Cépage * : Toward Computer-Aided Design of Software

Bertrand Meyer
Interactive Software Engineering, Inc., Goleta, California

The Cépage system for structural document manipulation combines the techniques of structural editing with modern concepts about user interfaces. Cépage may be used to produce and modify documents in any language. Adaptation to a new language or to variants of a previously described language are carried out using a simple notation, LDL (Language Description Language). The system relies on an elaborate display mechanism that automatically produces structural representations, adjusted to the current window size, with facilities for quick document traversal. The interface allows both menu-driven and text-driven entry; the built-in parser is able to complete partial input into syntactically correct forms.

1. WHY CEPAGE?

For many years, software engineers have been providing the engineers of other fields with advanced design tools that considerably facilitate their jobs, relieve them of tedious tasks, put them in control of the design process, and help them turn out quality products. But the tools used by software designers themselves look quite primitive in comparison: The standard design environment includes a text editor, a compiler, perhaps a debugger, but hardly anything that could in fairness be characterized as a design system.

This article presents a system 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.

Cépage (pronounced *say-pajj*) is intended to make the full power of computer-aided design available to software developers in practical environments. It frees its users from many of the chores traditionally associated with the construction of programs and other software-related documents; thus it allows them to concentrate on the really important aspects of software design. Beyond software design, Cépage may be applied to the construction of many kinds of documents with a rich enough structure, such as specifications, designs, technical reports, and configuration files.

1.1. Structural Editing

Cépage relies on the technology of structural editing, which was introduced as early as 1971 by W. J. Hansen in the Emily system [1]. A structural editor is a software tool that makes it possible to manipulate documents in terms of their structure.

Structural editing is best characterized by comparing it to traditional text editing. A standard text editor—say Vi under Unix, or SPF under TSO—treats any document as a sequence of characters or lines; it does not make any distinction between, say, a Pascal program and a technical report. In contrast, a structural editor is driven by a language description and will support the production of documents that conform to this language.

The potential advantages of using a structural editor are numerous:

All documents produced are guaranteed to be syntactically correct.

As syntactic aspects are handled by the editor, users may concentrate on more interesting issues. A programmer, for example, has better things to do than typing keywords, worrying about proper indentation, checking balancing of parentheses, carrying out commenting standards, etc.

Many language-dependent operations, which are hard or impossible to achieve with text editors, become possible. For example, text editors are good at performing operations of the form "replace all occurrences of the letter *a* by the letter *b*" but notoriously bad at operations such as "replace all

Address correspondence to Bertrand Meyer, Interactive Software Engineering, Inc., 270 Storke Road, Suite 7, Goleta, CA 93117.

*Cépage and Eiffel are trademarks of Interactive Software Engineering, Inc. Unix is a trademark of AT&T. A preliminary version of this paper was published in *Computer Language*, September 1986.

The Journal of Systems and Software 8, 419–429 (1988)

0164-1212/88/$3.50

occurrences of the variable *a* that are part of an expression by the variable *b*." Because a structural editor embodies knowledge about such notions as variable and expression, it can carry out such operations.

When the languages considered are software languages such as programming or design languages, a structural editor simplifies the task of writing other software tools. Most software tools acting on programs, designs, specifications, and the like must at some point perform some parsing to get the input documents into a suitable internal form. If a structural editor is used, this task is no longer necessary; the structural editor has its own internal form, usually some kind of tree structure, that can be used by other tools. Examples of tools that may benefit from this approach include program analyzers, profilers, test generators, specification checkers, and version and configuration managers. Thus a good structural editor is a promising candidate to serve as the basis for an integrated programming environment.

As we shall see in the case of Cépage, a flexible structural editor is a powerful tool for implementing programming or documentation standards.

More generally, a structural editor brings all the benefits of a "smart" tool that knows about the structure of the document it manipulates.

1.2. Cépage and Previous Work

There have been structural editors before. Some of the best known developments include Mentor [2, 3], Gandalf [4], and the Cornell Program Synthesizer [5]. A more recent tool with graphical facilities is Pecan [6, 7].

Why do we think that Cépage goes beyond these previous efforts? The combination of features that makes Cépage original and (we think) more practical than its predecessors includes:

The ease of adaptation to any new language thanks to the LDL formalism (see Section 5.2)

The ability to support languages with elaborate or irregular syntax, such as Fortran or languages describing the structure of technical reports

The double mode of entry (menu-driven or text-driven), providing for a flexible user interface, and supported by a general-purpose parser that does not enforce any of the restrictions of common parser generators

An elaborate display mechanism, which ensures that meaningful structural views are produced at every stage of the document construction process

The open architecture of the system, which allows it to be interfaced with other tools and makes it a promising candidate as a basis for a complete software development environment

More generally, the ambition, which pervades the whole system design, to take structural editing out of the laboratory and make its exciting potential available to practicing programmers in ordinary environments

The rest of this paper describes how these goals have been pursued in the design of the system.

