CEYX,a multiformalism programming environment

J.M. Hullot

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP105
78153 Le Chesnay Cedex
France
Tél.(3)954 90 20

Rapports de Recherche

N° 210

CEYX,
A MULTIFORMALISM PROGRAMMING ENVIRONMENT

Jean-Marie HULLOT

Mai 1983

CEYX,

A MULTIFORMALISM PROGRAMMING ENVIRONMENT

Jean-Marie HULLOT

This paper was presented at IFIP 83, 9th World Computer Congress, International Federation for Information Processing (Sept.83),Paris - FRANCE

ABSTRACT

CEYX is a system which provides mechanisms for:

- defining both the logical structure and the bit-level internal representation of arbitrary hierarchies;
- creating, accessing and modifying such hierarchies through high level functions;
- interactively editing such structures.

The system is an extension to LISP, and it allows the mixing of programs, which macro-generate hierarchies using the full descriptive power of the language, with actual data.

The structured editor has facilities to define extensions, which have been extensively used to tailor it to various special purpose applications, including VLSI design aids.

CEYX currently runs in Le_Lisp {1} on 68000 systems, MacLisp {12} and Franz Lisp {4}.

RESUME

CEYX est un systeme qui permet :

- de définir tant la structure logique que la représentation interne jusqu'au niveau des bits de hierarchies arbitraires ;
- de creer, acceder et modifier de telles hierarchies par des fonctions de haut niveau ;
- d'éditer interactivement de telles structures.

Ce système est une extension au langage LISP, et il autorise le melange de programmes qui macrogénerent des hiérarchies en utilisant la pleine puissance de LISP, avec les données courantes. L'éditeur structuré donne la possibilité de définir des extensions, possibilité qui a été utilisée pour l'adapter à de nombreuses applications particulieres, notamment dans le domaine de la CAO VLSI. CEYX fonctionne actuellement sur Le_Lisp {1} sur des machines à base de 68000, MacLisp {2} et Franz Lisp {4}.

1. INTRODUCTION

Editing is a major activity in computer software. Typing and modifying documents or programs is usually achieved via a text editor. A lot of efforts have been focused on writing such systems. The well known EMACS model {6,7, 8,14} seems to become a culminating point in this area. The most powerful feature of EMACS is its extensibility: it is written in a high-level programming language (LISP) and all basic constructions of the implementation are made available to the user so that he can extend the power of the editor by means of LISP programs calling these constructions as usual LISP functions. Another reason for its success is its user's interface based on a full screen video terminal.

Programming language oriented editors like INTERLISP {16}, MENTOR {2,3} and The Cornell Program Synthetizer {15} have given a new direction in researches. Instead of working on the textual representation of a program they work on its abstract syntax so that the user manipulates the structure of the program instead of pieces of text.

CEYX has been developped at INRIA as part of our design aids for VLSI circuits. Although they are eventually engraved on tiny pieces of silicon, the most striking charasteristic of VLSI objects is their enormous size. Hierarchical structuring of such objects is the only known way to master their correct design and efficient processing.

Moreover VLSI circuits must be considered from numerous viewpoints: architectural, functional, logical, geometrical ... All these visions must be communicated to the computer and this requires the definition and use of several formalisms. Most existing CAD systems provide ad hoc processors, one for each formalism, while CEYX deals with all these formalisms in a uniform manner.

Within CEYX it is possible to describe interactively any hierarchy and to produce a computer representation for it. Thus CEYX is not only a VLSI programming environment but a general system for manipulating hierarchies. Hierarchies are a central concept in computer's world: hierarchical data bases (file system, libraries, packages, ...), documents (chapters, sections, paragraphs, lines, characters, ...), programs ... Thus the domain covered by CEYX is quite large.

