i \text{ in } 1..n) \quad \text{(PC)}$$

We may note that the difficulty of ensuring the validity of PC stems from the presence of the interval $1..n$. If n was replaced with a smaller value, then the problem would be easier; in particular, if n was replaced with 0, then PC would be trivially true by taking *isthere* to be **false**. (note that the interval $a..b$ is empty if $b < a$).

We thus have our embedding, or weakening: introduce a fresh variable, say j, which will take its values in the interval $0..j$, i.e. is declared as

variable $j : 0..n$

and rewrite the postcondition as

$$\boxed{(isthere \iff (x = t[i] \text{ for some } i \text{ in } 1..j))} \text{ and } j = n$$

which is certainly equivalent to PC. Now the first (boxed) part of this condition, say INV, has all the qualifications of an invariant: it is easy to ensure initially (take **false** for *isthere* and 0 for j); it is a weakened form of the postcondition PC, since it coincides with PC for $j = n$; and it is not too hard to maintain it while bringing j a little closer to n, as will be seen shortly. Thus we look for a solution of the form:

```
from
    j := 0 ; isthere := false  {then INV}
keep
    INV
decrease
    n-j
until
    j = n
loop
    "Get j closer to n, maintaining the validity of INV"
end loop
        {then PC}
```

The loop body is easy to obtain. It must be a statement T such that the following is a correct annotated program fragment:

```
    {given INV and not E; V0 := n-j}
T
    {then INV and n-j < V0}
```

where E is the loop exit condition $j = n$. Since j is declared as ranging from 0 to n, the condition **not** E is equivalent to $j < n$.

The simplest way to *Get j closer to n* is to increase it by 1. Thus we are looking for a statement T' such that the following is correct:

```
        {given
            (isthere <=> (x = t[i] for some i in 1..j))
            and j < n}
    j := j+1 ; T'
        {then
            (isthere <=> (x = t[i] for some i in 1..j))}
```

The postcondition is very close to the precondition; more precisely, the precondition implies that after execution of the statement $j := j+1$ the following holds:

(isthere $\iff$ *(x = t[i] for some i in 1..j-1))}*

so that the specification for T' is:

```
        {given (isthere <=> (x = t[i] for some i in 1..j-1))}
T'
        {then (isthere <=> (x = t[i] for some i in 1..j))}
```

The obvious solution is to take for T' the following statement:

isthere := isthere **or** *(t[j] = x)*

which is easily shown to satisfy this specification by applying the substitution rule (3.2.4).

We thus get a correct implementation of procedure *Find*:


```
procedure Find (t, x, isthere←) implementation
variable j : 0..n ;
from
    j := 0 ; isthere := false
keep
    {(isthere <=> (x = t[i] for some i in 1..j))}
decrease
    n-j
until
    j = n
loop
    j := j + 1 ;
    isthere := isthere or (t[j] = x)
end loop
    {then isthere <=> (x = t[i] for some i in 1..n)}
end procedure implementation Find
```

The reader is invited to investigate for himself how the obvious improvement (stop the loop if *isthere* is found to be true) may be carried out in the same rigorous framework.

The embedding method which we have evidenced on this example may be called constant relaxation: it entails replacing a constant of the postcondition (n in our example) by a variable, thus "relaxing" the goal and making it possible to get started by assigning an appropriate initial value (usually far away from the final one) to the variable.

This method is of very general applicability. In particular, it underlies the algorithms which may be described by "for" loops, as studied in the next chapter.

3.8.4. - Uncoupling: Searching sequentially an ordered table

Our second example is a variation of the first and will allow us to evidence another embedding strategy, which we call "uncoupling". This time we assume that the array t which we are searching for an occurrence of x is initially sorted (and will remain so since it is an in parameter that cannot be changed). This requirement implies that there is an order relation on the type T, written $\leq$; by saying that the array t is sorted, we mean that

t[k] $\leq$ *t[l]* **for** $1 \leq k \leq l \leq n$

As before, we decide that the result of the search will be a boolean variable *isthere* and the postcondition is the same as with the non-sorted array example. The specification of the new procedure, which we call *Search*, is thus:

```
procedure Search specification
    in x: REAL, t: array [1..n] of REAL
        {assume t[k] ≤ t[l] for 1 ≤ k ≤ l ≤ n};
    out isthere: BOOLEAN
        {ensure isthere <=> x = t[i] for some i in1..n}
end procedure specification - - Search
```

Note again that by specifying **in** x, t we require that x and t be left unchanged by *Search*, which takes care of some trivially wrong apparent solutions, like assigning the value of x to $t[1]$ and the value **true** to *isthere*. This also ensures that the precondition (t is sorted) will remain invariantly satisfied throughout the program body.

How should we proceed? The previous solution is of course still applicable, but here we would like to take advantage of the fact that the **array** is sorted.

One rather natural remark is that, when *isthere* is true, checking this fact is the same as looking for the index i which appears in the postcondition. If we know such an i in $1..n$, then *isthere* is **true**; i is then such that $x = t\,[i]$.

So when i exists, it gives a simple way of computing *isthere* (by just checking whether $x = t[i]$). It would be quite nice to be able to extend the definition of i so that it always exists and *isthere* may be computed in a simple way by just looking at this new i.

If we use the precondition, i.e. the fact that t is sorted, and define i as the largest index in $1..n$, if any, such that $t[i] \leq x$, then x belongs to t if and only if i is defined and $x = t[i]$. Since the latter condition is very simple to check, it is tempting to start from it when looking for an extended definition $\int i$.

i as defined above does not exist only in the case when x is smaller than the minimum value of t, i.e. $x < t[1]$. It is quite natural in this case to take 0 as the value for i. In this way i is always defined; its precise definition (which is equivalent to the previous one when $i > 0$) may be expressed as:

$\{i \in 0..n$ **and**

$(x \geq t[k]$ **for all** k **in** $1..i)$ **and**

$(x < t[k]$ **for all** k **in** $i+1..n)\}$

(It is essential here to recall that any property of the form "$p(x)$ *holds* **for all** x **in** E" is trivially true whenever E is empty, regardless of what the property $p(x)$ is, and that the interval $a..b$ is empty for $a > b$).

So the problem of writing *Search* may be replaced by that of writing *Searchindex* with the following specification:

procedure *Searchindex* **specification**
  in x: *REAL*, t : **array** $[1..n]$ **of** *REAL*)
    {**assume** $t[k] \leq t[l]$ **for** $1 \leq k \leq l \leq n$};
  out i: $0..n$
    {**ensure** $x \geq t[k]$ **for all** k **in** $1..i$
     and $x < t[k]$ **for all** k **in** $i + 1..n$}
end procedure specification - - *Searchindex*

Indeed, once we know such a correct *Searchindex*, the implementation of *Search* may be simply written as:

procedure *Search* $(x, t,$ *isthere*$\leftarrow)$ **implementation**
  variable i: $0..n$;
  Searchindex $(x, t, i\leftarrow)$;
  isthere := $(i \neq 0)$ **and then** $(t[i] = x)$ }
end procedure *Search*

Note how careful we must be in expressing the postcondition of *Searchindex*: by not giving the right interval, $0..n$, for i, or writing $\leq$ at one of the places where $<$ is required, etc., we would have obtained a

specification which either does not always have a solution, or does not uniquely determine a solution. This, of course, does not mean that the specification above is the only possible one; the reader is invited to derive for himself the specification obtained from initially defining i as "the smallest index, if any, such that $x \leq t[i]$".

How are we to build *Searchindex*? Looking at the postcondition, we see that it has the particular form

$p(i)$ **and** $q(i)$

where

$p(i)$ is $t[k] \leq x$ **for all** k **in** $1..i$

$q(i)$ is $t[k] > x$ **for all** k **in** $i+1..n$

The "uncoupling" strategy applies to postcondition of this general form ($p(i)$ **and** $q(i)$). It is based on the remark that the reason such a postcondition may be hard to ensure is that it is the same i that appears in both p and q. The uncoupling idea thus entails replacing i with **two variables**, say i and j ("uncouple") and looking for an algorithm which will, first, ensure separately the validity of $p(i)$ and $q(j)$; then, bring i and j closer together while maintaining the validity of $p(i)$ **and** $q(j)$, until j becomes equal to i.

In other words, the uncoupling strategy means that we rewrite the postcondition in the exactly equivalent form

$\boxed{p(i) \text{ and } q(j)}$ **and** $(i = j)$

and choose the first (boxed) part of this new postcondition as the invariant, the second part as the exit condition, and the distance between i and j (defined in a suitable way) as the variant for a loop of the following form:

from
  $i := i_0$; $j := j_0$
keep
  $p\,(i)$ **and** $q\,(j)$
decrease
  distance (i,j)
until
  $i = j$
loop
  bring i *and* j *closer*
end loop

This program is correct if $p(i_0, j_0)$ is satisfied, the action "*bring* i *and* j *closer*" conserves $p(i)$ and $q(j)$, and *distance* (i,j) is an integer variant.

Applying this strategy to our problem, we express the postcondition, using a second variable j, also of range $0..n$, as

$p(i)$ and $q(j)$ and $i = j$

with

$$\begin{cases} p(i) = (t[k] \le x \textbf{ for all } k \textbf{ in } 1..i) \\ q(j) = (t[k] > x \textbf{ for all } k \textbf{ in } j{+}1..n) \end{cases}$$

Looking for a loop of type (3) above, we see that the initialization

$i := i_0 \; ; \; j := j_0$

will be correct by taking 0 for i_0 and n for j_0, since $p(0)$ and $q(n)$ are trivially true.

What remains now is to find a way to "*bring i and j closer*" while maintaining the truth of $p(i)$ and $q(j)$ if it is satisfied and the exit condition $i = j$ does not hold.

Since we start from $i = 0$ and $j = n$ and we want to "*bring i and j closer*" until $i = j$, we shall include the property $0 \le i \le j \le n$ in the invariant, which will thus be:

p (i) **and** *q (j)* **and** $(0 \le i \le j \le n)$

The most obvious way to shorten the interval is to increment i by 1, or alternatively decrement j by 1, and see what must be done in order that the invariant still be true. Since the problem is symmetric in i and j (or, more precisely, in i and $j-1$) and there is no clear reason at this point to upset this symmetry, we will consider these two actions on a par.

Assume $p(i)$ **and** $q(j)$ is true, and that the exit condition is not true, i.e. $i < j$. Under what conditions may we execute $i := i + 1$ or $j := j - 1$ and preserve the invariant?

Clearly, we may execute the first of these operations if and only if $p(i+1)$ is true, and the second if and only if $q(j-1)$ is true.

Consider i first. Recalling that

$p(i) = (t[k] \le x$ **for all** k **in** $1..i)$

we see that

$p(i+1) = p(i)$ **and** $t[i+1] \le x$

Note that this is only meaningful if $t[i+1]$ is defined, that is if $i < n$. Thus starting from a state where $p(i)$ is satisfied, we see that we may perform the assignment $i := i + 1$ if and only if

$i < n$ **and then** $t[i+1] \le x$

Similarly, by expressing $q(j-1)$ in terms of $q(j)$, we find that the condition for the second assignment $(j := j-1)$ to preserve the invariant is

$j > 0$ **and then** $t[j] > x$

The extra conditions ($i < n$ and $j > 0$) are indispensable but it turns out that they are satisfied when the loop body is executed: the invariant includes $0 \le i \le j \le n$, and the negation of the exit condition is $i < j$, so that their combination implies that both $i+1$ and j lie in the interval $1..n$. Thus we may use just

$t[i+1] \le x$

and

$t[j] > x$

as respective guards for the two statements. A tentative loop body is thus the following conditional statement:


```
if
    t[i+1] ≤ x  ⟶  i := i+1 []
    t[j] > x  ⟶  j := j-1
end if
```

We have almost found a solution, since we know that each branch of this conditional, executed under the condition that its guard is true as well as the invariant and that the exit condition is false, will maintain the invariant and decrease the variant. But we have to be careful: as mentioned in section 3.5 when we introduced the conditional statement, a conditional is only correct if at least one of its guard is satisfied whenever it is executed. Is it always the case that $t[i+1] \le x$ or $t[j] > x$ when the loop body is executed? The answer is yes: this property is implied by the fact that the array is sorted; since $i < j$ in the loop, the negation of the first guard, namely $t[i+1] > x$ implies $t[j] > x$, namely the second. Thus the conditional statement is safe.

So we have a simple and correct version of *Searchindex:*

```
procedure Searchindex (x, t, i←) implementation

    variable j: 0..n;

    from
        i := 0; j := n
    keep
        p(i) and q(j) and 0 ≤ i ≤ j ≤ n
    decrease
        j - i
    until
        i = j
    loop
        if
            t[i+1] ≤ x  ⟶  i := i+1 []
            t[j] > x  ⟶  j := j-1
        end if
    end loop
end procedure implementation - - Searchindex
```

Now we are in for a small surprise: the program which we have obtained is not exactly the way sequential search is usually written! The reason is that we kept the symmetry between i and $j-1$. If we forget about this (esthetic) constraint, we may note that if the first guard is false, i.e. $t[i+1] > x$, then the invariant will still be preserved if we immediately assign the value of i to j. The loop body may thus be replaced by:

```
if
    t[i+1] ≤ x  ⟶  i := i+1 []
    t[i+1] > x  ⟶  j := i
end if
```

(of course, we could have performed the symmetric change instead). Now we may dispense with variable j altogether, by noting that loop termination occurs when either $i = n$ or $t[i+1] > x$, thus yielding the following form, which is sequential search, written in the more usual fashion:

```
from i := 0
keep ... decrease ...
until i = n or else t[i+1] > x loop
        i := i+1
end loop
```

The reader is requested to complete the **keep** and **decrease** clauses of this loop.

3.9. Uncoupling revisited: binary search

The efficiency improvement which we obtained by removing the symmetry between i and $j-1$ was marginal at best. There is a much more promising avenue for improving the efficiency of sorted table searching based on the fact that t is an array. The basic property of arrays is "direct access": any element may be accessed or modified directly (in constant time) if its index is known.

The idea here, which you will have recognized as as the principle leading to the well-known idea of **binary search**, is to try to "*bring i and j closer*" faster than just one step at a time. Before reading the rest of this section, the reader is urged to try to write a simple binary search program and make sure that it is correct. For a start, you may take a look at the four programs in figure 3.7; it turns out that these four programs are all *wrong*; you should convince yourself of this by finding, for each of them, a case for which it fails to terminate, exceeds the array bounds, or yields a wrong answer.

The remark on the basic property of arrays can be interpreted in the framework of the previous discussion. Since the array is sorted, comparing x with any element whose index lies between i and j in t (not just $t[i+1]$ or $t[j-1]$) makes it possible to discard a whole interval for the rest of the search. More precisely, let m be such that

$$i \leq m \leq j$$

Assume that $t[m]$ is defined, i.e. $m \in 1..n$. Then if $t[m] \leq x$, we can infer that

$t[k] \leq x$ **for all** k **in** $1..m$

that is, $p(m)$ is true. In the other case, $t[m] > x$, we have that

$t[k] > x$ **for all** k **in** $m...n$

or, to put this in "q" form:

$t[k] > x$ **for all** k **in** $(m-1)+1..n$

That is, $q(m-1)$ is satisfied.

Figure 3.7: Four programs for binary search

```
variable i, j, m : INTEGER ;

from
    i := 1 ; j := n
until
    i = j
loop
    m := ⌊(i + j) / 2⌋ ;
    if
        x ≤ t[m] ⟶ j := m []
        x > t[m] ⟶ i := m
    end if
end loop ;
isthere := (x = t[i])
```

```
variable i, j, m : INTEGER ;

from
    i := 0 ; j := n
until
    i = j
loop
    m := ⌊(i + j) / 2⌋ ;
    if
        x ≤ t[m] ⟶ j := m []
        x > t[m] ⟶ i := m + 1
    end if
end loop ;
isthere := i ∈ 1..n
    and then (x = t[i])
```

```
variable i, j, m : INTEGER ;
    found : BOOLEAN ;

from
    i := 1 ; j := n ; found := false ;
until
    i = j or found
loop
    m := ⌊(i + j) / 2⌋ ;
    if
        x < t[m] ⟶ j := m - 1 []
        x = t[i] ⟶ found := true []
        x > t[m] ⟶ i := m + 1
    end if
end loop ;
isthere := found
```

```
variable i, j, m : INTEGER ;

from
    i := 0 ; j := n+1
until
    i = j
loop
    m := ⌊(i + j) / 2⌋ ;
    if
        x ≤ t[m] ⟶ j := m []
        x > t[m] ⟶ i := m + 1
    end if
end loop ;
isthere := i ∈ 1..n
    and then (x = t[i])
```

The following class of algorithms will thus be a correct implementation of *Searchindex:*

```
variable m: 1..n
from i := 0; j := n
keep
    p(i) and q(j) and 0 ≤ i ≤ j ≤ n
decrease
    j - i
until
    j = i
loop
    m := "some value in 1..n such that i < m ≤ j";
    if
        t[m] ≤ x  ⟶  i := m []
        t[m] > x  ⟶  j := m - 1
    end if
end loop
```

An important detail should be noted here: we have specified that m should be such that $i < m \leq j$, not only $i \leq m \leq j$ as was previously suggested. This is necessary to ensure termination: if both actions $i := m$ and $j := m - 1$ are to decrease the variant $j - i$, then m should be lesser than or equal to j and strictly greater than i.

Note that the requirement $i < m \leq j$ entails that m indeed belongs to the interval $1..n$, since $0 \leq i \leq j \leq n$ is invariant.

Any policy for choosing m which meets this requirement is acceptable. Two simple ones are choosing $m = i + 1$ and $m = j$ respectively; both lead to variants of the above sequential search algorithm. A more balanced choice is to take m as the average of i and j; to obtain a correct program, we should take not $\left\lfloor \frac{i+j}{2} \right\rfloor$ but $\left\lfloor \frac{i+j}{2} \right\rfloor + 1$ so that it is guaranteed that $m > i$ (see exercise 3.2 as to what modifications will allow choosing $m = \left\lfloor \frac{i+j}{2} \right\rfloor$).

We thus obtain the following program for *Searchindex*, a version of the "binary search" algorithm:


```
variable m: 1..n;

from
    i := 0; j := n
keep p(i)
    and q(j) and 0 ≤ i ≤ j ≤ n
decrease
    j - i
until
    j = i
loop
    m := ⌊(i + j)/2⌋ + 1;

    if
        t[m] ≤ x -> i := m []

        t[m] > x -> j := m - 1
    end if
end loop
```

The advantage of this algorithm over sequential search, from the point of view of efficiency, comes from the fact that the searching interval is approximately divided by two, rather than reduced by one, at each pass through the loop; thus, the number of iterations will be bounded by $\log_2 n$ rather than n.

To verify that the loop is executed at most $\lfloor \log_2 n \rfloor$ times, we prove that $\lfloor \log_2 (j - i) \rfloor$ is a variant. For any real numbers a and b, $\lfloor \log_2(a) \rfloor < \lfloor \log_2(b) \rfloor$ if $a \leq \frac{b}{2}$. Here, $j - i$ is indeed at least divided by 2 in both possible cases in the loop, since whenever $i < j$ (i and j being integers):

$$j - \left(\left\lfloor \frac{i+j}{2} \right\rfloor + 1 \right) \leq \frac{j-i}{2}$$

$$i - \left\lfloor \frac{i+j}{2} \right\rfloor \leq \frac{j-i}{2}$$

(This is trivially seen by looking separately at the cases $i + j$ odd and $i + j$ even).

Some remarks

From the examples seen so far of assertion-guided program construction, the following points are worth pondering.

- Several algorithms, quite different in their actual working, some sequential, some binary, were derived in the same framework. Actually, it is only at the last step (choosing how to "*bring i and j closer*") that different design choices lead to different computing methods.
- The heuristics used, "uncoupling", is very general and quite independent from the particular problem of table searching, as will be seen below when we apply it to a completely different problem, array partitioning.

• We have built all versions in such a way that we can be convinced they are correct, and know exactly why they are. It may be noted that binary search, although quite simple in its principle, is not so easy to write down correctly; this has apparently been known for a long time, since Knuth, in his well-known treatise in algorithmics [Knuth 73], felt it necessary to write: "Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried". We now understand that the reason for this is all the delicate ponts in the analysis above, in particular the care which must be exercised when writing $<$ or $\leq$, when ensuring that $j - i$ actually decreases every time through the loop, when assigning an interval (0..n or 1..n) to each variable. Typical programming errors, for this problem, such as writing a loop which will yield an incorrect result when x does not lie between the minimum and maximum values of t, or will sometimes try to access an undefined value such as $t[0]$ or $t[n+1]$, or will not always terminate, may be traced to such oversights in the analysis. It is instructive in this respect to look at the programs of figure 3.7 (did you comply with our request at the beginning of this section and try honestly to come up with your own version?).

3.10. - AN EXAMPLE: ARRAY PARTITIONING

We will end this discussion of assertion-guided program construction with a slightly more difficult example, corresponding to a classical algorithm. We will show how the method can be applied to the derivation of several variants of the same algorithm; the first variant is fairly simple but not very efficient; we will then improve the efficiency in two successive refinements, while relying on the assertion-guided approach to check at every step that correctness is preserved.

3.10.1. Specification and usage

The problem we study is array partitioning; it arises in connection with sorting and so-called "order statistics". Let x be an element of a certain type and t an array of the same type, with size n. Every element of the array has a **key**. For simplicity, we shall abbreviate the key of element i, normally written *key (t[i])*, as $key_i(t)$ or just key_i if there is no ambiguity as to what array is meant.

Keys are ordered, that is to say, we can compare keys of elements using an order relation written $\leq$ (less than or equal to; "greater than or equal to" is $\geq$ and "less than" is $<$). We use the abbreviation

$$key_{a..b}(t) \leq key_{c..d}(t)$$

to mean "all keys of elements in t *[a..b]* are smaller than or equal to all keys of elements in t *[c..d]*" (true if either interval is empty). Similarly, $key_{a..b}(t) \leq k$ means "all keys of elements in t *[a..b]* are less than or equal to k" (true if $a..b$ is empty). Again, we omit (t) if the context makes it clear what array is meant.

The partitioning problem is to split the array in two parts, where the elements in the first part have smaller or equal keys than the elements in the second part. The final state is described by figure 3.8.

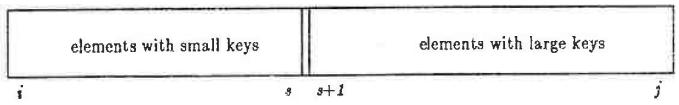


Figure 3.8: Final state of Partition

To reach this desired state, the partitioning algorithm will move around some elements and find an index s such that, in the end, $key_{i..s}(t) \leq key_{s+1..j}(t)$. The value of s will be computed by the program as it reorganizes the array, since there is no way to know it in advance.

As it turns out, what will be required in most applications will be to partition some subarray of t, not necessarily the whole of it, so that we will have a specification of the following form. We call the procedure *Partition1*, reserving the name *Partition* for a slightly modified form, to be introduced soon.

procedure *Partition1* **specification**
in *a, b : 1..n,*
in out *t* **: array** *[1..n]* **of** *T;*
out *s : 0..n*
end procedure specification - - *Partition1*

The problem may be solved trivially, if we have an array sorting program at our disposal, by writing

Sort (t[a..b]⟷);
Searchindex (k, t[a..b], s←)

where *Searchindex* as given above is modified to search for an element with given key k.

We do not consider this solution acceptable, however, since partitioning is an inherently simpler problem than sorting. In fact, one of the best known sorting algorithms, Quicksort, has the following recursive form (for sorting array t between indices a and b):

```
procedure Quicksort (a, b, t⟷) implementation
    variable s : 1..n ;
    if
        b-a ≤ 0  ⟶  skip []
        b-a > 0  ⟶
            Partition1 (a, b, t⟷, s⟵);
            - - Do in some order:
                Quicksort (t[a..s−1]),
                Quicksort (t[s+1..b])
    end if
end procedure implementation - - Quicksort
```

Another interesting application of *Partition1* is to compute an i-th smallest element of an array without having to perform a complete sort[1]:

```
procedure Find_ith_smaller specification
    in    t : array [1..n] of T,
          i : 1..n ;
    out   x : T
          {ensure x is an element of t with i-th smallest key}
implementation
    variables t1 [1:n] : T,
              a, b, s : 1..n ;
    t1 := t ; - - Copy array parameter to local array
    from
        a := 1 ; b := n ;
    keep
        t1 is a permutation of t and 1 ≤ a ≤ b ≤ n and
        there is an element of t with i-th smallest key in t1 [a..b] and
        key_{1..a−1}(t1) ≤ key_{a..b}(t1) ≤ key_{b+1..n}(t1)
    decrease
        b-a
    until
        a = b
    loop
        Partition1 (a, b, t1⟷, s⟵) ;
            {then a ≤ s < b and key_{a..s}(t1) ≤ key_{s+1..b}(t1)}
        if
            s < i  ⟶  a := s+1 []
            s ≥ i  ⟶  b := s
        end if
    end loop ;
    x := t[a]
end procedure - - Find_ith_smaller
```

We may now give the complete specification of *Partition1*:

[1] The details of this procedure are far from trivial and the reader is invited to check its correctness. See exercise 3.15.


```
procedure Partition1 specification
        in i, j: integer ;
        in out t: array [1..n] of ELEMENT ;
                {assume   j > i ; t0 [i..j] := t [i..j]}
        out s: integer
                {ensure   t [i..j] is a permutation of t0 [i..j] and i ≤ s < j and
                          key_{i..s}(t) ≤ key.lb_{s+1..j}(t)}
end procedure specification - - Partition1
```

Note that on procedure return s should be such that $i \leq s < j$, i.e. that none of the two slices is empty. This is important in view of a requirement on *Partition1* that the careful reader will have noted when seeing the two applications mentioned (sorting and finding the i-th element): in both cases, the programs would fail to terminate if *Partition1* returned an empty slice. For procedure *Find_ith_smaller*, this is readily seen by checking the variant (b-a); for *Quicksort*, this follows from the corresponding rule for termination of recursive procedures, which will be studied in chapter 7.

In order to implement *Partition1*, what is normally done is to choose a "pivot key" in the subarray *t[i..j]* and to use it as a separator between "small" and "large" elements; in other words, *Partition1* is implemented as:

```
choose index p in i..j ;
pivot := key_p ;
Partition (i, j, pivot, t, s)
```

where the procedure *Partition*, which is really the one of interest here, is specified by:

```
procedure Partition specification
        in i, j: integer ;
           pivot : KEY ;
        in out t: array [1..n] of ELEMENT ;
                {assume   j > i and
                          pivot = key_p for some p in i..j ; t0 [i..j] := t [i..j]}
        out s: integer
                {ensure   t[i..j] is a permutation of t0 [i..j] and i ≤ s < j and
                          key_{i..s} ≤ pivot and key_{s+1..j} ≥ pivot}
end procedure specification - - Partition
```

Note that the reason for taking as *pivot* the key of some element in *t[i..j]* is to make sure that *pivot* is not outside the range of keys in *t[i..j]*, in which case there could be no solution satisfying the requirement, mentioned above as essential, that none of the two slices must be empty ($i \leq s < j$).

3.10.2. Choosing an invariant

Let us first remark that if the only changes ever performed on the array are element swaps, of the form

swap (u, v) for $i \leq u, v \leq j$

then the property that *t[i..j] is a permutation of what it was initially* will remain true throughout the partitioning process. We shall only use operations of this kind so that this part of the postcondition will hold. Thus we won't consider it any more in the sequel.

A loop will clearly be needed for *Partition*. If we look at the rest of the postcondition, we notice another example of **coupling**: the hard part comes from the fact that s appears in both operands of the and. Why not try uncoupling again?

The uncoupling heuristics prompts us to associate two variables u and v to the **out** parameter s, and to include the following assertion in the loop invariant:

$key_{i..u-1} \leq pivot$ **and** $key_{v+1..j} \geq pivot$

In this formulation, we have chosen the bounds $u-1$ and $v+1$ rather than u and v to make initialization easy; indeed, the above assertion is trivially satisfied after the following initialization:

$u := i$; $v := j$

This invariant will coincide with the postcondition of the procedure if

$v\text{-}u = -1$ **and** $i < u$ **and** $v < j$

and we take as value of s, to be returned by the procedure, the final value of v. It thus seems fit to add the following constraints to the invariant:

$v\text{-}u \geq -1$ **and** $i \leq u$ **and** $v \leq j$

The reason we had to choose $\leq$ for the last two inequalities here, rather than $<$ which would be more in line with the postcondition of the procedure, is that that we want the invariant to be satisfied after the initialization, when $u = i$ and $v = j$. For the procedure body to be correct with respect to the specification, however, we shall have to make sure that s receives a final value strictly in the interval $i..j\text{-}1$ (whereas if the final value of v is assigned to s, the above constraints only guarantee that it belongs to $i\text{-}1..j$).

So we embark on the construction of the loop with the following tentative invariant, later referred to as *INV*:

(*INV*) $v\text{-}u \geq -1$ **and** $i \leq u$ **and** $v \leq j$ **and**

$key_{i..u-1} \leq pivot$ **and** $key_{v+1..j} \geq pivot$

Since the invariant includes $v\text{-}u \geq -1$ and the exit condition will be $v\text{-}u = -1$, it is natural to use $v\text{-}u+1$ as variant. Note that this implies that the two cursors u and v will "cross over" just before loop termination. The loop will have the form:

from
  $u := i$; $v := j$
keep
  $v\text{-}u \geq -1$ **and** $i \leq u$ **and** $v \leq j$ **and**
  $key_{i..u-1} \leq pivot$ **and** $key_{v+1..j} \geq pivot$
decrease
  $v\text{-}u+1$
until
  $v\text{-}u = -1$
loop
  BURN_CANDLE
end loop ;

The loop body *BURN_CANDLE* should be so designed as to maintain the invariant (if the exit condition is not satisfied) and to decrease the variant, keeping it non-negative. In other words, using a snapshot d_0 to record the value of the variant before an execution of *BURN_CANDLE*, *BURN_CANDLE* should satisfy the following precondition-postcondition specification:

{**given** *INV* **and** $v\text{-}u+1 > 0$}
BURN_CANDLE
{**then** *INV* **and** $v\text{-}u+1 \geq 0$}

If we can invent a statement *BURN_CANDLE* which satisfies this specification, then the above loop will result in a state in which the invariant *INV* and the exit condition $v\text{-}u = -1$ both hold. Their

conjunction yields the following condition:

$u = v+1$ **and** $i-1 \leq v$ **and** $v \leq j$ **and**

$key_{i..u-1} \leq pivot$ **and** $key_{v+1..j} \geq pivot$

If we take $s = v$, this almost yields the postcondition of the *Partition* procedure, with one small restriction, already mentioned: the actual postcondition requires $i \leq s < j$ (because no slice should be empty), whereas here we only have $i-1 \leq v \leq j$. Given the second line of the above assertion, $v = j$ may only occur if all the elements of $t[i..j]$ have a key lesser than or equal to *pivot*. In this case, we may safely take $v-1$ as the value assigned to s. Similarly, if $v = i-1$, i.e. $u = i$, then the pivot is a minimum key and we may assign to s the value of i. Thus if the above loop is followed by

$s := max\ (min\ (v, j-1), i)$

then we shall have obtained a correct implementation of *Partition* provided *BURN_CANDLE* satisfies its specification. So we now turn to the refinement of *BURN_CANDLE*.