2. THE EXPANSION PROCESS

The basic mechanism for document construction is known under Cépage as expansion. Expansion comes in two styles: menu-driven and text-driven. The following extracts from a sample session illustrate them.

2.1. Menu-Driven Expansion

Figure 1 shows a picture of a Cépage window at some point during a session. In this example, Cépage is being applied to a Pascal-like language. The document being constructed is an incomplete program, containing some unexpanded elements such as ⟨Instruction⟩, and ⟨Boolean_expression⟩. These are placeholders representing substructures that are yet to be expanded. Unexpanded elements appear in angle brackets if they are required, or in square brackets if they are optional, as the label declaration part of a Pascal program. A

Figure 1. A Cépage document window.

```
program JSS_example;
        (*A demonstration of Cépage*)
    [labels];
    constant
        pi = 3.141592; [more_constants];
    [types];
    [variables];
    {3 subprograms};
begin
    <instruction>; ⇐
    xx := <expression>;
    repeat
        if x = y then
            {instruction}
        else
            x := x+1
        end
    until <boolean> end;
    {7 instructions}
end. (*JSS_example*)
```

document containing unexpanded elements is called a *partially expanded document*.

Eventually a document should be totally 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 formatter such as troff on Unix.

One way to develop an unexpanded element is to use the menu-driven interface provided by Cépage. Assume, for example, the user wants to expand the first ⟨instruction⟩ of the program; he may bring the cursor to this element and enter the ^X (eXpand) command (we use the notation ^a to denote the result of pressing letter *a* while holding down the Control key). This brings a pop-up menu of available choices for instructions in Pascal (Figure 2). Once the proper choice has been made, the form of the chosen expansion will override the ⟨instruction⟩ placeholder (Figure 3). This form of document construction is particularly efficient for high-level constructs whose expansion may be long and involve many keywords. Software developers have more interesting things to do than typing. The problem is not so much typing itself as the fact that it detracts the user's attention from other, more important concerns.

2.2. Text-Driven Entry

Cépage is not dogmatic about menu-driven entry vs. typing. As mentioned above, menu-driven entry is useful for high-level constructs; it is also precious for novices or users who do not remember all the syntactical details of a particular language. However, this style of entry may become tedious for low-level constructs such as simple expressions and more generally when the expansion is short and the user knows exactly what it is. When what you want is $x + y$, simply typing it is often just as convenient as first requesting a sum from the "expression" menu and then entering the two operands in sequence.

Cépage allows the user to do just that: move the cursor to the element to be expanded and type the text. For example, the ⟨expression⟩ placeholder that appears in Figures 1–3 may be expanded by moving the cursor to it and typing $x + y$. (The reason for using control codes for commands, for example ^X for eXpand, is precisely to allow normal characters to stand for themselves. Typing a normal character triggers the text entry mode.) As soon as text has been typed, the built-in Cépage parser will analyze it on the fly, so that the result is exactly the same as if the phrase had been entered in the menu-driven mode.

An important feature of the Cépage parser is that the phrase typed need not be complete. For example, the text typed for an ⟨expression⟩ may be just

```
x * (a+
```

Figure 2. Selecting an expansion in a menu.

```
program JSS_example;
      (*A demonstration of Cépage*)
   [labels];
   constant
      pi = 3.141592; [more_constants];
   [types];
   [variables];
   {3 subprograms};
begin
   <instruction>; ⇐
   xx := <expression>;
   repeat
      if x = y then
         {instruction}
      else
         x := x+1
      end
   until <boolean> end;
   {7 instructions}
end. (*JSS_example*)
```

EXPAND: <Instruction>
if...
while...
repeat...
for...
call...
... := ...
CANCEL

```
program JSS_example;
      (*A demonstration of Cépage*)
   [labels];
   constant
      pi = 3.141592; [more_constants];
   [types];
   [variables];
   {3 subprograms};
begin
⇒ if <Boolean> then
      <Instruction>
   else
      <Instruction>
   end;
   xx := <expression>;
   repeat
      if x = y then
         {Instruction}
      else
         x := x+1
      end
   until <Boolean> end;
   {7 Instructions}
end. (*JSS_example*)
```

Figure 3. Result of expansion.

which the Cépage parser will echo back as

x * (a + ⟨factor⟩)

that is to say, closing the parentheses and requesting the user to expand (now or later) the missing ⟨factor⟩. Similarly, the above expansion of ⟨instruction⟩ (Figures 1-3) might have been obtained by typing just

if

over the ⟨instruction⟩ placeholder. As this is the unambiguous beginning of a conditional instruction, the result would have been exactly the same as that obtained in Figure 3 by choosing the conditional instruction from the instruction menu.

