CEPAGE: a full-screen structural editor

Bertrand MEYER

Computer Science Department, University of California, Santa Barbara, California 93106, U.S.A.

Jean-Marc NERSON

CIMSA, 10 Avenue de l'Europe, 78140 Vélizy, France

ABSTRACT

This paper describes CEPAGE[1], an editor for structured documents, designed for ease of use on modern terminals. The design of CEPAGE is the result of work on syntax editors, full-screen editors and advanced software environments. CEPAGE is a universal editor, in which the language description is merely a parameter; its external interface is designed for the children of the video age. Although itself a prototype, CEPAGE embodies the properties which a structural editor usable in an industrial environment should possess.

Keywords: structural editors, syntax editors, human interfaces, man–machine communication, programming environments, program formatting, ergonomics.

1. Objectives

CEPAGE is a structural editor (a term we prefer to 'syntax editor'), whose human interface was designed with particular care. It is completely parameterizable and can be applied to any language defined by a grammar: a programming language or specification language, but also languages for describing structured documents of any kind (we shall from now on use the word 'documents' for the objects the editor is used to construct). CEPAGE belongs to a tradition of structural editors developed over the past few years [Hansen 71; Wilander 80; Donzeau-Gouge 81; Teitelbaum 81; Habermann 82; Allison 83; Donzeau-Gouge 84]. Structural editors, as opposed to normal text editors, permit documents to be manipulated, not as simple sequences of lines or characters, but as structured objects, by applying operations to them defined in terms of their structure. Among the main advantages of this method are:

1. The guarantee that only syntactically correct documents will be produced.
2. The possibility of carrying out transformations that may be complex but are guaranteed correct, e.g. transformations to improve portability or efficiency of programs.
3. The possibility of relieving the user of some of the routine tasks associated with the necessity, in a normal text editor, of supplying all the details of the 'concrete' syntax of the documents.
4. The possibility of automatically translating documents from a syntactic framework into another framework (e.g. in the case of conversions between programming languages).
5. The use of standardized data structure (generally the abstract syntax tree) which can serve as a support for other software tools (e.g. [Schroeder 83]), or even complete programming environments [Habermann 82].

[1] Work carried out initially at: Electricité de France, Direction des Etudes et Recherches, 1 avenue du Général de Gaulle, 92141 Clamart, France.

In spite of these qualities, structural editors have not made much headway in industry. One of the main reasons for this is, in our opinion, their external interface, which, in most cases, is of the 'line-by-line' type; i.e. dialogue with the user takes place by way of exchanges of commands and responses. Today's programming environments, however, increasingly frequently offer **full-screen** text editors such as SPF (on IBM), EMACS (on MULTICS and VAX-UNIX) or VI (on VAX-UNIX), which take full advantage of the potential provided by modern terminals. The characteristics of these systems can include the following [Meyer 83a]:

1. Use of the whole screen instead of a line as a unit of communication between the system and the user, giving the user a much wider view of the document during construction, and consequently permitting him to exercise better control on the whole editing process.
2. The possibility of providing, more easily than in a line-by-line system, personalized dialogue, by storing individual user profiles.
3. Providing users with several simultaneous views of the document being handled.
4. Finally, and more generally, the application of the 'direct manipulation' principle [Schneiderman 83], according to which, the user has a better command of the system if he is provided, at any given time, with a clear representation of the current state of the objects at hand.

The advantage of these different characteristics is such that it is almost impossible to persuade a user of a full-screen editor to go back to a line-by-line editor, regardless of its other qualities. In our experience, this also goes for structural editors: line-by-line type editors will not find favour amongst those who are used to full-screen systems.

The objectives of CEPAGE derive from these considerations. It was a matter of combining the advantages of structural editors, as regards security and power, with the power of full-screen text editors and the benefits of modern terminals.