To achieve such a generality, we need a uniform data structure for representing text, trees, pictures, ..., in which the structure manipulation mechanism is identical for all types. However a text line and a tree are not dealt with in a same manner. Thus we must be able to associate to each type some particular semantical properties. Such object oriented programming style {5,17} is made available in CEYX: CEYX is a LISP extension which provides the user with a set of LISP functions to create and manipulate objects; such objects are the combinaison of a record like structure with a set of semantical properties. Objects are arranged by families in a hierarchical manner a la SIMULA so that they

inherit properties of their ancestors. The kernel of the editor consists in a basic management system for keeping pointers onto objects, associated with a screen manager. Printing and displaying properties are associated to objects either by the system or by the user so that each instance of an object knows how it must be displayed. Moreover the basic actions allowed at each step are recoverable from the type of the pointed instance: these are the semantical properties of the corresponding objects together with the ones it inherits. Saving an object can always be achieved by saving the LISP program generating this object. This allows us to work without parser.

Basic families of objects have conducted to the design of specialized working modes of the editor, the current mode being the one of the currently pointed object:

- LISP mode, for editing LISP programs. It closely resembles the INTERLISP editor {16} with a video interface.

- TEXT mode, which is closely related to EMACS {6,7,8,14}.

- TREE mode, which focuses on a special structure called tree.

There are a lot of subsystems derived from the tree mode. For instance:

- DIRECTORY/BUFFER/LIBRARY modes, for editing hierarchical data bases.

- FLIP mode. FLIP is a small language to describe color slides {10}.

CEYX is the basis of our VLSI design CAD system, where it can be used to edit all circuit representations:

- mask level representation, LUCIFER is the hierarchical language we use for VLSI mask specification {11},

- electrical connexions level, HELL is the hierarchical language we use for VLSI electrical design {13}, ...

Each time a new family of objects is defined, CEYX uses a basic way to edit it, which can be refined by adding particular properties to these objects. This makes the editor fully extensible by the user.

2. THE STRUCTURE GENERATOR

CEYX is divided in two main parts: a structure generator and an editor. All instances of structures defined through the structure generator can be edited through the editor. The entire system is written in LISP which is both the implementation and the command language. The structure generator provides the user with an expressive way to define abstract machines: they consist of a memory mechanism (record) together with a set

of LISP functions which are the basic actions that can be performed on such records.

2.1 Basic Structures

The purpose of the basic construction called DEFRECORD is to allow the definition of PASCAL-like records. A record is an object composed of named fields which store values. To define a record called RECT with fields xorg, yorg, xdim, ydim and color, one just has to evaluate the expression:

```
(defrecord RECT xorg yorg
               xdim ydim
               color)
```

It associates to the symbol RECT a record definition. This information is used by the instantiation function rmakeq and the access functions rputq and rgetq:

```
(setq mybox
      (rmakeq RECT xorg 0 yorg 0))
(rgetq RECT yorg mybox) -> 0
(rputq RECT color mybox 'green)
(rgetq RECT color mybox) -> green
```

Within CEYX it is possible to type record fields, that is to allow only certain kinds of objects to be the value of a particular field. Defining types is achieved by using the DEFTYPE construction which associates a boolean function to a symbol. For instance the type color is defined by:

```
(deftype color
  (lambda (x)
    (memq x
      '(black white blue green red))))
```

A general discrimination function called IS-INQ is used to determine if a LISP object is of a particular type; it calls the associated type function. For instance x is of type color if and only if:

```
(is-inq colorx) -> t
```

Records fields are then typed in the following way:

```
(defrecord RECT xorg~integer
               yorg~integer
               xdim~integer>=0
               ydim~integer>=0
               color~color)
```

assuming types integer and integer>=0 have been defined.

Moreover each time a record is defined a new type is defined having the name of the record. The discrimination function uses a general pattern matching algorithm to recognize if a LISP object is isomorphic to instances of a record. However it is not possible to discriminate between isomorphic records since record instances do not keep trace of their type. If one wishes to introduce such self-typed records, he has to use abstract records instead of records.