This ability to enter incomplete phrases is one of the key advantages of Cépage over text editors or more primitive structural editors.

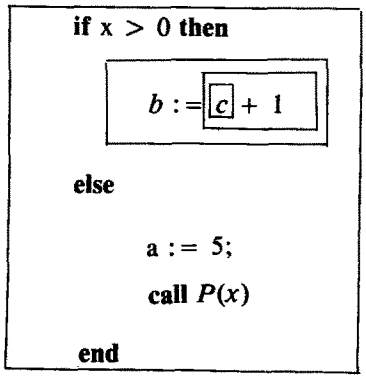
If the beginning of the phrase is ambiguous, as when *x* is entered for an instruction [this could be the beginning of an assignment, x := ⟨expression⟩, or of a procedure call, x (⟨parameters⟩)], the parser will mention it and allow the user to cyclically examine all the legal possibilities in order to select the appropriate one.

3. OTHER CAPABILITIES

The previous section has described the basic process: expansion. In this section, we introduce some other commands of Cépage, which should give a representative, if incomplete, idea of the interface with the system.

3.1. Document Traversal

The user interacts with Cépage in terms of structures rather than characters. These structures are represented by windows on the screen. The notion of "window" used here is a general one; the window structure is hierarchical. In the following extract, all windows surrounding *c* are boxed.



One of the aims of Cépage was to make the technology of structural editing available to a wide population of users, many of whom are not computer scientists. Because of this requirement, it was essential to make sure that the concepts could be explained in simple terms. Whereas the interface to many structural editors can be understood only in terms of abstract or concrete syntax trees—and Cépage's implementation, as described in Section 6.2, indeed relies on tree structures—, we have been very careful to avoid any such terminology in the documentation. As a result, the word "tree" does not appear anywhere in the user documentation; the interface is described in terms of the hierarchical structure of the document as reflected by the hierarchical window structure.

At each point during a session, one window is the active window; it appears in reverse video. (To make up for the difficulty of showing reverse video in print, the active window is marked by an arrow in Figures 1-3). The cursor keys move between windows; the cursor is a logical, "window" cursor, not a physical, character cursor. The basic traversal commands are:

→ or ^N (Next): Go to next window.

← or ^P (Previous): Go to previous window.

↓ or ^D (Down): Go to first window on a subsequent line.

↑ or ^U (Up): Go to first window on a preceding line.

^I (In): Go to first enclosed window.

^O (Out): Go to closest enclosing window.

The first four operations wrap around the structure upon reaching the edges. The last of these operations may entail an abstraction on the display (see Section 4).

3.2. Selection

Most operations use the current window as argument. Sometimes it is necessary to select another substructure of the document, for example, a sublist (say three consecutive instructions out of a block). This is done using the selection commands.

Selection is entered by typing ^S; the smallest enclosing window becomes the selected substructure. The following commands are then available:

^K (Keep): Accept the currently selected substructure and exit selection mode.

^C (Cancel): Cancel selection and exit selection mode.

→ or ^N: Add next list element to selected substructure.

← or ^P: Add previous list element to selected substructure.

↑ or ^O: Make enclosing substructure the selected one.

Backspace: Cancel the effect of the last command (if one of the last three above).

3.3. Replication

As is proper with a structural editor, replicating operations apply to substructures rather than arbitrary subsets of the document. To allow for a flexible interface, source and target may be selected in any order. The command ^W (Whence) designates the currently selected window or sublist as source for the next replication or move; ^T (Thence, or Target) designates the current window as target. Command ^R (Replicate) copies the current source to the current target. The source and target must be of compatible types (for example, both of them could be instructions).

3.4. Deleting and Changing Expansions

Command ^F (Forget) will take the current window back to the unexpanded state or, if applied to an optional unexpanded element (such as the list of label declarations in a Pascal program), remove it. Command ^V (Vary), applied to an expanded window, will change the expansion. For example, ^V applied to a window

while x = y **do** x := x + 1 **end**

will present the user with the menu for ⟨instruction⟩ again. If the user chooses, say, ⟨conditional⟩, the result will be

if x = y **then** x := x + 1 **else** ⟨instruction⟩ **end**

In other words, the system attempts to retain whatever substructures it can from one choice to the next. This way, the user does not need to reenter long substructures.