The CEPAGE project was not intended to be a research project, but rather an exercise in technology transfer intended to win industrial support for an idea, structural editing, which has been the subject of considerable research. In fact, we had to 'invent' a little more than we had originally imagined.

The main sources of our inspiration were, as far as structural editors were concerned, GANDALF and (to a lesser degree) MENTOR and CPS; SMALLTALK also influenced us, as a man–machine interface model.

By any objective standards, CEPAGE is a small-scale project. Specification and design were carried out by the two authors of this article. Its implementation was due almost entirely to Jean-Marc Nerson (in addition, CEPAGE includes a small text editor, written by N. Triquet). Initial discussions were held at the beginning of 1983 with the aim (which was met) of producing a working prototype by 20 December 1983. Programming consists of about 6000 lines of PASCAL; in addition, it uses the Gescran package for handling the screen interface [Audin 80], produced by the same group and made up of 4000 lines of FORTRAN 77 (Gescran is a collection of subprograms permitting 'full-screen' interaction to be described easily by manipulating only objects belonging to four abstract types, called *screen, window, zone* and *terminal*, and accessible only via primitives specific to the package [Meyer 82]; it uses the ENSORCELE I/O package [Meyer 81; Brisson 82]. The rather special conditions in which this project was carried out no doubt explain the fact that all this hardly corresponds to good practice as defined by the most eminent authorities [Boehm 82].

It may be of interest to note that partial use of formal specifications, based on the language Z [Abrial 80], then on the M method [Meyer 84a] were of some assistance.

2. Using CEPAGE

2.1. The screen

The screen allocated to a CEPAGE session is divided into a certain number of **windows** (Fig. 1). Each window fulfils a specific function:

1. A 'Document' window contains a representation of the current state of the document being constructed or modified; some elements of this representation, displayed between chevrons (e.g. <*statement*>), correspond to elements of the document that have not yet been polished and are said to be 'non-terminal'.
2. A 'Text' window is intended to receive non-structured texts which it may be necessary to supply at certain stages of a session (e.g. identifiers, comments).
3. A 'Menu' window provides a list of options available at any stage.
4. A 'Type' window gives the syntax type of undefined elements (see discussion below).
5. 'Reserved' windows (not shown in Fig. 1) give information on documents or elements of documents other than the document being edited; these windows are used to change document during a session, as well as for copy and write operations.
6. A 'Message' window displays diagnostics.