3.10.3. The loop body

The name which we have chosen for the loop body is suggestive of the method used for *Partition*: "burn the candle at both ends". In other words, *BURN_CANDLE* will contain two internal loops, say *Lu* and *Lv*, one which will increment u, the other decrementing v:

(*Lu*) **until** ... **loop** $u := u+1$ **end loop**

(*Lv*) **until** ... **loop** $v := v-1$ **end loop**

We must now see if more statements are needed in *BURN_CANDLE* and replace the dots with actual conditions. It is easy to find exit conditions which are such that *INV* will be invariant for both internal loops: just take conditions which will stop execution of either loop whenever the validity of *INV* would be endangered. This yields:

(*Lu*) **until** $u = v+1$ **or else** $key_u > pivot$ **loop** $u := u+1$ **end loop**

(*Lv*) **until** $v = u-1$ **or else** $key_v < pivot$ **loop** $v := v-1$ **end loop**

The **or else** connectives are necessary because of the conditions on u and v: in the last execution of the outermost loop, $v+1$ or $u-1$ might run out of the interval $i..j$.

The above two loops maintain *INV* but are not suitable, if taken alone, as code for *BURN_CANDLE* because they may fail to decrease the stated variant: if the exit condition $v\text{-}u = -1$ is not met but $key_u > pivot$ and $key_v < pivot$, then both internal loops will be equivalent to null statements, which in plain English means that nothing will happen. And rightly so: it would be an error to either increase u or decrease v in this case. *BURN_CANDLE* has found an inversion, and the thing to do is to remove it. The following statement should thus be added after the two internal loops:

if
  $v\text{-}u \neq -1 \longrightarrow$
    {**then** $v\text{-}u \geq 0$ **and** $key_u > pivot$ **and** $key_v < pivot$}
    swap (u, v) □
  $v\text{-}u = -1 \longrightarrow$ **skip**
end if

Apparently, this statement still does not ensure that the invariant decreases, since nothing happens to either u or v when the swap is executed. One possible solution is to add the statements

$u := u+1$; $v := v-1$

after *swap (u, v)* in the first branch of the **if** statement. Such an addition is permitted because it will not invalidate *INV*; in particular, there is no danger that u and v might cross over "too far", i.e. that $v\text{-}u+1$

might become negative. To see this, note that if the following conditions are met

$v\text{-}u \geq 0$ **and** $key_u > pivot$ **and** $key_v < pivot$

then u and v cannot be equal (the key corresponding to their common value would then be both lesser and greater than *pivot*), so that in fact $v\text{-}u > 0$ in this case, implying

$(v\text{-}1) - (u+1) \geq -1$

So we may insert statements for incrementing u and decrementing v into the first branch of the **if** statement, after the swap. We choose not to do it, however, in order not to complicate the program; it is nicer to keep the incrementing of u in just one place, the body of the *Lu* loop, and similarly the decrementing of v in just the body of the *Lv* loop. We thus tentatively refine *BURN_CANDLE* as:

{**given** *INV* **and** $v\text{-}u+1 > 0$}

until $u = v+1$ **or else** $key_u > pivot$ **loop** $u := u+1$ **end loop** ;

until $v = u\text{-}1$ **or else** $key_v < pivot$ **loop** $v := v\text{-}1$ **end loop** ;

if

$v\text{-}u \neq -1 \longrightarrow$

{**then** $v\text{-}u > 0$ **and** $key_u > pivot$ **and** $key_v < pivot$}

swap (u, v) []

$v\text{-}u = -1 \longrightarrow$ **skip**

end if

{**then** *INV* **and** $v\text{-}u+1 \geq 0$}

BURN_CANDLE as it stands now, with just the swap in the first branch of the **if** statement (and no change to either u or v), clearly preserves the invariant *INV*. The reason it is also correct as a loop body, with respect to termination of the loop, is, informally, that when the swap is executed the external loop is not complete ($v\text{-}u$ is not equal to -1 yet); so *BURN_CANDLE* will be executed one more time at least; next time it is executed, the exit test for the first internal loop, *Lu*, namely

$u = v+1$ **or else** $key_u > pivot$

must evaluate to **false** since $u - v > -1$ and an element of key less than *pivot* has been placed in position u, so that the body of *Lu* will be executed at least once (and the body of *Lv*, too, unless u crosses v at the end of *Lu*); thus the process won't stop. Formally, to prove that *BURN_CANDLE* terminates, we note that its variant is not the one proposed initially, namely $v\text{-}u+1$, but

$(v\text{-}u+1) + nbinv$

where *nbinv* is the number of inversions in $t[i..j]$; an inversion is a pair of indexes a, b such that

$i \leq a < b \leq j$ **and** $key_a > key_b$

Every execution of *BURN_CANDLE* decreases this variant, since whenever the exit condition is not satisfied and both internal loops result in a null statement, *BURN_CANDLE* executes the statement *swap (u, v)*, which removes an inversion and thus decreases *nbinv*.

We have thus obtained now a correct version of *Partition*.

procedure *Partition* **specification** - - *Version 1*

in *i, j:* **integer** ;
pivot : KEY ;

in out *t: array [1..n]* **of** *ELEMENT* ;
{**assume** $j > i$ **and** $pivot = key_p$ *for some* p *in* $i..j$;
$t_0[i..j] := t[i..j]$}

out *s:* **integer**
{**ensure** *t[i..j] is a permutation of* $t_0[i..j]$ **and**
$i \leq s < j$ **and**
$key_{i..s} \leq pivot$ **and** $key_{s+1..j} \geq pivot$}

variables *u, v:* **integer** ;

implementation

from
$u := i$; $v := j$
keep
$v\text{-}u \geq -1$ **and** $i \leq u$ **and** $v \leq j$ **and**
$key_{i..u-1} \leq pivot$ **and**
$key_{v+1..j} \geq pivot$
decrease
$v\text{-}u+1$
until
$v\text{-}u = -1$
loop - - *BURN_CANDLE*

until $u = v+1$ **or else** $key_u > pivot$ **loop** $u := u+1$ **end loop** ;

until $v = u\text{-}1$ **or else** $key_v < pivot$ **loop** $v := v\text{-}1$ **end loop** ;

if

$v\text{-}u \neq -1 \longrightarrow$

{**then** $v\text{-}u > 0$ **and** $key_u > pivot$ **and** $key_v < pivot$}

swap (u, v) []

$v\text{-}u = -1 \longrightarrow$ **skip**

end if

end loop ;

$s := max\ (min\ (v, j\text{-}1), i)$;

end procedure - - *Partition, Version 1*

3.10.4. Improvements needed

The above procedure body has been obtained naturally by working from the assertions; we might just be satisfied with it and, in many cases, the best thing to do is to stop here and code the algorithm as we have it now.

If, however, we are interested in getting the most efficient (but still correct!) program, it is easy to see that the above version is not optimal.

A minor remark is that, given the precondition of the procedure, the exit condition of the outermost loop cannot be **true** the first time it is tested, so that this first test is redundant. In other words, this outermost loop is the equivalent of a Pascal **repeat...until** loop (the various kinds of loops will be studied in detail in the next Chapter), where the exit test is performed after the execution of the loop body rather than before. Following a C-like convention, we shall distinguish such a loop from the standard one by just exchanging two clauses: we write

...**loop**...**until**...

instead of

...**until**...**loop**...

the other clauses (**from, keep, decrease**) being unchanged.[2] Thus we may write the body of *Partition* as a **loop**...**until**... loop. This is a minor point, however. Two more serious complaints may be voiced against the previous version:

- The test $v\text{-}u = -1$, which is the exit condition of the outer loop, is also performed in each of the internal loops. Except possibly in the last iteration, the value of this condition in the internal loops will always **false**, so that the internal tests are wasteful. This is worrying because these internal tests will, of course, be executed much more often than the outer test. If key comparisons are not particularly expensive, the overhead due to the almost redundant internal tests may be estimated to be around 10-20%.

- The test in the **if** statement is again the same as the external loop exit condition. In fact, this statement looks a little like a **goto** (out of the loop) in disguise.

Can we do anything about these problems?

3.10.5. Improving the internal loop exit conditions

Let us first concentrate on the first deficiency, which is the most serious. It would be nice if we could just remove the tests on $v\text{-}u$ from both internal loops. But would they terminate then? The answer with *BURN_CANDLE* as it stands now is probably no: if we rewrite *Lu*, for example, so that it reads

until $key_u > pivot$ **loop** $u := u+1$ **end loop**

then this loop will only terminate if there is an element at position u or to its right with key greater than *pivot*. To ensure this, we must choose the pivot in a special way; but we may only do so under the assumption that not all elements of the subarray have equal keys. Although such an assumption will be required for the next improvement of the procedure's efficiency, we prefer to avoid the need for it here.

What is much easier to ensure, however, is that there is an element at position u or to its right with key greater than **or equal to** *pivot* (and, similarly, an element at position v or to its left with key lesser than or equal to *pivot*). Thus it seems interesting to see if we can remove the $v\text{-}u$ tests from a slightly different version of *BURN_CANDLE*, where the tests on keys will stop the internal loops on equality as well. Here is this new version:

[2] Note that in general the proof rules associated with a true **loop**...**until**... loop are slightly different from those associated with the loops which we write **until**...**loop**... (equivalent to Pascal **while** loops). In particular, they require that the invariant be true after each execution of the loop body, but not necessarily upon entry. The ...**loop**...**until**... loops which we consider here, however, are just plain **while**-like loops with the extra property that the exit condition is always **false** right after the initialization.

until $u = v+1$ **or else** $key_u \geq pivot$ **loop** $u := u+1$ **end loop** ;

until $v = u\text{-}1$ **or else** $key_v \leq pivot$ **loop** $v := v\text{-}1$ **end loop** ;

if

$v\text{-}u \neq -1 \longrightarrow swap\ (u, v);\ u := u+1\ ;\ v := max\ (v\text{-}1,\ u\text{-}1)$ []

$v\text{-}u = -1 \longrightarrow$ **skip**

end if

This version is still correct as loop body, since it retains both the invariant *INV* and the variant $v\text{-}u+1$; for the latter, however, we had to make the decrease of v in the first branch of the **if** statement conditional upon the fact that the two cursors do not cross over too far; the argument used previously to show that this could not happen does not hold any more with non-strict inequalities on keys ($key_u \leq pivot$ **and** $key_u \geq pivot$ is not a contradiction). Alternatively, we could have let the cursors cross over "one too far", i.e used $v\text{-}u+2$ as variant, $v\text{-}u \geq -2$ as first clause of the invariant and and $v\text{-}u \leq -1$ as loop exit condition.

The main difference between the above version and the previous one is that the new one may perform a few more swaps; these will only occur for elements whose key is equal to *pivot*, so that the loss may be assumed not to be significant in practice (unless there is an significant amount of equal keys in the array).

This new version has a very interesting property in that it keeps invariant the following assertion:

there is at least one index a in $u..v+1 \cap i..j$ *such that* $key_a \geq pivot$ **and**

there is at least one index b in $u\text{-}1..v \cap i..j$ *such that* $key_b \leq pivot$

Note that this assertion is initially satisfied since the pivot is chosen as the key of an element in $i..j$. It is easy to see that the new version of *BURN_CANDLE* keeps it invariant.

The reason this property is interesting is that it makes the first part of the exit conditions for *Lu* and *Lv* unnecessary, since it implies that both loops will encounter an element satisfying the other part of their exit conditions ($key_u \geq pivot$ and $key_v \leq pivot$ respectively) before they get a chance to make $v\text{-}u+1$ negative. We may thus dispense with the extra tests and rewrite *BURN_CANDLE* as just:

until $key_u \geq pivot$ **loop** $u := u+1$ **end loop** ;

until $key_v \leq pivot$ **loop** $v := v\text{-}1$ **end loop** ;

if

$v\text{-}u \neq -1 \longrightarrow swap\ (u, v);\ u := u+1\ ;\ v := max\ (v\text{-}1,\ u\text{-}1)$

$v\text{-}u = -1 \longrightarrow$ **skip**

end if

We thus get our second version of *Partition*. It is interesting to note that in this version both u and v will lie in the interval $i..j$ upon loop exit, so that the value assigned to s by the procedure may be just v, whereas the first version required a final correction *max (min (v, j-1), i)*. This property is readily derived from the new invariant property emphasized above; it is equivalent to the fact that, in this version, the bodies of both *Lu* and *Lv* will be executed at least once.

```
procedure Partition specification      -- Version 2
        in i, j: integer ;
           pivot : KEY ;
        in out t: array [1..n] of ELEMENT ;
                {assume  j > i and
                         pivot = key_p for some p in i..j ;
                         t0 [i..j] := t [i..j]}
        out s: integer
                {ensure  t[i..j] is a permutation of t0 [i..j] and
                         i ≤ s < j and
                         key_i..s ≤ pivot and
                         key_s+1..j ≥ pivot
    variables u, v: integer ;
implementation
    from
        u := i ; v := j
    keep
        v-u ≥ -1 and i ≤ u and v ≤ j and
        key_i..u-1 ≤ pivot and
        key_v+1..j ≥ pivot and
        there is at least one index a in u..v+1 ∩ i..j such that key_a ≥ pivot and
        there is at least one index b in u-1..v ∩ i..j such that key_b ≤ pivot
    decrease
        v-u+1
    loop -- BURN_CANDLE
        until key_u ≥ pivot loop u := u+1 end loop ;
        until key_v ≤ pivot loop v := max (v-1, u-1) end loop ;
        if
            v-u ≠ -1  ⟶  swap (u, v); u := u+1 ; v := v-1 []
            v-u = -1  ⟶  skip
        end if
    until
        v-u = -1
    end loop ;
    s := v ;
end procedure -- Partition, Version 2
```

3.10.6. Getting rid of the internal if statement

Let us now deal with the problem of the internal **if** statement.

What we have here is a loop belonging to the category of so-called "$n + \frac{1}{2}$" loops, meaning that the last iteration is not complete (we don't perform the swap last time through the loop). Some languages offer an "exit" construct which makes it possible to break a normal loop structure somewhere in the middle; for example, in Ada, we would write:


```
loop
    ...Lu'...
    ...Lv'...
    when v-u = -1 exit ;
    swap (u, v); u := u+1; v := max (v-1, u-1) ;
end loop
```

The problem of "$n + \frac{1}{2}$" loops will be studied in the next chapter. Here, if we are concerned about removing the conditional statement local to the loop body, there is a way to do it, but it will turn out that we need an extra hypothesis on the array and the pivot.

First, we would like to get rid again of the operations on u and v in the body of the conditional statement. The reason we reintroduced them in the last version was our concern about termination: the loops on u and v, with their exit conditions rewritten with non-strict inequalities (i.e. $key_u \geq pivot$ and $key_u \leq pivot$ respectively) would be equivalent to null statements after two elements whose key is equal to *pivot* had been swapped.

It is enough, however, that **one** of these loops should be non-null. So far, we have been very careful to keep the treatment of u and v completely symmetric; the only place where we had to breach this principle was in the last version, when we introduced a *max* in the expression assigned to v in order to avoid $v\text{-}u+1$ becoming negative. Let us see what happens if we introduce some more dissymmetry by writing *BURN_CANDLE* as:

```
until key_u ≥ pivot loop u := u+1 end loop ;
until key_v < pivot loop v := v-1 end loop ;
if
    v-u ≠ -1  ⟶  swap (u, v) []
    v-u = -1  ⟶  skip
end if
```

Note that the use of *max (v-1, u-1)* is no longer necessary when assigning *v-1* to v in the second internal loop ($key_u \geq pivot$ precludes $key_u < pivot$).

The only potential problem is termination of the loop on v. It is easy to see, for example, that this loop may not terminate (and will try to access elements of the array outside the interval *i..j*) in the case where all keys in the subarray are equal to *pivot*. This may **not** occur, however, if we add the following assumption:

there is at least one index q in i..j such that $key_q < pivot$

If this assumption is satisfied when *Partition* is called, then the following property will remain invariant throughout:

there is at least one index a in u..v+1 ∩ i..j such that $key_a \geq pivot$ **and**
there is at least one index b in u-1..v ∩ i..j such that $key_b < pivot$

so that the new version of *BURN_CANDLE* will terminate. Note that the variant, as in section 2, involves the number of inversions; here we have to use a slightly different definition: an inversion is a pair of indexes a, b such that

$$i \leq a < b \leq j \text{ and } key_a \geq pivot \text{ and } key_b < pivot$$

How practical is the assumption that *there is at least one index q in i..j such that* $key_q < pivot$? To be able to use it, we must rely on the hypothesis that, whatever method is used to find the pivot, it does not return an element with maximum key; if, when trying to find such a pivot, it discovers that none

exists, i.e. that all elements have equal keys, then the new version of *Partition* should not be called at all (which is not a problem since, in this case, any value *s* in the interval *i..j-1* may be returned to *Quicksort* or *Find_ith_smaller*). So, in practice, the extra assumption on the pivot means placing a small overhead on the part of the procedure that we have called *Partition1*, which is responsible for finding a pivot. This overhead will usually be acceptable **unless** there is a significant number of elements with equal keys in the array, in which case it is better to forget about it and be content with the last version we have obtained for *Partition*. It should also be noted that this new requirement on the pivot slightly complicates the mathematical analysis of the algorithm in the average case.

In the sequel, we assume that the assumption on the pivot not being an element with maximum key may be made and that *BURN_CANDLE* has consequently been rewritten as above. To avoid any confusion, we call *Lu'* and *Lv'* the new versions of the internal loops.

An execution of *Partition* will consist of a certain number of executions of its loop body, that is:

Lu'; Lv'; **if** *v-u* ≠ *-1* ⟶ *swap (u, v)* [] *v-u = -1* ⟶ **skip end if** ;
Lu'; Lv'; **if** *v-u* ≠ *-1* ⟶ *swap (u, v)* [] *v-u = -1* ⟶ **skip end if** ;
.....
Lu'; Lv'; **if** *v-u* ≠ *-1* ⟶ *swap (u, v)* [] *v-u = -1* ⟶ **skip end if** ;

Note, however, that the last **if** statement is a null one (**skip**) because, on the last iteration, *v-u* has value *-1* (this is the loop exit condition). So we can remove it.

Now let us see if we could **add** an instance of the conditional statement at the **top** of this execution sequence. Since the condition *v-u* ≠ *-1* is initially satisfied, such an addition implies is that an extra *swap (i, j)* will be performed initially. If not immediately useful, such a swap is certainly harmless. So we may insert an extra conditional swap at the top of the execution sequence; if we also remove the unnecessary one at the bottom, the execution sequence becomes:

if *v-u* ≠ *-1* ⟶ *swap (u, v)* [] *v-u = -1* ⟶ **skip end if** ; *Lu'* ; *Lv'* ;
if *v-u* ≠ *-1* ⟶ *swap (u, v)* [] *v-u = -1* ⟶ **skip end if** ; *Lu'* ; *Lv'* ;
.....
if *v-u* ≠ *-1* ⟶ *swap (u, v)* [] *v-u = -1* ⟶ **skip end if** ; *Lu'* ; *Lv'*

The point of this seemingly strange game is that the new execution sequence corresponds to another loop, which may be written:

from
    u := *i* ; *v* := *j*
loop
    if *v-u* ≠ *-1* ⟶ *swap (u, v)* [] *v-u = -1* ⟶ **skip end if** ;
    Lu' ; *Lv'*
until
    v-u = -1
end loop ;

But now we notice that the body of this new loop begins with a conditional statement whose conditional expression is the negation of the loop exit condition, which we know is not satisfied on loop entry; the test in the **if** statement at the head of the loop is thus redundant. So *swap (u, v)* may be executed unconditionally, and the loop body becomes simply

swap (u, v); Lu' ; Lv'

We thus get a final version of our *Partition* procedure; please note the new clause in the precondition, which restricts the applicability of this version. Also note that, as with the previous version, the invariant guarantees that *v* belongs to the interval *i..j* on loop exit, so that no correction is necessary in the assignment to *s*.

procedure *Partition* **specification** - - *Version 3*
    in *i, j*: **integer** ;
        pivot : KEY ;
    in out *t*: *array [1..n]* **of** *ELEMENT* ;
        {**assume** *j* > *i* **and**
            pivot = key_p *for some p in i..j* **and**
            pivot > key_q *for some q in i..j* ;
            t_0 *[i..j]* := *t [i..j]*}
    out *s*: **integer**
        {**ensure** *t[i..j] is a permutation of* t_0 *[i..j]* **and**
            i ≤ *s* < *j* **and**
            $key_{i..s}$ ≤ *pivot* **and**
            $key_{s+1..j}$ ≥ *pivot*
    variables *u, v*: **integer** ;
implementation
    from
        u := *i* ; *v* := *j*
    keep
        v-u ≥ *-1* **and** *i* ≤ *u* **and** *v* ≤ *j* **and**
        $key_{i..v-1}$ ≤ *pivot* **and** $key_{v+1..j}$ ≥ *pivot* **and**
        there is at least one index a in u..v+1 ∩ *i..j such that* key_a ≥ *pivot* **and**
        there is at least one index b in u-1..v ∩ *i..j such that* key_b < *pivot*
    decrease
        v-u+1
    loop - - *BURN_CANDLE*
        swap (u, v) ;
        until key_u ≥ *pivot* **loop** *u* := *u+1* **end loop** ;
        until key_v < *pivot* **loop** *v* := *v-1* **end loop** ;
    until
        v-u = -1
    end loop ;
    s := *v*
end procedure - - *Partition, Version 3*

MODULARITY

(Chapter 6)

Bertrand Meyer

Pre - draft

This is a first draft of chapter 6 of a book in preparation. The working title of the book is **Applied Programming Methodology**.

The book follows the spirit of **Méthodes de Programmation**, which I co-authored with Claude Baudoin (from Schlumberger); this text was published in 1978 by Eyrolles in Paris. The present work is not, however, a translation of the former one; shortly after publication of the French book, we did consider translating it into English, but for various reasons this project was delayed and it soon became clear that an entirely new design was needed. Claude did not wish to participate in such an endeavor; what follows is thus my sole responsibility.

The projected audience of the book includes practitioners (engineers, programmers, etc.) who are looking for a readable survey on modern programming concepts, as well as students, for whom it is intended as a textbook to be used in connection with courses on programming methodology, programming languages, programming techniques or software reliability.

The book uses several programming languages as a means to exemplify the programming concepts discussed and to deepen their analysis. The languages studied include Fortran, Pascal, Simula 67, Ada, Modula, Lisp and, to a lesser extent, PI/I, Cobol, Algol W, Smalltalk and APL.

The tentative plan of the book is as follows.

Chapter 1: The challenge of software engineering
A short introduction recalling the basic problems of software engineering, summarizing the current state of the art, and describing the "two schools" of software engineering.

Chapter 2: The structure and role of programming languages
A description of the structure of programming languages, introducing the basic issues in language design and discussing the role of languages in programming.

Chapter 3: Control structures: Fundamentals
An introduction to the basic control structures of sequential programming, using from the outset a systematic, semi-formal approach. Includes a discussion of specification-directed program construction.

Chapter 4: Control structures: Techniques
All elaboration on the concepts introduced in the previous chapter: variants of the basic patterns; control stuctures as implemented in various languages; technical problems associated with procedures.

Chapter 5: Data structures and their description
An introduction to the practical use of abstract data structure descriptions. Emphasizes hierarchical definition of types and reuse of previously written descriptions (through mechanisms of enrichment and restriction derived from those of Simula, Z and Clear). Offers three levels for the description of data structures: implicit (i.e. by one or more abstract data types), constructive, physical.

Chapter 6: Modularity (this chapter)
A discussion of some of the main requirements for modular programming and of existing techniques.

Chapter 7: Recursion and Functional Programming
An introduction to the "other culture" of programming, with hints for the practitioner as to how to use its concepts.

Inclusion of the next two chapters is still a matter of discussion.

Chapter 8: Some fundamental data structures
A systematic presentation of some of the most useful data structures, from specification to implementation, the latter including coding examples in various programming languages.

Chapter 9: Some fundamental algorithms
A systematic presentation of some important algorithms, chosen both for their methodological interest, elegance and practical usefulness.

NOTE ON CHAPTER 6

A first version of this chapter was prepared as the text of an invited presentation at the September, 1982 Conference of the Association of Simula Users. Adaptation and generalization of this paper to the framework of the book have not been completed. Besides filling some of the "blank" sections, the next version will contain the following corrections:

- better balance between Simula and other languages, in particular Ada and Smalltalk;
- better distinction between **type** and **object**, on the one hand, and between **algorithm** and **process**, on the other hand (the wording of the present version does not distinguish clearly enough between general categories and their instances);
- Conformance to the programming notation introduced in chapter 3;
- conformance to the conventions of the rest of the book (e.g. few references in text, bibliographic notes at end of chapter, format of references, etc.)

Note (January 1985): This text was essentially written in 1982-1983 and will be extensively reworked.

6.1. - INTRODUCTION: A MULTI-FACET DEFINITION

Among the buzzwords in software engineering, "modularity" is one of the most misused; as a catch-phrase, it is probably as popular as "structured" or "reliable". The proponents of any specification, design or programming method will always claim that it leads to "truly modular" systems; seldom, however, is there any convincing evidence supporting such a claim. As a matter of fact, there does not exist a simple, widely accepted definition of what the concept of modularity means with respect to programs. On the other hand, anyone with some programming experience will have an intuitive feeling for what it means for a program to be modular, and agree that this is an important quality which programs should indeed possess.

The purpose of this chapter is to try to elucidate what modularity means as far as programming is concerned, why it is so important a property, and what can be done to promote it when designing and building programs. This last point will take us into the study of several fruitful concepts.

Our first task will be to give a sufficiently precise definition of what modularity is. The general idea, of course, is that a program is modular if it is made out of a number of elements, or modules, in such a way that each of them enjoys some **internal homogeneity** and **conceptual autonomy**, and the set of their interrelationships is **structurally coherent**. This problem is not unlike those which are described (if not solved) by general systems theory; some of the vocabulary which will be used reflects this similarity.

The above characterization is, however, much too vague to be considered technically satisfactory; it does not offer much help in determining whether or not a given program is modular, or, even more fundamentally, whether or not a programming (or program design) method or language is an aid in the construction of modular programs.

As it turns out, one of the reasons that there is not a single universally accepted definition of modularity is that different viewpoints will lead to different, equally justifiable definitions. It is not so much that modularity means many things to many people, but rather that there are several inherently different facets to it. In fact, we will not concentrate on a single definition; we will instead define modularity through a set of ten requirements: five **criteria** and five **principles**. The "principles" are as important in practice as the criteria; the distinction stems from the fact that the latter may be logically deduced from the former. On the other hand, even though the five criteria are not totally mutually independent, no one follows directly from the others. In our view, no understanding of modularity will be complete if it does not satisfy these ten requirements.

After discussing the criteria and principles, we will introduce five **keywords**, denoting technical concepts which help in the practical realization of modularity. We will then study how these concepts are compatible with traditional approaches to modular programming, based on the subprogram concept; other techniques will be discussed, such as the Jackson and Warnier design methods, the coroutine concept and the use of abstract data types.

Several examples will be given of applications of these concepts, one of them describing an important practical class of problems, the design of interactive menu-driven programs. We will conclude by studying the relational structure of systems seen as networks of modules.

Although much of the discussion concentrates on what "modular" means for programs and how modular programs may be achieved, it generalizes for the most part to modularity as applied to earlier stages of the software life-cycle, so that one can also use part of it when studying modular **specifications** and modular **design documents**.

6.2. FIVE CRITERIA

Our five criteria are properties which a method or language for programming or program design should possess if it is to be considered an aid in the construction of modular programs. They are termed:

- modular decomposability;
- modular composability;
- modular understandability;
- modular continuity;
- modular protection.

6.2.1. - Modular decomposability

The first criterion, decomposability, has to do with the construction of programs. A method or language may be said to be modular with respect to this criterion if it helps in the decomposition of a problem into several subproblems, whose solution may then be pursued separately. From the point of view of systems theory, the problem is to help decompose systems into subsystems (Fig. 6.1).

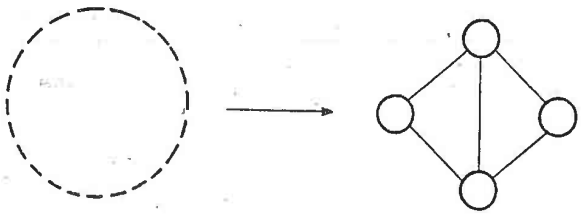


Figure 6.1 - Decomposability

In the general case, the decomposition process will be repetitive: each of the modules will give rise to the development of new modules. Decomposability is thus the condition that must be met in a top-down design process.

It should be noted that this requirement is essential in view of one of the fundamentals conditions of "programming-in-the-large": the ability to split a programming task between several persons. Of course, it is also important in the case of single-person programming, as a basis for an orderly approach to the solution of a problem.

Example: The top-down program design method was designed to meet this criterion.

Counter-example: Many modules require some kind of initialization, i.e. a set of steps to be taken before the module may perform its first useful tasks. In some design methods, however, it is required that all such module initializations be concentrated in a common "initialization module". Since a module's initialization is so closely related to the rest of the module, such a rule is clearly quite anti-modular as far as decomposability is concerned.

6.2.2. - Modular composability

The second criterion, composability, is a mirror image of decomposability. A method or language will be modular according to this criterion if it favors the production of software elements which may be freely combined with each other to produce new programs, possibly in an environment which is quite different from the one in which they were initially developed (figure 6.2).

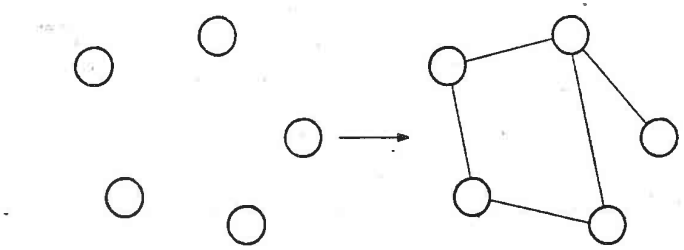


Figure 6.2 - Composability

The composability criterion corresponds to one of the most serious problems which confronts the software industry: the need for **reusable software.** Because this problem has not been solved, people now write similar programs or program elements over and over again; most of the time, they are not able to use solutions which have already been devised, by others or even by themselves, to answer the same (or almost the same) questions. Software engineering will not progress significantly unless we become able to avoid this constant re-inventing of the wheel. We **must** find a way to design pieces of software performing definite tasks, which should be usable by any program requiring these tasks outside the initial context.

It is a pity that the fashion for top-down design (or, rather, for naive interpretations of this methodology) has led many people to underestimate the importance of the reusability issue.

The criterion of composability reflects an old dream (which, perhaps, is more a program user's or a programming project manager's dream than a programmer's one): that of having the software design process look like a construction box activity, which would consist in combining existing standard elements. Although such an extreme view will probably remain unrealistic for a long time, much progress is indeed possible towards composability (we will just mention here, without elaborating on it at all, a phrase which we think contains one of the keys to this evolution: **Computer-Aided Software Design**).

Example 1: Subprogram libraries are designed as sets of composable elements. Even though they are under-used in practice, mathematical libraries (such as IMSL, NAG, LINPACK, EISPACK, Harwell, to name some of the best) represent the most advanced developments in this area.

Example 2: Commands in the Unix operating system all operate on an input viewed as a sequential character stream, and produce an output with the same standard structure. They are thus designed

so as to be composable; the operator denoted as | is used for the purpose of composition, so that $A|B$ represents a program which will take A's input and have it processed by A, A's output being sent to B as input and processed by B.

Counter-example 1: Switch on your terminal, or pull out a card drawer, and have a look at some of the programs you wrote last year.

Counter-example 2: One popular way to "extend" the facilities of languages like Fortran (which in most cases means in effect to correct some of their most blatant deficiencies) is to use so-called **preprocessors** which will accept an extended syntax as input and map them into the standard form of the language, producing programs (e.g. Fortran ones) as output. Such software tools are usually not composable with one another.

Counter-example 3: In the initial definition of Pascal, as well as in the proposed ISO level 0 standard, the bounds of an array must be compile-time constants; if they array is a procedure parameter, the actual bounds must be declared in the procedure as constants all the same. This means that a library procedure for summing two vectors with 103 elements may be written in the language; a procedure for summing vectors with 104 elements may also be written, but it has to be a different one. In practice, this precludes the use of these versions of Pascal to write reusable software involving arrays - and, in particular, numerical software. The problem is corrected through the notion of **conformant array** (which does not lift the requirement for constant bounds, but allows the bounds of an array parameter to a procedure to be specified by the calling programs only, as e.g. in Fortran) in level 1 of the ISO standard. Adherence to this level is, however, not compulsory.

6.2.3. - Modular understandability

The third criterion, as well as the next one, is important with respect to the "maintenance" phase in the software life-cycle, which many studies have estimated to account for 60 to 90% of software costs. A program (or program specification, or program design) will be modular with respect to this criterion if the text of each of its modules may be understood by a human reader all by itself, without making any reference to other modules, or by making reference to as few other modules as possible (figure 6.3).

Example: The method which at first sight seems the least modular one of all, yielding only one-module complete programs, satisfies this particular requirement. . . providing the resulting programs are understandable at all.

Counter-example: If a set of modules has been so designed that its correct functioning depends on these modules being activated in a certain prescribed order, then they will not be individually understandable.

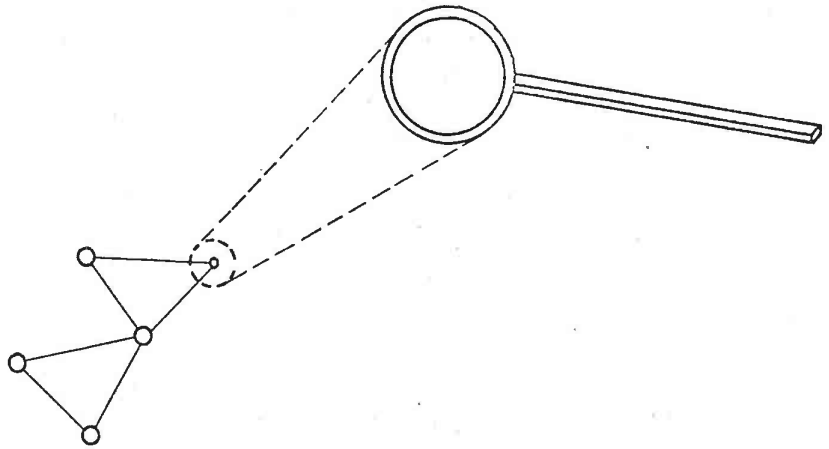


Figure 6.3 - Understandability

6.2.4. - Modular continuity

The fourth criterion corresponds to another characteristics of software projects, which, although perhaps regrettable, is just about universal and must be dealt with openly: the fact that the requirement specification for the problem to be solved will always vary during the lifecycle of the project. A programming method is modular with respect to the continuity criterion if a small change in a problem specification results in a change of just one module, or few modules, in the program obtained from the specification through the method.

The term "continuity" is drawn from an analogy with the notion of a continuous function in mathematical analysis; the "function" which should be "continuous" here (of course these words should not be taken too literally) is (see figure 6.4) the function:

Programming method: Specification → Program

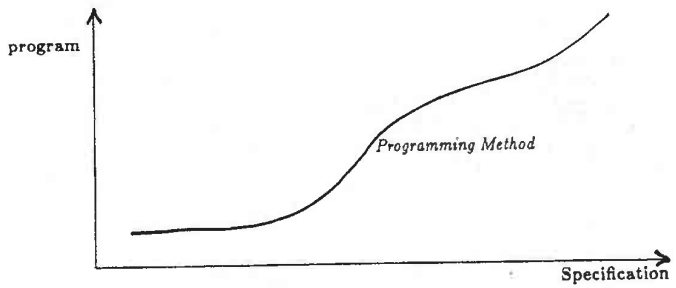


Figure 6.4 - Continuity

Continuity is one of the main benefits we may feel entitled to draw from a truly modular method. The problem here is that whereas the notion of a "small change" has a fairly intuitive meaning with respect to changing a program (there may even exist reasonable ways of measuring the size of a program change), it is much more difficult to define for specifications. Worse yet, anyone who has had the opportunity to discuss the urgency of a change with a customer knows that programmers and program users have very different perceptions of what "small" means in the context of changes to program specifications.

Example 1: Some programming projects enforce the rule that no numerical or textual constant should ever be used in the statements of a program: all constants must be referred to through symbolic names, whose associated value only appears in a constant definition clause (*PARAMETER* in Fortran 77, **constant** in Pascal or Ada, = in Algol 68). Thus, if the value has to change, only the definition is impacted. This is a very wise precaution as regards continuity.

Example 2: The Uniform Referent property: Let x be the name of an object, and a the name of an attribute possessed by objects of the same type as x. For example, x might be the name of a bank account, and a the "current balance" attribute of bank accounts. Let t be the type of x ("bank account" in our example).

In programming terms, a may be represented by a field designator if objects of type t are implemented as records (e.g. a bank account is represented by a record with fields such as account holder's name, credit, debit, current balance, etc.); alternatively, a may be associated with a procedure or function working on objects of type t (if the credit and debit are stored for every account, the balance may be computed as their difference whenever required, rather than stored permanently). Choosing between these two representations is a space-time tradeoff; the former economizes on computation, the latter on storage.

In many languages, the notation used to refer to attribute a of object x will not be the same in both cases: access to a record field will be written $x.a$ (as in Pascal, Simula, PL/I, Ada), or x **of** a (as in Algol 68, Cobol), whereas function call will be written $a(x)$. On the other hand, in Algol W, the notation $a(x)$ is used in both cases (in Simula, the notation $x.a$ will also serve both purposes if a, whether a variable or a function, is local to the class of which a is an instance).

This property of languages such as Algol W (and Simula to a certain extent) is known as the **Uniform Referent** property; the phrase means that there is a uniform way of referring to certain elements, independently of their implementation.

Clearly, a language possessing the uniform referent property will favor modular continuity, since a reversal of the initial implementation decision (using a function rather than a stored attribute, or vice versa), which is quite possible in the lifetime of a project, will entail changes only in the module(s) where the implementation of the objects at hand, such as x and its attributes, is described - not in those where they are used.

Counter-example 1: A method in which program designs are patterned after the physical implementation of data, e.g. customer's address begins on byte 27 of customer record, or printer interrupts are recorded at address 2351, will yield designs which are very sensitive to slight changes in the environment.

Counter-example 2: Languages such as Fortran or Pascal which, in contrast to Algol, Simula or Ada, do not allow to declare arrays whose bounds will only be known at run-time, make program evolution much harder.

6.2.5. - Modular protection

The last criterion corresponds to another fundamental issue: errors, and, more precisely, propagation of errors. A system is modular with respect to this criterion if the consequences of an error remain confined to the module in which the error occurred, or this module and few others.

As far as programs are concerned, the kinds of errors which are of concern here are run-time errors, e.g. errors resulting from hardware failures, erroneous input, lack of needed resources (like exhaustion of available storage). In a broader context, e.g. if we consider the modular structure of specifications as well as program designs and programs, protection also implies that a logical error made in, say, one module of a specification (stemming for example from an incorrect understanding of the problem to be solved), has consequences in a small number of modules in the design document or eventual code.

Example: A methodology which imposes that every module which contains input statements also contains statements to check the conformity of input data and correct abnormal values is good for modular protection.

Counter-example: Languages such as PL/I and Ada have the notion of "exceptions", with special statements to "raise" an exception and statements to "intercept" an exception; when an exception is raised, control will be transferred to the intercepting statement, which may be anywhere in the program. Unless used with a strict discipline, such facilities may lead to programs with bad modular protection (which is of course better than no protection at all).

6.3. - FIVE PRINCIPLES

From the above set of criteria, certain principles follow which must be observed to ensure proper modularity. We shall study five of them:

- linguistic modular units;
- few interfaces;
- small interfaces (weak coupling);
- explicit interfaces;
- privacy (information hiding).

All but the first have to do with the basic issue of **intermodule communication.**

6.3.1. - Linguistic Modular Units

The principle of linguistic modular units is fairly obvious but worth recalling anyhow.

> Modules must correspond to syntactic units in the language used.

The language mentioned here may be a programming language, a program design language, a specification language etc.

What this precludes is the possibility of having the module structure described from the outside, with no correspondence with the linguistic structure of the program (e.g. module X extends from lines 47 to 283 of procedure P).

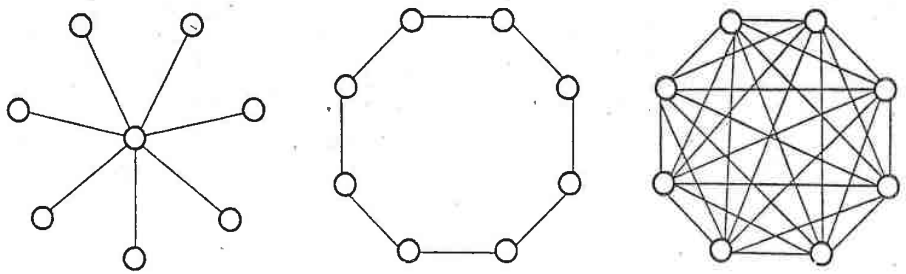
This principle clearly follows from the criteria of decomposability (if we want to separate tasks, then every one must result in a clearly delimited syntactic unit), composability (how can we combine anything else than closed units?), and protection (we can hope to be able to control the scope of errors only if modules are syntactically delimited).

6.3.2. - Few Interfaces

An important characteristics of the structure of a purportedly modular system is the number of intermodule relations. A relation may exist between two modules in a variety of ways: they maybe be procedures, one of which can call the other; they may access common information; etc. (see section 6.11 for a classification of these possible relations). The "Few Interfaces" principle limits the number of such connections:

> Every module should communicate with as few others as possible.

More precisely, if a system is composed of n modules, then the number of intermodule connections should remain much closer to the minimum, $n-1$ (see figure 6.5 (a)) than to the maximum, $n(n-1)/2$ (see figure 6.5 (c)).



(a) (b) (c)

Figure 6.5 - Types of module interconnection structures

This principle follows in particular from the criteria of continuity and protection: if there are too many relations between modules, then the effect of a change or of an error may propagate to a large number of modules. It is also connected to the other criteria: composability (if we want a module to be usable by itself in a new environment, then it should not depend on too many others), understandability and decomposability.

It should be noted that, whereas figure 6.5(a) shows a way to reach the minimum number of interconnections ($n-1$) through an extremely centralized structure (one "boss", everybody else talks to him and to him only), there are also much more "libertarian" or "anarchistic" structures, like that of figure 6.5(b) which is almost as good with respect to the Few Interfaces principles (n connections) but looks quite different in its organization (everybody talks to two immediate neighbors). Although the latter style of design is sometimes viewed with suspicion, it may yield very interesting and solid results, as we shall see later.

6.3.3. - Small Interfaces (Weak Coupling)

The "Small Interfaces" or "Weak Coupling" principle (see figure 6.6) relates to the size of intermodule connections rather than to their number:

> If any two modules communicate at all, they should exchange as little information as possible.

Using an expression from electrical engineering, what this principle says is that all channels must be of limited bandwidth. This requirement clearly stems, in particular, from the criteria of continuity and protection.

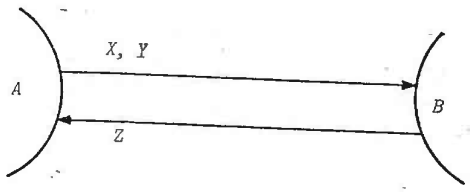


Figure 6.6 - Intermodule communication

An extreme but all too frequent counter-example is a Fortran practice which some readers will probably recognize: the "Garbage Common Block". Programmers who use this technique find it very convenient to put at the beginning of every program unit an identical, gigantic *COMMON* directive which lists all significant data objects (variables, arrays), so that every unit has access to every datum. Sure this is very convenient: no need for declarations in a new unit, just copy the Garbage Block! And program debugging is so much of an excitement ...

It should also be noted that the block structure of Algol-like languages is quite dangerous with respect to the Small Interfaces principle: any block has access to all the objects belong to higher level enclosing blocks, including many which are of no interest to it, there does exist a risk of unjustified access to such objects. Thus, the equivalent of the "garbage common block" may be found in Pascal or PL/I programming (too many variables declared at the outermost level), in C programming (too many "external" variables), etc.

6.3.4. - Explicit Interfaces

With the fourth principle, we go one step further in enforcing a totalitarian regime upon the society of modules: not only do we require that everyone talks with few others, and that any such conversation be limited to the exchange of a few words; we also impose that it must be held in public and loudly!

> Whenever two modules *A* and *B* communicate, this fact must be obvious from the text of both *A* and *B*.

Behind this principle stand the criteria of decomposability and composability (if a module is to be decomposable into or composed with others, any outside connection should be clearly marked), continuity (what other element might be impacted by a change should be obvious) and understandability (how can one understand *A* by itself if its behavior is influenced by *B* in the same tricky way?).

As we shall see later, the principle of Explicit Interfaces plays a very important part in the search for adequate modular structures. One of the problems is that, whereas one of the most obvious kinds of intermodule coupling is the classical procedure call, another, much less explicit kind occurs through data sharing.

Figure 6.7: Data Sharing

Assume for example (figure 6.7) that module *A* modifies and module *B* uses the same data element *X*; *X* may be a variable, an internal structure, a file, an attribute of an external device (e.g. a sensor), etc., accessible to both modules through a sharing mechanism such as the Fortran *COMMON*, the Cobol Data Division, or the block structure of Algol-like languages. *A* and *B* are in fact strongly coupled through *X* even though there may be no apparent connection, such as procedure call, between them.

6.3.5. - Privacy (Information Hiding)

The Privacy principle, also known as Information Hiding, relies upon the assumption that there exists for every module an official description of its purpose, which is normally different from (and less detailed than) the description of its function, i.e. of how the module may rely upon properties of another module which are not part of such official description for this other module[1].

> All information is private unless it is declared public.

This principle prohibits a module from relying on the way another module works internally. The fundamental criterion here is continuity: if a module changes, then other modules may not be affected by its changes if the interface remains the same. If the specification of interfaces is of a sufficiently high level, i.e. describe function, not implementation, they will remain unaffected by technical changes.

For example, a procedure used for retrieving the attributes associated with a key in a table (e.g. in a personnel file or the symbol table of a compiler) will internally be very different depending on the way the table is stored (sequential array or file, hash table, binary or B-Tree, etc.). The document describing the usage of this module should only contain information which describes the services it offers (storing an element, searching for the element associated with a given key, asking for the current number of elements), rather than details about the particular implementation techniques chosen. A module which uses it only through these official properties, without relying on the implementation, will not suffer when the initial implementation choice is modified - a very common event in software projects.

The need for separating the description of function from that of realization and the principle of information hiding which it implies are also related to decomposability, composability and understandability: to separately develop the modules of a system, to combine various existing modules, or to understand individual modules, it is indispensable to know exactly what each of them may and may not expect from the others.

---

[1] We use the word "description" here, rather than "specification", to avoid any confusion with the specification phase of the software life-cycle; indeed we feel the principle may be applied to the modularity of specification documents as well: a complex specification should have a modular structure, and it may be deemed necessary to distinguish between the "public" and "private" parts of every module in the specification.

6.4. - FIVE KEYWORDS

We now come to more technical aspects of the modularity concept in programming. Our five keywords describe features which are useful, if not necessary, in order to build truly modular programs. They are:

- synchronization;
- data transmission and sharing;
- genericity;
- persistence;
- separate compilation.

The first three have to do with inter-module connections; the last two with the autonomous development and behavior of each module.

6.4.1. - Synchronization

Our first keyword is relative to module connections occurring through transfer of control. An obvious way for two modules A and B to be related is when the execution of some program code belonging to A depends on the execution of some other code from B.

The best-known example of such connection is subprogram call: execution of A is suspended; execution of B starts at the beginning of B, and proceeds until the end of B; then execution of A resumes when it had stopped.

This is not, however, the only possible kind of control connection. The more general mechanism may be called **synchronization**. Synchronization extends to its full scope in the case of parallel programming, where more than one module may be active at any given instant. The study of synchronization among parallel processes brings some interesting concepts back into the domain of sequential programming as well, as we shall see in section 6.9.

6.4.2. - Data transmission and sharing

The other obvious facet of intermodule communication is through the exchange or sharing of data.

The delicate point regarding explicit transmission of data (as via argument passing), in particular with respect to the continuity and protection criteria, is to be able to assess precisely what may happen to a datum which is passed through from one module to another or, more generally, what *access rights* are associated with a datum being transmitted. This problem has received a great deal of attention for operating systems. In programming languages, the least which should be required is that the rights any module has on arguments transmitted to it be defined precisely.

Communication of data between modules occurs not only through explicit transmission, but also through sharing. Data sharing is allowed, for example, by the *COMMON* mechanism of Fortran and by the block structure of Algol-like languages. This feature is both convenient and quite dangerous; as we have seen, it tends to breach the Explicit Interfaces principle. Block structure, in particular, has been often criticized because it provides unlimited upward visibility: a statement belonging to a block may access and/or modify data belonging to any enclosing block. The basic elements of a remedy may be contained in Dijkstra's notation [Dijkstra 76] which defines at block inception, for each imported or exported element, the rights that the new block may exert on this element.

In any case, data sharing should be used with great caution, since it gives basically the same power as data transmission, but in a much less visible fashion.

6.4.3. - Genericity

[To be completed. Examples from Ada and LPG (a language developed at IMAG, Grenoble.]

6.4.4. - Persistence

The data elements used by a program module are **persistent** if they retain their values from one activation of the module to the next.

For example, the internal variables of a class object in Simula are persistent ("activation" as used in the above definition here includes creation by the **new** construct, call of any of the procedures of the class relative to the object, and re-activation by **resume**). Some entities may be explicitly designated as persistent in languages such as Fortran 77 (*SAVE*) Algol 60 (*OWN*) and Algol 68 (heap). On the other hand, no data element of a Pascal procedure is persistent through the successive activations of this procedure; the only way to have it retain its value into move its declaration outwards to the enclosing block, which is obviously very harmful to modularity.

Allowing persistence of data local to a module is fundamental with respect to the construction of modular programs. If we want modules to be truly autonomous and homogeneous entities, then they should be allowed to "possess" and manage their own data. If this is not done, all the information which is necessary to re-start a module will have to be provided by the module which triggers the re-starting; this means that information pertaining to one module will be disseminated among other modules, possibly many of them (since the propagation of information may be carried quite far). Such a scheme obviously breaks the Small Interfaces and Privacy principles.

Although the necessity for persistence of local data as an important asset for modularity clearly follows from the previous discussion, it is interesting to note that this idea contradicts some commonly found ideas of programming methodology. For example, the methodology known as "structured/composite design" [14], which has modularity as one of its central themes, advises against the use of persistent data. although this approach may be in part justified by the technical difficulties encountered when dealing with persistence in programming languages, we feel that is based on a partial and short-sighted view of modularity, taking the procedure as the only possible kind of module; this view as we shall see in the next section, encompasses only part of the problem.

What the persistence concept really contradicts is the usual notion of "pure procedure", viewed as an idealization of the subprogram and thus of the module. A pure procedure is a mechanism which performs some computation which may entirely be specified in terms of the relation between input (arguments communicated to the procedure) and output (results computed by the procedure and returned to the calling programs). It is thus the programming equivalent of a mathematical function (except that one may, without too much trouble, accept as pure procedures mechanisms which have not only **in** arguments and **out** results, but also some **in out** arguments).

A pure procedure is thus like what some people view as the ideal subordinate: memoryless, historyless, capable of performing a well-defined task whenever it is ordered to do so and provided with the requisite information, and ready to disappear again once it has delivered the expected results. In contrast, a module with persistent data manages its own information and accumulates experience, a feature which not all "bosses" are enthusiastic about. (The two approaches are not unlike what is known in social science as "theory X" and "theory Y").

The necessity of persistence and the insufficiency of the pure procedure concept are obvious even for very simple problems. The following two examples are typical:

- write a module of some kind, which upon request will print an integer passed to it, in such a way that two successive items are separated by a blank; there should thus be a way for the module to remember whether it has already been requested to print an integer (so that a number is preceded by a blank if and only if it is not the first one);
- write a "pseudo-random number generator", the algorithm used by such tools generally a relies on a sequence of the form $a_0 = x$, $a_{i+1} = f(a_i)$, where x is a "seed" provided initially be the requesting

module, a_i is returned as "pseudo-random" number at the i-th activation of the module, and f is a function designed so as to provide as much apparent "randomness" as possible. If every activation of the module is to return a new number a_i, then the module should be able to retain the last computed value.

It should be noted that expression of persistence raises some problems as far as programming languages are concerned. One of them is the conflict between persistence and dynamic allocation; historically, it is known as "the Algol 60 Dynamic *OWN* array problem", since the initial Algol report allowed both the declaration of any object as persistent (*OWN*) and the presence of array with adjustable bounds (i.e. the bounds of an Algol 60 array declared in a block B may involve variables, provided these variables are declared in a block enclosing B and have been assigned proper values whenever B is executed). An array having adjustable bounds must clearly be re-allocated every time B (which may be a procedure) is executed; but then what does it mean for the array to be persistent? Later versions of Algol 60 removed this contradiction by forbidding adjustable *OWN* arrays. Descendents of Algol usually do not have an *OWN* declaration; one of them, Simula includes persistence in a nice way through classes.

A problem which is intimately connected with persistence is that of **initialization**. If we have variables which retain their values, then we must have a way of describing the value they have when the module to which they belong is first activated. Note that naively writing

if *first_call* **then**
    initialize ; first_call := **false**
end if

does not provide a solution since *first_call* itself has to be a persistent variable.

Some languages provide syntactic support for the expression of initial values. In Fortran, if X is a persistent variable, we will write

SAVE X
DATA X /x_0/

where x_0 is a constant. In Algol 68 we will write the initialization in the declaration statements, e.g.

loc int $X := x_0$

None of these solutions, however, is entirely satisfactory. The limitation to constants may be too restricting: for example, one might want x_0 to involve previously initialized elements, as in $i := \cdots ; j := 2*i$ or subprogram parameters. On the other hand, allowing x_0 to be any expression opens dangerous possibilities: by destroying the traditional fence between the entirely static nature of declarations and the dynamic nature of expression evaluation, are may run into trouble; for example, the evaluation of x_0 might entail side-effects (in Ada, where one may write X: **integer** $:= x_0$ in a task, x_0 could be a call to a function which starts another task which communicates with the initial one ...).

In fact, a satisfactory solution to the problems raised by persistence and initialization implies going beyond the concept of procedure and considering modules as **processes**. This will be studied in sections 6.8 and 6.9.

6.4.5. - Separate compilation

The possibility to compile modules separately is a necessity for any module structure provided by a programming language.

The need for separate compilation is **not**, as often argued, motivated primarily by efficiency concerns. This is shown a contrario by the fact that, on many systems, separately compiled modules may only be executed after they have been assembled by a special system program called a **linker** (or linkage

editor, or linking loader), which often takes as much time per module as the compiler. In fact, the main argument for separate compilation, even when it is not explicitly formulated, it **information hiding:** once a module has been compiled, then it becomes a closed object; its internal details are no longer easily accessible (to use a popular term, they have been "encapsulated"). Actually, it is only usable through some external description; such a description thus **has** to exist.

A related property of compiled code which also plays a role is the fact that it is not modifiable any more. This helps in making a compiled module into "packed material", known only from its label.

Separate compilation is also important for decomposability (separate development of modules makes it necessary to work as far as possible on each individual module, up to testing) and, obviously, composability. Separate compilation is a natural extension of the "linguistic modular units" principle.

>From this discussion, it is interesting to note that compilation here is meaningful not only as code production, but also (and perhaps more importantly) as a global operation on a program or program fragment, which applies to it tools for verification (syntactic and semantic checking) and archival. Thus the notion of separate compilation, replaced by "separate analysis and archival", may be transposed to such inherently non-compilable objects as specifications or program designs; it is useful to constitute "libraries" of, for instance, specification modules, each of which has been processed by a set of tools for consistency checking and hiding of internal details.

Studying separate compilation not as a minor technical issue but as an essential requirement in the broad context of information hiding, modular design, software reusability, program libraries and programming-in-the-large has some interesting consequences. One of them has to do with the required software tools. With the above approach, a traditional linker - which, essentially, takes as input "quasi-object code" which still contains some symbolic elements corresponding to resolved external references, and replaces these by the actual addresses of the referenced modules - is no longer sufficient; one will be confronted with a variety of modules, subsystems and systems, each existing in several versions, and perhaps being split into a description part and an implementation part. What will be needed, then, includes:

- a **"module interconnection language"** which makes it possible to describe particular combinations of modules;
- a set of tools for what is called **"configuration management"**, i.e. keeping track of the respective states of the various modules, subsystems and systems, and their interconnections.

Such tools are essential in supporting a truly modular design methodology.

Separate compilation raises some non-trivial technical problems at the border between language and compiler design. The most important issue is to make it compatible with strong typing, a fundamental feature of modern languages. At the other extreme, in Fortran (a language for which separate compilation was one of the most important design criteria), the compiler stops at the module border: the compilation unit is the Fortran subprogram (subroutine or function); once a subprogram has been compiled and stored in a library, the compiler is not required by the language definition to check the compatibility of the formal argument list with the actual argument lists in calling programs; very few compilers will do it. In the code generated for each module, all typing information has been lost. The linker, when combining subprograms, cannot then perform any significant checking.

In a strongly typed language like Pascal, Simula or Algol 68, it is always possible to implement separate compilation by doing no more than in Fortran: provided a procedure does not use any global variable, it can be compiled separately. Such a simple scheme will not allow, however, for checking type compatibility; i.e. if we compile separately

procedure $P\ (x_1: t_1, x_2: t_2, \ldots, x_n: t_n)$

and procedure Q containing the call

$$P\ (a_1, a_2, \ldots, a_m)$$

we have no way, once P and Q have been compiled, to check that $m = n$ and that every a_i is of a type

compatible with that of z_i. What may be felt acceptable by Fortran programmers for the sake of "efficiency" (?) should be rejected by users of languages which have been explicitly designed so that their type systems allow for complete static verification of type compatibility. Accepting it would be accepting that compatibility is checked for entities within a given module, but not for objects belonging to different modules, which is all the more absurd that the most serious errors, and the ones which are the most difficult to detect and correct, precisely occur at inter-module connections (this is one of the reasons which make "programming-in-the-large" so much harder than "programming-in-the-small").

An apparent solution is to modify the usual notion of compilation, so that the result of compiling a module will contain not only some object code but also an **"interface symbol table"** which carries all the information necessary to check the conformity of calls to the module. In our example, the interface symbol table for P would contain, in an appropriately coded form, the list of argument types $t_1, t_2, \ldots, t_n$.

With this technique, module linking has to be performed by a special tool which must know about the structure of the interface symbol table, and hence about the type system of the programming language used. Such a tool may thus not be a general-purpose linker; it has to be "language-oriented" - a solution which raises some difficulties when one has to combine modules written in different languages, but of course this requirement is not compatible with that of intermodule type checking.

Unfortunately, there is a major flaw in this approach. It may be viable for a language like Fortran where the type system is essentially finite; in a language like Pascal, however, it cannot be applied to the separate compilation of **procedures** because of the presence of constructed (user-defined) types. Assume that t_1 and t_2 above are constructed types, e.g.

$$t_1 = \textbf{array}\,[1..k]\,\textbf{of}\ (a, b, \ldots);$$

$$t_2 = \textbf{record}\ (m_1\ p_1;\ m_2\ p_2; \ldots)$$

Thus, if the argument list of P and the call to P in Q are to make sense, then both P and Q must have access to the above type definitions (and others, such as the definitions of types P_1, P_2, ...). They must also be able to access common constants such as k (and its value), the names a, b, ..., etc.

Note that the definitions of these objects may **not** be part of the body of procedure P (or Q), since they are needed outside.

The consequence of this discussion is clear: **Pascal does not allow for separate compilation of procedures** if static type checking is to be retained. the same is true of other languages of the same class (rich type system, block structure) such as Algol W or Algol 68. Although many existing Pascal systems purport to provide separate compilation, they all either do not perform inter-module type checking or extend the language.

One should not draw the hasty conclusion that the static type checking and separate compilation are incompatible for modern languages: the contradiction which we have analyzed is relative to **procedures** only. What we must then deduce is that the procedure is not an adequate modular structure for separate compilation in modern languages.

We may note at this point that the impossibility would not have arisen if we had had our disposal modules which contained not only one (or more) procedures, but also constant definitions, and type declarations defined so as to be accessible from other modules. We shall, in fact, be looking for such kinds of modules, containing other elements as well, such as variable declarations and even statements[1]. But let's not anticipate ... .

[1] Note that Pascal offers a language structure which satisfies these requirements and embodies all the necessary information (whereas the procedure does not because external information may be needed to understand the argument list): the Pascal concept of **program**. Unfortunately, there is no interconnection mechanism between programs: a program may not call another one. There have been, however, implementations of separate compilation in Pascal based on the "program" concept. (extensions of

6.5. - SIMPLE-MINDED APPROACHES: THE FIVE SINS OF THE PROCEDURE

The simplest way to decompose a program into modules is to use the ordinary concept of procedures. It means that the task to be performed is divided repetitively into subtasks, each of which will correspond to a program unit.

In fact, the term "module" is still considered by many people as a synonym for "procedure". "Modular programming", in this context, simply means decomposing programs into as many short procedures as possible -- which may be better than no division at all, but does not really further the aims of modularity as we have analyzed them. Several books popularized this idea during the late sixties and seventies (see e.g. [9]); but even for more recent authors ([14]) the equation *module = procedure* still holds.

Procedures are of course one possible modular structure. There are, however, many problems with taking the procedure as the paramount type of module. We shall study five of them; some have already been mentioned and will be only briefly reviewed. The Five Sins of the Procedure are:

- lack of control of data propagation;
- incompatibility with separate compilation in the presence of static
- type checking;
- lack of flexibility of the argument mechanism;
- unequal control relationship;
- imbalance between process and data structures.

6.5.7. - Lack of control on data propagation

Whereas the direct interprocedural connections of "control" type, occurring through the procedure call mechanism, are obvious and easy to spot, the various procedures in a system may also be indirectly related via the data which they exchange or share. In many cases, these connections will be too big (violating the Small Interfaces principle), not as visible as the control connections (contrary to the Explicit Interfaces principle) and hard to document properly, especially in the case of sharing (endangering the observance of the Privacy principle).

6.5.8. - Incompatibility with separate compilation in the presence of static type checking

We have seen that the use of procedures as modules does not blend well with separate compilation if complete, static type checking is also required.

On the other hand, separate compilation, at least if taken as the ability to separately analyze modules and then make them into closed units, appears to be a fundamental component of any programming language or methodology if it is to be applicable to anything else than toy problems, student exercises and experiments.

6.5.9. - Lack of flexibility of the argument mechanism

Another serious flaw of procedures as modules is the bad performance of the argument passing mechanism with respect to the continuity criterion. In a procedure call

$$P(b_1, b_2 \cdots b_n)$$

corresponding to the procedure heading

procedure P (**in** a_1: ..., a_2: ..., ..., a_i: ...;

out a_{i+1}: ..., ..., a_n: ...)

the argument list $a_1, \cdots a_n$ describes the interface between p and the calling program units. Now in the evolution of a system, it very commonly happens that a new use for an existing procedure requires new procedure arguments or, to the opposite, makes some arguments meaningless. A typical example is the case when p is an output statement. and some of the **in** parameters describe terminal characteristics; then at some point comes a terminal with new options, such as the possibility to choose between several colors for outputting an element.

In such a case, a new argument will have to be added to p (in other situations, a argument might have to be removed); the body of p will be modified in order to take this parameter into account. The problem is that p's interface with the outside world will be changed; so **every program unit which uses** p must be updated, in order to include the right arguments in the call to p. The change will affect p itself and the units which require the new facility -- and so far this is perfectly natural -- , but also all previously written units, which do not use the extension. This is unacceptable in a large system: as functions are added or removed, existing code would have to be constantly updated even if it is logically unaffected by these changes.

It should be noted that the presence of keyword arguments, with default values, as in Ada, provides only a partial solution to this question.

Some remedies to this problem have been studied in [12]. A proper solution implies going beyond the concept of procedure.

6.5.10. - Unequal control relationship

We have already remarked, when studying persistence (4.4), that the relationship from calling program to called procedure was a very dissymmetric one, of the master-slave kind. A procedure is a perfect specialized servant, which does not exist except when it is required to execute its master's orders. In our search for ways of allowing the development of systems as sets of autonomous and coherent parts, we may hope to find module types which lend themselves to relations more oriented toward the "self-management" style.

6.5.11. - Imbalance between process and data structures

A central duality in programming is that of process *vs.* data, or algorithm *vs.* objects. Data processing (a suggestive expression) consists in applying some algorithms to some objects; a data processing system can only be deeply understood through an analysis the structure of both process and data structures. A representation of this situation may be constructed by associating with the system a graph (fig. 6.8) which has the data elements as its nodes, and the data transformations as its vertices (a multigraph is actually needed in the general case).

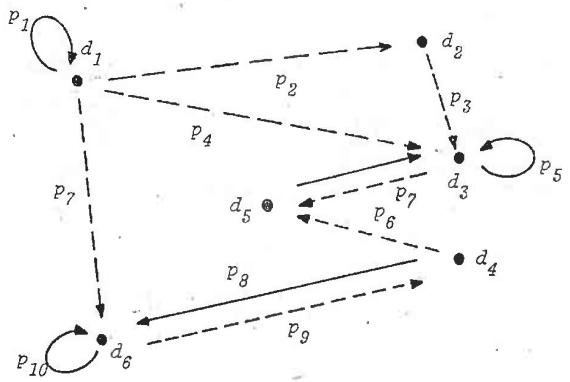


Figure 6.8 - A SYSTEM GRAPH

Building programs as systems of procedures is a serious break of the process/data duality, since procedures correspond exclusively to divisions of the process part of the system: such a modularization completely neglects the other half. As might be expected, the data half takes its revenge in the form of insufficient coherence of the resulting modules: this is apparent in several of the problems which we have encountered and discussed such as sneaky connections through data sharing.

More satisfactory modular policies will have to restore the balance and take into account the data as well. We shall in fact see the usefulness of introducing **data units** for system modularization, along with the traditional "program units", i.e. procedures.

A data unit is a module which is organized around a data structure, whereas a procedure is based on a processing step. Because of the dual relationship of data and process, a data unit will contain, along with the description of a data structure, the procedures which act upon it, in the same way as a procedure unit attaches to the description of a procedure that of the data structures on which it operates. The modularization choice which faces the designer is thus, at each step, whether to put the data into the procedures or the procedure into the data.

For example two extreme policies for the system of fig. 6.8 would be having six data units, corresponding to $d_1, d_2, \ldots d_6$ or ten procedure units, corresponding to $p_1, p_2, \ldots p_{10}$. All kinds of intermediate solutions are also possible.

It may seem at first sight that each of these groupings will be just as artificial as the others. The first extreme solution means that p_2, for example, has to be split between d_1 and d_2, or arbitrarily attached to one of them; the other implies the converse situation for d_2 with respect to p_2 and p_3. But not all choices are equivalent with respect to modularity.

Although there is not absolute rule for decomposing along one line before the other, some serious arguments tend to favor the "modularize around-data" method at the outermost level of a system. The main criterion here is continuity. During the evolution of a software system, there is a very good chance that the **tasks** which are required from the system will change. A payroll processing system, for instance, will be required to provide new outputs, e.g., statistics, and the payroll computation will change due to new rules, evolving legislation, etc. On the other hand, the **objects** which are manipulated by the system will probably remain roughly the same, **at least if viewed from a sufficient level of abstraction**: e.g. the payroll system will always work on things like employee records, salary scales, social security files etc. It is thus probably a better bet on the future to base the overall structure of the program on the objects than on the tasks.

This argument -- which lies at the basis of "object-oriented programming" -- plays an important role in the rest of this discussion. Again, we should mention that it contradicts simplistic presentations of the top-down method. The usual description of top-down design goes somewhat like the following: begin by

describing the overall purpose, or "top" of your system, e.g.,

compute monthly payroll

this gives "the first version of the system", unfortunately too abstract yet to be implementable. So it must be **refined**; the first refinement could lead to something like

```
-- compute monthly payroll:
        input employee data ;
        input monthly work data :
        for each employee do
            compute monthly pay ;
        output payroll
```

You should then apply the same process to the still unresolved pseudo-statements, like "*input employee data*", etc. and proceed repetitively until you are only left with implementable statements.

The problem with this approach is that it does not apply to real size programs. It is meaningless to talk about "**the**" purpose of a nontrivial program: **real systems have no top** in the above sense. Any serious system fulfills a **set of functions**. An operating system, for example, answers requests from various devices; pretending that "the" purpose of the system is described at the topmost level by

answer device request

and basing the development of the system on refinements of this "statement" will result in an entirely artificial structure. The same is true of systems which superficially seem more single-goal-oriented, such as the above payroll processing program, or a classical scientific computation; when one looks more closely and considers the full life cycle of any such program, it becomes clear that there are always several variants of the program, which are closely related by the **facilities** they use but pursue different aims. Among the many parameters which may characterize a system variant are, for instance, whether it is for actual production or for program test, and whether it is interactive or batch-oriented.

Let us concentrate on this last criterion and consider a program which has both a batch and an interactive form. The batch variant will have as its "purpose"

solve a complete instance of the problem

which, according to the top-down design credo, could be initially refined into something like

```
read input values ;
compute results ;
output results
```

The interactive version, on the other hand, may be described as

process one transaction

which could be refined into


```
if new information provided by the user then
        input information ;
        store it
else if request for information previously given then
        retrieve requested information ;
        output it
else if request for result then
        if necessary information available then
            compute requested result ;
            output it
        else
            ask for confirmation of the request ;
            if yes then
                obtain required information ;
                compute requested result ;
                output result
            end if
        end if
else ...
```

Thus at the outermost level there is apparently no relation whatsoever between these two programs; but in reality they are two facets of the same system, and their essential components will be the same.

What this discussion shows, in our opinion, is not that the top-down method is inadequate, but that it is an error to apply it to the process part only. The structure of a system should be analyzed in terms of both the objects it manipulates and the tasks (the plural being fundamental here) that it performs. At every step in the development of a system, the designer has the choice between refining data and refining processes. This choice must be made consciously; choosing the former alternative first at the basic levels of system development will, according to our experience, yield more coherent, solid and change-proof designs.

It is interesting to note that Wirth's 1971 paper [16], the first description of the top-down method, clearly stated that the refinement process should be applied to data as well as to algorithms. It does not however mention that the designer must often **choose** between the former and the latter.

We shall show in sections 6.6 to 6.8 how these ideas can be applied. We will see in particular when introducing "abstract data types" that a proper approach restores the process/data balance, which may seem to be destroyed by going too far in the modularize-around-data direction; indeed, a good way of describing a data structure is to use as abstract description the properties of the set of **functions** which may be applied to it, so that the loop is closed and the process/data complementarity appears clearly again.

Before we come to this, however, we must study a little more closely what "structuring around data" means. We will first take as example a design method which is well-known in business data processing circles: the "Jackson method". Although it has some obvious limitations, Jackson's method will help us introduce some of the most fruitful concepts.

6.6. - MODULARIZING ON THE BASIS OF THE PHYSICAL DATA STRUCTURE

The method developed by M. Jackson [7], (similar to the one preached by J.D. Warnier) has been essentially used for business data processing applications.

One of the basic ideas of the method is that the structure of a program should only depend upon the structure of the data it manipulates. In particular, the tasks to be carried out do not influence this structure; they only determine how it will be filled.

Since "data" is a somewhat overused term, it is necessary to say what it means in the particular context. Let P be a program using some input I, producing output O and having some internal data structure D (fig. 6.9). The word "data" is used in the literature to mean at least four things:

- 1- I
- 2- D
- 3- $I \cup O$
- 4- $I \cup D \cup O$

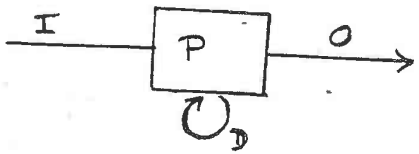


Figure 6.9 - DATA MANIPULATION

The data considered by Jackson are apparently those of the fourth category, i.e. input and output files. The structure of such files will be described in Jackson's method by a tree using a combination of the base schemata of figure 6.10: for instance, an element type of A is the concatenation of one element of type $A1$, one of type $A2$, one of $A3$ and one of $A4$; a B is either a $B1$, a $B2$ or a $B3$; a C is zero, one or more occurrences of a $C1$.

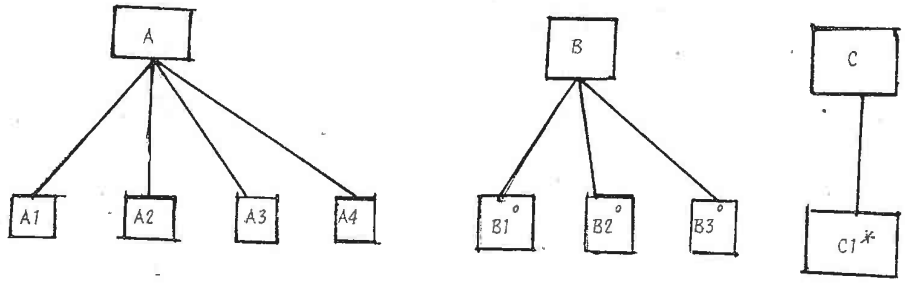


Figure 6.10 - BASIC JACKSON STRUCTURES

The central idea in Jackson's method is that a program processing a file whose structure uses these elements will have a corresponding structure, with sequence corresponding to concatenation, conditional to alternative, and loop to repetition.

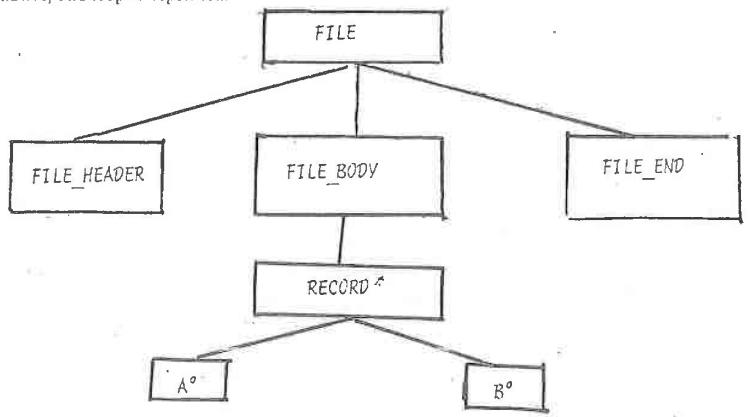


Figure 6.11 - A FILE STRUCTURE

For example, a file having the structure given by figure 6.11 will be processed by a program of the form:

```
-- process_FILE:
        process_FILE_HEADER ;
        -- process_FILE_BODY:
            until file_end do
                read record ;
                -- process_RECORD:
                    type_A: process_A,
                    type_B: process_B ;
            end do
        process_FILE_END
```

The study of the tasks to be performed will be carried out separately, resulting in a list of necessary elementary "actions". These actions will then be spread out into the "holes" of the above program structure *(process_A, process_B,* etc.), to obtain a complex program.

The following example is taken, slightly simplified, from Jackson's book. An input file of card images is to be analyzed. There are three card types, *A*,*B* and *C* (distinguished by a character in some position of the card). The required analysis is as follows:

- count the cards preceding the first *A* (the resulting count will be held in variable *initial*);
- print the first *A*;
- print the last card, which is always the first *C* following the first *A*;
- count the "batches", where a batch is either a contiguous sequence of *A* cards or a contiguous sequence of *B* cards, starting with the first *A* (this count will be held in variable *batches*).
- count the number of "batches" of *B* (this count will be held in variable *Bbatches*).

As implied by the method, only the structure of the data determines the structure of the program, not the tasks to be carried out.

It is very easy to draw the diagram corresponding to this structure (figure 6.12)

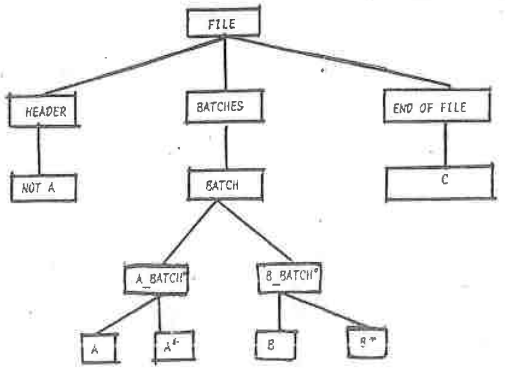


Figure 6.12 - A FILE STRUCTURE

The program operating on such data **must** then have an isomorphic structure:

```
process_FILE:
        process_HEADER ;
        process_BATCHES ;
        process_END_OF_FILE
```


where the refinements of the statements are:

```
process_HEADER:
        init_HEADER ;
        until A do
            process_not_A
        end do

process_BATCHES:
        init_BATCHES ;
        until not batch do
            process_A_BATCH ;
            process_B_BATCH
        end do

process_A_BATCH:
        process_starting_A ;
        until not A do
            process_batch_A
        end do

process_B_BATCH:
        process_starting_B ;
        until not B do
            process_batch_B
        end do

process_END_OF_FILE:
        process_C
```

The method then requires that we list the necessary "executable actions". To read the file, we will need tests and actions. First we need to express the three tests two distinguish between card types; they may be refined as follows:

A : (type(card) = A)
B : (type(card) = B)
C : (type(card) = C)

type (card) may be given for example by the first character of *card*.

The "actions" are of two different kinds: input-output and computation. The first kind comprises the statements necessary to handle the input file ("open" and "read") and to print the special cards and the requested counts; thus we find four kinds of input-output statements:

open
read *(card)*
print *card*
print *counter*

The computation statements are needed to compute the various requested counts; for each of these counts, we need a statement to initialize the corresponding counter and a statement to increase it:

initial := 0	*initial := initial + 1*
batches := 0	*batches := handler + 1*
Bbatches := 0	*Bbatches := Bbatches + 1*

What remains is then to "spread out" these actions into the program structure by refining the remaining pseudo-statements and tests. It is easy to obtain in this way the refinements of the yet underdeveloped statements:

```
init_HEADER:
        open ;
        initial := 0 ;
        read card

process_not_A:
        initial := initial + 1 ;
        read card

init_BATCHES:
        print card ;
        batches := 0 ;
        Bbatches := 0

process_starting_A:
        batches := batches + 1 ;
        read card

process_batch_A
        read card

process_starting_B:
        batches := batches + 1 ;
        Bbatches := Bbatches + 1 ;
        read card

process_batch_B:
        read card

process_C:
        print card
```

By moving all these statements into the required position in the program structure, we get a solution to our problem.

This example of the application of Jackson's method (we shall see another one in section 6.9) is quite representative of the benefits and limitations of this method. The nice part is that the design process goes very smoothly by just applying the basic recipe: use the data structure first. The "actions" are quite easy to map into the overall structure (with the partial exception of the "read" operations, whose proper placement requires some insight).

On the other hand, some obvious criticisms may be voiced. The first is that things may not go so well when there is more than one data structure involved; in particular, "interesting" problems are those in which the input and output structure are different! This case, called "structure clash", is dealt with in detail by Jackson, who suggests a coroutine-like solution schema.

Of more direct interest to us for this discussion is the study of the method with respect to continuity and protection. It is all very nice to base the program structure on the data structure, but it also means that the data structure will be "wired" into the program structure, which will thus have to change just as much as the structure of the data, in case the latter evolves. The problem, however, is that the way the data structure has been modeled here is so strictly fixed that it will be very difficult to change anything without changing everything; consider for example what happens to the structure depicted in fig. 6.12 if we change the specification just slightly: the header part now consists of cards which are not only not A but also not B; the sequence of batches may begin with either an A-batch or a B-batch. Although this new problem is very close to the initial one, the structure will have to be completely reworked. The same is true if we try to extend the program in order to cater to possible input errors (e.g., no A-card in the like). Such disruptions of the program structure for conceptually small extensions are not acceptable in

light of the requirements for modular design.

These difficulties reflect two of the most serious limitations of Jackson's method:

- the restriction to tree-like structures for the modeling of data, which is too constraining in many cases; for instance, even in simple problems, a general graph (e.g., a transition table) may be needed;
- even more importantly, the fact that the structure used is the external, physical one: figures 6.11 and 6.12 describe a file in a way which is very close to how it appears on a disk or deck or cards. This is the main reason why the designs which are obtained by using the method are so sensitive to surface changes in the appearance of the data.

One of the lessons which may thus be learned from Jackson's method is that data should be described by deep properties, not by physical structure. This will be the topic of the next section.

Despite its limitations, Jackson's method (which the above presentation described only in part) provides some important insights into the structure of programs. Some of its principles are worth pondering over carefully; one of the most interesting ones, although it seems at first sight paradoxical, is the idea that the actions required by the problem specification will be found in the **contents** of the program but should exert no influence at all on its **structure**, which should be entirely determined by the structure of the data. Data structures, however, should be described in a deeper way than what we have seen so far.

6.7. - ABSTRACT DATA TYPES AS A BASIS FOR MODULARIZATION

6.7.1. - Overview

[SECTIONS 6.7.1 AND 6.7.2 WILL BE REPLACED BY REFERENCES TO CHAPTER 5, WHICH DESCRIBE ADTs IN A DEEPER & MORE COMPLETE FASHION.]

If data structures are to be used as a basis for proper modularization, we must find a method to describe them in a sufficiently abstract way.

How can we describe a data structure? Let us take the simple example of a "table" structure, used to store keys, which we will assume are just character strings (of type *STRING*), and to retrieve them. Such a table is necessary in many applications, e.g. compiler writing ("symbol tables" will contain program identifiers), business data processing, etc.

The most obvious way to describe such a structure is to use its implementation: the table might be represented by an array of strings, a linked list, a file, etc. Such a method is clearly not acceptable for our purpose since, as discussed above, it relies on properties of the structure which are far too superficial and change-sensitive. The physical representation does not capture the true nature of the table; by looking at two different representations (e.g. array and linked list), one may get the impression that they correspond to very different structures, whereas they serve the same purpose, namely representing the table. On the other hand, one of them, e.g. the array, can also be used for representing very different things (an array of strings could be the internal representation used by, say, a text editor for the part of a text file which is accessible at any point in time).

What constitutes the true "nature" of the table structure is in fact its intended usage, that is, essentially, the store and retrieve operations. It is these operations which make it possible to distinguish an array (say) used for representing a table from an array used for something else, and, conversely, provide the commonality between various representations of the same abstract structure.

By combining a modularize-around-data approach with a description of data structures based on operations, we can (as announced at the end of 6.5.5) restore the process_data balance.

There remains, however, the problem of how to describe the operations which characterize a structure. We could use a description of the algorithms themselves, e.g. the procedures for "store" and "retrieve" in the table example. This approach has taken in the systems implementation language BLISS [17], where data structures are described by their access mechanisms, for instance, an $[n, n]$ triangular matrix is characterized by the fact that the $[i, j]$ element lies at relative address $i\frac{(i-1)}{2} + j$. Such a solution, however, is again not acceptable for out purpose since, as when describing a structure by its actual memory layout, it relies on physical, superficial characteristics of the algorithms. In our example, there are many possible different algorithms for table management (e.g. sequential storage and retrieval, hash-coding, binary tree or B-tree search and insertion, etc.). The procedures "store" and "retrieve" will be externally very different depending on the particular policy chosen.

What we should do is to characterize the available operations not by their implementations but their **abstract properties;** for example, "store" and "retrieve" are characterized by the fact that, roughly speaking, a string will be retrievable if and only if it has been previously stored into the table. This fundamental property transcends all particular representations.

It is useful here to be a little more formal. We will define classes of data structures having the same general properties as **abstract data types.** An abstract data type T is characterized by:

- a list of *functions* $f_1, f_2, \cdots f_n$, which correspond to operations available on objects of the class;
- a list of **predicates** (or assertions) relative to $f_1, f_2, \cdots f_n$, which give the abstract properties of the operations.

Every f_i has zero or more input domains, and one or more output domains:

$$f_i : D_{i_1} \times D_{i_2} \ldots\ldots \times D_{i_p} \longrightarrow D_{i_{p+1}} \times D_{i_{p+2}} \times \ldots\ldots \times D_{i_q}$$

$(p \geq 0,\ q > p)$, where at least one of the D_{i_j} $(1 \leq j \leq q)$ is T (the type being defined). The following cases arise:

- If T appears only in the right-hand side (that is, T is D_{i_j} for one or more j such that $p + 1 \leq j \leq q$), then f_i corresponds to an operation which produces objects of type T, possibly using objects of other types (D_{i_j} for $1 \leq j \leq p$); it is thus termed a **creation function.** If the left-hand-side is empty ($p = 0$), then f_i has no argument and corresponds to a constant function.
- T appears only on the left-hand side, then f_i takes one or more objects of type T objects: it is thus called an **access function.**
- Finally, if T appears on both sides, then f_i, among other things, takes objects of type T and yields other objects of type T: it is a **modification function.**

In our example case, we may describe the abstract data type *table* as in figure 6.13.

type *TABLE:*

FUNCTIONS

creation

create-empty: → *TABLE*
(a constant function which yields an (empty) table)

modification

insert: *STRING* × *TABLE* → *TABLE*
(models the "store" operation. Note that, functionally speaking, an insertion yields a new object of type *TABLE)*

access

present: *STRING* × *TABLE* → *BOOLEAN*
(should yield **true** if and only if the string is in the table)

ASSERTIONS

for all *s, s':* *STRING; t: TABLE:*

not *present (s, create-empty)*
(as its name suggests, an empty table has nothing present in it)

present (s', insert(s, t)) = ((s' = s) **or** *present (s', t))*
(when inserting a string into a table, the strings present in the new table are the one just inserted and those already present in the old table)

Figure 6.13 - Specification of string tables

This description captures the essential properties of the table structure in a formal way, without implying any particular implementation choice.

An **implementation** of an abstract data type T defined as above consists in a physical representation for objects of type T, and a set of procedures $p_1, p_2, \cdots p_n$, such that each p_i has the same functionality as f_i and the assertions relative to the f_i functions are satisfied between the input and output arguments of the respective p_i procedures. In our example, we need to choose a representation for table (e.g. array, etc.) and procedures corresponding to *create-empty, insert* and *present.*

What we can do with these elements with respect to modularity should now be clear: in a modularize-around-data approach, they should not be scattered, but be grouped together into a single module, which will constitute the **data unit** we are looking for.

An essential requirement of such a data unit is that it should be usable only through the official procedures (*create-empty, insert* and *present* in our example), **not** by the internal representation which will have been chosen. Only by observing this rule shall we be able to obey the principle of information hiding and allow for continuity (if the implementation is changed, only the module implementing the abstract data type must be updated: other modules use it through the official interface consisting of the public procedures, which, although changed internally, will still be called in the same way).

6.7.2. - Data type refinement

An extremely important requirement with regard to the criteria of composability and decomposability is the ability to describe objects as refinements, extensions or particularizations of others. This applies to data as well as to procedures.

Such an approach may be applied in a way to abstract data types. We will use a "refine" relation: B is said to refine A if B possesses all the functions of A, enjoying the same properties. B may have other functions and/or other assertions.

A simple example may be found in the case treated above. So far, there is no obligation on any *TABLE* implementation to do anything when a request is made for insertion of an already present item; an implementation may ignore the request or do something else which does not change the further behavior of the operation *present*. We now consider a new structure, *COUNTED_TABLE*, for which it should be possible to know how many times a given item has been inserted.

Rather than redefining *COUNTED_TABLE* from scratch, we consider it to be a refinement of *TABLE*, with a new function *count*, and new assertions. The definition is given in figure 6.14. Since *COUNTED_TABLE* refines *TABLE*, everything which was expressed in figure 6.13 applies to *COUNTED_TABLE*; only the new features have been added. Note that the last assertion expresses a property of the *TABLE* function present which is only true of *COUNTED_TABLE*.

type *COUNTED_TABLE* **refines** *TABLE*

FUNCTIONS

access

count: STRING × *COUNTED_TABLE* → *INTEGER*

ASSERTIONS

for all *s, s': STRING, T: COUNTED_TABLE*

count (s, create-empty) = 0 ;
count (s', insert (s, t)) =
if *s' = s* **then** *count (s, t) + 1*
else *count (s, t) ;*
present (s, t) = count (s, t) > 0)

Figure 6.14 - Specification of a counted table

6.7.3. - The Simula class

The Simula 67 language embodies a concept called "class", a very interesting programming language structure for the implementation of abstract data types.

A class is characterized by the following elements:

- a name;
- zero or more arguments (similar to procedure argument);

- two kinds of **attributes:** variables (or arrays), and procedures;
- a (possibly compound) statement.

The general syntactic structure of a class with name C is the following:

class *C (argument list); argument types ;*
begin
declarations of variables and arrays ;
declarations of procedures ;
statement ;
end

To apply a class for the implementation of an abstract data type, we will use the variables and arrays for the physical representation of objects of the type and the procedures for the operations.

The statement represents actions to be taken when creating an object of the class. An object of the class, say x, is declared by writing:

ref *(C) x*

and is actually "created" at execution time by the statement:

x:- **new** *C*

(:- is used instead of := for assignment to class objects). The *new* construct allocates the storage space which is necessary for the attributes of x and executes the statement of the corresponding class definition.

Once x has been created in this way, it behaves as an object of the abstract data type associated with C. It possesses attributes written $x.a$, where a is any of the variable or procedure names appearing in the definition of C.

For example, we could implement our *TABLE* type in a straight forward (and inefficient) way by using an array of strings, managed sequentially, as in figure 6.15.