3.5. Insertion

Command ^A adds a new list element after the current one; ^B adds one before. So if the current window is an instruction in a block, either of these commands will add a new ⟨instruction⟩ placeholder, after or before the current instruction.

3.6. Marking

^M (Mark) will mark the current window for later return; ^G (Go) will return to the last marked window. ^G may be repeated to return to previously marked windows.

3.7. Generating and Writing

The basic commands, as seen above, have codes of the form ^letter. Other commands are introduced by a tab; note that this is an exception to the rule that typing a normal character starts the text entry mode. Typing two tabs in a row will start the text entry mode and initialize the window with one tab.

Tab-G (Generate) produces an archive of the current document in an external file. This archive may be used later to restart Cépage. This command may be applied to a partially expanded document; a suitable external form is used for storing such documents and retrieving them.

Tab-O (Output) produces a text form of the document. This command is applicable only if the document has been completely expanded.

3.8. Other Commands

Other available commands include operations for searching and replacing substructures and for alternating between various documents. Apart from the main document window, Cépage also maintains a "catalog" of substructures. An entry of the catalog may become the active document at any time; substructures may be moved between catalog entries and the active document.

4. SHOWING THE STRUCTURE, OR, HOW TO DO AWAY WITH THE SCROLLING SYNDROME

A structural editor manipulates documents in terms of their structure. Cépage takes the view that this idea should also be applied to the interface: The user should be presented with faithful representations of this structure.

It is surprising in this respect to note that many structural editors rely either on a line-oriented interface or on a screen-oriented interface that does not take the document's structure into account. A facility for "holophrasting," that is to say, collapsing the representation of some substructures to show more details of others, is often provided, but this leaves to the user the burden of deciding what to show and what to hide. The principle applied by Cépage is different. At each point in the editing process, Cépage ensures that the view displayed on the screen is structurally meaningful.

How can this be done? A standard text editor displays some contiguous excerpt of the text being handled. If the editor is "full-screen," like IBM's SPF, or Vi on Unix, this excerpt is going to fill a screen or window—say, 24 lines of contiguous text. Figure 4 shows a typical screen from a session with such an editor, where the text being edited is an Ada program. Such selection of displayed information has little to do with the underlying structure of the document. Programmers who use text editors for composing programs know this well: They spend much of their time going back and forth from one end of the file to the other, chasing for structurally related but

```
REMAINDER: REAL;
NB_USERS: INTEGER := 0;
TERMINALS: constant := 5;
package STOCK is
   LIMIT: constant := 1000;
   TABLE: array (1..LIMIT) of INTEGER;
   T_COUNT: INTEGER := 0;
   procedure RESTART;
   procedure BACKUP (F: FILE);
end STOCK
package body STOCK is
   procedure RESTART is
   begin
      INITIALIZE;
      CHECK_INVENTORY;
      for N in 1..LIMIT loop
         TABLE (N) := N;
         T_COUNT := T_COUNT + N;
      end loop
   end;
procedure BACKUP (F: FILE) is
   begin
      TABLE (X) := TABLE (X) + SMALL;
      while ACTIVE loop
         if NB_USERS > 5 then
```

Figure 4. A contiguous program fragment.

```
procedure PROCESSOR is
   {5 declarations};
   package STOCK is {specification} end STOCK
   package body STOCK is
      {2 declarations};
      procedure RESTART is {body} end;
      procedure BACKUP (F: FILE) is
         {12 instructions}
      end;
   begin RESTART end STOCK;

   procedure UPDATE (X: INTEGER) is
      {4 declarations}
   begin {7 instructions} end;

   procedure REMOVE (F: FILE) is
   begin {23 instructions} end;
begin -- PROCESSOR
   RESTART;
   while ACTIVE loop
      if NB_USERS > 5 then
         {6 instructions}
      else
         BACKUP
      end
end PROCESSOR;
```

Figure 5. A structured program view.

physically remote elements. This is the infamous scrolling syndrome. When designing Cépage, we came to the conclusion that such an approach was unacceptable for a structural editor. What we had to provide was a structural view, at varying levels of abstraction.

The driving metaphor is computer-aided design (CAD); with a good CAD system for, say, electronic design, one may traverse the structure of the system, choosing to see at each step a graphical representation at the level of the whole system, a subsystem, a wafer, a circuit, a gate, a transistor, etc. For programs the same principles should apply. For example, a representation of the Ada program of Figure 4 at the top level should be something like the one shown in Figure 5.