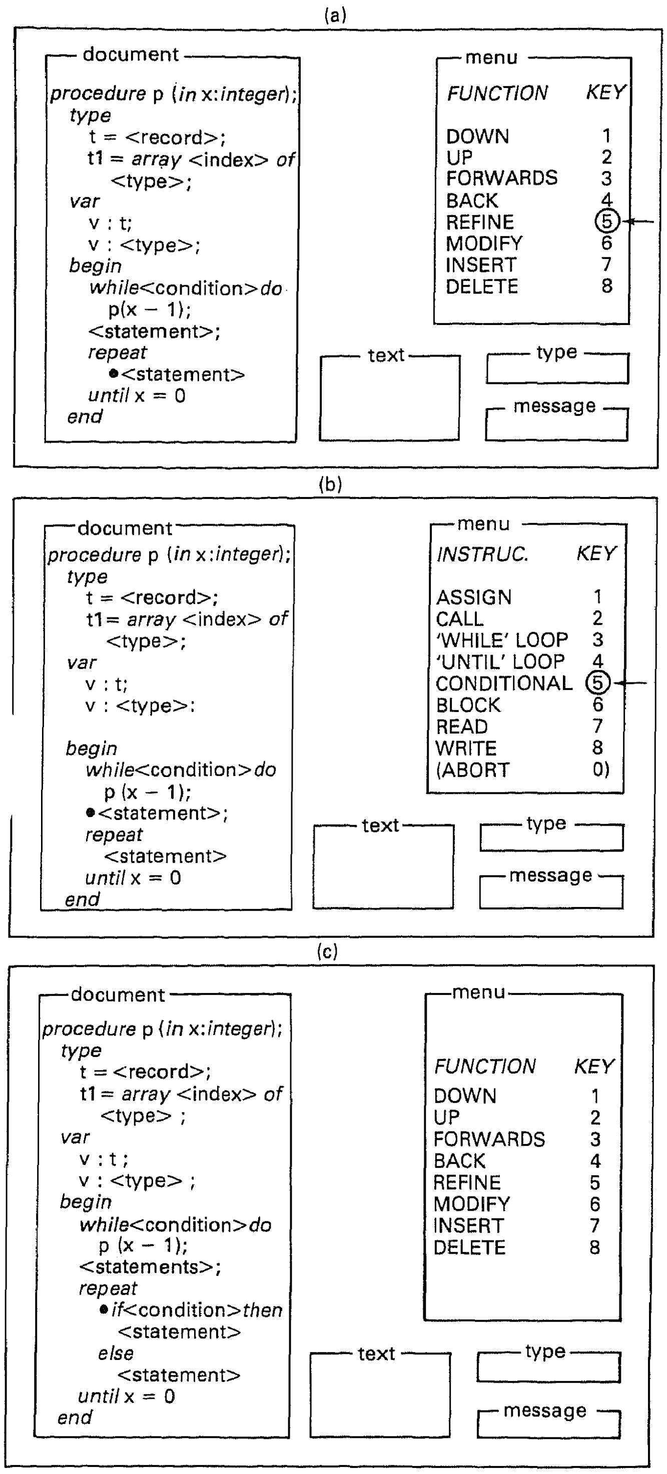
2.2. Dialogue

At each stage of a CEPAGE session, the system gives the user a choice between a certain number of options on a menu. The menus are sufficient for the basic CEPAGE functions; a user manual is not necessary once the user has a reasonable grasp of the basic concepts of the system. In the current IBM version, choices between the various elements of the menu are made using function keys on the terminal. On more advanced terminals, a mouse might be used.

The cursor is used whenever an element of the document has to be selected (e.g. to indicate which terminal will be used for editing, as in Fig. 1(a)), by placing it over the element in question. This is the only mode of access to a document (the concept of a line number, for example, is not used). A more rapid device, such as a mouse, would be particularly valuable.

There are some more advanced functions using commands; these commands are made up of a single word, and their inclusion is a result only of the limited number of function keys available (there are twelve). CEPAGE therefore has no 'command language' in the normal sense of the word; all interactions within the system are made by 'pointing and touching'.

In particular, the user writing a program text with CEPAGE, e.g. in PASCAL, *never* has to strike a key for concrete syntax elements such as key words like 'if,



(a)

```
— document ————————————          — menu ——————————
procedure p (in x:integer);      FUNCTION      KEY
  type                           DOWN            1
    t = <record>;                UP              2
    t1 = array <index> of        FORWARDS        3
         <type>;                 BACK            4
  var                            REFINE         (5)←
    v : t;                       MODIFY          6
    v : <type>;                  INSERT          7
  begin                          DELETE          8
    while<condition>do
      p(x – 1);
    <statement>;                 — text —   — type —
    repeat
      •<statement>                          — message —
    until x = 0
  end
```

(b)

```
— document ————————————          — menu ——————————
procedure p (in x:integer);      INSTRUC.      KEY
  type                           ASSIGN          1
    t = <record>;                CALL            2
    t1= array <index> of         'WHILE' LOOP    3
         <type>;                 'UNTIL' LOOP    4
  var                            CONDITIONAL    (5)←
    v : t;                       BLOCK           6
    v : <type>:                  READ            7
                                 WRITE           8
  begin                          (ABORT          0)
    while<condition>do
      p (x – 1);
    •<statement>;                — text —   — type —
    repeat
      <statement>                           — message —
    until x = 0
  end
```