A particular table *t1* will then be declared by

ref *(TABLE) t1 ;*

To create it, one must execute

t1 :- **new** *TABLE*

t1 can then be used through the appropriate procedures, e.g.:

t1.insert ("KEY1") ;
t1.insert ("KEYWO") ;
if *t1.present ("KEY2")* then...

```
class TABLE(n); integer n ;
begin
        comment variables ;
            text array strings (1:n) :
            integer top ;
        comment procedures ;
            boolean procedure present(t); text t ;
            begin integer i; boolean isthere ;
                    isthere := false; i := 1 ;
                    while i ≤ top do
                        begin
                            isthere := (strings(i) = t) ;
                            i := i + 1
                        end search ;
                    present := isthere
            end present ;
            procedure insert(t); text t ;
                    if not present(t) then
                        begin
                            if top = n then error
                            else
                            begin
                            top := top + 1
                            strings (top) := t
                            end actual insertion
                        end insert ;
        comment statement ;
            top := 0
end TABLE
```

Figure 6.15 - *TABLE* implementation in Simula

6.7.4. - Class prefixing and virtual procedures

Class B will inherit all the properties of a class A if its declaration is prefixed by the name of A:

```
A class B (...); ... ;
begin
        attributes which objects of class B
        have on top of those of class A;
        statement
end B
```

B is called a subclass of A.

A facility which is needed in connection with prefixing is **virtual procedures.** If a function f in the specification of A may only be refined at the B level, then in the corresponding Simula classes the name of the procedure corresponding to f should appear in A, but the procedure implementation may only be given in B. The necessity for such a feature is particularly clear if we consider a class A with several subclasses B, C, D, each of which may have further subclasses (figure 6.16).

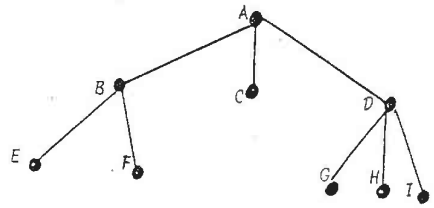


Figure 6.16 - A class hierarchy

An operation may be defined at the A level, its implementation depending on whether and A object is a B, a C or a D (or even on whether it is an E, F, etc). We may, for instance have the following relations in a graphics system:

```
                    class figure; ...
          figure class plane-figure; ... ;
   plane-figure class triangle; ... ;
   plane-figure class circle; ... ;
   plane-figure class rectangle; ... ;
       rectangle class square; ... ;
```

Some operations, such as rotation, will have to be defined at the *figure* level; their implementation can only be given if we know what kind of figure we have. such procedures will appear in a special **virtual** paragraph in the parent class; their presence there means that subclasses are required to provide a concrete implementation of the virtual procedures.

The example of figure 6.18[1], corresponding to the **refines** relation seen above between *TABLE* and *COUNTED_TABLE*, should by now be sufficiently clear. Figure 6.17 gives the structure; + indicates the "actual" procedures, * the virtual ones.

[1] We trust the reader will pardon us for the conflicting uses of the word "figure" here.

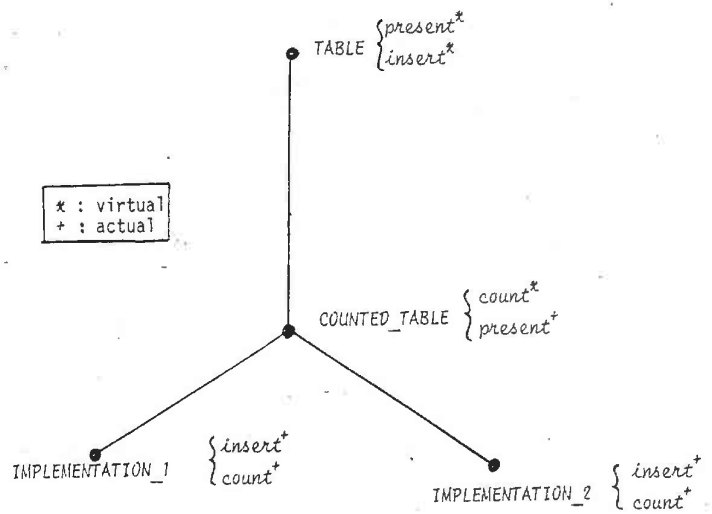


Figure 6.17 - Hierarchical system of the "table" classes


```
class TABLE ;
        virtual:
                boolean     procedure present ;
                            procedure insert ;
begin end TABLE;            comment no actual attribute or statement at this level ;

TABLE class COUNTED_TABLE ;
        virtual:
                integer     procedure count ;
begin
                boolean     procedure present (t); text t ;
                            present := (count (t) > 0)
end COUNTED_TABLE ;

COUNTED_TABLE class IMPLEMENTATION1 ;
begin
        ...     physical representation of the counted table ...
        integer procedure count (t); text t ;
                begin ... actual implementation of count .... end count ;
        procedure insert (t); text t ;
                begin ... actual implementation of insert ... end insert ;
        .....
end IMPLEMENTATION1
```

Figure 6.18 - Hierarchy of tables

Several remarks are in order here.

- 1. The "modification" procedure *insert* does not recreate a new table, but modifies the existing one.
- 2. All the procedures have an implicit argument of type *TABLE*: the class describes what operations may be applied to any table, so the corresponding argument need not be indicated within the class. The following function of the abstract data type specification:

  present: *STRING* × *TABLE* ⟶ *BOOLEAN*

  is thus represented by a procedure with only one argument:

  boolean procedure *present (t);* **text** *t*

  Indeed, the other argument appears in the calls through the dot notation; i.e. the function application

  present (s, t1)

  becomes in Simula

  t1. present (s)
- 3. As could be expected, we had to introduce a limitation on the possibilities offered by the abstract specification: our "table", implemented with an array, may only contain up to *n* strings, where *n* is the argument of the class.
- 4. There is no procedure corresponding to the creation function *create_empty*: creation is ensured by the **new** construct.
- 5. There is no way in current standard Simula to distinguish between "public" attributes (the procedures *insert* and *present*) and internal, "private" ones (*top* and *strings*). So any calling module may, for example, execute:

  *t1.top := 7 * t1.top*

  which is, of course, contrary to the principle of information hiding. There is no way to enforce this principle in Simula other than programming discipline. A simple language extension, allowing the

module implementor to specify what is public and what is private (the so-called "hidden-protected feature") has been designed but is not provided by common Simula compilers.

Of course, on any leaf of a tree such as that of figure 6.17, there should not remain any virtual procedure which does not have a matching actual procedure on the path from the root of the tree; such a procedure would not be executable.

This prefixing mechanism is extremely elegant and powerful. It may be compared to the "variant record" mechanism of Pascal and Ada, with the important difference that the former is **closed** (once a set of variants has been designed, no further extension of the record type is possible), whereas in Simula any class can always be extended by being used to prefix a new class.

One may point out two limitations of this mechanism:

- 1. The specification of virtual procedures includes the type of their results for function procedures (e.g. present, count above), but not the type of their arguments, which will only be given in the subclass which actualizes each procedure. This absence is motivated by the fact that such an argument type may depend on the entities defined only in a subclass, and thus not textually available in the parent class; it is, however, rather unpleasant for readability.
- 2. A class may have at most one prefix, which restricts type networks to trees. It is as though the mechanism allowed to define vector spaces and topological spaces, but not to combine them into the concept of "topological vector space". Technically, however, there exist ways to "cheat" with this limitation.

6.7.5. - An example

We now present a slightly more ambitious example of the use of Simula classes as a module built around the implementation of an abstract data type. This example does not use the concept of prefixing. The example of figure 6.20 is a class implementing the concept of complex number. It corresponds to the abstract data type definition of figure 6.19 (where some complex operations have been circled, e.g. +, =, etc., to avoid confusion with their real equivalents).

The basic idea behind the implementation is that, among the two main representations for complex numbers (cartesian and polar), the proper one should be used for each operation. The benefit of this approach will be appreciated by trying to derive the formula for complex addition in polar representation. On the other hand, the necessary conversions are performed only when necessary; outside the class, if a proper discipline is followed (i.e. if the class objects are only accessed through the official procedures), which representation is available at any given instant is irrelevant.

We have chosen to implement the operations (plus, minus, etc.) as modifying an element rather than computing a new one; that is, the call

z1 plus (z)

will assign to *z1* the value of *z1 + z*. We could alternatively create a new element; procedure *plus* would then become:

```
ref (COMPLEX) procedure plus (z); ref (COMPLEX) z ;
begin ref (COMPLEX) C ;
        C :- new COMPLEX ;
        C.cartesian-assign (x + z.x, y + z.y) ;
        plus :- C
end plus
```

and similarly for the other procedures. this is closer to the abstract data type specification but more prolific in storage.

The Simula class of figure 6.20 assumes the existence of a boolean hardware-dependent boolean procedure *may_divide (y, z)*, which returns *true* if and only if the division of *y* by *z* may be carried out (i.e. $\frac{|x|}{|y|}$ is not too small relative to $\frac{|x|}{|y|}$), and of a variable *pi* initialized with an approximation of π (Simula has no symbolic constants).

This example clearly shows one of the not so happy consequences of the "modularize around one type" policy: since a Simula class describes what may be done to any class instance, there is always an implicit parameter, here of type *COMPLEX*. This means that procedures like *plus, minus*, etc., which operate on **two** elements of the class, must be dissymmetric; *plus* will be called by:

z1.plus (z2)

or *z2.plus (z1)*

If we want to be able to resort to the more reassuring form

plus (z1, z2)

then we have to declare *plus* outside the class *COMPLEX*, which of course is not compatible with modularity.

The Ada approach has "packages" which are poorer in structure than the Simula class, but do not raise the above problem (see below 6.7.6).

type *COMPLEX*

FUNCTIONS

creation

$0: \rightarrow COMPLEX$

$cart, pol: REAL \times REAL \rightarrow COMPLEX$

$real, imaginary: REAL \rightarrow COMPLEX$

access

$x, y, \rho, \Theta: COMPLEX \rightarrow REAL$

$\neq, =: COMPLEX \times COMPLEX \rightarrow REAL$

$distance: COMPLEX \times COMPLEX \rightarrow REAL$

modification

$+, -, *, /: COMPLEX \times COMPLEX \rightarrow REAL$

$**, \sqrt{}: COMPLEX \times INTEGER \rightarrow COMPLEX$

$\bar{}: COMPLEX \rightarrow REAL$ -- *conjugate*

PROPERTIES

for all z, z': *COMPLEX*, a, b, r, t: *REAL*:

$0 = cart\ (0,0) = pol(0,t)$;

$x\ (cart\ (a,b)) = a$; $y\ (cart\ (a,b)) = b$;

$\rho\ (pol\ (r,t)) = r$, $\Theta\ (pol\ (r,t)) = t$ **mod** 2Π ;

$real\ (a) = cart\ (a,\ 0) = pol\ (|a|,$ **if** $a \geq 0$ *then* 0 **else** $\Pi)$;

$imaginary\ (b) = cart\ (0,b) = pol\ ($

$\rho(z) = \sqrt{x(z)^2 + y(z)^2}$; $x(z) \neq 0 \Rightarrow \Theta(z) = arctg\frac{y(z)}{x(z)}$;

$x(z) = \rho(z) * cos\ (\Theta(z))$; $y(z) = \rho(z) * sin(\Theta(z))$;

$(z = z') = (x(z) = x(z')$ **and** $y(z) = y(z'))$

$= (\rho(z) = \rho(z')$ **and** $\Theta(z) = \Theta(z')$ **mod** $2\Pi)$

$z \neq z' =$ **not** $(z = z')$

$distance\ (z,\ z') = \rho(z - z')$;

$z + z' = cart\ (x(z) + x(z'), y(z) + y(z'))$;

$z - z' = cart\ (x(z) - x(z'), y(z) - y(z'))$;

$z * z' = pol\ (\rho(z) * \rho(z'), \Theta(z) + \Theta(z'))$;

$z' \neq 0 \Rightarrow z/z' = pol(\rho(z)/\rho(z'), \Theta(z) - \Theta(z'))$;

$z**n = pol(\rho(z)^n, n \times \Theta(z))$;

$\sqrt[n]{z} = pol\ (\sqrt[n]{\rho(z)}, \frac{\Theta(z)}{n})$

$\bar{z} = cart\ (x(z), -y(z)) = pol(\rho(z), -\Theta(z))$

Figure 6.19 - Specification of the type *COMPLEX*

```
class COMPLEX ;
begin
    comment Representation invariants:
        (cartesian ⇒   zrep and yrep have the values of the cartesian
                       coordinates of the current complex,
                       as resulting from previous operations)
        and (polar ⇒   rorep and thetarep have the values of the polar
                       coordinates of the current complex,
                       as resulting from previous operations,
                       rorep ≥ 0, 0 ≤ thetarep *2* pi) ;
    comment Internal (hidden) variables ;
        real zrep, yrep, rorep, thetarep ;
        boolean cartesian, polar ;
    comment Internal (hidden) procedures ;
        procedure make-cartesian ;
            comment make cartesian form available ;
            if not cartesian then
                if not polar then error ("uninitialized complex") else
                    begin
                        zrep := rorep * cos (thetarep) ;
                        yrep := rorep * sin (thetarep) ;
                        cartesian := true
                    end cartesian ;
        procedure make-polar ;
            comment make polar form available ;
            if not polar then
                if not cartesian then error ("uninitialized complex") else
                    begin
                        rorep := sqrt (z**2 + y**2) ;
                        if may_divide (y,z) then theta := arctg (y/z)
                        else theta := sign(y) * (pi/2)
                        polar := true
                    end make-polar ;
    comment The procedures which follow are public ;
    comment Initialization procedures ;
        procedure cartesian_assign (a,b); real a,b ;
            begin
                zrep := a; yrep := b ;
                cartesian := true; polar := false
            end cartesian-assign ;
        procedure polar_assign (r, t); real r, t ;
            begin
                rorep := r, thetarep := modulo (t, 2*pi) ;
                cartesian := false; polar := true
            end polar_assign ;
```



```
        procedure real_assign (a); real a ;
            begin
                zrep := a; rorep := abs(a) ;
                yrep := 0 ;
                thetarep := if a ≥ 0 then 0 else pi ;
                cartesian := polar := true
            end real_assign ;
        procedure imaginary_assign(b); real b ;
            begin
                yrep := b; rorep := abs(b) ;
                zrep := 0 ;
                thetarep := if b ≥ 0 then pi/2 else -pi/2
                polar := cartesian := true
            end imaginary_assign ;
    comment Access procedures ;
        real procedure x ;
            begin
                make_cartesian; x := zrep
            end x ;
        real procedure y ;
            begin
                make_cartesian; y := yrep
            end y ;
        real procedure ro ;
            begin
                make_polar; ro := rorep
            end ro ;
        real procedure theta ;
            begin
                make_polar; theta := thetarep
            end theta ;
        ref (COMPLEX) procedure copy ;
            begin ref (COMPLEX) c ;
                c :- new complex ;
                if cartesian then
                    c.cartesian_assign (z,y)
                else if polar then
                    c.polar_assign (ro, theta)
                else error ("uninitialized complex") ;
                copy :- c
            end copy ;
    comment Operations ;
        procedure plus (z); ref (COMPLEX) z ;
            cartesian_assign (z + y.x, y + z.y) ;
        procedure minus (z); ref (COMPLEX) z ;
            cartesian_assign (z - y.x, y - z.y) ;
        procedure times (z); ref (COMPLEX) z ;
            polar_assign (ro*z.ro, theta + z.theta) ;
```

```
        procedure divide(z); ref (COMPLEX) z ;
            if not may_divide (ro, z.ro) then error
                    ("impossible complex division")
            else polar_assign (ro/z.ro, theta-z.theta) ;
        procedure conjugate ;
            if cartesian then cartesian_assign (-z, -y)
            else if cartesian then polar_assign (- ro, - theta) ;
        boolean procedure equal(z); ref (COMPLEX) z ;
            if cartesian then
                    equal := (z = z.z) and (y = z.y)
            else if polar then
                    equal := (ro = z.ro) and (theta = z.theta)
            else error ("uninitialized complex") ;
        boolean procedure not equal(z); ref (COMPLEX) z
;
            not equal := not equal (z) ;
        procedure power (n); integer n ;
            polar_assign (ro **n, n*theta)
        procedure power (n); integer n ;
            polar_assign (ro**n, n*theta)
        procedure root (n); integer n ;
            polar_assign (ro**(1/n), theta/n)
        real procedure distance(z); ref (COMPLEX) z ;
            begin ref (complex) C ;
                C :- copy ;
                D.minus (z) ;
                distance := C.ro
            end distance ;
end COMPLEX ;
```

Figure 6.20: A Simula class for complex numbers

6.7.6. - The Ada package facility

[To be completed. Will describe Ada packages and show that they are more general in nature than Simula classes, since they may be used to gather virtually any set of objects, but are static in nature and do not lend themselves to object-oriented design. Emphasis will be put on genericity.]

6.8. - MENU-DRIVEN INTERACTIVE PROGRAMS: A CASE STUDY IN MODULAR DESIGN

6.8.1. - Full-screen interactive applications

We shall illustrate some of the concepts introduced above by an example which is quite representative of an important class of problems, and yields an elegant modular solution based on modularizing-around-data: menu-driven interactive systems.

In such systems, any user, working at a terminal, sees at every step a certain full-screen "menu", i.e. a set of questions; the answers he provides to these questions will either trigger an error message if they are incorrect, or cause the program to take some actions (such as updating a data base) and proceed to some other step in the dialog, displaying the appropriate menu.

An execution of such a system may thus be described as a traversal of a transition graph associated with the system. Such a graph depicts the system with nodes, or **state**, and labeled vertices, or **transitions**. An example graph, an for imaginary (and simplistic) flight reservation system, is pictured on figure 6.22.

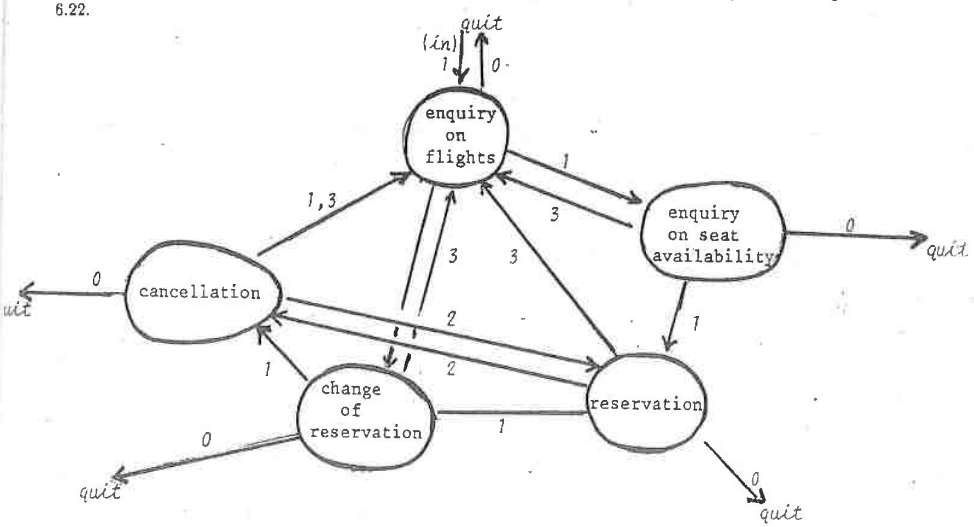


Figure 6.22 - An interactive system

We shall assume for the sequel that one of the answers provided by the user at each state is an indication of what he wants to do next, i.e. what transition should be taken; we will call such an indication an exit **label** and assume that possible exit labels are expressed by integers between *0* and some value $m-1$. Quite frequently, the exit label is determined by hitting one among m special **function keys** on the keyboard. The labels of the graph vertices on figure 6.22 correspond to such exit choices.

We should point out that such a transition graph is often quite complicated and intricate, and there is no obvious way to "structure" it (there have been some attempts, however -- see e.g. [82]). In particular:

• there is often a possibility to "quit" at any time during the dialog by pressing a special key (label *0* on figure 6.22); this facility, which is almost indispensable from the users' standpoint, breaks all nesting convention (figure 6.23(b)).

• there is often a "help" facility which suspends the current execution to display an explanation menu (from which it is sometimes possible to ask for more help) (figure 6.23(b));

• even though one may have tried to separate a system into different components, it is not unusual that users will demand a direct connection between two states far apart in the structure (figure 6.23(c)), in order to avoid the necessity of going back through the top level all the time (see the direct transition from cancellations to reservations in figure 6.23).

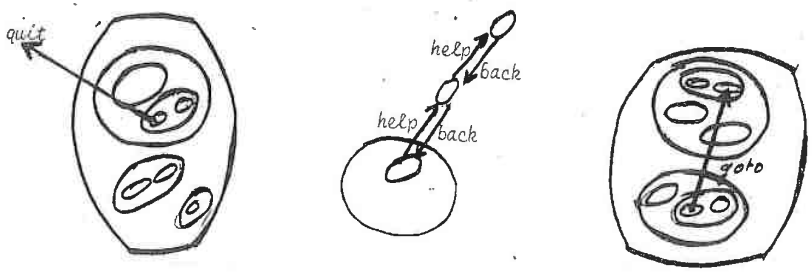


Figure 6.23 - Examples of state graph sub-structures

Our purpose in this section is to try to design a general framework for such an interactive system.

6.8.2. - A first attempt

Let us try to write a program for a schema of the above type, without any attempt at sophistication. We just forget anything we may have heard about "structured programming" in its vulgar-sense, and let ourselves be guided by our intuition. Quite naturally, we shall get a program which is made of a certain number of blocks, or paragraphs, $P_1.P_2 \cdots P_n$, each corresponding to a state in the above graph; the i-th paragraph has the form:


```
P_i (program for state i):
        output screen for state i ;
        repeat
                read user's answers and his exit choice C for the next step ;
                if error in answer then
                        output message
                end if
        until
                no error in answer
        end repeat ;
        record answer ;
        case C in
                C_0: goto P_t(i,1)
                C_1: goto P_t(i,2)
                .....
                C_m-1: goto P_t(i,m-1)
        end case ;
```

where $t(i,k)$ is the state to be entered when leaving state i with exit label k, and m is the number of possible exit labels after each step.

6.8.3. - A hierarchically procedural solution

What is bad with the above scheme? An obvious defect is that it will result in programs with an intricate branching structure, belonging to the well-known "bowl of spaghetti" type; remember that the state graph may be very complicated. Probably even more serious is the bad performance that such a program will have with respect to the "continuity" criterion: among the requirements changes which are most likely to occur during the development of an interactive system of the kind we are interested in are changes in the structure of the state diagram; e.g. users will request that it become possible to go directly from state A to state C using a certain function key, whereas it was originally planned that one should always to through state B. So the above design is unsatisfactory not only because it uses *goto* statements but, more importantly, because it has "wired" the structure of the dialog into that of the program, so that any change in the former will imply rewriting the latter.

As a result of this remark, any technique for "structuring" programs by eliminating *goto* statements and replacing them with equivalent loops is bound to have no more than a superficial effect on the quality of the program above. A similar comment may be applied to the use of exceptions (as in Ada, PL/I, CLU), which are provided by some languages especially designed for the construction of the interactive programs, such as PLAIN [14]; such constructs seem here only marginally preferable to ordinary jumps.

The limitations of any a posteriori attempt to improve the structure of the above program are further evidenced by the fact that this structure reproduces that of the underlying transition graph which, as we have seen, is in many practical cases inherently intricate.

A much better solution is to completely disconnect the description of what happens at every step, i.e. the operations performed while in a given state, from the description of the overall structure of the dialog, i.e. the traversal of the graph. In other words, we will replace the constants $t(i,k)$ above by calls an explicit function

TRANSITION (i,k)

which will be used to specify the transition diagram associated with any particular interactive application.

Using this idea, we get a simpler version of our program schema, based on nine program units on three levels of abstraction (figure 6.24): level 3 is that of a general system for executing interactive applications; level 2 is that of the individual applications; level 1 is that of the individual states in an

application.

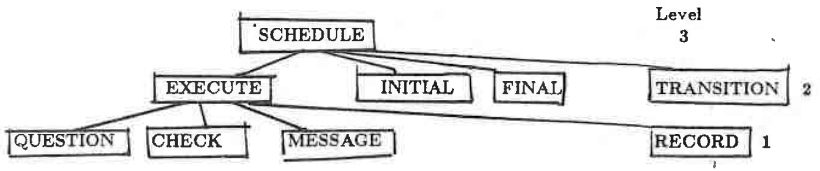


Figure 6.24 - A hierarchical procedural solution

SCHEDULE only defines the traversal of the transition graph; it knows nothing about the particular screens of a given application, and should be identical for all applications[1].

```
SCHEDULE:
        var current: STATE, next: CHOICE ;
        current := INITIAL ;
        repeat
                EXECUTE (current, next ←) ;
                current := TRANSITION (current, next)
        until
                FINAL (current)
        end repeat
```

Any application is described by a *TRANSITION* function associated with its graph, by a particular state called *INITIAL* which is the first state to be entered when starting the execution, and by a predicate on states, *FINAL*, indicating whether execution should cease in a given state. Procedure *EXECUTE* is refined below; it has two arguments, a state *current* and a choice *next*; *next* is a result computed by the procedure (this property is denoted by the mark ← in the notation above), indicating the choice made by the user for the next step, i.e. the exit label.

Note that from a practical point of view *TRANSITION* may be implemented either as a function subprogram or by a two-dimensional array, the latter technique leads to a "table-driven" program which will be more easily adaptable and is strongly recommended here. It is widely used in some application areas such as business data processing
("decision tables"), compiler writing, real-time programming.

EXECUTE does what is required in a given state: ask the right question, check the answer, perform the necessary actions and return the exit label *C*

[1] An exclamation mark following and actual procedures argument, as in *next* ←, serves in our notation as a reminder that this argument is of out mode, i.e. computed by the procedure.


```
EXECUTE (in s: STATE, out c: CHOICE):
        var a: ANSWER, correct: BOOLEAN ;
begin
        repeat
                QUESTION (s, a ←)) ;
                correct := CHECK (s, a) ;
                if not correct then
                        MESSAGE (s, a)
        until correct ;
        end repeat ;
        RECORD (s, a) ;
        c := exit_label (a)
end EXECUTE
```

The specification of the yet unwritten procedures is the following. *QUESTION* (*s*, *a*←) is a procedure which outputs the question associated with state *s* and reads the user's answer which it returns in *a*; *CHECK* (*s*, *a*) returns **true** if and only if *a* is a correct answer for the question asked in state *s*; if so, *RECORD* (*s*, *a*) processes answer *a*; if not, *MESSAGE* (*s*, *a*) outputs the relevant error message. An answer *a* is assumed to include the exit label, written *exit_label* (*a*).

6.9. - GOING MODULAR: THE LAW OF INVERSION

Is this new solution satisfactory? At first sight it may seem so; but if we look at it more carefully from the standpoint of modularity, we will soon discover that it is in fact unacceptable.

The problem here is (among others) continuity, and the keyword is data transmission. Let us recapitulate the functionalities of the procedures and function which are application-specific:

```
            EXECUTE (in s: STATE; out c: CHOICE)
            QUESTION (in s: STATE; out a: ANSWER)
function    CHECK (in s: STATE; a: ANSWER): BOOLEAN
            MESSAGE (in s: STATE, a: ANSWER)
            RECORD (in s: STATE, a: ANSWER)
```

All these procedures have a state as in argument. This means that they must all discriminate on this argument, i.e. have the general form

```
case s of
    State1: .....,
    State2: .....,
    ..............,
    Staten: .....
end case
```

This will make them long and complicated (note in particular that all but *EXECUTE* have a second-level discrimination on *a*), but it is not the worst: much more annoying with respect to modularity is the fact that all these procedures and functions will know about one complex interactive application. Thus, if we change one transition, or add a state (e.g. a "help"), we must change all the procedures involved. This is not acceptable in view of continuity and information hiding. Altogether, there is far too much data transmission in this program: variable *current* (alias *s*) is passed from *SCHEDULE* (level 3) to all procedures and functions on level 2 and on to those on level 3. This is contrary to everything we have seen.

The situation is in fact even worse than one might think, since there is another argument to **all** procedures, which has remained implicit so far, namely the particular interactive application which we are implementing (so that *MESSAGE* for example, has a three-level discrimination structure: on application, state and answer). If we are thinking of "cataloging" procedures *SCHEDULE*, *EXECUTE*, *QUESTION*, etc., as general-purpose tools in a library, then they should all know about all interactive applications which use them, and all the states of every such application! This of course would be impossible to implement and we must look for another modular structure.

The fact that the state (*current*, *s*) lies at the basis of the problem should alert us: maybe we have decomposed along the wrong line and we should try the "modularize-around-data" path. We may apply here a "law" of modularity, which we call the "**law of inversion**"(1):

> If you pass too much data in your procedures,
> then put your procedures into your data.

Here an obvious candidate for a "data unit" based on an abstract data type is the state. We will thus introduce a class *STATE* (from now on we use the Simula notation, which is of sufficiently high level

(1) This use of the term "inversion" is not the same as in the expression "program inversion" introduced by Jackson in [Jackson 75].

to describe both the abstract data types which we need and their implementation).

Among the attributes of a state are:

- the four procedures of level 1 in figure 6.24, which in the general class *STATE* can only remain virtual;
- the procedure *EXECUTE*, as seen above, but without the state parameter.

We assume that the user's answer is described by an object of class *ANSWER*, which has the integer "exit-label" as one of its attributes. We thus get the class of figure 6.25.

```
class STATE ;
        virtual:    procedure QUESTION ;
                    boolean procedure CHECK ;
                    procedure MESSAGE ;
                    procedure RECORD ;

begin
        procedure EXECUTE(c); integer c; name c ;
        begin
                ref (ANSWER) a; boolean correct ;
                correct := false ;
                while not correct do
                    begin
                        QUESTION (a) ;
                        correct := CHECK (a) ;
                        if not correct then
                        MESSAGE (a)
                    end checking ;
                RECORD (a) ;
                c := a.exit-choice
        end EXECUTE
end STATE
```

Figure 6.25 - Making the state a class

To describe a particular state of a particular application, we must refine this class by giving the particular realization of the above procedures, e.g.:

```
STATE   class ENQUIRY_ON_FLIGHTS ;

begin
        procedure QUESTION (a); ref (ANSWER) a;      begin ... end QUESTION ;
        boolean procedure CHECK (a); ref (ANSWER) a; begin ... end CHECK ;
        procedure MESSAGE (a); ref (ANSWER) a;       begin ... end MESSAGE ;
        procedure RECORD (a); ref (ANSWER) a;        begin ... end RECORD
end
```

Proceeding to the next level, we now have all the elements to describe any complete interactive system, which we will call an "application". Rather than writing a main program, we will again introduce an abstract data type, represented by a class, which will allow us to solve the problem of having a single system for describing, building and executing many applications.

An application is described by the remaining elements at levels 2 and 3 of figure 6.24: the *TRANSITION* function which describes the state graph, the *INITIAL* state, the predicate allowing to determine whether or not a state is *FINAL*, and the *SCHEDULE* procedure.