Here the elements in curly brackets (such as {*5 declarations*}) represent program elements that are present but may not be shown in detail due to lack of space. We say that such elements have been **abstracted**. By abstracting elements, it is possible to give a clear view of the structure at a certain level in the document. Here we see immediately what procedure PROCESSOR is made of.

Abstracted elements, such as {*instruction*}, also appear in Figures 1-3, distinguished by italics. They should not be confused with as yet unexpanded elements, appearing in angle brackets and in roman type, as in ⟨instruction⟩. An abstracted element is expanded but cannot be shown in detail for lack of screen "real estate."

If one wants to see an abstracted element, say the *procedure_body* of procedure RESTART in Figure 5, some of the context will have to be sacrificed. This will be achieved in Cépage simply by moving the cursor or mouse to the corresponding place on the screen and requesting a zoom with the ^X command (which has the effect of "expand" on an unexpanded structure and that of "zoom" on an abstracted one). Since the screen is not large enough to show everything, some details of the enclosing context will disappear to make space for the procedure body; this loss of context is indicated by lines of dots at the top and bottom of the screen. In all cases the view shown will be a structural one, corresponding to a meaningful syntactic structure rather than an arbitrary grouping of contiguous elements.

In Cépage, the abstraction mechanism is entirely automatic. From the user requests, the system determines what should be shown and what should be abstracted, taking into account both the document structure and the available window space.

When starting the design of Cépage, we were surprised to discover that few previous projects had

addressed the question of structural document display. One contribution was Mikelsons' [8], which unfortunately proved too specific for our purposes. The Cépage display mechanism relies on a set of data structures and algorithms that have been described elsewhere [9]. It is fair to say that, together with the parser, the display mechanism is the most elaborate part of Cépage.

5. LANGUAGE INDEPENDENCE

Independence vis-à-vis the language is one of the main aims of the design of Cépage. Why this interest in preserving adaptability to various languages?

5.1. Applications of Language Independence

The first reason simply comes from an assessment of the situation. Clearly, many programming languages are being used today, and there is little prospect for unification in the near future. A tool tied to a single language would have had a limited practical interest.

But switching to entirely different languages is not the only application of language adaptability. Very often, local environments (companies, laboratories, universities, etc.) have specific variants of programming languages (e.g., Turbo-Pascal, VMS Pascal, IBM Pascal.) that differ in small but significant details. Adapting a structural editor to such variations should be a simple matter, and indeed it is with Cépage.

Third, methodology-conscious projects are increasingly defining programming standards (such as comment conventions and exclusion of certain constructs). It is good to have such standards and even better to control their application; but nothing can beat using a program construction tool where the standards are built in, having been integrated into the language description. A modest example shown by the session extracts in Figures 1–3 was the automatic addition of a comment at the end of a program unit, repeating the name of the unit; but much more interesting conventions can be supported.

A fourth and equally promising application of language independence is the ability to support many types of document structures, which might not initially be thought of as languages. For example, many companies or departments have defined standardized structures for technical reports; such structures may be described as languages and then automatically supported by Cépage. As another example, installing a Unix system requires editing various configuration files (/etc/termcap for terminal descriptions, /etc/ttys for external lines, etc.), each of which has its own peculiar and sometimes confusing structure; each of language adaptability makes Cépage a good candidate for supporting this process. Examples include (such as a standard structure for technical reports) may benefit from a versatile tool that will automatically enforce the observation of the structure.

5.2. LDL

The language adaptability of Cépage is supported by the use of the LDL formalism (Language Description Language) [10]. Any Cépage session relies on a language description, or grammar, written in LDL. Only through this grammar is the session associated with the language; the Cépage system itself is entirely language-independent.

An LDL grammar is a sequence of "construct paragraphs," each of which describes the form of instances of a given construct (syntactic type). As an example of a construct paragraph, the following describes Pascal While loops:

```
construct While_loop
        --"While" loop in Pascal
short "while..."
aggregate
    test: Expression;
    body: Instruction
format
    "while" test "do" body
end
```

This description comprises the following elements:

- A full name, *While_loop*, used to refer to the construct in the grammar
- A short name, "while...", used to represent the construct when abstracted, unexpanded, or appearing as one possible choice in a menu
- A description of the abstract syntax of while loops as being "aggregates" with two components: an expression, the *test*, and an instruction, the *body*
- A description of the concrete syntax, which gives the external format of a loop with these components, indicating where the keywords should be placed

In such construct specifications, the abstract syntax refers to the abstract structure of construct instances. For example, a while loop is made of two components. The concrete syntax refers to the way construct instances are displayed or printed, based on the components of the abstract structure. Note that an abstract component may appear more than once in the concrete form; this is how the convention for final comments in Pascal programs or procedures, repeating the name of the unit (see Figures 1–3), may be automatically enforced.

Aggregates are but one type of constructs. Others

include "choice" and "list" constructs, as in the following self-explanatory examples:

```
construct Instruction
        --Instruction in Pascal
short "instruction"
choice
    Assignment, While, Repeat, If, Call, Case,
    Goto
end
```

```
construct Compound
        --Compound instruction in Pascal
short "begin..."
list Instruction
format
    header "begin"
    delimiter ";"
    tail "end"
end
```

Note how the concrete syntax for a list construct is specified by a header, a delimiter to be inserted between elements, and a tail.

Other types of constructs include atomic constructs (basic elements such as identifiers or constants, described by regular expressions) and predefined constructs (integers, strings, etc.).

We felt very strongly that writing a language description should be easy. Using LDL, the description of a language should take from a few hours (when working on a variant of a previously defined language) to at most a week for an entirely new language. We expect many language descriptions to be obtained by imitation or modification of existing ones.

Of course, LDL descriptions may be done using Cépage, as LDL is one of the languages supported by the system. A grammar for LDL is given in Ref. 10.

5.3. Using the Grammar for Document Layout

The LDL grammar is used by Cépage for two purposes: It drives both the display mechanism described above and the Cépage parser.

The display mechanism relies on the grammar to decide when abstractions should be performed and to format the document on the screen. The formatting process is entirely automatic; in particular, it is usually not necessary to add any special formatting indications to the LDL grammars.

For languages with a regular enough structure, such as most modern programming languages, the default display policy will perform necessary line returns, indentations, and so on in an entirely satisfactory manner, similar to the examples of Figures 1-3 and 5. The basic principle of the display policy is to use the *abstract* structure to construct the layout of the *concrete* form. As a result, any operand (element of the concrete text that corresponds to a component of the abstract syntax, such as the body of a While loop) will be displayed either on a single line together with preceding and/or subsequent elements or, if this proves impossible, just by itself, indented on one or more lines. Details of the Cépage display policy may be found in Refs. 9 and 10.

This fundamental property of the display mechanism is a key factor in Cépage's ease of language adaptability. The grammar writer may concentrate on describing the syntax proper and let Cépage take care of the appropriate formatting.

If a specific nonstandard layout is desired, formating codes are available; they are also useful to describe languages with bizarre formats such as Fortran. Examples of formatting codes are the following:

$Ai: Move to absolute position i. For example, the "C" of a Fortran comment must appear in position 1 and should be preceded by $A1 in the concrete syntax clause.

$Ri: Move to relative position i. A relative position is computed with respect to the left border of the enclosing window, whereas an absolute position refers to the left border of the topmost window in the hierarchy.

): Start new indentation level.

(: End indentation level.

These and other formatting codes span a wide range of possible formatting requirements.

5.4. Using the Grammar for Parsing

The same grammar that drives the display mechanism is also used by the built-in Cépage parser. The parser relies on the concrete syntax clauses of the grammar to analyze input entered in text mode and build the corresponding abstract structures.

LDL does not place any special requirements on the class of acceptable languages. This is another key property in achieving Cépage's language independence. Other structural editors that embody a parser often rely on parser generators like Unix's YACC, which enforce strange restrictions such as LALR (k) or SLR (k) and make language description a lengthy trial-and-error process. In contrast, the Cépage parser is fully general; the underlying algorithm [11] applies to any context-free

grammar. This algorithm is also the key to parsing incomplete phrases (Section 2.2).

Beyond on-the-fly parsing of user input, the parser may also be used to input existing documents for subsequent manipulation by Cépage.

6. THE IMPLEMENTATION

6.1. State of the System

An initial version of Cépage was developed in Pascal at Electricité de France [12]. This version, running on IBM 370 hardware and 3279 terminals, was a prototype aimed only at validating some of the design ideas. The current version was undertaken late in 1985 by Interactive Software Engineering under Unix. It is now a commercial product. The system has been developed with portability in mind and runs on both System V and BSD variants of Unix. Versions for other systems are being investigated.

The current version is written entirely in Eiffel [13], an object-oriented language featuring multiple inheritance, generic classes, strict typing, assertions, and information hiding. The availability of Eiffel has proved to be a key factor in completing the construction of Cépage. Of particular importance was the Eiffel library, which provided ready-made, robust software components built around basic data abstractions such as lists and trees.