Abstracts records are defined by using the DEFABSRECORD construction. The only difference between this construction and the DEFRECORD one is that the produced instances are tagged with their type, and thus always know about this type. If RECT has been defined as an ABSRECORD and mybox is an instance of RECT, we have: (get-type mybox) -> RECT

2.2 Semantical Properties

As in Smalltalk {5} or when using flavors {17}, we would like now to be able to bind some functions to the structures. They will be the menu of actions one can apply to a given structure instance. To attach a display-text property to the structure RECT, whose action is to display text in the middle of the rectangle, we use the DEFSEM construction:

```
(defsem (RECT display) (rectangle text)
        <function-body>)
```

Activating a given instance of RECT is achieved by using the SEND construction: (send 'display <instance> <text>) which evaluates <function-body> in the appropriate environment. Note that in the presented form, the construction works only for abstract records where objects are self typed.

A structure can be defined as an extension of another structure. Such a composite object shares with the initial structure all its fields and semantical properties together with some new ones. In this case the access functions and the send construction will first look for the required field or semantical property among its personal one and then among the ones of its father and so on. It implements hierarchical dependancies. More elaborated ways of combining structures are allowed by the structure generator, but we will primarily focus on this one which fits especially well for designing the editor.

3. BUILDING THE EDITOR

We have presented the basic constructions of our LISP extension. This new section is devoted to show how we use them to build the different parts of the edition system. Our purpose in this part is not to describe specialized ways of editing. As we have said much work has been done in this area and the main problem is now to design a system powerful enough to allow the implementation of these ideas in a uniform way. We propose

an editor model, based on our structure generator, which has already permitted the implementations of numerous editing schemes.

3.1 Abstract Syntax Trees

One major feature of CEYX is the ability to define elaborated structures from lower level ones, using the LISP macrogeneration mechanism. For instance the record structure presented before is itself implemented as a particularization of a structure called pattern which allows to describe precisely the internal LISP representation of the structures. We show on an example how to introduce the specification of a programming language. First let us build a tree structure from a record structure. The basis of this construction consists in an abstract record called tree defined by: (defabsrecord tree sons) where the sons field is intended to point to a list of subtrees. Very often, we need some extra informations about trees. For instance we would like to give a name to a tree or to handle special fields; they are used for later evaluation of the tree structure. A special construction called DEFUNIVERSE is provided for that. It creates an extension of the abstract record tree, or of a previously defined universe of trees, with some extra named fields.

Example. LUCIFER is the language we use for describing circuits at the mask level. This language is based on a juxtaposition of rectangles. LUCIFER objects can be:

(box x y w h layer)
(union <obj>1 ... <obj>n)
(juxtx <obj>1 ... <obj>n)
(juxty <obj>1 ... <obj>n)

To each LUCIFER object is associated an origin and a bounding box. Atomic objects are colored boxes (box ...). Overlapping of several objects (union ...) is made by identifying the origins of these objects. (juxtx ...) means juxtaposition along the x coordinate of the component objects, thus the origin of the composite object is the one of the first component, and juxtaposition is achieved by putting aside the bounding boxes of all components. Similarly juxty is juxtaposition along the y coordinate.

Translations, rotations and mirror operations are achieved by:

(transl x y <obj>)
(rot n <obj>)
(miroirx <obj>)
(miroiry <obj>)

To build LUCIFER under CEYX, we first have to define a universe of trees called lucifer. We choose to implement the transformations as attributes of the objects, thus we need four attributes:

```
(defuniverse lucifer rot sym dx dy)
```

Any tree instance in this universe will then share all these fields.

Next, we define the operators of this language i.e. special lucifer trees having types such as box, juxtx, ... This is achieved through the DEFCONS special form which defines both a new structure of name box, juxtx, ... and a LISP function sharing this name. It takes as arguments a list of trees of some required form. For LUCIFER operators, we define:

```
(defcons box~lucifer
          (integer integer
           integer>=0 integer>=0
           layer~layer))
```

and for variable arity operators:

```
(defcons juxtx~lucifer (&rest lucifer))
(defcons juxty~lucifer (&rest lucifer))
(defcons union~lucifer (&rest lucifer))
```

Transformations are then implemented as LISP functions modifying the attributes fields. It is now easy to generate LUCIFER objects by LISP programs by calling the LISP functions so generated juxtx, juxty, union, box. For instance (box 1 2 3 4 'poly) generates a box instance in the lucifer universe and (juxtx <lucifer>1 ... <lucifer>n) is used to juxtapose lucifer structures instances.