(c)

```
— document ————————————          — menu ——————————
procedure p (in x:integer);
  type                           FUNCTION      KEY
    t = <record>;                DOWN            1
    t1 = array <index> of        UP              2
         <type> ;                FORWARDS        3
  var                            BACK            4
    v : t ;                      REFINE          5
    v : <type> ;                 MODIFY          6
  begin                          INSERT          7
    while<condition>do           DELETE          8
      p (x – 1);
    <statements>;
    repeat
      • if<condition>then
        <statement>
      else                       — text —   — type —
        <statement>
    until x = 0                             — message —
  end
```

● Cursor postion ○←Function key selected by user

Fig. 1.—Refinement.

procedure, record', etc. Instead, a menu allows him to choose between *conditional, procedure, declaration, record type declaration*, etc. and the system produces the correct syntax for him (routine tasks should be done by machines, not people).

The only case when the keyboard (apart from function keys) is necessary is when the user has to supply a text that the system could not invent by itself, such as an identifier or a comment. The 'text' window is used for this. The text is written in it, using the full-screen text editor included in CEPAGE.

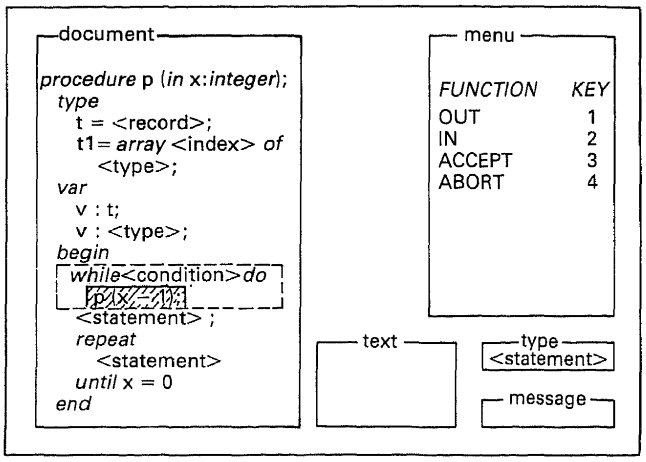
2.3. Basic functions

The main functions offered by CEPAGE fall into the following categories:

1. 'browse': scan the document (up or down within the hierarchy of syntax entities, backwards or forwards within lists);
2. 'construction-modification': creation of entities, modification of earlier versions, insertion and deletion within lists;
3. 'copy-transfer': reproducing or moving an element of text (using the 'definition' operation, as described below);
4. 'save-retrieve': using a file, in a suitable form, to save the present state of a partially or completely refined document; retrieval of a document previously saved;
5. 'generation': production of the final form of a fully defined document;
6. 'session management': document selection, change of document selection, library definition, etc. (a library is a collection of documents; in the course of a session, one can work on several documents, but only one of them is active at a given moment; one can move freely from one to another).

2.4. Delimiting elements of text

Delimiting text (Fig. 2) is an essential operation for functions that require the user to define a syntactic subsection of the document: copying or writing, for example, make it necessary to define the part of the document to which the operation applies. The mechanism provided applies direct manipulation principles.



```
— document ————————————          — menu ——————————
procedure p (in x:integer);      FUNCTION      KEY
  type                           OUT             1
    t = <record>;                IN              2
    t1= array <index> of         ACCEPT          3
         <type>;                 ABORT           4
  var
    v : t;
    v : <type>;
  begin
  ┌ while<condition>do ┐
  └   p(x–1);          ┘
    <statement> ;                — text —   — type —
    repeat                                  <statement>
      <statement>
    until x = 0                             — message —
  end
```

Fig. 2.—Delimiting.

In order to delimit an element, the cursor is placed anywhere within it, and the extent of the element is given by a series of commnds, issued using function keys (as indicated by the appropriate menu); at each stage the system highlights the element by means of special display attributes (e.g. colour or reverse video).