6.2. System Structure

The general structure of the system is given by Figure 6. Since Cépage was designed and implemented according to object-oriented techniques, the structure is best described by explaining the main object classes (abstract data type implementations) used.

The "abstract syntactic forest" is a hierarchical data structure that provides an internal representation of the document being manipulated. As mentioned above, Cépage itself is entirely language-independent. Thus an abstract syntactic forest is meaningful only with respect to a certain language. This language is known internally through another fundamental data structure, the "grammar graph." A Cépage component called LDLT (LDL translator) is charged with producing a grammar graph from an LDL grammar. Note that the grammar graph is functionally equivalent to a grammar; the user may think of the grammar as being directly interpreted by Cépage, although, for efficiency reasons, the interpretation in fact uses the grammar graph.

The "external form" is a representation of the document conceptually equivalent to the abstract syntactic forest but suitable for storage on external devices.

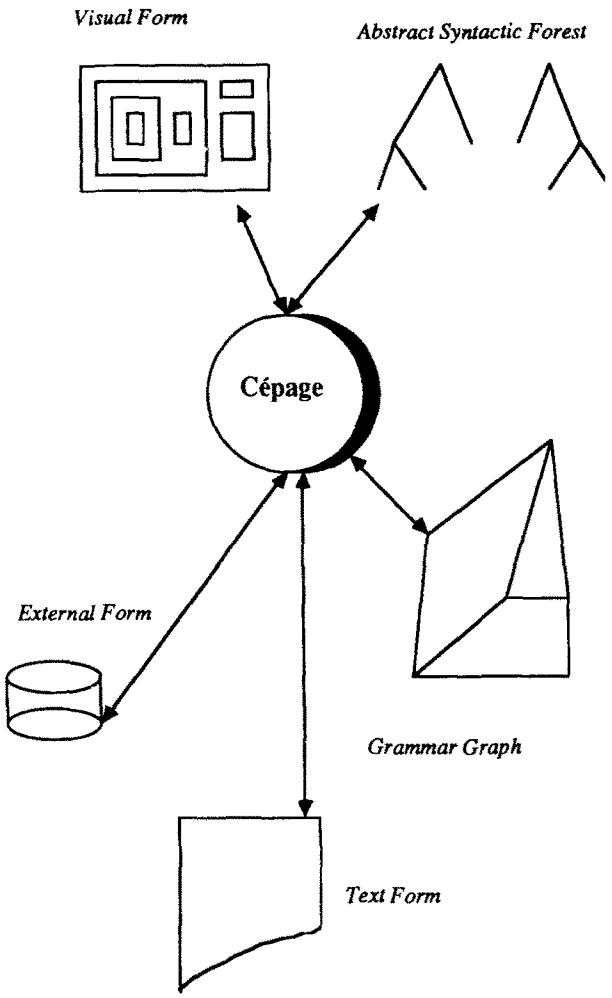


Figure 6. Cépage system structure.

The "visual form" is produced by the Cépage display system by mapping the internal document structure, as represented by the abstract syntactic forest, onto the available window space, choosing the elements that will be abstracted for lack of screen space, and performing the necessary formatting and indentation operations. As mentioned above, this process is entirely automatic; the user does not need to intervene but will get an appropriate view at each stage of the computation. The visual form is a list of windows processable by another of our software tools, Winpack (a general multiwindowing screen package), also written in Eiffel.

Finally, the "text form" may be generated for a totally expanded document for possible processing by other tools.

Note that one may envision applications involving more than one grammar graph. For example, two windows may be used concurrently to run two instances of Cépage, one for a PDL (Program Design Language) and the other for a programming language; if the grammar graphs are connected, changes to the former

may be immediately reflected in the latter. Another obvious application is to language translation.

7. APPLICATIONS OF CÉPAGE

As is now stands, Cépage may be applied to the following goals.

Program construction, relieving programmers of syntactic details and ensuring syntactic correctness at each stage.

Computer science education.

Production of standardized documents of just about any type: specifications, designs, technical reports, budgets, operating system configuration files, etc.

Implementation of programming and documentation conventions, such as standardized comments appearing at predefined places (for example, structured header comments in subprograms), and other style standards.

Language extension. If one wishes to add constructs to a language, for example While loops to Fortran, Cépage may advantageously replace a traditional preprocessor. It suffices to add a "While" construct to the LDL grammar with a concrete syntax that specifies the appropriate expansion (If and Goto). When the user chooses "While" from the instruction menu, the expansion will be performed immediately on the screen.