3.2 Evaluation

Some aspects of structure evaluation are:

- either modifying by programs some attributes in the structure,

- or generating a particular representation of this structure (textual representation, graphical representation, ...),

- or generating a new structure which represents a particular aspect of the initial structure. This last kind of evaluation is widely used in the VLSI area: from an abstract representation of the VLSI circuit under development, all levels of description (functional, logical, geometrical, ...) are macrogenerated via a simple tree traversal control over the structure.

Structure evaluation is made by means of LISP programs using access functions and semantical properties of structures. For instance, textual and graphical representations will be produced by a general display algorithm walking through the structure and calling for each substructure the associated printing property. These properties are attached to

structures as semantical properties which are functions activating printing virtual machines.

More specialized evaluators have to be defined by the designer of a particular subsystem.

3.3 Editor Internal Structure

The editor is itself an abstract machine implemented as an abstract record called EDITOR. As any object it maintains pointers on other objects and has some semantical properties which are the commands of the editor. It may be defined by:

```
(defabsrecord EDITOR
              screens curscreen
              buffers curbuffer)
```

At the initialization of the system, an instance of this object is created. All structures will be attached as descendant of this initial structure.

The screens field points to a list of screens objects which are abstract representations of concrete screens. We distinguish between three families of screens: alphanumeric screens, bitmaps and color bitmaps. Each type of screen object has some characteristics memorized in named fields and some semantical properties which define the associated output virtual machine. All CEYX outputs are directed through these virtual machines. Installing a given terminal of one of these three families is achieved by giving appropriate definitions to the semantical properties.

Buffers are also abstract machines which keep named pointers on the structures being currently edited. They manage a pointer, called the current pointer of the buffer, on subparts of these structures. A list of buffers instances is kept in the buffers field of the current instance of EDITOR. The curbuffer field contains a pointer on one of these buffers; it is called the current buffer.

There are two basic semantical properties defined for buffer objects:

- DOWN-BUFFER, which takes as argument an occurrence. It may be either the name of a field in the object instance pointed by the current pointer, or a number when a list is pointed. The associated action is to move the current pointer on the required substructure.

- UP-BUFFER, which conversely moves the current pointer to its structural father.

They are the basic functions for moving around structures; they represent the first part of the editing mechanism. Specialized way of moving, like moving on LIST or TREE structures, are provided as standard tools.

Searching mechanisms have been defined as LISP functions using these basic properties together with general pattern matching algorithms.

3.4 The General Display Algorithm

Screens objects point to a hierarchy of windows, which are abstract representations of regions on the physical screen. The user manipulates the windows hierarchies by the standard mechanisms of the editor (he can do the same with buffers and screens). To each window instance is associated a buffer in the current buffer list.

The role of the general display algorithm is to maintain on each active screen a representation of all structures on which the attention is centered: all windows are displayed together with some accurate representation of the structure pointed in the associated buffer. What a window looks like, depends on its graphical display properties. It may be pictured with a shape, or a who line or in some color ..., according to these properties activated by the display algorithm. Structures are pictured in windows according to their own semantical display properties. To each buffer is associated a visualisation pointer which indicates which subpart of the structure is to be displayed. Moreover the representation is holophrasted in the sense the structure is displayed only up to some level.

3.5 User Input

The attention of the editor is centered on the current pointer of the current buffer. Changing the value of the current pointer is primarily made by using a pointing device. To recover the expression attached to a given position on the screen we use the general display algorithm in a special mode which does not generate any output on the screen: it just generates the coordinates where objects are pictured and compares them with the currently pointed coordinates.

To each type of structure is associated a set of actions (its semantical properties) which can be applied to instances of this structure. These actions are the basic menu of the structure. Moreover each structure inherits of the menus of its ancestors in the type hierarchy. For instance all LUCIFER structures are TREE structures, thus all TREE actions become automatically LUCIFER actions. The most general way used by the editor to activate these actions is via question/answering through the QUERY-SEND construction. It works essentially as SEND but applies always to the current pointer. The user has then only to give the name of the required action; if additional arguments are needed the editor will ask for them by their name in a special window.