Commands provided for delimitation are as follows:

1. Out: extends the element delimited to the structure immediately enclosing it syntactically (e.g. if a statement has just been defined, the whole of the block that contains it is included).
2. In: cancels the effect of the previous out function, by returning to the next lower level.

3. Extend left: includes the immediately preceding element (applicable when the current element is a sublist; the three complementary operations are: exclude left, extend right, exclude right).

4. Accept (the currently delimited element is accepted).

5. Cancel.

2.5. Language modifications

CEPAGE is completely language-independent; the syntax (concrete and abstract) is a parameter which can be modified as required. In the present version, description and modification of the language is carried out by inputting a grammar. It is expected that this operation will eventually by implemented as an interface to the system itself, i.e. that one of the languages that CEPAGE will be defined for will be a syntax description language (it is entirely in keeping with the general principles of the design of CEPAGE, that the user need not have any knowledge of the concrete syntax of this 'language').

Language modifications may seem fairly useless in practice, as long as CEPAGE is delivered with descriptions of the main languages. However, the possibility of easily adapting the language description to local conditions seems highly desirable to us. In particular, it permits programming standards to be set up more conveniently (and in a way that makes them more easily acceptable) than by using *a posteriori* validation tools. Language subsets, conventions relating to comments and to the structure of programs, etc. can also be enforced in this way.

3. CEPAGE: technical choices

3.1. Basic data structures

In the course of a session, CEPAGE works (Fig. 3) on two main data structures:

1. The internal description of the language, or grammar graph.

2. The internal description of a set of documents, or abstract syntax forest.

It is important to note that these two data structures are treated in the same way. This is what makes CEPAGE a completely parameterized system with respect to language: the description of the language is interpreted repetitively by the system. This distinguishes CEPAGE clearly from a system such as GANDALF, which is parametrizable but uses a 'compiled' description of the language; i.e. a 'kernel' version of GANDALF and a description of a language X (or Z or C) is used as a basis to obtain a specific tool, GANDALF-X or GANDALF-C, adapted to the chosen language. The approach adopted by CEPAGE offers greater flexibility and makes it possible to modify a language easily. On the other hand, it does not permit semantic actions to be taken into account as easily as in, say, GANDALF.

The grammar graph is a data structure representing the grammar of the language.

Abstract syntax is used as a basis for language description; it is described by a collection of **syntax types** and **production rules**. Each syntax type appears on the left-hand side of at most one production rule; any that do not appear to the left of a production rule are said to be terminal. There are three types of production rules, known as 'aggregate', 'choice' and 'list', illustrated respectively by the following examples:

conditional = c: boolean; st 1, st 2: statement;
statement = assignment |conditional| compound
*compound = statement**

Concrete syntax is obtained by 'decoration' of abstract syntax productions; every list type production rule is associated with a header, delimiter and end (e.g. **begin**, the semicolon and **end** in the case of *compound*). The grammar graph contains all this information.

The **abstract syntax forest** is a set of abstract syntax trees, each associated with a document, or an element of a document, in the course of elaboration.

There are four types of internal nodes on an abstract syntax tree (Fig. 4), corresponding to the four types of production rules:

1. 'Aggregate' nodes have a fixed arity.

2. 'Choice' nodes simply represent a choice in a production rule of the choice type.

3. 'List' nodes can have any number of sons.

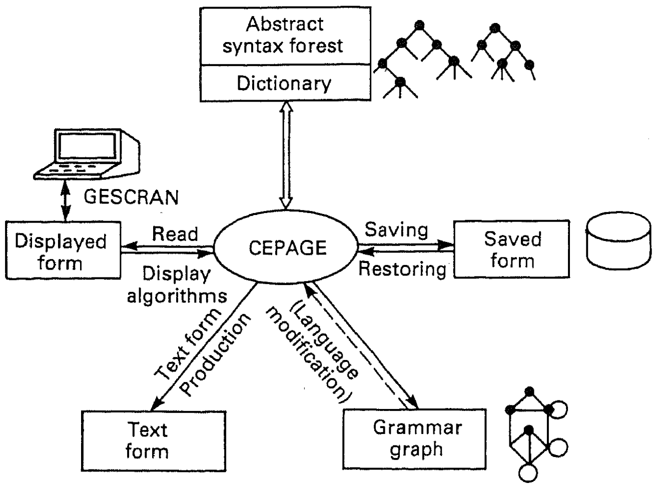


Fig. 3.—Data structures.