As a basis for other software tools that manipulate documents in terms of their structure, such as program or specification analyzers, profilers, complexity analyzers, and transformation tools. To make this goal a reality, many of the Cépage functions are accessible independently of the user interface through a library of primitives.

With respect to the last goal, we think that Cépage, with its data structures (abstract syntactic forest, grammar graph) and the library of routines that allows manipulating these structures, is a particularly worthy candidate to provide the standard interface for all tools that handle structured documents. For this reason we will publish the detailed specifications of the library for use by any such tools.

8. FURTHER DEVELOPMENTS

Current work on Cépage builds on the above results to extend the system and its applicability. A first range of extensions is to add semantic facilities to the syntactic stem. Both static semantics (for example, type-checking facilities) and dynamic semantics are being considered. The latter facility should make it possible to execute programs even if they are not completely expanded; if execution hits an unexpanded substructure, the system will revert to the editing mode to allow the user to enter a temporary stub allowing execution to proceed. Of course, dynamic semantic specifications are only meaningful for executable languages such as programming languages. This facility should make Cépage an invaluable tool for program testing and rapid prototyping.

The challenge with respect to semantics is to allow more advanced processing to be performed without impairing one of the key qualities of Cépage, the ease of language adaptation.

Another set of extensions concerns the possibility to use more than one grammar in the same Cépage session. This opens rich possibilities of using Cépage to support design in a high-level language in one window with immediate automatic code generation in another. Another promising application is as a tool for automatic translation.

A more advanced application would be to use Cépage to support program construction from general reusable patterns. The idea here is to treat parameterized software elements as abstract structures that could be modeled as LDL constructs. A sorting routine, for example, has a number of options (type of elements to sort, comparison criterion, swapping routine, sorting algorithm used); these could be viewed as components of an LDL aggregate construct *sort*. Cépage could thus be used as a tool for the construction of reusable software.

On a more short-term basis, we plan to use Cépage, and especially its standardized data structures, as a basis for adding a range of new software engineering tools, and to promote it as a standard for all software tools that manipulate structured documents.

ACKNOWLEDGMENTS

Jean-Marc Nerson was the co-designer of Cépage and its prime implementor. A key contribution was made by Reynald Bouy, who wrote the parser. The application to configuration files (Section 5.1) was suggested by David Farber.

REFERENCES

1. W. J. Hansen, Creation of Hierarchic Text with a Computer Display, ANL-7818, Argonne National Laboratory, Argonne, Illinois, 1971. (Also as dissertation, Computer Science Department, Stanford University, June 1971.)
2. V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, Environnement de Programmation Mentor: Présent et Avenir, in *Actes des Troisièmes Journées Francophones sur l'Informatique*, Geneva, 1981.
3. 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, E. Sandewall, eds.), McGraw-Hill, New York, 1984, pp. 128–140.

4. N. Habermann et al., *The Second Compendium of Gandalf Documentation,* Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1982. See also the special issue of *J. Syst. Software* (Vol. 5, no. 2, May 1985) on Gandalf.
5. T. Teitelbaum and T. Reps, The Cornell Program Synthesizer: A Syntax Directed Programming Environment, *Commun. ACM* 24(9), 563–573 (1981).
6. S. P. Reiss, PECAN: Program Development Systems that Support Multiple Views, *Proc. 7th Int. Conf. on Software Eng.,* Orlando, Florida, March 26–29, 1984, pp. 324–333.
7. S. P. Reiss, Graphical Program Development with PECAN Program Development System, *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments, Pittsburgh, 1984,* (P. Henderson, ed.), 19(5), 30–41 (1984). [This issue is also *Software Eng. Notes,* 9(3).]
8. M. Mikelsons, Prettyprinting in an Interactive Programming Environment, *SIGPLAN Notices,* 16(6), 108–116 (1981).
9. B. Meyer, J.-M. Nerson, and S. H. Ko, Showing Programs on a Screen, *Sci. Comput. Program.* 5(2), 111–142 (1985).
10. Interactive Software Engineering, Inc., *LDL User's Manual,* Tech. Rep. TR-CE-8/LD, 1986.
11. J. Earley, An Efficient Context-Free Parsing Algorithm, *Commun. ACM* 13(2), 94–102 (1970).
12. B. Meyer and J.-M. Nerson, CÉPAGE, a Full-Screen Structured Editor, in *Software Engineering: Practice and Experience* (E. Girard, ed.), North Oxford Academic, Oxford, 1984, pp. 60–65.
13. B. Meyer, Eiffel: A Language and Environment for Software Engineering, *J. Syst. Software,* 1987. To appear.