To simplify matters and avoid leaving major choices to the designer of each application, we take the following implementation decisions:

- the *TRANSITION* function will be represented by an array, with n rows and m columns;
- n (the number of states) and m (the number of possible exit labels) will be the parameters of the class;
- since *TRANSITION* is represented by an array, every application will have to number its states from 1 to n; we need an array *ASSOCIATED_STATE* to find the state associated with a given number (but **not** the reverse: class *STATE* does not have a "state number" attribute: this would bind a state to a particular application, which we do not want);
- to avoid a virtual boolean procedure *FINAL*, we take the general convention that a transition to state number 0 denotes system termination, "normal" states being numbered 1 to n. We could also systematically take a certain value, e.g. 1, as the number of the initial state, but this would be too constraining for system evolution and we prefer to leave *INITIAL_NUMBER*, the number of the initial state, as an explicit attribute.

We thus get the class of figure 6.26.

```
class INTERACTIVE_APPLICATION (n, m); integer n, m ;
begin
        ref (STATE) array TRANSITION (1:n, 0:m-1) ;
        ref (STATE) array ASSOCIATED_STATE (1:n) ;
        integer INITIAL_NUMBER ;
        procedure SCHEDULE ;
                begin integer current_number; comment 0 ≤ current_number ≤ n ;
                    current_number := INITIAL_NUMBER ;
                    while current_number /= 0 do
                        begin ref (STATE) current ;
                            integer next; comment 0 ≤ next ≤ m-1 ;
                            current := ASSOCIATED STATE (current_number) ;
                            current.EXECUTE (next) ;
                            current_number := TRANSITION (current_number, next)
                        end Loop
                end SCHEDULE
end INTERACTIVE_APPLICATION
```

Figure 6.26 - Application as a class

In this framework, building an application consists in first constructing its various states as refinements of the *STATE* class; note that they may be designed independently of each other, or taken from previous applications. Then the application itself is constructed as an object of class *INTERACTIVE_APPLICATION*; this is done simply by assigning a number to each state (i.e. filling the array *ASSOCIATED_STATE*), constructing the state graph (by filling *TRANSITION*) and choosing the initial state.

During system evolution, it will then be quite easy to add a new transition, add a new state, delete a state and the associated transitions, change the actions performed in a given state, etc.

As a final remark, note that we have written *SCHEDULE* not as "the" body of *INTERACTIVE_APPLICATION* but as just one procedure of the class. In fact, we can imagine situations where one wants to do something else with an application than executing it; if we plan to build a serious systems for the development of interactive menu-driven applications, we will soon discover many other facilities which have to be associated with an application, for instance:

- procedures to build and modify an application: add a state, add a transition, delete a state, etc.;
- procedures to simulate an application, e.g. in batch mode, or in interactive but line-oriented mode on a terminal which does not provide full-screen facilities;
- procedures to store a complete application, in an appropriately coded form, into a file (or more generally into a data base of interactive applications), and to retrieve it.

All this can be done in the above framework by adding new procedures to the class *INTERACTIVE_APPLICATION* in a progressive way, without having to destroy the structure or change existing uses of the class. The system we have sketched is thus very **open** to evolution. This is so because we have concentrated on the **objects** of our problem (state, transition graph, application), not on the apparent "purpose" of the system, which would have closed it with respect to evolution:

Real systems have no top.

6.10. - THE PROCESS APPROACH AND COROUTINES

After this review of one of the most interesting approaches to modularity, object-oriented design, we turn to another useful direction: the process concept. We will concentrate on one of its variants, coroutines, which provides and answer to one of the main objections to subprograms: their lack of autonomy.

Coroutines are the transposition to sequential programming of the concept of **process** as it is known in the programming of applications involving parallelism, e.g. operating systems or real-time programs. A process which manages, say, a printer, has much more conceptual autonomy than a classical subroutine; it communicates with other processes only through well-defined parts and functions to a large extent as an independent entity which is entirely responsible for what happens to the printer but knows very little, for instance, about the way the CPU, the files or the terminals are managed. The reason for this situation is not that real-time programmers take special courses in modular design, but that the inherent difficulty of parallel programming is such that one may simply not indulge in careless design and uncontrolled interprocess communication if one hopes to be able to construct any system at all.

A coroutine is best understood as a process; the only difference between coroutines and parallel process is that the scheduling of the former will be sequential. In both cases, we have an autonomous entity, which will go through a "birth" and "life" of its own, terminated in some cases by a "death"; many processes or coroutines, however, are conceptually infinite. Our printer manager, for example, could have the following rough structure:

```
initialization ;
while true do
        get a file to be printed ;
        print it
end while
```

Instead of being "called" by a master program, a coroutine will usually be **activated** by another coroutine; this means that the coroutine's execution starts again, not from scratch as in the case of a subprogram (which always resumes execution at its textual beginning), but at the point where its execution stopped last. This execution will be suspended (but not abandoned) whenever the coroutine itself activates another one.

Note that technically this implies that a coroutine must retain the value of all its internal elements (i.e. variables, parameters, execution counter, etc.) when its execution is suspended. This is the exact opposite of the notion of "pure procedure" which is often pinpointed as the idealization of the subprogram concept, but is quite in line with the keyword of section 6.4.4: **persistence.**

With the activation mechanism, one is able to construct a system as a set of coroutines which are on a par with each other, rather than under a master-slave relationship. This mechanism may, of course, be combined with more hierarchical ones.

Again, we will use Simula as a notational vehicle here; it was the first important general purpose language to include a coroutine facility. Coroutines in Simula are represented by class instances; activation is represented by the *resume* statement:

resume *coroutine*

The reader may have noted that the activation mechanism as described above must be complemented with a special mechanism for **starting** a coroutine: upon initialization, one may not "activate" something which does not yet exist. In Simula, this is performed via the **new** instruction which creates a new class instance. This, however, is not enough: after it has been initialized, a coroutine must not yet begin its activity proper (it should only do so when it gets activated by a **resume** statement); it should first relinquish control to whoever started it. This is done via a special parameterless instruction:

detach

Thus, a system built as a set of cooperating coroutines will have the following form:


```
begin
    class C1; begin ... end C1; ref (C1) coroutine1 ;
    class C2; begin ... end C2; ref (C2) coroutine2 ;
    .....
    class Cn; begin ... end Cn; ref (Cn) coroutinen ;

    coroutine1 :- new C1;
    coroutine2 :- new C2 ;
    .....;
    coroutinen :- new Cn ;

    resume Ci
end System of coroutines
```

Note that since in Simula a class is not an object, but a pattern for a set of objects, each coroutine is represented by two things: a class C_j and an object *coroutine*$_j$ of type **ref** (C_j).

Each of the classes will have the following general form:

```
class Cj; comment pattern for coroutinej ;
begin
    initialization part ;
    detach ;
    comment below, the "active life" of the coroutine ;
    { ... actions ...;
      resume coroutinej1
      ... actions ...;
      resume coroutinej2;
      ... actions ...;
      .....
    }
end
```

Often, the coroutine's "active life", indicated by a large bracket above, will have the form of a loop:

```
while cond do
        begin
                actions ;
                resume somebody
        end Loop
```

cond will sometimes be the constant **true**: there may be potentially infinite coroutines in a program which itself terminates; coroutines will simply stop being activated at a certain time.

We now study an example of modular design using these concepts. This example is a variant of a problem studied by Jackson in [Jackson 78].

Jackson's problem is as follows. A well-known proof in modern mathematics was devised by Cantor to show that the set of rationals is enumerable. Cantor used an explicit enumeration of rationals, illustrated in figure 6.27.

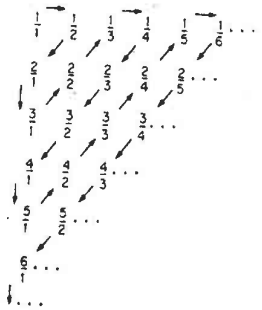


Figure 6.27 - Cantor's table

The problem is to write in the order of this enumeration the rational numbers appearing on the first 100 diagonals:
1/1 1/2 2/1 3/1 2/2 ··· 1/99 1/100 2/99 3/98 ··· 99/2 100/1

Jackson uses this problem as an example application of his method. The "data" is indeed nicely modeled by a Jackson diagram (figure 6.28): the part of the table to be printed may be described as a sequence of 50 *zigzags* where every *zigzag* is an up *zig* followed by a down *zag*. Zigs and zags are sequences of rationals, which are easy to compute because for all rationals a/b on a given zig or zag the value of $a+b$ is the same.

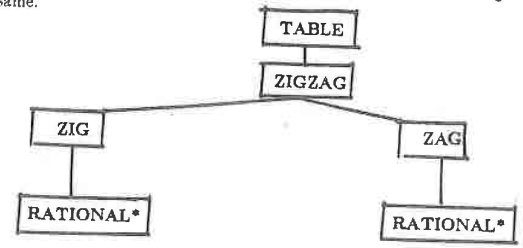


Figure 6.28 - A model of Cantor's table of rationals

The solution is thus:

ENUMERATION:

for *i: 1...100 (2)* **do** *ZIGZAG(i)*

with

ZIGZAG (k):

ZIG (k) ;
ZAG (k+1)

and

ZIG (m):

$a := m$; $b := 1$;
while $a > 0$ **do**
print_rational (a,b) ;
$a := a-1$; $b := b+1$
end while

ZAG (m):

$a := 1$; $b := m$;
while $b > 0$ **do**
print_rational (a,b) ;
$a := a+1$; $b := b-1$
end while

This is a very nicely structured program (figure 6.29). Consider now variants of the problem where we ask to print the first 100,000 elements of the enumeration, or all the elements up to the first are a/b such that $a^5 + b^7 \geq 10^{45}$, or any other criterion. The above solution does not generalize easily: this is because the actual termination may occur either in a zig or in a zag, and we have no easy way of telling in advance which.

What we are looking for is a properly structured solution (avoiding *GOTO*s to a common code for termination), with no inelegant duplication of code; we are also preoccupied with the continuity criterion, so we would like the solutions to the variants of the problem (including the initial one) to differ only slightly, since they are the same except for the termination condition.

ENUMERATION

ZIGZAG

ZIG *ZAG*

Figure 6.29 - Program structure for Jackson's solution

To meet this requirement, we will replace the previous hierarchical structure by a coroutine structure (figure 6.30). Arrows correspond to *resume* activations between instances of the classes shown; containment stands for blocks embedding; the names of class instances are in parenthesis.

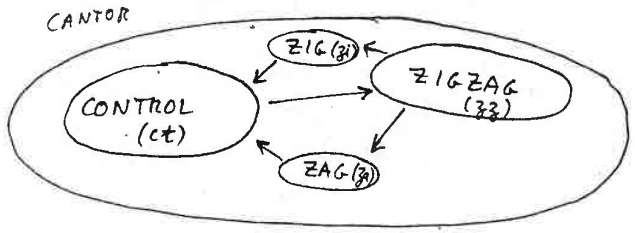


Figure 6.30 - Program structure for a coroutine solution

For convenience, we use a class to describe rationals:

```
class rational ;
begin
    integer numerator, denominator ;
    procedure print ;
        begin ... statements for printing the rational
                  numerator/denominator ...
        end print ;
    procedure assign (a,b); integer a, b ;
        begin numerator := a; denominator := b end assign
end rational
```

The main program is given in figure 6.31.


```
begin
    class zigzag; begin ... see below ... end; ref (zigzag) zz ;
    class zig (m); integer m; begin ... see below ... end zig ;
    class zag (m); integer m; begin ... see below ... end zag ;
    class control; begin ... see below ... end ;
    ref (zigzag) zz; ref (control) ct ;
    ref (rational) current; current :- new rational ;
    zz :- new zigzag; ct :- new control ;
    resume ct
end enumeration
```

Figure 6.31 - Main program for coroutine solution

Class *control* (more precisely, its instance *ct*) is in charge of controlling the continuation of the whole process (figure 6.32).

```
class control ;
begin integer i ;
    detach ;
    for i := 1 step 1 until 100 000 do
        begin
        resume zz ;
        current.print
        end active life of control
end control ;
```

Figure 6.32 - Class *Control*

Note that for other variants of the program it suffices to replace the *for* loop by something else (e.g. the Simula equivalent of **repeat ... until** *(current.numerator)*^{5} + **(current.denominator)**$^5 \geq 10^{15}$).

The rest of the solution is independent of the stopping criterion. Traversal of the Cantor table is performed by instance *zz* of class *zigzag* (figure 6.33).

```
class zigzag ;
begin
        ref (zig); ref (zag) za ;
        integer i ;
        detach; i := 0 ;
        while true do
            begin
                comment up one diagonal:    ;
                i := i+1; zi := new zig (i) ;
                while not zi.over do resume zi ;
                comment down one diagonal:
                i := i+1; za := new zag (i) ;
                while not za.over do resume za
            end (infinite) active life of zigzag
        end zigzag ;
```

Figure 6.33 - class *zigzag*

The boolean variable *over* will be present in classes *zig* and *zag*, and will be set to **true** in these classes when and only when the corresponding diagonal traversal has been completed. Note that this variable is superfluous: we know that *zi* and *za* must be **resumed** exactly *i* times each so that the **while** loops could be transformed into **for** loops. The principle of information hiding tells us, however, that it is better not to let *zigzag* rely explicitly on this property.

An up diagonal will be traversed by repeated resumptions of instance *zi* of class *zag* (figure 6.34).

```
class zig (m); integer m ;
begin integer a, b; boolean over ;
        a := m; b := 1; over := false ;
        detach ;
        while a > 0 do
                begin
                current.assign (a,b) ;
                resume ct ;
                a := a-1; b := b+1
                end active life of zig ;
        over := true
end zig
```

Figure 6.34 - *zig*

Finally *zag* is the exact symmetric of *zig* (figure 6.35).


```
class zag (m); integer m ;
begin integer a, b; boolean over ;
        a := 1; b := m; over := false ;
        detach ;
        while b > 0 do
                begin
                        current.assign (a,b) ;
                        resume ct ;
                        a := a+1; b := b-1
                end life of zag ;
        over := true
end zag
```

Figure 6.35 - *zag*

This solution deserves a few comments.

First one might argue that classes *zig* and *zag* should be declared within *zigzag*, giving the structure of figure 6.36.

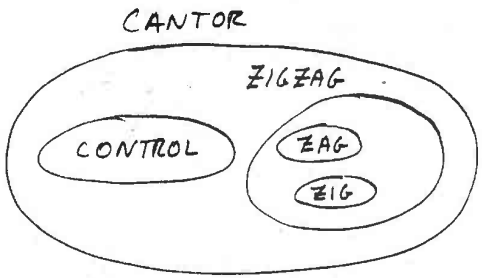


Figure 6.36 - A variant of the previous structure

The kinds of relationships which Simula allows us to combine may be both "egalitarian" (**resume**) and hierarchic (static: block embedding and class prefixing, dynamic: procedure call); this gives much (too much?) freedom to the designer.

Note also an instance of a well-known deficiency of Simula with respect to information hiding: in classes *zig* and *zag*, a public variable, *over*, is on the same footing as *a* and *b* which are only locally needed. The need for the "hidden/protected" feature is clear here.

We also notice here what is probably the most serious problem with the Simula coroutine facility: the lack of provision for transmission of information. It is well-known in parallel programming that two kinds of interaction occur between processes: **synchronization** and **communication.** There exist mechanisms (e.g. semaphores) for synchronizing two processes, i.e. making sure they reach compatible states at a certain time, and others (e.g. parameter transmission in procedure calls) for making them exchange information. Any complete mechanism should, however, cater to both needs.

Here, it is clear that the **resume** mechanism only provides for the sequential equivalent of synchronization. Communication has to occur by other means: in our example, information is transmitted through the global variable *current*. This method is, of course, unsatisfactory with respect to our modularity criteria and principles. What would be needed would be for the **resume** statement to be able to send information from the activating to the activated process, as one does with parameters in procedure calls. The reason why this is not possible in a simple way, however, is that whereas a call statement has a syntactic match in the program (the subprogram heading with the list of formal arguments), there cannot be any such thing in **resume** statements; indeed, by the very definition of this mechanism, execution may continue at any statement in the resumed coroutine, so that there is no simple way to devise a language construct to express how information could be transmitted upon coroutine activation.

A partial solution to this problem may be found in a descendant of Simula, Smalltalk, whose classes communicate by structured messages: More complete answers are given by Hewitt's ACTOR mechanism, also based on message passing, and by Hoare's notation for Communicating Sequential Processes, known as CSP, which will be studied in the next section.

6.11. - ABSTRACT DATA TYPES WITH A SCENARIO

This section [to be completed] will show how the ideas of modules built around a data structure, on the one hand, and of modules associated with processes, on the other hand, may be combined by considering an object with its associated history or, equivalently, a process together with the object(s) it is responsible for.

A simple example will be that of a table whose life includes two phases: in the first one, both insertions and searches are allowed; after a certain signal has been sent (triggering perhaps an internal reorganization of the physical representation), only searches will be permitted (figure 6.37).

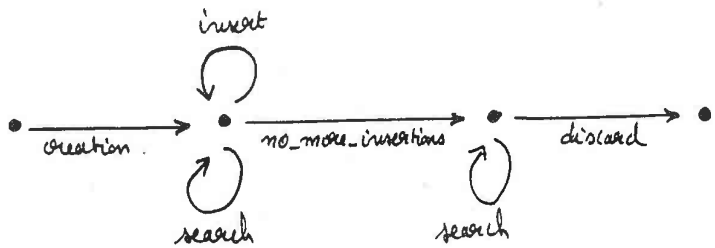


Figure 6.37 - THE LIFECYCLE OF A TABLE MANAGEMENT SYSTEM

The use of Actors & CSP to model such uses will be demonstrated.

6.12. - THE STRUCTURE OF INTERMODULE RELATIONS

6.12.1. - Possible relations

This chapter has explored several kinds of module structure.

From the systems viewpoint, it is important to note that the main issue in designing for modularity may not be what the various modules are, but how they are interconnected. In the course of this chapter, we have encountered several kinds of intermodular relations; it is interesting as a conclusion to recap them and try to study them more systematically.

Figure 6.39 gives some of the most useful intermodule connection mechanisms. As one goes down the picture, one encounters first more abstract, conceptual relations, then more concrete, technical ones, as found in programming languages. Every relation, described by an arrow from left to right (where $A \longrightarrow B$ means A is connected to B by relation r) is accompanied by the inverse relation, labeling a right-to-left arrow.

A **models** (or **specifies**, or **abstracts**, or **describes**) B, and B **instances** (or **implements**, or **realizes**) A if A contains a description of what B does, or viewed the other way around, B is one way to do what is prescribed by A. Conceptually, then, A gives the same information as B, or less, but is at a higher level of abstraction. For example, the user's manual for a machine models this machine; or, the abstract data type *TABLE* (fig. 6.13) is a model implemented by the Simula class *TABLE* (figure 6.15). Similarly, a Simula class C is used to model all instances of this class (objects of type **ref** (C)).

A **complements** B (the relation is symmetric) if A and B cooperate toward the realization of some higher-order aim. For example, workers of the same grade in a company "complement" each other; the same is true of procedures *present* and *insert* in the class *TABLE* (fig. 6.13), of the various subprograms in a library or of composable Unix programs. In fact, this relationship is closely connected to the composability criterion.

Note that the **models** and **complements** relation corresponds to two well-known figures of rhetorics: metaphor (denoting a concept by a corresponding concept at another level of abstraction), and metonimy (using a neighboring concept).

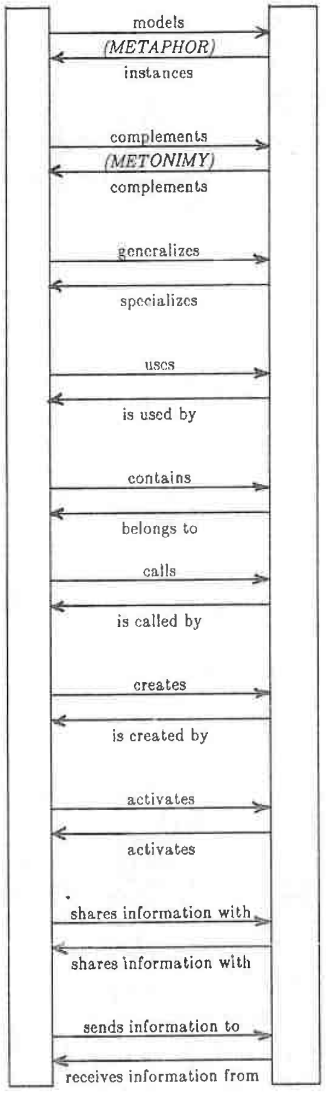


Figure 6.39 : Intermodule relations

A **generalizes** (or **extends**) B, and B **specializes** A, if anything which is described by B is also described by A (but some things may be described by A which are not described by B). The **generalizes/specializes** relation lies at the basis of the Linnaean classification of plants and is, in fact, an important component of the scientific method. The prefixing mechanism of Simula (*A* **class** ..., *A* **begin** *B* **end**) is an elegant programming language implementation of the same concept which, unfortunately, was not carried through to later languages embodying the concept of abstract data type implementation, with the notable exception of Smalltalk. A different application of this relation is provided by the concept of generic instantiation in Ada.

A **uses** B, and B **is used by** A, if A refers to B by its name. This occurs in Algol-like languages through the rules of block structure, extended in Pascal and Simula by special facilities which make it possible to "peep" into the names of entities local to an object (*with* in Pascal, *inspect* in Simula). It is only in Ada, however, that this concept takes its full meaning with respect to intermodular communication, thanks to the *use* clause, which allows explicit appropriation of a set of external names.

The **contains/belongs to** relation corresponds to textual inclusion in block-structured languages.

The **calls/is called** by relation is the standard boss-to-subordinate, officer-to-private etc. relation; in common programming languages, it is represented by the caller-subprogram relation.

The **creates/is created** by relation exists in languages which provide for dynamically allocated objects (Pascal, Algol 68, Simula, etc.).

The symmetric relation **activates** corresponds to the coroutine or process activation mechanism (**resume** in Simula).

The relation **shares information with**, also symmetric, corresponds to data sharing by any mechanism, e.g. COMMON (Fortran) or block structure (Algol, Simula, etc.).

The relation **sends information to/receives information from** is included in the **calls/is called** by relation for common programming languages; not so in CSP or Ada, for example, where it also has elements in common with the **activates** relation.

6.12.2. - Properties of the relations

The various relations are of course not unconnected to each other. In fact, a system such as a programming or specification language is characterized in part by the set-theoretical properties which it assigns to these relations; for example, the last remark could be written more formally as

- (1) *sends information to* $\cup$ *receives information from* $\subset$ *calls* $\cup$ *is called by*
  (in Algol, Pascal, Simula, etc.)
- (2) *sends information to* $\cup$ *receives information from* $\subset$ *calls* $\cup$ *is called by* $\cup$ *activates*
  (in CSP or Ada)

The following properties are also of interest. Operator $\circ$ will denote relational composition; for any relation r, we note r^{-1} the inverse relation ($r \circ r^{-1} = r^{-1} \circ r = identity$) and r^* the transitive reflexive closure of r($r^* = identity \cup r \cup r^2 \cup r^3 \cup \cdots$, where $r^n = r \circ r \circ r \circ \ldots$, n times). We use the relation *samescope* defined as

$$\begin{aligned} samescope &= contains \circ contains^{-1} \\ &= contains \circ belongs\ to \end{aligned}$$

- (3) *uses* $\subset$ belongs to * $\cup$ samescope
  (in Algol 60)
- (4) *uses* $\subset$ *belongs to* * $\cup$ *samescope* $\cup$ *contains* $\circ$ *users*
  (in Ada)
- (5) *calls* $\cup$ *creates* $\cup$ *activates* $\subset$ *uses*
  (in all common languages).
- (6) *shares information with* $\subset$ *belongs to* * $\cup$ *samescope*
  (in Algol 60, Pascal, Simula, PL/I, Ada, etc., excluding external files).

Etc. It would be worthwhile to explore further this "relational algebra" for important programming languages.[1]

6.12.3. - Intermodule relations in Fortran

6.12.4. - Intermodule relations in Pascal

6.12.5. - Intermodule relations in Simula 67

Figure 6.40 gives a representation of the allowable relational structure of modular facilities in Simula 67. Only the "direct" sense of the relation (from left to right on figure 6.39) is pictures. Because of properties (2) and (6) above, relations "sends information to" and "shares information with" have not been drawn explicitly.

Connections on this graph indicate possible relationships: $A \xrightarrow[r,r']{} B$ means "an A may be connected by r and r' to a B". For instance one sees from the graph that a procedure may create and activate a coroutine.

The richness of this graph, which one may interpret as reflecting the power of the language or its complexity, is in itself interesting.

6.12.6. - Intermodule relations in Ada

[1] See: *The Software Knowledge Base*, B. Meyer, UCSB technical report, January 1985.

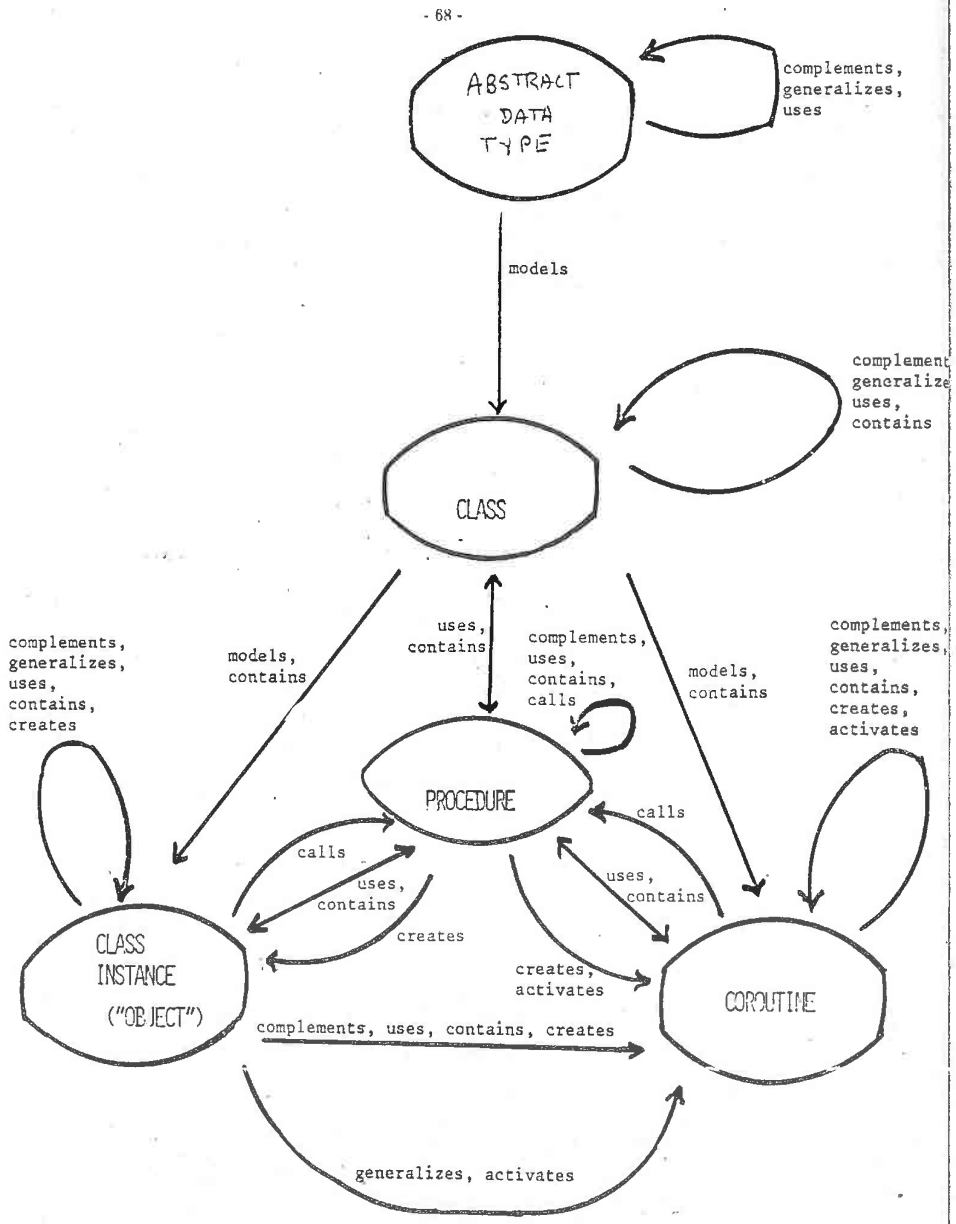


Figure 6.40: Possible intermodule relations in Simula.

INCREMENTAL STRING MATCHING

Bertrand Meyer[1]

Computer Science Department
University of California
Santa Barbara, California 93106

Telephone: (805) 961 4321
uucp: ...!ucbvax!ucsbcsl!bpm
csnet: bpm@ucsb

ABSTRACT

The problem studied in this paper is to search a given text for occurrences of certain strings, in the particular case where the set of strings may change as the search proceeds.

A well-known algorithm by Aho and Corasick applies to the simpler case when the set of strings is known beforehand and does not change. This algorithm builds a transition diagram (finite automaton) from the strings, and uses it as a guide to traverse the text. The search can then be done in linear time.

We show how this algorithm can be modified to allow incremental diagram construction, so that new keywords may be entered at any time during the search. The incremental algorithm presented essentially retains the time and space complexities of the non-incremental one.

To appear in Information Processing Letters.

KEYWORDS

String matching, word processing, bibliographic search, indexing, program correctness, analysis of algorithms.

---

[1] On leave from Electricité de France, 1 avenue du Général de Gaulle, 92141 Clamart France

1. INTRODUCTION

The problem of searching a text for all occurrences of one or more strings (hereafter called *search strings*) has been well researched. Several algorithms have been published [3, 1, 2] .

This paper considers a variant of the problem which, to our knowledge, has not been addressed by previous publications: the case when the set of search strings may change as the search proceeds. In fact, in the practical application that led to this work, the search strings are found in the text itself as it is being searched.

In the next section, we describe that application, an interactive book indexing program. In section 3, we give the algorithm by Aho and Corasick which serves as a basis for our solution. Section 4 shows why this algorithm does not readily apply to the incremental case. A solution is proposed in section 5; its correctness is proved in section 6 and its efficiency analyzed in section 7.

2. AN INDEX PROGRAM

We first present the concrete occasion for which we developed the algorithm below. Of course, there may be other applications of incremental string searching.

The occasion is a program for making book indexes. An index is a sorted list of all the "interesting" words which appear in a text, each word being accompanied by a sorted list of the pages where it occurs.

As anyone who has tried knows, preparing indexes is a tedious and error-prone task, and it is natural to look for computerized aids. Much of the work (searching for interesting words in the text, sorting the lists) can indeed be done automatically; but one critical step requires human intervention: deciding which words are "interesting" and which are not. We call the person who will make this decision the *indexer*; the best indexer is usually the author.

The best place to look for "interesting" words is of course the text itself, which we assume to be available as a computer file. Our indexing program thus has a phase called the **collector** which presents the indexer with the text and asks him to select words for indexing.

The collector is an interactive program. It displays the text one screen at a time; each word appearing on a screen belongs to one of the following three categories:

- "rejected" words, which the indexer has already designated as not interesting;
- "retained" words, which the indexer has designated as interesting (these words appear underlined on the screen as the collector is being executed)
- "undecided" words, whose fate has not yet been sealed (these appear highlighted).

Thus whenever a screenfull of text is displayed, the indexer must choose to either reject or retain each "undecided" word on the screen. This decision process is the essential object of the collector.

From the program point of view, then, what the collector must do is to search each successive portion of text for occurrences of words belonging to the union of the "rejected" and "retained" sets. Both these sets change as new words are being classified by the indexer: thus the string searching method must allow for incremental construction of the set of search strings.

3. A NON-INCREMENTAL ALGORITHM

A very efficient algorithm by Aho and Corasick [1] applies to the case when there is more than one search string. The principle of this algorithm is that one first builds a "transition diagram" from the set of search strings, and then traverses the text using this diagram as a guide. Thus in the standard algorithm all search strings must be known at the outset.

Since our algorithm is based on a modification of Aho and Corasick's, we shall first present the key aspects of theirs. Our presentation is slightly different from the one in their original paper; it is close to the one we gave in [4] .

3.1. Data Structures

Aho and Corasick's algorithm uses three data structures as internal representation of a search string set: a tree T, called "goto function" in [1] (it is in fact a trie); an "output table" O; and a "failure function" F. Together, these three structures constitute what may be termed the "transition diagram" associated with the search string set. We describe them in turn.

3.1.1. The Tree

The branches of the tree T are labeled by characters. Tree T is associated in a natural way with the set of search strings: for example, the search string set $\{A, CAN, AN\}$ may yield the tree of Fig. 1.

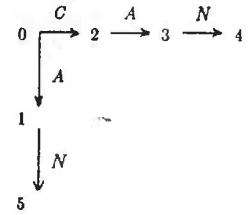


Figure 1: A String Matching Tree

An important property of this tree is that each node has an associated character string. For example, in the above tree, node 0 is associated with the empty string, node 3 with string CA, etc. From now on, **we will not make the distinction between a node and the associated string**; for example, we say that string AN is a suffix of node 4 (that is, of the associated string CAN), or that string CAT does not appear in the above tree (that is, no node of the tree is associated with this string).

If CC is the character set and the nodes are numbered from 0 to N, i.e. we define

type $NODE = 0..N$;

then the tree may be represented as a two-dimensional array

T : **array** $[NODE, CC]$ **of** $NODE$;

where the child of node n through branch labeled c is $T[n, c]$. By convention, the root is numbered 0 and $T[n, c]$ is 0 if there is no branch leading from node n with label c. We will not distinguish between the tree and the associated array; note that in practice the array will usually be sparse, requiring a suitable implementation (hashed, linked etc.).

3.1.2. The Output Table

For any node n, the output set of n, written $O[n]$, is the set of suffixes of n which are search strings (we shall carefully distinguish between the **suffixes** of a string, which include the string itself, and its **proper suffixes**, which do not). On Fig. 2 below, corresponding to the search string set $\{A, CAN, AN\}$, the output sets (some of which are empty) have been written next to the corresponding nodes.

3.1.3. The Failure Function

The failure function F is defined on all nodes except the root. For such a node n, $F[n]$ is the longest proper suffix of n that appears in tree T. $F[n]$ may be node 0, the root (i.e. the empty string). It is important to note that the inverse of F is a tree spanning T, with node 0 as its root.

The dashed lines on Fig. 2 represent the failure function for the search string set $\{A, CAN, AN\}$.

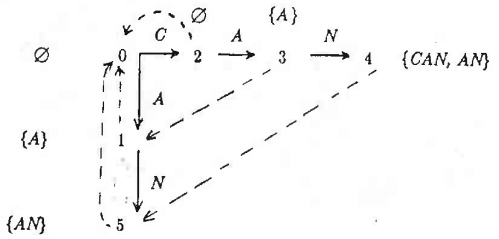


Figure 2: A Completed Transition Diagram

3.2. The String Searching Algorithm

Assuming a transition diagram consisting of the above three data structures has been constructed, the string searching algorithm is a simple traversal of the text, guided by the transition diagram:

```
procedure Recognize (text : in STRING)
    -- Search text for strings represented by T, O and F.
    n : NODE ;
begin
    n := 0 ;    -- Start at the root
    for c in text loop    -- Examine next character
        while n ≠ 0 and T [n, c] = 0 loop    -- Find worthy successor node
            n := F [n]
        end while ;
        n := T [n, c] ;
        Report all strings in O [n] as occurring at this point of the text
    end for
end procedure Recognize
```

This algorithm clearly makes one T transition per character of the text. It is proved in [1] that the number of F transitions is at most l, the length of the text. Thus the time complexity of the searching algorithm is O (l).

3.3. Constructing the Transition Diagram

To execute the above algorithm, the three data structures T, O and F must have been built from the search string set S. This is done in two steps:

Build_tree ; Build_failure

The first step builds T and initializes O; the second builds F and completes O.

We describe these two steps below. We assume that the data structures T, O and F are global to all the procedures given and have been properly dimensioned (an upper bound for N is *lsum*, the sum of the lengths of the search strings) and initialized (all values of T and F to zero, all values of O to the empty set).

3.4. Constructing the Tree

The first step may be done as follows:

```
procedure Build_tree ;
    last := 0 ;  -- Number of the last entered node
    for s in S loop  -- enter search string s into tree
        Enter_in_tree (s);
    end for
end procedure Build_tree
```

where *Enter_in_tree* (local to *Build_tree*) is given below.

```
procedure Enter_in_tree (s : in STRING) ;
    -- Enter new search string s into tree T.
    n, n' : NODE ;
begin
    n := 0 ;    -- Start at root
    for c in s loop  -- Examine next character
        n' := T [n, c] ;
        if n' = 0 then  -- Create new node
            last := last+1 ; n' := last ;
            Enter_child (n, n', c)
        end if ;
        n := n'
    end for ;
    Enter_output (n, s)
end procedure Enter_in_tree
```

The auxiliary procedures *Enter_child* and *Enter_output* each consist of a simple assignment; the only reason for pulling them out of the body of *Enter_in_tree* is to ease adaptation to the "incremental" case later.

```
procedure Enter_child (n, n' : in NODE ; c : in CHARACTER) ;
    -- Add to the tree a branch from n to n' labeled c.
begin
    T [n, c] := n'
end procedure Enter_child ;

procedure Enter_output (n : in NODE ; s : in STRING) ;
    -- Define the output set of n as consisting of the sole string s.
begin
    O [n] := {s}
end procedure Enter_output
```

3.5. Constructing the Failure Function

For any node n other than the root, let *lps (n)* be the longest proper suffix of n which appears in the tree. To complete the transition diagram, procedure *Build_failure* must set *F [n]* to *lps (n)* for every non-root node n, and complete *O [n]* accordingly.

To do this, the *Build_failure* algorithm uses a loop that considers all nodes of T in order of increasing length of the associated strings (i.e. first the root, then its children, then their children etc.).

```
procedure Build_failure ;

    Precondition : T is a tree associated with the given string set
    Postcondition : For all nodes i ≠ 0, F [i] = lps (i)

    - - Build the failure function F corresponding to T.
begin
    for n in NODE in order of increasing length loop  - - Compute F for the children of n
        - - loop invariant
            - - For any child i of a node previously considered, F [i] = lps (i)
        - - end invariant
        for c in CC such that T [n, c] ≠ 0 loop
            Complete_failure (n, c) ;     - - Compute F [T [n, c]]
        end for
    end for
end procedure Build_failure
```

with *Complete_failure* defined as:

```
procedure Complete_failure (n : in NODE ; c : in CHARACTER) ;
    - - Precondition : For any i ≠ 0 which is either n or a shorter node, F [i] = lps (i)
    - - Compute F [T [n, c]].
    n', m, m' : NODE ;
begin
    n' := T [n, c] ; m := n ;
    repeat
    m := F [m]
        - - loop invariant
            - - m is a proper suffix of n,
            - - and for any proper suffix p of n longer than m in the tree, T [p, c] = 0
        - - end invariant
        until m = 0 or T [m, c] ≠ 0
    end repeat ;
    m' := T [m, c] ;
    F [n'] := m' ;
    O [n'] := O [n'] ∪ O [m']
end procedure Complete_failure
```


(Note that the invariant of a **repeat...until** loop is written at the end of the loop body since it may not be satisfied until after the first iteration).

The correctness of *Build_failure* will follow from the loop invariant of that procedure since any node but the root is a child of another.

To show that this invariant is indeed preserved by the loop body, assume that $n \neq 0$ and that *F (i) = lps (i)* for all nodes i considered before n. We have to show that for any child n' of n, say $n' = T[n, c]$, execution of *Complete_failure (n, c)* results in *F [n'] = lps (n')*.

Using the notation xy for the concatenation of x and y (where x is a string and y is a string or a single character), we have $n' = nc$. Thus the longest proper suffix of n' in the tree is either $m' = mc$, where m is the longest proper suffix of n in the tree such that $T[m, c]$ is not 0, or 0 if there is no such m.

Now the list of all proper suffixes of n which appear in the tree is precisely, in order of decreasing length, the list of nodes examined successively by *Complete_failure*, namely *F [n], F [F [n]], F [F [F [[n]]]*, etc. This is a direct consequence of the inductive assumption: since n is not the root, n is the child of a previously considered node; thus *F[n]* is the longest proper suffix of n in T.

It is essential for this correctness argument that the set of nodes be explored in order of increasing length. This can be ensured in three different ways:

- The above *Build_tree* algorithm may be modified so that the number assigned to any node is smaller than the number assigned to nodes on the following level in the tree. To do this, *Build_tree* should consider successive character positions in all search strings rather than successive search strings (that is, the order of the embedded loops in *Build_tree* should be reversed). Then the loop in *Build_failure* will just consider nodes in order from 0 to N. This is the solution presented in [4] .
- Another solution is to add a topological sort step between *Build_tree* and *Build_failure* which will re-number the nodes according to the rule stated above.
- The solution given in [1] uses a FIFO queue of nodes in *Build_failure* to make sure that nodes are processed in order of increasing levels, without imposing a special node numbering.

3.6. Efficiency

It is shown in [1] that construction of the complete transition diagram, as given above, takes no more than O(*lsum*) time and space, where *lsum* is the sum of the lengths of the search strings.

4. THE PROBLEM WITH INCREMENTAL CONSTRUCTION

Let us now assume that instead of being all known beforehand, the search strings become available as the search proceeds.

There is no particular problem with the construction of the tree; we can execute *Enter_in_tree (s)* as each new string comes along. The real difficulty is associated with the failure function (and with the associated "completion" of the output table).

From a practical point of view, it should be noted that the failure function is only useful when some search strings may be proper suffixes of others. Referring to the book indexing application mentioned in section 2, this will not occur if all index entries correspond to **words**, always enclosed in delimiters in the text. If such is the case, the *Build_tree* procedure as given above is sufficient, and one may apply Aho and Corasick's method without a failure function. In many cases, however, one needs to have **phrases** as well as single words in index entries, so that a search string may be the proper suffix of another; for example an index to the present paper might include entries for both strings *suffix* and *proper suffix*. If such is the case, one must find a way to build the F function incrementally, as new search strings are entered into T.

Now when a new node $n' = T[n, c]$ is entered, one must compute *F [n']*; this in itself raises no difficulty since the longest proper suffix of n' in the tree must be of the form $T[m, c]$ for some proper suffix m of n, and we may assume inductively as before that all proper suffixes of n are accessible through F. But this is not the whole story: when adding n' we may also have to update the failure value of **existing** nodes.

Fig. 3 illustrates the problem. The dotted lines represent the current values of the failure function. Assume we initially had the two search strings A and CAN and we add AN, corresponding to the transition represented by the double arrow. Then $F\,[4]$ must be updated to point to the new node 5.

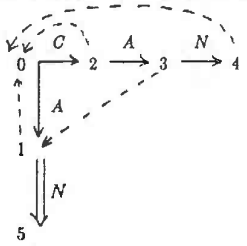


Figure 3: Node Insertion

The next section presents a solution to this problem.

5. THE INCREMENTAL ALGORITHM

When a new node $n' = T\,[n, c]$ is entered, the value of $F\,[x']$ must be changed for some existing node x' if and only if n' is the longest proper suffix of the string associated with x'. This may only be the case if $x' = T\,[x, c]$ for a node x such that $n \in F^+\,[x]$, where $^+$ denotes (non-reflexive) transitive closure.

Thus to be able to find all the nodes whose F value needs to be updated, we must keep a record IF of the inverse function of F. Note that F is single-valued but IF may be multi-valued. As indicated in section 3.1.3, IF is a tree.

If we have access to IF, then to enter a new search string we use the procedure *Enter_in_tree* as given in section 3.4. We only need to change procedures *Enter_output* and *Enter_child*. Both now have a slightly different specification and become recursive, as follows. We assume that IF, as the other data structures, is properly initialized: $IF\,[n]$ should be initially empty for all nodes n.

```
procedure Enter_output (n : in NODE ; s : in STRING) ;
    - - Add s to the output set of any node which has n as a suffix.
begin
    O [n] := O [n] ∪ {s} ;
    for x in IF [n] loop Enter_output (x, s) end for ;
end procedure Enter_output ;

procedure Enter_child (n, n' : in NODE ; c : in CHARACTER) ;
    - - Add branch from n to n' labeled c ;
    - - update failure and inverse functions to account for the insertion of n'.
begin
    T [n, c] := n' ;
    Complete_failure (n, c);        - - Compute F [n']
    IF [F [n']] := IF [F [n']] ∪ {n'} ; - - Update IF for m' = F [n']
    Complete_inverse (n, n', c)  - - Compute IF [n'] and change to n' the corresponding values of F
end procedure Enter_child
```


Procedure *Complete_failure* is as before (section 3.5). Procedure *Complete_inverse* finds all nodes which will "fail to" a new node and is defined as follows:

```
procedure Complete_inverse (y, n' : in NODE ; c : in CHARACTER) ;
    - - Recursive Precondition : n' is a suffix of yc
    - - and T [βn, c] = 0 for any proper suffix βn of y in the tree (where n' = nc)
    - - Record n' as new failure value for all x' such that yc = lps (x')
    x, x' : NODE ;
begin
    for x in IF [y] loop
            - - {y = lps (x)}
        if T [x, c] ≠ 0 then
            x' := T [x, c] ;   - - {yc is a proper suffix of x' ; thus so is n'}
            - - Remove previous failure value of x' ; install new one.
            IF [F [x']] = IF [F [x']] − {x'} ;    - - Set difference
            F [x'] := n'; IF [n'] := IF [n'] ∪ {x'} ;
        else
            - - Try recursively with nodes having x as proper suffix
            Complete_inverse (x, n', c) ;
        end if
    end for
end procedure Complete_inverse
```

6. CORRECTNESS

To prove the correctness of the above incremental algorithm, we first note that termination of the two recursive calls follows from the fact that IF is a tree, and that the changes brought to T and O when a new search string is inserted are the same ones that Aho and Corasick's algorithm would have performed. Thus we concentrate on the partial correctness of the modifications to F and IF.

It suffices to prove that an execution of *Enter_in_tree* as given above leaves the following two properties of the transition diagram invariant:

- (INV) IF is the inverse of F.
- (SUFF) For any node $x \neq 0$, $F\,[x] = lps\,(x)$ (that is, $F\,[x]$ is the longest proper suffix of x in the tree).

Property (INV) is trivially invariant since any modification to F in *Enter_in_tree* is accompanied by the corresponding modification to IF and conversely.

To show that property (SUFF) is invariant, we study in what way function *lps* changes when $n' = T\,[n, c]$ is inserted and check that F follows suit. There may be two reasons for change:

- 1- The definition of *lps* has to be extended for n' in accordance with (SUFF); the call to *Complete_failure* takes care of this case.
- 2- The value of $lps\,[y']$ for some previously existing nodes y' may now become n'. These nodes are those which have n', i.e. nc, as a proper suffix, and have no longer proper suffix in the tree.

Assuming inductively that (SUFF) was satisfied before the insertion, any such y' is of the form αnc (see Fig. 4), such that no node of the form βnc, where β is a proper suffix of α, exists in the tree.

Thus $y' = T\,[y, c]$, where $y = \alpha n$. As expressed by the "recursive precondition" to procedure *Complete_inverse*, the required y nodes are exactly those which are considered (in order of increasing length) by the successive recursive calls to that procedure, starting with $y = n$.

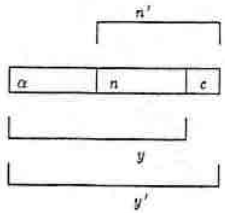


Figure 4: Strings and Suffixes

7. EFFICIENCY

The recognition algorithm (procedure *Recognize*) is not affected by the modification; nor is the performance of the part of the algorithm which builds the tree. We must consider the impact of the modification on the building of the failure and output functions.

Regarding space efficiency, since we must store *IF [n]* for every node *n* (presumably as a linked list), the space requirement for the representation of the failure function is doubled, remaining O (*lsum*).

Regarding time, we first notice that for each search string the operations performed by *Complete_failure* are a subset of those performed in the original algorithm; there may be fewer operations because some proper suffixes may not have been entered yet. Thus the total time for this procedure will be O(*lsum*). For a given search string set, the operations performed by *Enter_output* (adding elements to the output sets) are also the same in the incremental algorithm as in the original, although the former may do them in a different order and will follow inverse *F* chains.

The case of *Complete_inverse* is more delicate since this procedure may follow void chains which the direct algorithm would never have explored. When inserting $n' = T\,[n, c]$, the maximum number of nodes which may be searched in this fashion is the number of elements in the set $IF^+\,[n]$. Thus the maximum number of extra operations is $K * \sum_{n \in NODE} descendants(n)$, where *K* is the size of the character set *CC* and *descendants (n)* is the number of proper descendants of node *n* in the *IF* tree.

It is easily proved that, for any tree with *M* nodes and height *h*, $\sum_{n \in NODE} descendants(n) \leq M * h$. Here *IF* has the same nodes as *T*, thus $M \leq lsum$, and $h \leq lmax$, where *lmax* is the maximum search string length.

Thus the overall complexity of the algorithm is now O ($K * lmax * lsum$), which is identical to the original O (*lsum*) if we consider *K* and *lmax* as constants. In normal practical cases, the constant factors to apply are much smaller than the above analysis would seem to imply.

Acknowledgement: We are grateful to Man-Tak Shing and Don Brady for useful comments and for pointing out errors in a previous version of this paper.

References

1. Alfred V. Aho and Margaret J. Corasick, "Fast Pattern Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, June 1975.
2. Robert S. Boyer and J. Strother Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, October 1977.
3. Donald E. Knuth, J. H. Morris, and Vaughan R. Pratt, "Fast Pattern Matching in Strings," TR CS-74-440, Stanford University, Stanford (California), 1974.
4. Bertrand Meyer and Claude Baudoin, *Méthodes de Programmation*, Eyrolles, Paris, 1978. (new edition, 1984).

[85f]

THE SOFTWARE KNOWLEDGE BASE

Bertrand Meyer

Computer Science Department, University of California
Santa Barbara, California 93106 (USA)

(On leave from Electricité de France, Clamart, France)

ABSTRACT

We describe a system for maintaining useful information about a software project. The "software knowledge base" keeps track of software components and their properties; these properties are described through *binary relations* and the *constraints* that these relations must satisfy. The relations and constraints are entirely user-definable, although a set of predefined *libraries* of relations with associated constraints is provided for some of the most important aspects of software development (specification, design, implementation, testing, project management).

The use of the binary relational model for describing the properties of software is backed by a theoretical study of the relations and constraints which play an important role in software development.

Keywords

Software engineering tools, configuration management, project management, formal description of software engineering concepts.

This paper appears in the Proceedings of the 8th International Conference on Software Engineering, London, 28-30 August 1985.

1. INTRODUCTION

Studies have repeatedly shown that management problems are one of the primary sources of delays and failures in large software projects (see e.g. [5]).

If bad management is due to bad managers, one can hardly expect that advances in software engineering will alleviate the problem. But bad management, or rather bad organization, often has another cause: the sheer difficulty of mastering the various aspects of a project, and in particular of controlling change. Project managers and project members alike have trouble keeping track of what is going on. As the project develops, its "entropy" increases and it becomes increasingly difficult to maintain a clear picture of the state of its various components. Here good tools can play a major role.

The effort reported in this paper aims at providing a unified base of supporting tools for various aspects of software development. To this end, we introduce the notion of a **software knowledge base**, that is to say a repository of all useful project information.

The software knowledge base is used by managers and programmers to keep track of all interesting properties of the software components and their relationships. The software *components*, as defined here, include all the relevant project elements: program modules, data definitions, requirements, user manuals and other documentation, specifications, design documents, test data, test results, schedules, tasks, personnel data, budgets etc. The *relations* between these components may be of diverse kinds: we may want to record the fact that a certain module of the design implements a certain module of the specification, that a certain program module uses a certain data definition module, that a certain task is assigned to a certain person, etc.

We use the expression "software knowledge base", or SKB, to denote the compendium of information associated with a software project. The system used to record, access and manipulate this information, as described in this paper, is called the SKB *system* whenever there might be a confusion.

Several aspects of the SKB system are present in previous project management systems. The ISDOS system [34] is a set of project documentation tools, which make it possible to record project information as relations between entities of various predefined kinds; these ideas were further developed in the SREM system [3] written at TRW, which particularly emphasized the notion of traceability (i.e. ability to locate over the whole data base the consequences of a change made to some element). Simpler yet very useful tools for **configuration management** and **version control** are gaining acceptance: Make [15] and SCCS [32] on Unix, DEC's CMS, Softool's CCC, the "System Version Control" component of Gandalf [20], Adèle [14], RCS, etc. The idea of collecting all useful project documents in a single database is expounded in the Stoneman report [10], and used in a current TRW development, the "Software Master Database" [6]. The use of relations in software environments is advocated in [11], which relies on general (n-ary) relational databases; [26] shows that binary relations may be applied to various aspects of programming. The SKB project builds on all these ideas but emphasizes some original, and in our view essential, design criteria, which we shall now describe.

2. DESIGN CRITERIA

2.1. Simplicity

The SKB system should be easy to learn and use. Managers and programmers have enough to do already; they should not be required to go through an extensive training period before they can effectively use the SKB system.

A necessary condition for ease of learning and use is to base the whole system on a simple and uniform conceptual framework.

2.2. Method-, language- and system- independence

The SKB system is a set of tools, not an integrated methodology. Although its consistent use naturally leads to some sound methodological practices, it should be viewed as a way of helping project managers and developers, not as a disruption of current development practices.

Thus the SKB system should not conceptually imply the use of any particular methodology, programming language, computer or operating system. It should blend well with other software engineering tools.

We will refer to this criterion as the "independence" criterion (as a shorthand for method-, language- and system-independence).

2.3. Adaptability

Not only should the system be compatible with existing methods or languages: it should be able to provide efficient support for specific methods or languages in use in, say, a company.

Thus the natural counterpart of independence is the ability to parameterize.

2.4. Whole life-cycle coverage

The SKB system should provide benefits across the entire life-cycle of a software project. Although this criterion may restrict the power of SKB tools as applied to a specific life-cycle stage, it is essential in view of the fact that non-trivial projects usually have a long history. A system that would only apply to, say, the initial phases of specification and design, would stand little chance of playing a significant role: so much of the software process is evolution, refinement and extension of systems occurring after the first "cycle" has been completed.

2.5. Support for system semantics

Many of the systems quoted in section 1 have little, if any, notion of what the objects being manipulated really "are". Most configuration management systems, for example, focus on just one attribute of objects, their time stamp, and know of just one relation, the dependency relation (there is usually also the notion of a "permission" attribute in the systems which support protection). These systems are not equipped to deal with other properties of the objects such as the "*A is an implementation of B*" relation quoted above.

On the other hand, some of the more complex systems do know about "types" of objects (e.g. specification, test data set, etc.), but then they violate the independence criterion since these types are defined once and for all. The problem is thus to be able to record semantic properties of software objects while retaining flexibility.

2.6. Formal analysis

The design of the SKB system was based on a systematic analysis of the properties of software project elements; some elements of this analysis are given below (section 5), in the form of a review of software relations and their abstract properties (constraints).

This approach contrasts with most published work on software engineering tools: although the necessity for a systematic requirements analysis is one of

the tenets of software engineering, it seems to have seldom been applied, let alone in a formal way, to software engineering tools and environments. The formal specifications we know in this field are *a posteriori* exercises applied to existing designs, e.g. [18], which describes some aspects of ISDOS, and [12] which describes the system version control component of Gandalf. The analysis outlined in section 5 is not a complete specification of the SKB system, but provides a sound (we hope) theoretical basis for the system.

2.7. Object-independence

A software knowledge base is a **model** of a certain set of software objects and their relations. The model is conceptually and physically distinct from the objects themselves; this is in contrast with systems that essentially add project and configuration management information to the object representations (usually files on a conventional host system). In our approach, the SKB is a separate entity; objects are modeled by SKB elements, called "atoms" below.

Thus a reference to the object modeled by an SKB atom (e.g. the file containing a program or other software object) will merely be considered as one of the *attributes* of the atom (the notion of attribute is made more precise below).

Such an approach has advantages and drawbacks. The advantages are simplicity and portability; the SKB system can be built on top of any operating system without undue modification to this operating system. The approach also makes it possible to keep the model (the SKB) on one computer and the objects themselves on another if it is deemed preferable to separate the development machine from the management machine.

On the other hand, the approach taken makes it impossible to ensure consistency: one cannot prevent users from modifying the objects without making the corresponding changes in the model. However, regardless of the decision taken, it is hard to ensure consistency anyhow unless one is to build a management system that replicates most of the functions of an operating system. For example, if one wants to guarantee that the management system knows about all changes brought to the objects, then the management system should include such utilities as text editors and the like. We did not want to follow this path.

Thus we prefer to stick to the more modest goal of providing a set of management tools on top of an existing operating system, with an open architecture which makes it possible to combine these tools with other software tools. It is the responsibility of the project members to maintain an accurate SKB about the project. In other words, we accept the possibility that the SKB system may be fooled, as a price to pay for the simplicity, flexibility and independence (in the above sense) of that system.

Obviously, efforts should be made to improve the consistency of the SKBs. In particular, specific interfaces may be built between the SKB system and the host system so that information may be entered automatically into the SKB, as a result of operations performed in the host system (e.g. a compilation or an editing session).

Our approach thus follows the example set by the Make system [15], which achieves simplicity by relying on dependency information provided explicitly by programmers; this system having proved to be useful, efficient and easy to use, other researchers have been able to come up with tools [35] that automatically feed dependency information into Make for specific cases (source programs in C, Pascal, Fortran, Lex and Yacc in the reference cited).

3. THEORETICAL BASIS

The notion of software knowledge base is based on a small number of concepts: atoms, attributes, relations, constraints and actions.

3.1. Atoms

The objects in the knowledge base, associated with physical objects of the software project, are called atoms. As implied by the "object-independence" criterion discussed above, the atoms have no immediate connection with the objects they represent; they are meaningful for the SKB operations only, and their properties are only defined through their attributes, relations with other atoms, and constraints on these relations.

3.2. Attributes

Atoms may have attributes. Attributes are user-definable, although some predefined attributes are available. The value of an attribute may only belong to one of a small number of predefined types such as *Integer, String, Time, File.* The values of the last type are are references to files supported by the operating system (in a non-standard system that does not have files, we may have to replace this by a more general notion of "object").

Typical predefined atom attributes are *time_of_last_change*, yielding values of type *Time*; *atom_type*, yielding values of type *String* (some possible types for atoms are predefined, e.g. "procedure", "requirement", "test data", etc., but new ones may freely be added); and *representation*, yielding values of type *File.*

Attributes may not be of complex types; in particular, they cannot yield atoms. For anything but simple properties of atoms, relations should be used instead (see below).

3.3. Relations

The heart of an SKB consists of a series of facts about the software project, expressed as links between atoms. Each of these links expresses the fact that a certain relation holds between two atoms *a* and *b*. Examples (complementing those in the introduction) are "*a* is defined in *b*" (where *a* is a procedure and *b* a package in Ada), "*a* is a member of *b*" (where *a* is a person and *b* a project), "*a* is the formal expression of *b*" (where *a* is a module in a specification and *b* a paragraph of the requirements document). More examples will be found in section 5.

The SKB system only uses binary relations; the reason is that binary relations are mathematically simple, have nice properties, and provide an intuitively appealing way to describe structural properties of systems. From the theoretical standpoint, any system that can be described using general relations (as e.g. with a relational data base management system) can be described with binary relations [8], and algorithms have been proposed to efficiently translate a binary schema into a more efficient *n*-ary one [31].

In practice, we have indeed found binary relations to be adequate for modeling properties of software objects. This is illustrated by the analysis in section 5 - where it will be seen that we did find one case where a ternary relation seems necessary.

3.4. Constraints

Attributes and relations constitute by themselves an empty shell; they describe the structural connections between software objects (the "syntax" of the project), but not their deeper properties (the "semantics"). The latter may be expressed by defining *constraints*, or conditions on the relations and attributes, which must be satisfied for the SKB to be in a consistent state. A simple and important example of constraint

is the "dependency" constraint maintained by tools such as Make, which expresses that the value of the *time_of_last_change* attribute should be greater (i.e. more recent) for every atom than for every atom to which it is connected by any relation that may be characterized as a "dependency" relation. But many other useful constraints may be defined on software systems; some will be given below.

Constraints will be expressed as mathematical relational predicates involving relations, attributes and atoms. The abstract formalism used to construct and manipulate a software knowledge base is called the Calculus of Relations, Attributes and Constraints (CRAC).

3.5. Actions

An action is associated with a constraint and specifies steps to be taken when the constraint is violated by certain objects, following manipulations of the SKB. Actions are not strictly part of the SKB system since they may involve commands to the operating system; the SKB system provides the interface, and ways to pass attributes of the atoms (e.g. file names) to the host system.

4. STRUCTURE OF THE SYSTEM

The structure of the SKB system follows from the design criteria of section 2 and the theoretical basis described in section 3. It is represented in figure 1.

The **kernel** of the SKB system provides the basic mechanisms for creating, accessing and updating the SKB entities: atoms, attributes, relations and constraints.

In connection with constraints, we introduce the concept of **daemon.** A daemon is a mechanism associated with a constraint, which monitors the SKB in order to detect possible violations of the constraint as the information in the SKB is being updated (i.e. links between atoms are modified, new atoms are entered, attributes are changed, etc.). When it finds that such a violation has been made, the daemon will report the inconsistency and trigger the action associated with the constraint, if there is one.

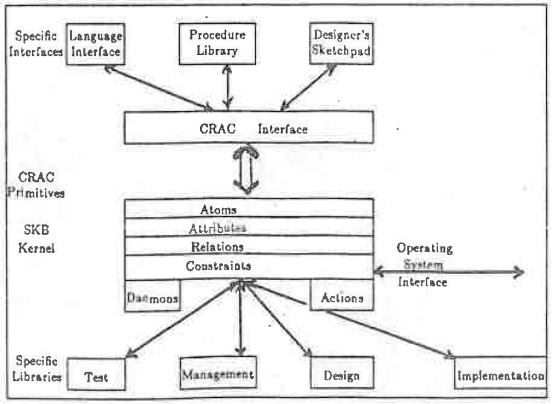


Figure 1: Structure of the SKB System

Daemons raise an interesting implementation problem: in a large SKB involving many atoms, attributes and relations, it is essential to find ways to avoid searching the whole structure (mathematically, a multigraph) for the consequences of a simple change. Work on related topics has been done previously in connection with artificial intelligence [25] and interactive graphics [17].

The SKB kernel is accessible through a set of primitives, the "CRAC primitives", which implement the calculus of relations and constraints, i.e. all the useful operations on the knowledge base. These operations are made available through a uniform interface, the "CRAC interface"; the idea here is that the SKB functions (like those of any good data base management system, or more generally of any good software engineering tool) should be equally accessible to interactive users, non-interactive users, and other programs[1].

Thus the CRAC interface does not favor any of these types of access. Several higher-level interfaces should be provided; figure 1 lists three:

- the procedure library, which makes CRAC primitives usable from programs (e.g. other software tools), written in ordinary programming languages;
- the CRAC language, which makes it possible to express CRAC manipulations in an appropriate notation;
- the **Designer's Sketchpad**, a graphical interface to the calculus, allowing for interactive description of the atoms and relations with a graphical display and a mouse. The aim here is to avoid the gap between high-level design decisions, which are often best expressed in pictures, and the rest of the software development process.

Finally, figure 1 includes a set of "CRAC Libraries", each of which provides a set of predefined relations, attributes and constraints corresponding to an important aspect of software engineering. Examples are project management (scheduling, personnel management etc.); design (a library might provide support for a specific PDL); implementation (an Ada library manager would fit here); and testing. This last point is particularly important in our view and we see test management as one of the main benefits of the SKB system: although there is an extensive literature on program testing, very little seems to have been published on the *management* of the testing process: how to keep track of test data sets for each module, record test results, etc.

[1] This is an implementation of what may be called "Strachey's principle" from the quotation of Christopher Strachey in Scott's preface to [33]: "decide what you are going to say before you decide how you are going to say it".

5. A TAXONOMY OF SOFTWARE RELATIONS AND CONSTRAINTS

5.1. Overview

The fact that useful relations exist among components of software systems has been pointed out by many authors. For example, Parnas [30] describes the "uses" and "invokes" relations among modules; systematic methodologies for software design have introduced the "abstraction" relation between a specification (e.g. an abstract data type) and an implementation [21]; the "isa" relation [9] is used in AI systems; the development of software development environments has recently led several researchers to consider using relational databases to keep track of the relations between the various objects needed in a software project [11,24]; at the program level, control and data dependencies play in important role in studies about code generation and optimization in compilers [2], program vectorization [23,22] , static analysis [16].

Despite this frequent use of relations for software-related issues, there have been very few systematic studies of these relations; most works dealing with relations just assume that they are there, and go on using them or discussing ways to compute or implement them (an exception is [26] which introduces some program-level relations and studies their properties).

The absence of a precise definition of software relations and their formal properties is regrettable, since relations are not just vague connections between objects, nor just "tables" as in simplistic presentations of the relational database theory, but useful mathematical objects with interesting properties. We feel that a systematic study of software relations is essential to advances in software configuration management. We have started such a study [28]; some elements from that study will now be reported. The aim of this section is to present some interesting relations and the associated constraints, giving support to our decision to base the design of the SKB system on binary relations.

Of course, the relations presented here are only some of the important relations that occur in software; the SKB system is an open system and the user may introduce any relations and attributes that may be needed for a particular application, together with the associated constraints. The normal way to do this is to define CRAC "libraries"; the relations and constraints presented below would normally be part of some basic, predefined libraries.

5.2. Basic Atom Types

As mentioned above, SKB atoms are not strictly typed; they simply have "atom type" as one of their attributes. Examples of atom types are "Requirement", "Specification", "Design", "Program", "Test_data", "Variable", "Statement", "Module", "Project", "Milestone", "Staff member", "Unit cost", etc. In the spirit of the theory of abstract data types, these types are only "defined" through the relations which may hold between the corresponding atoms and the associated constraints.

In the analysis that follows, we shall be talking about types of software **objects** and relations between these objects. For the SKB project, this analysis is only interesting insofar as these properties of objects can be modeled by properties of the corresponding atoms.

5.3. Relations between atoms of different types

- *a* **contains** *b*

This relation holds if and only if the object represented by *b* is a constituent of *a*. Typically, *a* will be a system, described at a certain level of abstraction (specification, design, code, documentation etc.) and *b* will be a component (module, chapter etc.) of that description. We call *part_of* the inverse relation $contains^{-1}$.

- *a* **models** *b*

This relations holds if and only if *a* includes a description of what *b* does, that is to say if *b* is one way to do what is prescribed by *a*. We call *instances* the inverse relation.

Examples: the user manual for a machine **models** that machine; an abstract data type description of a type **models** an implementation of that type as a class in Simula or Smalltalk, a package in Ada etc.

5.4. Relations between atoms of the same type

• *a* **complements** *b*

This relation holds if and only if *a* and *b* cooperate towards the achievement of some higher aim. For example, various procedures in the implementation of the same data type (class, package) complement each other; so do various subroutines in a numerical library, or Unix programs commonly used in a "pipe" fashion, e.g. for text processing the programs *refer, tbl, eqn, troff*.

Constraints: *complements* is a symmetric relation;

$$part\_of\ ;\ contains \subseteq complements$$

In this notation, the semicolon denotes the composition of relations: *part_of ; contains* is the relation which holds between any two elements *a* and *c* if and only if, for some *b*, *a* is **part_of** *b* and *b* **contains** *c*. Also, if *r* and *s* are two relations, then $r \subseteq s$ (*r* is a subset of *s*) means that any pair of elements connected by *r* is also connected by *s*. The appendix describes these and other notations.

• *a* **specializes** *b*

This relation holds if and only if anything which is described by *a* is also described by *b* (but some things may be described by *b* which are not described by *a*). The inverse relation, $specializes^{-1}$, may be written *generalizes*.

Examples: In other branches of science, the Linnaean classification of living beings is based upon this relation. In software, a particular elegant implementation of this relation is the prefixing mechanism of Simula and Smalltalk: if *a* is a class whose declaration is prefixed by the name of *b*, then any property which has been given in the declaration of *b* applies ipso facto to all objects of class *b*, but this does not prevent the declaration of *a* to add any further properties which may be needed; the mechanism can be iterated. A similar mechanism exists in the Z specification language [1].

Constraints: "linear" or "hierarchical" inheritance, as in Simula and Smalltalk, means that the relation is a forest. In Smalltalk, the introduction of the "metaclass" *Class* makes it a tree. "Multiple inheritance" would mean that a dag is acceptable.

An interesting variant of this relation occurs in many practical cases; it may be written *a specializes b except for c* (e.g., bats have all the properties of mammals except that they can fly). This seems very useful to model many aspects of software, e.g. Fortran 77 is "upward-compatible" with Fortran 66 (except for a few "minor" details), version 4.2 of the XXX operating system is almost compatible with version, say, 7, etc. This relation is also important in connection with modular, reusable system specifications [29]. It is a ternary relation.

• *a* **refers_to** *b*

This relation holds if and only if *a* refers to *b* by its name. It can happen in a variety of ways: *a* and *b* can be objects of the same type (i.e. procedures, where *a* **calls** *b*) but this is not necessary. In programming languages, a module can **refer_to** objects belonging to other modules (e.g. variables, etc.) either through the mechanism of block structure or sharing of data, or by special facilities which enable a module to "peep" into the names of entities belonging to another (**inspect** in Simula, **use** in Ada). We call *is_referred_by* the inverse relation.

• *a* **needs** *b*

This relation holds if and only if *a* cannot be understood (or, if a program element, executed) without *b*.

Constraint: we venture the following rule:

$$needs \subseteq is\_referred\_by^*;\ refers\_to^+$$

meaning that *a* possibly **needs** *b* if and only if some module *c* (which could be *a* itself) refers to both *a* and *b* directly or indirectly (the asterisk and plus sign denote transitive closures; see the appendix).

• *a* **declared_in** *b*

This relation holds in block-structured languages iff *a* is declared inside *b*.

Constraint: $declared\_in \subseteq part\_of$

• *a* **shares_information_with** *b*

This symmetric relation holds if and only if *a* and *b* may access some common information. In block-structured languages such as Algol 60 and Pascal, this is done through the block structure mechanism, as defined by the following constraint (valid for these languages):

$$shares\_information\_with \subseteq declared\_in^*\ ;\ has\_declaration^*$$

where *has_declaration* is the inverse of *declared_in*.

5.5. Relations between program modules

The following relations apply to modules of programs (procedures, classes, packages etc.).

• *a* **calls** *b*

This is the standard relation between procedures, which holds if and only if *a* may call *b*.

• *a* **creates** *b*

This relation holds if and only if *a* may create *b*. It exists in a language or systems where processes can start other processes (e.g. Ada tasks, Unix processes, PL/I tasks, Simula classes). The same relation also applies to the case where *b* is a data structure in languages where data can be allocated dynamically (e.g. **new** in Pascal).

• *a* **activates** *b*

This relation holds in systems supporting coroutines (e.g. Simula) or parallel processes (Ada) if and only if if *a* may re-start a suspended execution of *b*.

• *a* **sends_information_to** *b*

This relation holds if and only if *a* may pass information to *b*. Let *receives_information_from* be the inverse relation. The following constraint holds for common programming languages:

$$sends\_information\_to \cup receives\_information\_from \subseteq calls \cup is\_called\_by$$

However, this is not the case in CSP, for example, where information may also be passed through the "rendez-vous" mechanism; thus in these systems:

$$sends\_information\_to \cup receives\_information\_from \subseteq calls \cup is\_called\_by \cup activates$$

More Constraints

Many features of programming languages may be characterized as properties of the above relations. For example, defining

$$same\_scope = declared\_in\ ;\ has\_declaration$$

then in block-structured languages such as Algol 60:

$$refers\_to \subseteq declared\_in^* \cup same\_scope$$

but in Ada:

$$refers\_to \subseteq declared\_in^* \cup same\_scope \cup (declared\_in\ ;\ refers\_to)$$

In all common languages, we have

$$calls \cup creates \cup activates \subseteq uses$$

etc.

5.6. Time and system consistency

For the purpose of this study, only one property of the basic type Time matters: the fact that it is totally ordered by a relation which we call *before*. The inverse relation is predictably called *after*.

As mentioned in section 3.2, we define *time_of_last_change* as an attribute rather than a relation. This is

merely for convenience; mathematically, an attribute is a (possibly partial) function, thus a special case of a relation anyway. Let *changed_at* be the inverse of *time_of_last_change*.

Part of the problem of configuration management is due to the fact that no element in a system should be younger than any element which depends on it. This is expressed by the following constraint, which we may call the fundamental law of system consistency:

changed_at ; depends_on ; time_of_last_change $\subseteq$ *after*

where relation *depends_on* is defined as:

depends_on =
contains $\cup$ *instances* $\cup$ *generalizes* $\cup$ *refers_to* $\cup$ *needs*

5.7. Relations between program elements

Our last set of relations will contain relations between objects of a program. These relations play an essential role in static program analysis, whether it is for compiler optimization, supercomputer programming [7,23], or program debugging.

- ***a* follows *b***

This relation holds if and only if *a* is a statement whose execution may be immediately followed by that of statement *b*. It describes the flow of control.

- ***a* accesses *b***

This relation holds if and only if *a* is a statement or a program module, *b* is a program object (variable, etc.), and the value of *b* is needed for the execution of *a*. For example, if *a* is an assignment statement, it **accesses** the objects on the right-hand side.

- ***a* modifies *b***

This relation holds if and only if *a* is a statement or a program module, *b* is a program object (variable, etc.), and the value of *b* may be modified during the execution of *a*. For example, if *a* is an assignment statement, it **modifies** the variable on the left-hand side.

- ***a* needs_value_of *b***

This relation holds if and only if *a* and *b* are objects of a program (e.g. variables), and the value of *a* may be modified by a computation which uses the value of *b*.

The following constraint may be called the fundamental law of static analysis:

needs_value_of $\subseteq$
(modifies $^{-1}$ *; (follows ; modifies)* $\cap$ *accesses)* ; modifies* $^{-1}$ *; accesses*

To understand this constraint, it may be useful to look at figure 2, where *i* and *j* are statements, and *a, b, c* are program objects, and to note that the solution of

$d = r \cup (t ; d)$

is

$d = r \cup (t ; r) \cup (t ; t ; r) \cup (t ; t ; t ; r) \cup \ldots$

i.e

$d = t^* ; r$

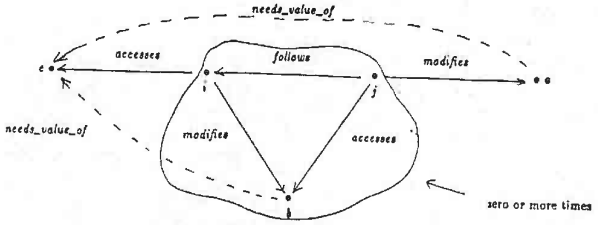


Figure 2: The Static Analysis Constraint

6. STATE OF THE SYSTEM

After the initial specification and design phase, the SKB project currently (June 1985) pursues the following tasks:

- The CRAC calculus has been defined precisely [13] and is being further refined to include diverse kinds of object manipulation and user queries.
- A prototype has been implemented in Prolog [27]; an alternative approach, using the relational data base management system Ingres, is pursued concurrently. An experimental graphical interface (the "designer's sketchpad" mentioned in section 4) is also being implemented.
- The study of useful software relations outlined in section 5 of this paper is being further refined.
- Two unrelated software projects, one at UC Santa Barbara and one in industry, have been the object of an in-depth analysis [19] with two complementary aims: to assess practitioners' needs from their current practices, and to evaluate the CRAC as a modeling tool.
- Finally, efficient multigraph algorithms for the incremental monitoring of constraints have been investigated [13].

Acknowledgments

Several UCSB students contributed useful ideas, notably Jacques Delort, Xavier Glikson and Lucio Mendes. The referee's comments were helpful. I also thank Peter Lohr for several important observations.

References

1. Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs*, ed. R. McNaughten and R.C. McKeag, Cambridge University Press, 1980.
2. Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading (Massachusets), 1979.
3. 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.
4. Dines Bjørner and Cliff B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Englewood Cliffs (New-Jersey), 1982.
5. Barry W. Boehm, "Software Engineering - As It Is," in *Proceedings of the 4th International Conference on Software Engineering, Munich (Germany)*, pp. 11-21, IEEE, September 1979.
6. Barry W. Boehm, Maria H. Penedo, E. Don Stuckle, Robert D. Williams, and Arthur B. Pyster, "A Software Development Environment for Improving Productivity," *Computer (IEEE)*, vol. 17, no. 6, pp. 30-44, June 1984.
7. Alain Bossavit and Bertrand Meyer, "The Design of Vector Programs," in *Algorithmic Languages*, ed. Jaco de Bakker and R.P. van Vliet, pp. 99-114, North-Holland Publishing Company, Amsterdam (The Netherlands), 1981.
8. G. Bracci, P Padini, and G. Pelagatti, "Binary Logical Associations in Data Modeling," in *Modeling in Data Base Management Systems, IFIP Working Conference on Modeling in DBMS's*, ed. G.M. Nijssen, 1976.
9. Ronald J. Brachman, "What IS-A and isn't: An Analysis of Taxonomic Links in Semantic Networks," *Computer (IEEE)*, vol. 16, no. 10, pp. 67-73, October 1983.
10. John Buxton, *Requirements for Ada Programming Support Environments: Stoneman*, US Department of Defense OSD/R&E, Washington, D.C., February 1980.
11. S. Ceri and Stefano Crespi-Reghizzi, "Relational Data Bases in the Design of Program Construction Systems," *SIGPLAN Notices*, vol. 18, no. 11, pp. 34-44, November 1983.
12. Ian D. Cottam, "The Rigorous Development of a System Version Control Program," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 2, pp. 143-154, March 1984.
13. Jacques Delort, "The Calculus of Relations and Constraints: Definition and Algorithms," Forthcoming Master's Thesis, University of California, Santa Barbara, 1985.
14. Jacky Estublier and Said Ghoul, "Un Système automatique de gestion de gros logiciels, la Base de Programmes Adele / An Automated Management System for Large Software, the Adele Data Base," *TSI (Technique et Science Informatiques / Technology and Science of Informatics)*, vol. 3, no. 4, pp. 253-260 (French edition), 221-240 (English Edition).
15. Stuart I. Feldman, "Make - A Program for Maintaining Computer Programs," *Software, Practice and Experience*, vol. 9, pp. 255-265, 1979.
16. Lloyd D. Fosdick and Leon J. Osterweil, "Data Flow Analysis in Software Reliability," *Computing Surveys*, vol. 8, no. 3, pp. 305-330, 1976.

17. Michael T. Garrett and James D. Foley, "Graphics Programming Using a Database System with Dependency Declarations," *ACM Transactions on Graphics*, vol. 1, no. 2, pp. 109-128, April 1982.
18. Susan L. Gerhart, "Application of Axiomatic Methods to a Specification Analyzer," in *Proceedings of the 7th International Conference on Software Engineering*, pp. 441-451, ACM-IEEE Computer Society, Orlando (Florida), March 26-29, 1984.
19. Xavier Glikson, "Analysis and Modeling of Software Configuration Management Practices," Forthcoming Master's Thesis, University of California, Santa Barbara, 1985.
20. Nico Haberman et al., *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh (Pennsylvania), 1982.
21. Cliff B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs (New-Jersey), 1980.
22. Ken Kennedy, *Automatic Translation of Fortran Programs to Vector Form*, Rice University, Department of Mathematical Sciences, October 1980.
23. David J. Kuck, R. H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, *Compiler Transformation of Dependence Graphs*, Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, Williamsburg (Virginia), January 1981.
24. Mark A. Linton, "Implementing Relational Views of Programs," in *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ed. Peter Henderson, pp. 132-140, Pittsburgh, Pennsylvania, April 23-25, 1984. Appears as Software Engineering Notes 9, 3 (May 1984) and SIGPLAN Notices 9, 3 (May 1984).
25. Alan K. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence*, vol. 8, pp. 99-118, 1977.
26. Bruce J. McLennan, "Overview of Relational Programming," *SIGPLAN Notices*, vol. 18, no. 3, pp. 36-44, March 1983.
27. Lucio Dimas dos Santos Mendes, "A Prolog Implementation of the Software Knowledge Base," Forthcoming Master's Thesis, University of California, Santa Barbara, 1985.
28. Bertrand Meyer, "Towards a Relational Theory of Software," Internal Report, University of California, Santa Barbara, July 1984.
29. Bertrand Meyer, "A System Description Method," in *International Workshop on Models and Languages for Software Specification and design*, ed. Robert G. Babb II and Ali Mili, pp. 42-46, Orlando (Fl.), March 1984. (also more detailed internal report available from the author).
30. David L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 128-138, March 1979.
31. Naphtali Rishe, "A Relational Database Design Methodology Using Binary Conceptual Schemata," Technical Report, Department of Computer Science, University of California, Santa Barbara, January 1985.
32. Mark J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364-369, December 1975.

33. Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, MIT Press, Boston, Massachussets, 1977.

34. Daniel Teichroew and Ernest A. III Hershey, "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.

35. Kim Walden, "Automatic Generation of Make Dependencies," *Software, Practice and Experience*, vol. 14, no. 6, pp. 575-585, June 1984.

APPENDIX

RELATIONS

Let X and Y be two sets. The set of binary relations (or just relations) between X and Y, denoted $X \leftrightarrow Y$, is defined as the powerset (set of subsets) of the cartesian product $X \times Y$:

$$X \leftrightarrow Y = \mathbf{P}\,(X \times Y)$$

In other words, a relation r between X and Y, i.e. an element of $X \leftrightarrow Y$, is a set of pairs

$$\{ [x_1, y_1], [x_2, y_2], \ldots \}$$

with $x_i \in X$ and $y_i \in Y$ for all i.

As a notational convention, the sets of interest (those between which relations are defined) will have names beginning with upper-case letters, e.g. X, *Specification*, etc. Names of set elements and those of relations will be written in lower-case, e.g. x, r, *part_of*. To express that a certain pair of elements $x \in X$, $y \in Y$ belongs to a relation r, i.e. that

$$[x, y] \in r$$

it is often convenient to use an infix notation, as in

$$x \ \mathbf{uses}\ y$$

(where x and y might be program modules), rather than

$$[x, y] \in uses.$$

We will use the convention that the name of a relation, written in **boldface** as in this example, may be used as an infix operator.

Since any relation in $X \leftrightarrow Y$ is a subset of $X \times Y$, we can talk about the intersection of two relations, denoted $r \cap s$, and their union, denoted $r \cup s$. We may also express that a relation is included in (is a subset of) another, by writing $r \subseteq s$.

The inverse of relation $r \in X \leftrightarrow Y$ is the relation r^{-1} in $Y \leftrightarrow X$ such that

$$y\ r^{-1}\ x \iff x\ r\ y$$

The domain of $r \in X \leftrightarrow Y$, written *domain (r)*, is the subset of X containing all elements x for which $x\ r\ y$ holds for some $y \in Y$. The range of r, written *range (r)*, is *domain* (r^{-1}).

The composition of two relations $r \in X \leftrightarrow Y$ and $s \in Y \leftrightarrow Z$, written $s \bullet r$, is that relation in $X \leftrightarrow Z$ which holds between elements x and z if and only if

$$x\ r\ y \text{ and } y\ s\ z \text{ for some } y \in Y$$

The order of the arguments to the composition operator is traditional in mathematics and has some justification; to many people, however, it is less confusing to write the relations in the order in which they are "applied"; thus rather than the dot notation we use the semi-colon notation, with $r\ ;\ s$ being defined as $s \bullet r$ (the use of the semi-colon is justified by the close connection which exists between statement sequencing in programs and composition of relations; see reference [4].

For any set X, the identity relation on X, denoted *id (X)*, or just *id* when there is no ambiguity, is the "diagonal" relation which holds only between each element and itself. We call *null* the empty relation.

Let $r \in X \leftrightarrow X$ (source and target set identical). The successive powers of r are defined as follows:

$$r^0 = id$$

$$r^i = r\ ;\ r^{i-1} \qquad (i > 0)$$

A relation $r \in X \leftrightarrow X$ has a transitive closure, denoted r^+, and a reflexive transitive closure, denoted r^*, defined as follows:

$$r^+ = r \cup r^2 \cup r^3 \cup \ldots$$

$$r^* = id \cup r^+$$

A relation $r \in X \longleftrightarrow X$ is:

- Transitive iff $r^2 \subseteq r$ (or equivalently $r^+ = r$)
- Reflexive iff $id \subseteq r$
- Symmetric iff $r^{-1} = r$
- Antisymmetric iff $r \cap r^{-1} \subseteq id$
- Functional iff $r^{-1} ; r \subseteq id$ (note that this characterizes partial functions)
- Total iff $id \subseteq r ; r^{-1}$

A (partial) order is a transitive, antisymmetric and reflexive relation. Such an order relation is total if and only if $r \cup r^{-1} = X \times X$.

A dag (directed acyclic graph) is a relation r such that r^* is a partial order. A dag is rooted if and only if, for any $y \in X$, the set of $x \in X$ such that $x\ r^*\ y$ is finite; a root is then an element of X - *domain* (r^{-1}). It is easily shown that in a rooted dag, for any $y \in X$, there is at least one root x such that $x\ r^*\ y$.

A forest is a rooted dag r such that r^{-1} is functional (note that r represents the relation between parent and child). It is easily shown that a non-empty forest has at least one root. A tree is a forest with at most one root.

[85g]

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.

8 22 85

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 *POINT*s, 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.

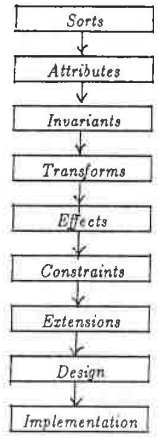


Figure 1: The Basic Paragraphs and their order.

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.

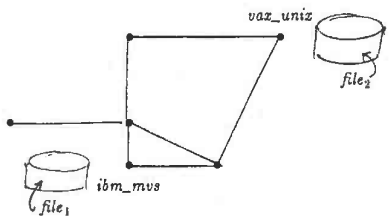


Figure 2: Files on a network

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 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


```
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 ;
```

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

$s \bullet locname$

The host on which it resides will be written $s \bullet host$, 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

$s \bullet file\_exists\ (l, c)$

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

$ct \bullet supporting\ (ft)$

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 $ibm\_mvs \bullet supporting\ (directory)$ to be false, $vax\_unix \bullet supporting\ (sequential)$ 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

$s \bullet file\_of\_name\ (l, c)$

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

If g is a global name, then $s \bullet used\_global\_name$ 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 $s \bullet globfile\ (g)$.

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 $s \bullet globname\ (g)$.

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 $s \bullet taken\ (g)$ is true for a global name g, then the user that has "taken" the corresponding file is given by $s \bullet owner\ (g)$ and and the key that was used to reserve it is $s \bullet key\ (g)$.

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* ;

i_1 : $f \bullet host \bullet make \bullet supporting\ (f \bullet ftype)$;

i_2 : $s \bullet file\_of\_name\ (l, c) \bullet locname = l$;

i_3 : $s \bullet globfile\ (s \bullet globname\ (f)) = f$;

i_4 : $s \bullet globname\ (s \bullet globfile\ (g)) = g$;

i_5 : $s \bullet used\_globname\ (s \bullet globname\ (f))$;

i_6 : $s \bullet mode\ (s \bullet globfile\ (g)) = global$;

j_1 : $ibm\_mvs \bullet supporting\ (ft) = (ft \in \{sequential, direct\_access, partitioned\})$;

j_2 : $vax\_unix \bullet supporting\ (ft) = (ft \in \{sequential, direct\_access, directory\})$;

j_3 : $multics \bullet supporting\ (ft) = (ft \in \{sequential, direct\_access, directory\})$;

j_4 : $vax\_vms \bullet supporting\ (ft) = (ft \in \{sequential, direct\_access, directory\})$;

j_5 : $apple\_2\_ms\_dos \bullet supporting\ (ft) = (ft \in \{sequential, direct\_access\})$;

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, $f \bullet host \bullet make$) 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 $(V_1, V_2, ..., V_n)$ ..... ;

..... ;

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

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

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 γ, δ,...

end *A* **use**

where γ, δ,... 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 γ, δ,...

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** $\beta 1$;

...

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

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

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 $x \bullet I\ (x)$, 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:

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

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 : $s \bullet globfile\ (s \bullet globname\ (f)) = f$;

The property to be proved is:

$\forall\ s \in STATE, f' \in FILE, g \in GLOBAL\_NAME,$

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

where i'_3 is i_3 with $s \bullet make\_global\ (f', g)$, as obtained from the effects paragraph, substituted for s. In other words, i'_3 is

$s' \bullet globfile\ (s' \bullet globname\ (f)) = f$

where

$s' = s \bullet\ make\_global\ (f', g)$

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

$lhs = s' \bullet globfile\ (g')$

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

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

Thus, factoring out the conditional expression, we get:

lhs = **if** $f = f'$ **then** $s' \bullet globfile\ (g)$ **else** $s' \bullet globfile\ (s \bullet globname\ (f))$ **end if**

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

$s \bullet make\_global\ (f', g) \bullet globfile\ (g)$

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

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

$s \bullet make\_global\ (f', g) \bullet globfile\ (s \bullet globname\ (f))$

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

if $g = s \bullet globname\ (f)$ **then** f' **else** $s \bullet globfile\ (s \bullet globname\ (f))$ **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 $\lessdot$ defined as follows: $f \lessdot g$ 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 $\lessdot$ 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

≠	0	1	2	$\perp$
0	f	t	t	f
1	t	f	t	f
2	t	t	f	f
$\perp$	f	f	f	f

¬	
t	f
f	t
$\perp$	f

∧	t	f	$\perp$
t	t	f	f
f	f	f	f
$\perp$	f	f	f

∨	t	f	$\perp$
t	t	t	t
f	t	f	f
$\perp$	t	f	f

⇒	t	f	$\perp$
t	t	f	f
f	t	t	t
$\perp$	t	f	f

Note that $(a \Rightarrow b) \equiv (\neg\ a \vee b)$. 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 involvés 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

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

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

domain $(h) = \{a \in$ **domain** $(f) \mid f(a) \in$ **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:

$x \bullet$ *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
- $x \bullet L_{t,a}[y,z] \ = \ x \bullet C_{t,a}[y] \ \wedge \ x \bullet$ *derived_constraint*$_{t,a}$ (y, z)

and $x \bullet$ *derived_constraint*$_{t,a}$ (y, z) is the expression obtained by substituting, for every attribute $\alpha \in$ *Attrib* (Γ_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:

$s \bullet$ *take* $(g, f, u, k) \bullet$ *owner* $(g') =$

if $g = g'$ **then** u

else $s \bullet$ *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*:

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

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

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

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

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

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

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

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

and not $s \bullet taken\ (g))$

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

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

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

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

and obtain:

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

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

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* • *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.