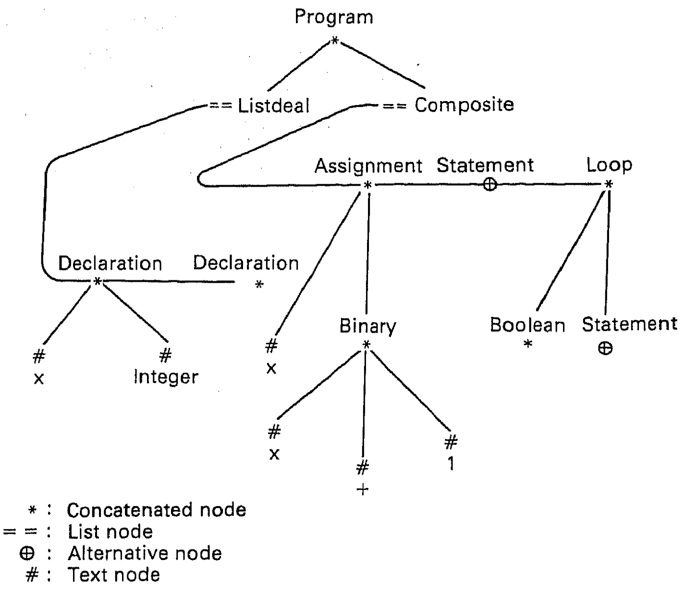


Fig. 4.—Abstract syntax tree.

4. 'Text' nodes correspond to the terminal elements defined by the user using the text editor included in CEPAGE.

3.2. Other data structures

Other data structures complement the first two. Apart from abstract syntax trees, the following three representations are necessary for documents:

1. A form that can be displayed, as a set of elements to be transmitted to the display package GESCRAN, for display on the terminal screen at each stage of the session.
2. A storable form for storage and later retrieval of the current state of documents.
3. Text form, the ultimate aim of the editing process.

The abstract syntax forest is also associated with a **dictionary**, containing the different text elements necessary (identifiers, etc.). The leaves of the syntax trees contain references to the dictionary.

3.3. Algorithms

Note that the objectives defined above imply **the absence of syntax analysis** (parsing) in CEPAGE. Construction of a text takes place by successive choices corresponding to the abstract syntax; the concrete syntax s constructed by the system which actually does the nverse of syntax analysis, sometimes known as 'unparsing'.

The freedom the user has in describing the language permits a good compromise to be established between ease of use and the degree of detail that the system permits; e.g. *expression* can be treated as a terminal. Another technique for this type of syntax entity, not used in the present version of CEPAGE, is that of [Kaiser 82], which is halfway between 'parsing' and 'unparsing'.

Though there is no syntax analysis, we still had one difficult algorithm to handle, for constructing the displayed form of documents. The issue here is to provide, at any given moment, a representation of the state of the document which is as clear as possible, while respecting the limits imposed by the physical size of the terminal.

Using a text editor, whether full-screen or not, we can usually only provide an extract of the document, giving a few contiguous lines (some editors have an option of excluding groups of lines from the part displayed in order to concentrate on the most important elements at any given moment). A structural editor must be capable of giving a global view of the document or part of it, even if it cannot represent all its details on the screen. The solution is **elision**: certain elements of the document can be replaced by an abbreviation—more exactly, by a simple indication of their type. In this way, a 2000-line procedure could be represented by the simple indication '*procedure*'; we call this type of abbreviation an **abstraction**. The second type of abbreviation carried out by CEPAGE is **collapsing**, which consists of an abstraction applied to one or more sublists of a list, as in:

"231 statements";
p:= *expression*;
"57 statements"

At each stage of the session, the system determines the **focus** of the user's apparent attention, after the last operations he has carried out, and seeks to display as detailed as possible a view of a portion of the document, around the focus, and decides on whatever abstraction and collapsing is necessary. It deduces the displayable form that is then transmitted to GESCRAN, for eventual display on the screen.

Finding a good representation for display purposes proved to be an unexpectedly difficult task. We were surprised by the lack of available documents; if the brief reference in [Barstow 84] is excluded, the only published reference, to the best of our knowledge, is [Mikelsons 81], which is difficult to use because of its lack of precision and the special characteristics of the environment described.

The abundant literature on program formatting ('pretty-printing'), is of little use here; the basic assumption, though it is not generally stated explicitly (cf. in particular [Oppen 80]) is that, though the length of individual lines may be limited, the number of lines is not. For screen formatting, however, **columns and lines are severely limited resources**. We were consequently led to design algorithms we have described elsewhere [Meyer 83b, 84b], which are beyond the scope of this paper. These algorithms behave in a linear way with respect to the number of nodes in the syntax tree. This is one of the fields in which we have had to resort to 'invention'.

4. The future of CEPAGE

As was indicated at the beginning of this article, the December 1983 version is a prototype which, however, covers the essential functions of the system. We foresee the following developments.

1. It will be necessary to study users' reactions. The design of CEPAGE rests on what we think is a good ergonomic basis for interactive systems, an opinion supported by recent studies resting on solid scientific bases [Card 83], but which, naturally, needs experimental confirmation.
2. We also plan to port the system to other environments. CEPAGE was designed to be portable; the decision to use PASCAL, in preference to an object-oriented language such as SIMULA 67 (used previously with success by the same group to implement high-quality interactive tools), was deliberately taken with just this in mind. In the short term we intend to adapt CEPAGE to a UNIX environment both on a VAX and on a SUN workstation (at the University of California); the SUN is a programmer's workstation based on a 68000 with a high-resolution bit-mapped screen and a mouse. This project is particularly important for us, as it is only in hardware environments of this quality that tools such as CEPAGE can, in our opinion, fulfil all their promise. We hope that CEPAGE will also be adapted to other similar systems (PERQ, APOLLO, SM90, etc.).
3. We also need to add the major functions missing from the prototype, particularly the language modification tool, and to prepare CEPAGE grammars for the main languages currently in use (the prototype was tested with a grammar of a language similar to PASCAL).

We expect to be able to answer some questions that have had to be left hanging, in the light of experience of the system in operation; syntax analysis is one such issue, and we have to decide whether to add a parser to later versions, allowing existing programs, obtained by other means, to be manipulated by CEPAGE.

We hope that the use of the first versions will confirm our view of the great potential importance of a powerful and ergonomic programming environment which could be derived from it.

REFERENCES

[Abrial 80] J.-R. Abrial, S. A. Schuman and B. Meyer: *A specification language;* In: On the Construction of Programs, C. A. R. Hoare and R. Perrot (Eds), 1980, Cambridge University Press, Cambridge (U.K.).

[Allison 83] R. Allison: *Syntax-directed program editing;* Software—Practice and Experience, **13**, 453–465, April 1983.

[Audin 80] E. Audin, G Brisson, B. Meyer and F. Vapné-Ficheux: *Gescran, Manuel de Référence,* Atelier Logiciel 22, Electricité de France, 1980. (Fourth Edition, 1984.)

[Barstow 84] D. R. Barstow: *A display-oriented editor for INTERLISP;* In: Interactive Programming Environments, D. R. Barstow, H. E. Shrobe and E. Sandewall (Eds), 288–299, 1984, McGraw-Hill, New York.

[Boehm 82] B. W. Boehm: Software Engineering Environments; 1982, Prentice-Hall, Englewood Cliffs, NJ.

[Brisson 82] G. Brisson, B. Meyer and F. Vapné-Ficheux: *Ensorcelé: Entrées et Sorties Sans Format (2ème partie);* Atelier Logiciel 6, Electricité de France, Dec. 1982.

[Card 83] S. K. Card, T. P. Moran and A. Newell: The Psychology of Human–Computer Interaction; 1983, Lawrence Erlbaum Associates, Hillsdale, NJ.

[Donzeau-Gouge 81] V. Donzeau-Gouge, G. Huet, G. Kahn and B. Lang: *Environment de programmation Mentor: présent et avenir;* In: Actes des Troisièmes Journées Francophones sur l'Informatique, 1981, Geneva.

[Donzeau-Gouge 84] V. Donzeau-Gouge, G. Huet, G. Kahn and B. Lang: *Programming environments based on structured editors: the MENTOR experience;* In: Interactive Programming Environments, D. R. Barstow, H. E. Shrobe and E. Sandewall (Eds), 128–140, 1984, McGraw-Hill, New York.

[Habermann 82] N. Habermann et al.: The Second Compendium of Gandalf Documentation; 1982, Carnegie-Mellon University, Pittsburgh, PA.

[Hansen 71] W. J. Hansen: *Creation of hierarchic text with a computer display;* ANL-7818, Argonne National Laboratory, Argonne, IL, 1971. (Also as dissertation, Computer Science Department, Stanford University, June 1971.)

[Lewis 81] J. W. Lewis: *Beyond ALBE/P: language and neutral form;* In: Proceedings of the 5th International Conference on Software Engineering, 422–429, 1981, San Diego, CA.

[Kaiser 82] G. E. Kaiser and E. Kant: *Incremental expression parsing for syntax-directed editors;* Computer Science Report, Carnegie-Mellon University, Pittsburgh, PA, Oct. 1982.

[Meyer 81] B. Meyer: *Ensorcelé: entrées et sorties sans format (1ère partie);* Atelier Logiciel 4, Electricité de France, April 1981. (Fourth Edition.)

[Meyer 82] B. Meyer: *Principles of package design;* Communications of the ACM, **25** (7), 419–428, July 1982.

[Meyer 83a] B. 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, P. Degano and E. Sandewall (Eds), 1983, North-Holland, Amsterdam.

[Meyer 83b] B. Meyer and J.-M. Nerson: *Showing programs on a screen;* Internal Report HI/4590-01, Electricité de France, Sep. 1983.

[Meyer 84a] B. Meyer: *A system description method;* In: Workshop on Specification Languages, Orlando, FL, March 1984 (to appear).

[Meyer 84b] B. Meyer and J.-M. Nerson: *Showing programs on a screen;* submitted for publication, 1984.

[Mikelsons 81] M. Mikelsons: *Prettyprinting in an interactive programming environment;* Sigplan Notices, **16** (6), 108–116, June 1981.

[Oppen 80] D. C. Oppen: *Prettyprinting;* ACM Transactions on Programming Languages and Systems (TOPLAS), **2** (4), 465–483, Oct. 1980.

[Schroeder 83] A. Schroeder: *Outils d'analyse des programmes sous Mentor;* Globule (AFCET), no. 4, 1983.

[Shneiderman 83] B. Shneiderman: *Direct manipulation: a step beyond programming languages;* Computer (IEEE), **16** (8), 57–69, Aug. 1983.

[Teitelbaum 81] T. Teitelbaum and T. Reps: *The Cornell Program Synthesizer: a syntax-directed programming environment;* Communications of the ACM, **24** (19), 563–573, Sep. 1981.

[Wilander 80] J. Wilander: *An interactive programming system for Pascal;* BIT, **20**, 163–174, 1980.