This can be facilitated by associating actions to keys. Then when the user hits a key, the corresponding action is applied to the current pointer. Still a same action name may have different meanings for distinct types of structure. For instance the forward action will move

the current pointer to its rightmost brother for a TREE, to the rightmost character for a text object or follow any previously defined scheme for other objects. The user can change key bindings interactively.

Another way is to display selected menus in special windows on the screen. Since menus are also structures, they are manipulated as any other objects. One can point on a menu item as it does for any substructure, using the pointing device.

4. PORTABILITY

CEYX has first been developped on Multics MacLisp {12}. The first implementation of the editor has been made on top of Multics Emacs {6,7,8}. It has then been moved on the Le_Lisp system {1} on MOTOROLA 68k based microprocessors and on VAX. It runs also on VAX Franz Lisp {4}. Moving from one system to another is achieved on a part via the LISP macrogeneration mechanism and on another part by giving new implementations to the basic virtual machines used in system dependant subparts of CEYX.

5. CONCLUSION

More than an editor, CEYX is a system which gives all basic capabilities to implement particular editors together with a general editing scheme. Each time new objects are defined through the structure generator, CEYX gives a basic way to edit them. Moreover the user has the full power of LISP and CEYX extensions to program new editing schemes.

We use it as the basic management system for our LISP based workstation. A lot of specialized VLSI processors are built on it {11} as well as special modes of general use like DIRECTORY/BUFFER/LIBRARY modes, LISP mode, TEXT mode, FLIP mode {10}, ... A very exciting direction of research, to the author's point of view, would be to build a CEYX based management system for operating systems written in LISP as the one existing on LISP Machines {17}.

ACKNOWLEDGEMENT

We are very grateful to Bertrand Serlet and Jean Vuillemin for their contributions to this work.

REFERENCES

{1} Chailloux J., Le_Lisp 68k: Le Manuel. INRIA report to appear (1983).

{2} Donzeau-Gouge V., Huet G., Kahn G., Lang B., Levy J-.J., A structure oriented program editor: a first step toward computer assisted programming. International Computer Symposium, North Holland Publishing Co, (1975).

{3} Donzeau-Gouge V., Huet G., Kahn G., Lang B., Programming environments based on structure editors: the Mentor experience. INRIA research report 26 (1980).

{4} Foderaro J.K., Slower K.L., The Franz Lisp Manual, (1981).

{5} Goldberg A., Kay A., Smalltalk-72 Instruction Manual, Xerox Palo Alto Research Center, SSL 76-6, (March 1976).

{6} Greenberg B.S., EMACS Text Editor User's Guide, Honeywell Information Systems Inc.,(1979).

{7} Greenberg B.S., EMACS Extension Writer's Guide, Honeywell Information Systems Inc., (January 1980).

{8} Greenberg B.S., Multics EMACS: an experiment in Computer Interaction. Proceedings, Fourth Annual Honeywell Software Conference Honeywell Information Systems, (March 1980).

{9} Hullot J.-M., CEYX Reference Manual. Preliminary Version (May 1982), INRIA report to appear (1983).

{10} Kahn G., FLIP user's manual. INRIA technical Report (June 1981).

{11} Levy J.-J., On the LUCIFER system. Advanced course on VLSI architecture, Bristol U., (July 1982).

{12} Moon D., MACLISP Reference Manual, MIT,Cambrige, Mass., (1974).

{13} Serlet B., HELL: Manuel d'Utilisation, INRIA report to appear (1983).

{14} Stallman R.M., EMACS: The extensible, customizable, self-documenting display editor, MIT, AI Memo 519, Cambridge, Mass., (June 1979).

{15} Teitelbaum T., Reps T., The Cornell Program Synthesizer: a syntax directed programming environment. CACM 24,9 (pp 563-573), (Sept 1981)

{16} Teitelman W., INTERLISP Reference Manual, Xerox Palo Alto Reseach Center, Palo Alto, California (1976).

{17} Weinreb D., Moon D., LISP Machine Manual, 4th edition, (July 1981).

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique