Liberating programming from the ASCII view

Harald WERTZ & Damien PLOIX

Département Informatique
Université Paris 8
2, rue de la Liberté
93526 Saint-Denis Cedex 02
FRANCE
hw@ai.univ-paris8.fr

Abstract

We present the xbVLISP programming environment currently under development in our laboratory which integrates textual, graphical and analogical representations of programs. This environment should be of help during the development of programs (including the handling of unfinished and/or erroneous programs), the observation of their execution (through the capacity of animating the various representations), their documentation and annotation (giving the user and the tools the possibility to enrich the representation of the program with textual or graphical comments, which then may be used by other tools), the explorative reading of programs and their maintenance (through such tools as integrated outstanding task lists, animating execution between two points of the program or displaying visual representations of selected characteristics).

Keywords: programming einvironments, multiple representations, observation of programs, documentation and annotation of programs, program maintenance, program animation, version control, views of programs, explorartive programmming.

1. Introduction

The xbVLISP programming environment developed at the University Paris 8 is an ongoing project which generalizes and formalizes the distinction between *representations* of programs and *views* of programs: at any given time there exists only one representation, but there may exist as many different views, each one highlighting different characteristics, as the user chooses. This environment integrates, in a unique framework, low level programming tools, such as text- and structural editors, tracers, breakpoints and on-line documentation, with high level programming tools, such as version control, automatically generated graphical representations of program- and data-structures, dynamic updating of the on-line help facility based on syntactic and semantic analysis of the program under development, generalized annotation facilities and analogical representation of programs.

A special emphasis is put on the need:

1) to have multiple views on the structure of programs and their data. These views, which may dynamically change during the execution of programs, include graphs of the data- and control-flow, program schematas or programmable analogical representations. The latter may be as varied as the automatic construction of cartoon faces whose features (form of the face, the ears, eyes and mouth, absence or presence of marks, etc.) are determined by criteria defined by the user, or the display of landscapes where the height of the mountains, their placement, their color, etc, translates such features as intensity of use, relation to other parts, or purely syntactic characteristics. This kind of representation is widely used in the display of statistical data and permits an immediate *visual* recognition of the underlying structures, without the necessity of entering into the details of the program. Naturally, several different views may be displayed and updated simultaneously.

2) to dispose of tools able to consider a program on different levels of abstraction: from the concrete textual view to maps, charts and such analogical views mentioned above. This multitude of views at different grain sizes permits careful interactive walkthroughs where the views are dynamically updated depending on the user's choice of level of detail.

3) to share a unique internal representation — it is only the views of this representation which change through the different tools. This criterion permits cross-fertilization of the different tools. By this we understand that knowledge about the program derived by one tool will be immediately available to all the other tools. In this way, over time, through the use of various tools, the representation of programs becomes more and more complex, including more and more descriptions.

4) to be able to use each tool for both the passive display of information as well as for interactive exploration. This last criterion is esssential, since we want — in the near future — to use our kind of analogical representations to translate the program into a virtual reality which will be explored and modified interactively by the programmer, transforming the activity of writing programs in an activity of arranging and sculpting pre-existing structures. We are convinced that this is the only way to construct, explore, understand and modify those large programs whose complexity is overwhelming to the individual programmer working only on textual representations.

In the following paragraphs we give a more precise description of the architecture of our programming environment, as well as a set of example representations we consider particularly useful for immediate recognition of chosen features and structures.

2. xbVLISP: the core environment

xbVLISP [Wertz85, Sendoya92] is a dialect of LISP which we have designed for the experimental development of programming environments. In addition to the standard features of LISP (mainly: program-manipulable representation of programs, self typing

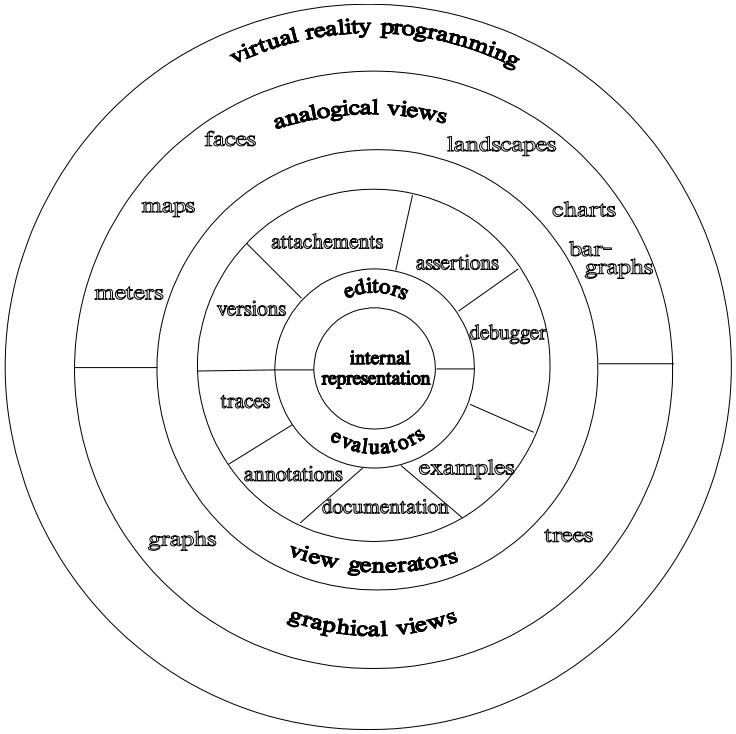


Figure 1 *global organisation of xbVLISP*

data structures, availability of an interactive interpreter), xbVLISP includes capacities to attach expressions to all structures of programs. Attached expressions are specified to be evaluated before and after the evaluation of the expression itself or at the consultation of data-structures to which they are attached. Clearly, this is xbVLISP's basic mechanism for the implementation of execution observing, monitoring or measuring tools. It is also the basic mechanism allowing easy implementation of such tools as input/output assertions (assertions attached to procedures or functions which are verified at each activation of the corresponding unit), input/output examples attached to the same units (whose correction is automatically verified after each modification of the corresponding code), tools for incrementally constructing a documentation of a specific execution of the corresponding program or part of program and tools for version control and dynamic version switching. Any number of expressions may be attached to any point or object in the program. Naturally, attachments may be attached to attachments. Since most of the attachments serve to observe the behavior of a program, this is giving the programmer the opportunity to observe the observer. xbVLISP is used as a teaching tool for advanced programming classes and as a programming tool by our artifial intelligence group. In this way the utility of new tools is easily estimated using the feedback provided, and we, the developers, are not (as is usual) the only users.

In Figure 1 we give an overview of the general structure of xbVLISP. At its center is the unique internal representation of the program. This is the basis on which all the tools act. The internal representation is both a standard LISP list-structure, enriched by the annotations (described in [Wertz85]) and a kind of distributed associative data-base containing information about all LISP objects, gathered by the different tools, and demons1 attached to those same objects. The information may concern symbolic descriptions of the object, indexing information indicating who uses this object and what other objects are used by it, comments describing the role and function of an object, which will be displayed by the on-line documentation, examples of its use, syntactic

(1) Demons are procedures activated automatically whenever a given condition about an object is true or whenever its value changes, or whenever the object is addressed.

and semantic definitions useful as well for the help facility as for the editors, in short, all information about the program derivable either by the tools or directly given by the programmer. Computationnally, the internal representation functions similarily to a blackboard system [Erman80]: all tools and demons find the necessary information in this representation and can delete, add or modify information in this internal representation.

The first layer built on top of this internal representation is composed of the two classes of objects which handle its construction and modification: the editors and the evaluators.

xbVLISP offers four types of editors: a form editor for the construction or modification of data (for example a list can be modified by graphically displaying its structure and then graphically changing pointers or cell contents), a structure editor for the construction or modification of programs which includes a standard emacs-like text editor and a graphics editor, a drawing program, permitting graphical annotations of programs. All these editors act directly on the internal representation of the program and data. The editors keep a history of the changes, thus giving the necessary foundation for version management which will be described later. In parallel to the editing process, the editor constructs a description of the edited objects. This description is the result of a syntactic analysis of the new piece of program and its integration in the already existing one. Thus for example, if the programmer defines a new function, the editor will invoke the syntactic analyser and add to the internal representation the modified indexing information, the syntactic description of this new function and the description of the types of the expected arguments. If such a description already exists and if it is contradictory with the newly derived one, the editor will inform the user of this difference and will mark all the dependent parts of the program as possibly subject to errors. This information may be used later by the debugger.

On the same level as the editors, xbVLISP offers the tools for evaluating expressions. Two evaluators are present: a normal evaluator, able to handle all the standard LISP evaluations at the same time as the evaluation of the attached expressions, and a symbolic evaluator, able to continue evaluation on symbolic descriptions whenever normal evaluation is not possible. Thus when the programmer wants to test the coherence of a partial program2, the normal evaluator computes the values of expressions until the encounter of, for example, a call to an undefined function. At that point, the user can choose between aborting the computation, defining the function or correcting the program and normal computation continues afterwards, interactively entering a return value for this function call, or asking the computation to be continued symbolically. In the latter case, the symbolic evaluator takes over, using the non symbolic values computed so far and generating new symbolic values whenever necessary. For the computation of symbolic values, the evaluator uses symbolic descriptions attached to all standard LISP functions and the symbolic descriptions constructed during the editing

(2) We call a *partial program* a program which is either intentionally unfinished or whose execution would, in a standard computation, abort with an error.

of the program. All information derivable from the source program is available during run-time!

In addition to the evaluation of expressions and their attachments, the evaluators complete the documentation of the program with information not easily derivable through static analysis, such as computed function calls or dynamic measurings. This permits, for example, to construct flow diagrams of specific executions of programs, thus giving the programmer a compact view on the history of a specific computation.

3. xbVLISP: textual environment tools

The next outer layer is composed of the textual tools xbVLISP offers. All these tools are implemented using the attachment facilities built into the language. Attachments are an original feature of xbVLISP. They may be associated to lists, that is mainly program code and sometimes data, through special cells which we call quasi-cons-cells. These quasi-cons-cells exist only at those points of the program where they are needed, so that the annotation capacity does not overly increase the program size, as would be the case if we would have them implemented with, for example, three address cons-cells. Quasi-cons-cells are cons cells whose contents are negative addresses.

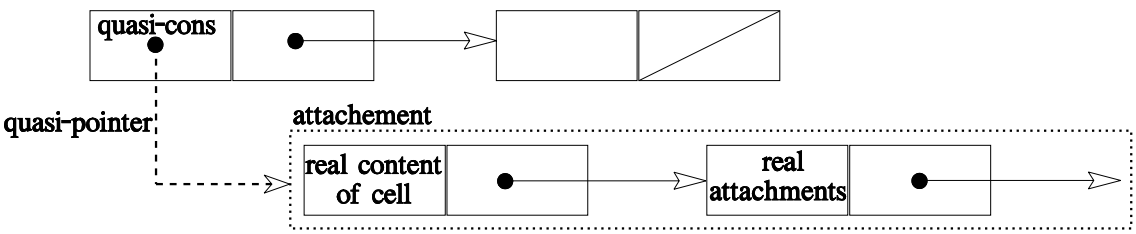


Figure 2 *a quasi-cons-cell*

In this way, whenever the evaluator encounters such a cell, the ill memory reference error engendered by this negative address is trapped and the negative address is translated into a positive one, which is the address of the attachment3. The attachment itself is a list, the CAR of which is the real content of the cell containing the quasi-pointer and its CDR is a property list, that is a set of keys and associated values. Standard keys available are **Entry**, to which the activities associated to the entry point of this piece of code are listed, **Exit**, the complementary key for exit activities, **Comment**, the key under which the comments are kept, thus permitting run time availability of comments, **Assert-Entry** and **Assert-Exit**, which are giving the input and output assertions, **Version**, which attaches other versions of this part and **Trace** for tracing information.

(3) In this way, program execution is not slowed down, since only at those places where an attachment was added the mechanism is activated.

Attachments may also be associated to LISP atoms. In this case they are in a special field of the atom, the internal-value field, which is also organized like a property list. We choose special fields because we want this attachment to be invisible to the normal user of xbVLISP, in order not to overload his vision of annotated programs. Except the comment attachment, attachments are only visible when explicitly asked for by the user. Attachement to atoms are primarily used for tracing and execution observing tools. They may also be used as a basis for *access oriented* programming [Stefik86].

In all cases, whenever the evaluator encounters an attachment, it searches the attachment's property list for additional activities and, if present, computes them before (or right after) the normal computation.

3.1. version management

As discussed in the paragraph describing the editors, no modification of the program is destructive, that is: all modifications can be undone, since modifications are annotated with the date and time the modification was done, the name of the programmer who did it and the previous version is kept as an attachment. Since keeping the entire history of the program's construction is soon very costly in memory and disk space, the programmer has the possibility to decide when to begin on a new version. If he does so, the actual state of the program becomes the new version. Future changes construct subversions of this new version, each subject to be undone, until the next creation of a new version, where the same mechanism takes place. Once a new version is created, one can go back to the previous version but one can not reconstruct subversions of the previous one.

Version attachments are only created (or deleted) through the editors and are not supposed to be changed during the execution of the program. The default behaviour of the evaluator is to always execute the current version of the program. But the programmer has the possibility to specify which version is the current version and he can dynamically (and locally) change the version. This permits experimenting with combinations of different versions.

This also permits a style of programming where the same piece of code can be used for different tasks, the latter corresponding to different versions of the program. To some extent, this corresponds to using the same expert system with different rule sets, since one case where this intentional version construction is very useful is when the programmer needs to use the same control structure for different activities, such as, for example, in the construction of a normal and a symbolic evaluator.

Version management is on the level of individual functions or procedures. A slight extension of this permits project management. A project is a snapshot on a program at a given moment of its development, that is: it is a collection of functions and data where for each entry the current version is specified.

3.2. assertions

Assertion attachments may be associated to functions, arbitrary pieces of code and data. In all cases they express constraints that should always be satisfied by the corresponding unit. We distinguish between input- and output assertions. Input assertions are logical expressions which should be satisfied before evaluation of the corresponding unit, output assertions should be satisfied after evaluation.

An input assertion for a variable may, for example, assert that the value of the variable should be in a given interval or of a given type. Then, whenever the program intends to change the value of this variable, the evaluator will first verify the correction of the assertion for the new value and if it is not satisfied, the user is informed and, if the user wishes, he can enter into a break where he has the opportunity to inspect the state of the computation. In the case of an output assertion, the value would first have been changed.

Assertions may be any LISP expressions which return either true or false. They are extremely useful during the development of programs since it is through this assertion facility that the programmer expresses constraints which he believes satisfied once the program is finished and supposedly correct, but which he can not guarantee while the program is still in an experimental and developmental phase.

Assertions are also useful for the exploratory reading of programs, i.e.: reverse engineering, since they give useful information about the programmer's intentions. The editors have special commands to make the assertions visible, since they are hidden by default, as all attachments. They resemble comments, except that they are *active* comments.

Like all attachments, the activation of assertions is possible only in the *careful* mode of the evaluator. In this mode the evaluator has exactly the same behavior4 as in normal mode, except that all attachments are activated whenever necessary. Since mode change may be done dynamically, this permits computation of the already finished parts of programs at a normal speed and to enter in the additional activities only at necessary places.

3.3. generalized Entry and Exit activities

We have generalized this notion of input/output assertions in giving the programmer the means to attach arbitrary expressions to functions, pieces of code and data. In the same way as for the assertions, whenever an entry or exit activity is attached some-

(4) This includes iterative interpretation of tail recursive functions, even if the function is traced or has input/output assertions attached. We think that too much programming support or debugging facilities do not sufficiently take care of this compatibility between development environment and production environment. At the SIGPLAN conference on debugging [ACM83] the wonderful term *Heisenbug* was coined for debugging environments which change the normal behaviour.

where, the evaluator activates it either before or after performing the associated action. The difference consists in that these activities do not need to return true or false: it may be any computation.

This was initially used for the tracing and stepping facility. The user attaches a start-step mark to the point where wants to enter into a step by step execution. If there is a specific point where he wants to stop this mode of execution, he attaches an end-step marker to that second point. He proceeds in a similar way for entering and exiting tracing the execution. Note that both the stepper as well as the tracer can function on different grain sizes, depending on what the user wants to observe. If one wants to observe, for example, only external function calls — that is calls to functions which come from outside the function itself — the trace facility attaches a conditional entry and exit activity to the function, if one wants to trace all calls to a function, the entry and exit activity will be unconditional, and if one wants to trace only one particular call, the tracing facility attaches entry and exit activities to the specific point in the program which corresponds to this call. In all cases, the tracing and stepping facility opens a new window in order to not distroy the contents of the actual LISP interaction window.

Once the stepping and tracing was implemented, we needed other observation capacities, such as tracing the content-change of a variable or measuring the intensity of use of a given part of the program. The natural way to do this was in annotations. Since we did not want to create too many specific keys, we introduced the generalized entry and exit keys. It is them which are used by all the higher level tools.

The entry and exit attachments may be freely used by the programmer for his own activities. We use them also as a programming tool to separate input/output programming from the purely algorithmic part. In this use, similar to the Model-View-Controler paradigm of Smalltalk programming [Lalonde91], the programmer first constructs his algorithm and, once it is correct and running, he attaches the input/output commands to the code. In this way, he does not modify the algorithm by introducing more sophisticated input/output, and he is always assured that whatever he adds, the algorithm will still be correct.

3.4. a (very) small example

Figure 3 gives an example of a small LISP function containing different attachments. There exist 2 versions of this function, a first one:

```
(define example (var)
      (cons var (cdr var))
```

and a second one:

```
(define example (var)
      (if var
            (cons var (example (cdr var)))
            nil))
```

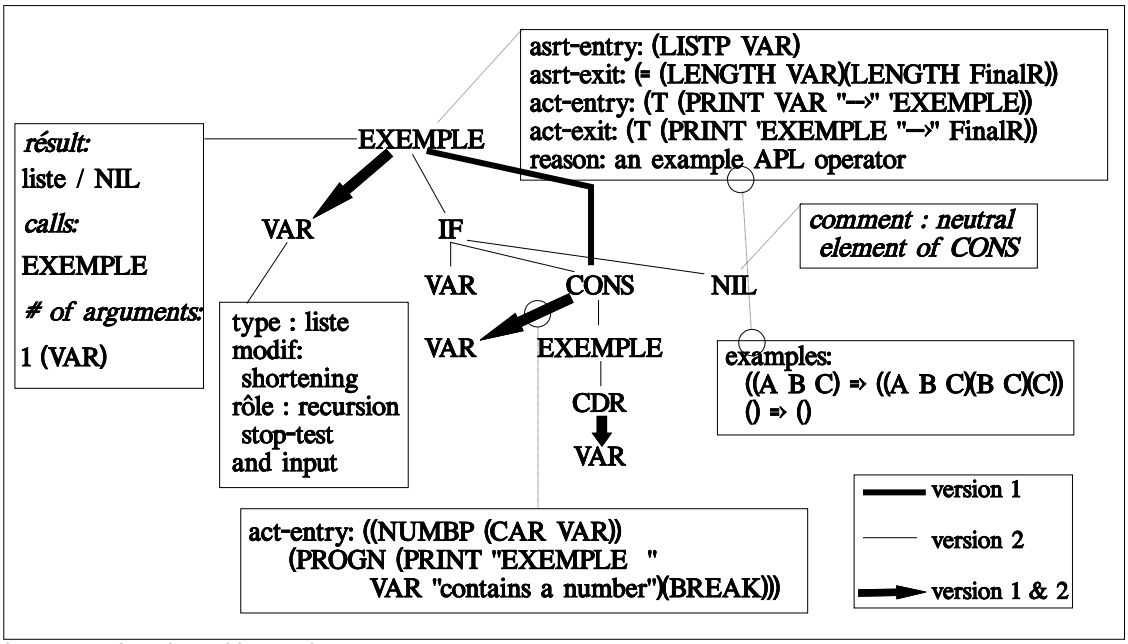


Figure 3 a function with attachments

where the original body has been embedded in a conditional expression and which contains a recursive call. It is clearly visible that the internal representation shares those parts which are common to both versions.

The box in the top right corner of Figure 3 displays the input/output assertions and generalized entry/exit activities attached to the function. It also contains a specific comment giving the reason for this function. Whenever the programmer defines a function (either interactively or with the editor) xbVLISP asks to give such a comment. While reading programs, the editor can restrict the display of functions to just those comments. There are also two examples attached to the function, giving a hint about what the function is computing. This too is a kind of active comment, since the examples are verified automatically after each modification of the function.

The upper left box contains information about this function which was automatically derived by the system. Specifically, it tells that the return value of this function is always a list or NIL, that this function calls itself and that it has one variable called VAR. To this variable VAR is also attached some information derived by the system: that its value should always be a list, that the function changes the original value so that the list shortens, and that it has two roles: it is the argument of the function and the recursion stop test.

There is one atom which has an attachment: the atom NIL has a comment attached indicating that it is considered to be the neutral element of the CONS function and there is one piece of code (the first argument of the call of the CONS function) which has a generalized entry activity attached. Note that all entry/exit activities are written in the form <test-part action-part>, thus the activity is started only if the test returns true. Here, a message is printed and the programmer gets into a break point if the first element of the variable VAR is a number, while the rudimentary tracing attached to the

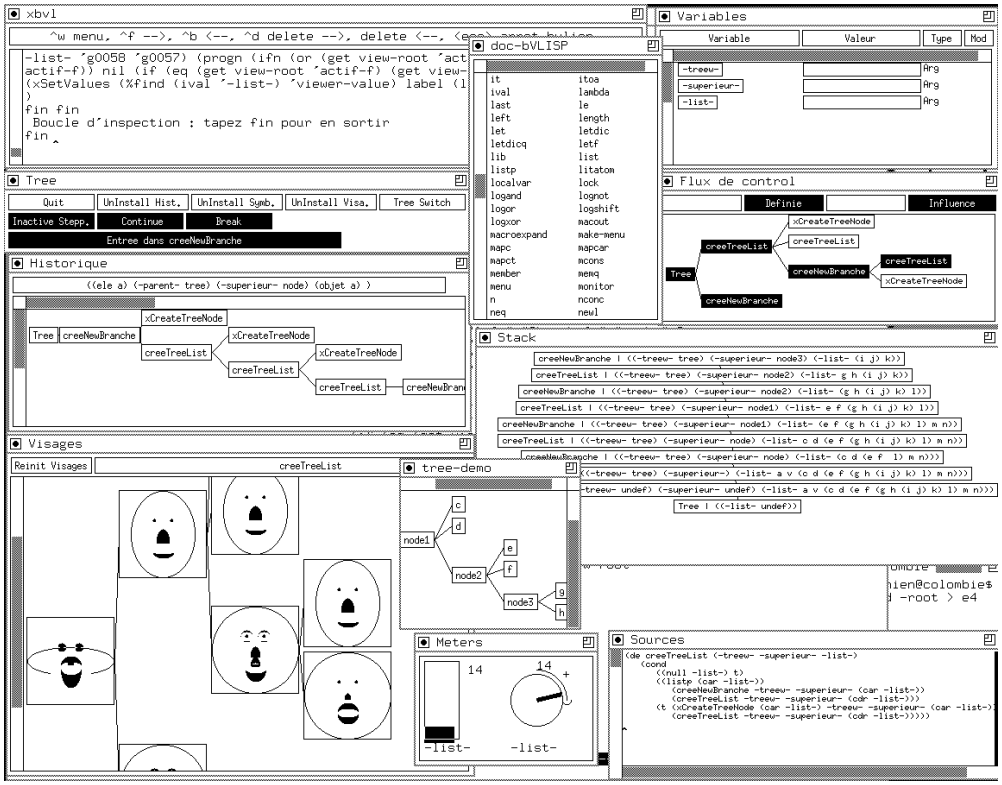


Figure 4 *t*ypical display sreen with xbVLI*SP*

function EXAMPLE is always activated, since the test is reduced to T, LISP's way of saying "true".

4. xbVLISP: graphical environment tools

The next layer of the xbVLISP programming environment is composed of the view generation tools (cf. Figure 1). This layer is mainly an interface with the X-window system [Nye90] permitting to use and program, in LISP, the entirety of the X-window system commands. This is a completely embedded sublanguage giving access to the Xlib library, to the X toolkit and to the X widgets. All graphical interaction is done through this interface, conforming to the X window system's client/server methodology. All of xbVLISP's window management is done through this interface.

Figure 4 shows a typical screen while working with our environment: the top left window is xbVLISP's interaction window. It has a pull-down menu for the most standard activities during normal interaction and on top a line reminding standard editing commands. These editing commands are the same as those of the xbVLISP editor and one can actually select parts (or all) of the window to write in a file for later use.

The small window underneath, labeled "Tree" is a environment control window for the careful execution or reading of programs. The label indicates the name of the program being run. The first line of this window is a row of buttons activating or deactivating special environment tools. We will come back to this below. Here, the user, after installing the history facility, the symbolic views and the cartoon faces representation, is interactively stepping through an execution. The precise point of the program where

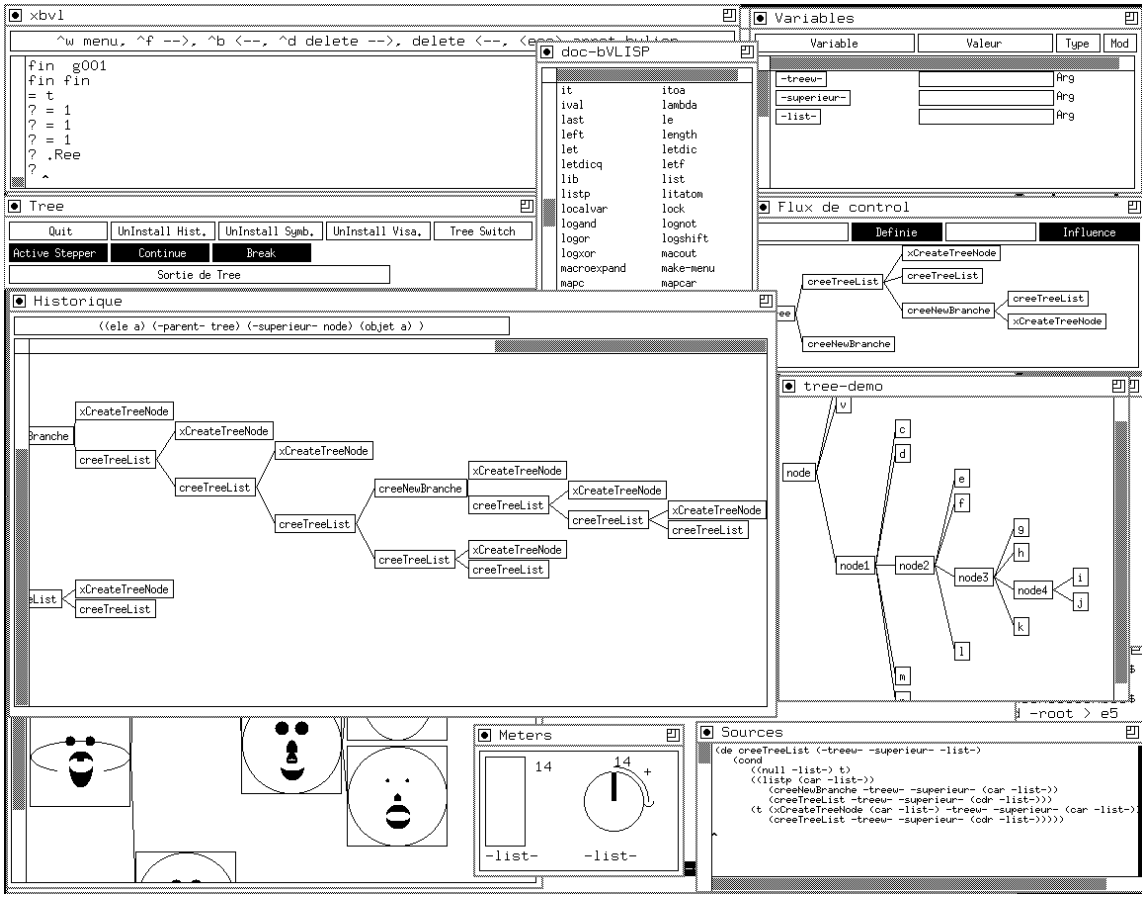


Figure 5 another typical display screen with xbVLISP

the execution currently is, is recognizable in the window labeled “Flux de control” (control flow). This window displays the control flow of the program. The boxes in inverse video indicate the currently active functions. The name of the function it is in currently is repeated in the last line of the environment control window. The contents of the variables of the last active function are displayed in the “Variables” window in the upper right corner.

The window in the middle of the upper portion of the screen, labeled “doc-bVLISP”, is a help window which, when the user clicks on one of the items listed, opens a window with either the manual entry of the corresponding item — if it is a standard lisp object — or a symbolic description of the item, if it is a user defined lisp object.

Below this window is a display of the current state of LISP’s control stack, another snapshot of the execution of the program.

The window labeled “tree-demo” displays a graphical representation of the data structure on which the program is currently working. As all the other windows, this too is continuously updated, so that the programmer can visually follow the effects of each step of the computation.

In the lower right corner of the screen is the “Sources” window which displays the source code of the function chosen by the user in clicking on one of the rectangles in either the flow-control or in the history window.

This last window, labeled “Historique” (history), is shown in the middle of the left side of the screen in Figure 4 and, at a later state, it is the large central window in Figure 5. It continuously updates a graphical representation of the specific control flow during the actual execution. This permits an easy walk through the program execution either after the program has finished execution (a kind of post-mortem dump) or during an interactive inspection while in a break-point. The difference between this window and the stack-window, which also may be used for inspection, is that in the stack window one can only visit states which are still active, while in this history window all previous computations may be inspected.

The small window labeled “Meters” displays analogical representations of chosen variables. In this case the length of the variable **-list-** is displayed once as a bar-chart and once as a gauge.

The window with the cartoon faces is another representation of the control flow. In this one, instead of giving boxes with the names of the functions, chosen characteristics of each function are displayed in an analogical manner. We will come back to this in the next few paragraphs.

Figure 5 shows the same screen after the execution of the program has finished when the programmer is inspecting the history of the computation.

4.1. the LISP interaction window

xbVLISP is a standard, interpreter based LISP system. The LISP interaction window is its main interface with the user. Each activation of xbVLISP creates one interaction window. Here commands are entered and the returned values are displayed. All this interaction takes place through a standard, emacs-like editor. This means that one can go back, edit and/or copy previous commands for other activations. The top line of this window is a kind of constant reminder of the most usual editing commands. It posseses a pull down menu through which programming environment tools, such as the help facility and the control window, are activated. Either the entire content or selected parts of the window may be saved whenever the user wishes. For later replay, he can also save all or only selected commands: this permits interactive construction of “batch” files. Note that all text windows use the same editor and offer the same capacities.

4.2. the help windows

All programmers, advanced as well as beginning ones, meet situations where they need some help, either about the system and the language they are using or some reminders about characteristics of the program they are currently working on. Naturally, all of the environment tools may be considered as help facilities. In this paragraph we consider only those tools which give help by consulting either pre-existing documentation or documentation the system derived about the user program. The help facilities we offer are:

- a menu driven help window listing all the entries of the xbVLISP manual. Clicking on one of the entries will display the corresponding paragraph in a special pop-up window. All manual entries are organized in three parts:
  1) a short description of the command, function or tool,
  2) some simple examples of its use and
  3) the long explanation corresponding to the real entry of the programming manual.

  Most of the time, a short description and two or three well chosen examples are enough information. Note that inside the help window, the user can try out his own examples which may use data structures and functions of his own environment. This permits a kind of fast checking of one's understanding, and this directly related to his actual task.

  At the end of this menu driven help window, xbVLISP continuously inserts user defined LISP objects. If the programmer chooses one of those entries, the help window displays the special comment and the examples the programmer has attached to this object (if this exists), the symbolic description the system automatically constructed during the editing or file-in of this object and information about where this object is defined and used and which other objects are used by it.

- a context sensitive help which is attached to each kind of special environment window. This help facility also displays paragraphs of the xbVLISP manual, but this time with information about the use and the possibilities of the environment. For example, if the programmer is in the LISP interaction window, the context sensitive help will offer him a menu with entries for how to edit and how to save all or parts of the contents of the window. Or, another example, when the programmer encounters an error during the execution of a program, he can ask the system to propose possible corrections. This facility is useful when the error is due to a typing error or to a "simple" programming error. This automatic error correction facility is described in [Wertz87]. Still another example of context sensitive help is the possibility, whenever an error occurs, to ask the system for possible places producing that specific error. This is very helpful, since the place an error manifests itself is usually different from the place the error was produced. This facility uses information gathered by the editor (cf. page 4), while backtracking in the trace of the actual execution of the program, and displays, as possible candidates for error

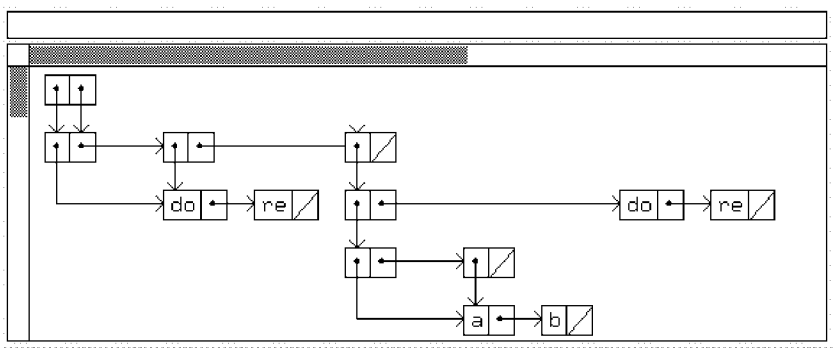


Figure 6 a list-editor

producing code, all functions which were not executed following their last modification.

4.3. graphical representation of data

One recurring feature of xbVLISP is the capacity to graphically view data-structures. Each list may be displayed graphically either as a linked list as in Figure 6 or as a tree, such as in the “tree-demo” window in Figure 5. These graphical views are used throughout the environment itself, primarily as views of the control- or data-flow. As all views, these graphical views may be either static, showing the underlying structure of a particular list (and being the only readable view of lists which share substructures, of circular lists or lists having circular substructures) or they may be dynamic: being updated whenever the displayed list changes. Different from other views, the graphical view of lists may also be used for the interactive modification of a data-structure. In this case, whenever the programmer selects a cell, he can modify its content, delete it or add another cell. When he deletes a cell, the arrow (a view of the link to this cell) is also deleted. One can also, just by selecting and clicking in the appropriate places, change the origin or the end-point of an arrow.

To increase the flexibility for changing views, xbVLISP offers tools for defining views of more complex data-structures. Figure 8 displays a view of an association-list. Such a list always has the form:

$$((name_1 . value_1) (name_2 . value_2) ... (name_n . value_n))$$

A standard view of the list ((a . 1) (b . 2) (c . 3)) would be:

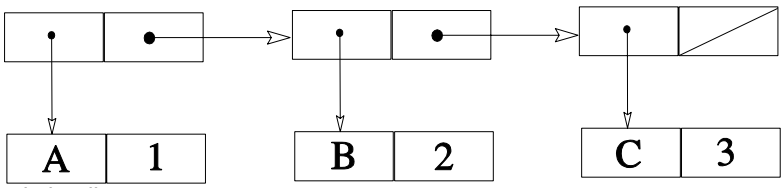


Figure 7 an association-list

We can change the standard view of data structures through redefining views. These redefinitions are given in a fully integrated sublanguage, similar to and influenced by PIC [Kernighan82], specialized for graphics programming. Below is the definition of an alternative view for association-lists and underneath the resulting picture for our example association-list:

```
(view alistView (x)
   (name alist:
      (tripleCell
            (if (null (cdr x)) nil “”)
            (caar x)
            (cadar x)))
      (if (null (cdr x)) (endlist alist:)
            (arrow right (center alist:)) (alistView (cdr x)))
```

with the auxiliary definition of tripleCell as follows:

```
(view tripleCell (first second third)
      (down)
      (name box1:
            (box (size StandardCell) first))
      (box (size StandardCell) second)
      (box (size StandardCell) third)
      box1:)
```

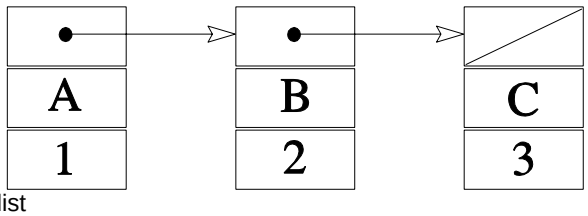


Figure 8 an association list

This sublanguage gives the programmer sufficient freedom to adapt views on data-structures to his conceptual understanding. Nothing prohibits simultaneously having several different views of the same object. Thus, one can both display an association-list as a standard list structure, which gives a clear view of the implementation and permits interactive graphical modifications and at the same time in the newly defined manner, showing more of the conceptual structure of the same object.

4.4. graphical views of programs

Similar to the graphical representation of data, xbVLISP offers tools for the graphical views of programs or parts of programs. In the basic version, those views show trees of the program's control-flow on which data-flow information can be superimposed. Figure 9 shows three graphical views of the same program: the first view displays the control-flow, giving a visual clue of the dependencies of the functions used by the program.

This view may be used to interactively explore the program: clicking on one of the function boxes displays (cf. Figure 4) in the associated sources-window the corresponding source code and in the associated variables-window all the variables used by this function, distinguishing between variables which are arguments to the function, locally defined variables and global variables. In the latter case, the variables-window indicates if this function modifies the variable or just accesses.

Clicking on one of the variables changes the control-flow view into a data-flow view. The second view in Figure 9 superimposes on the control-flow the data-flow for a chosen variable. This variable is *defined* in the function represented by a colored box and the data is transmitted to all the functions represented without a surrounding box, following the same paths as the control flow. This permits easy recognition of those functions which are involved in given subtasks. In the example of Figure 9, the program viewed is the program constructing graphical representations of lists and the data-flow is displayed for an argument which corresponds to the position where a cons-cell should be displayed; clearly, all functions touched by this data-flow are involved in calculating this position.

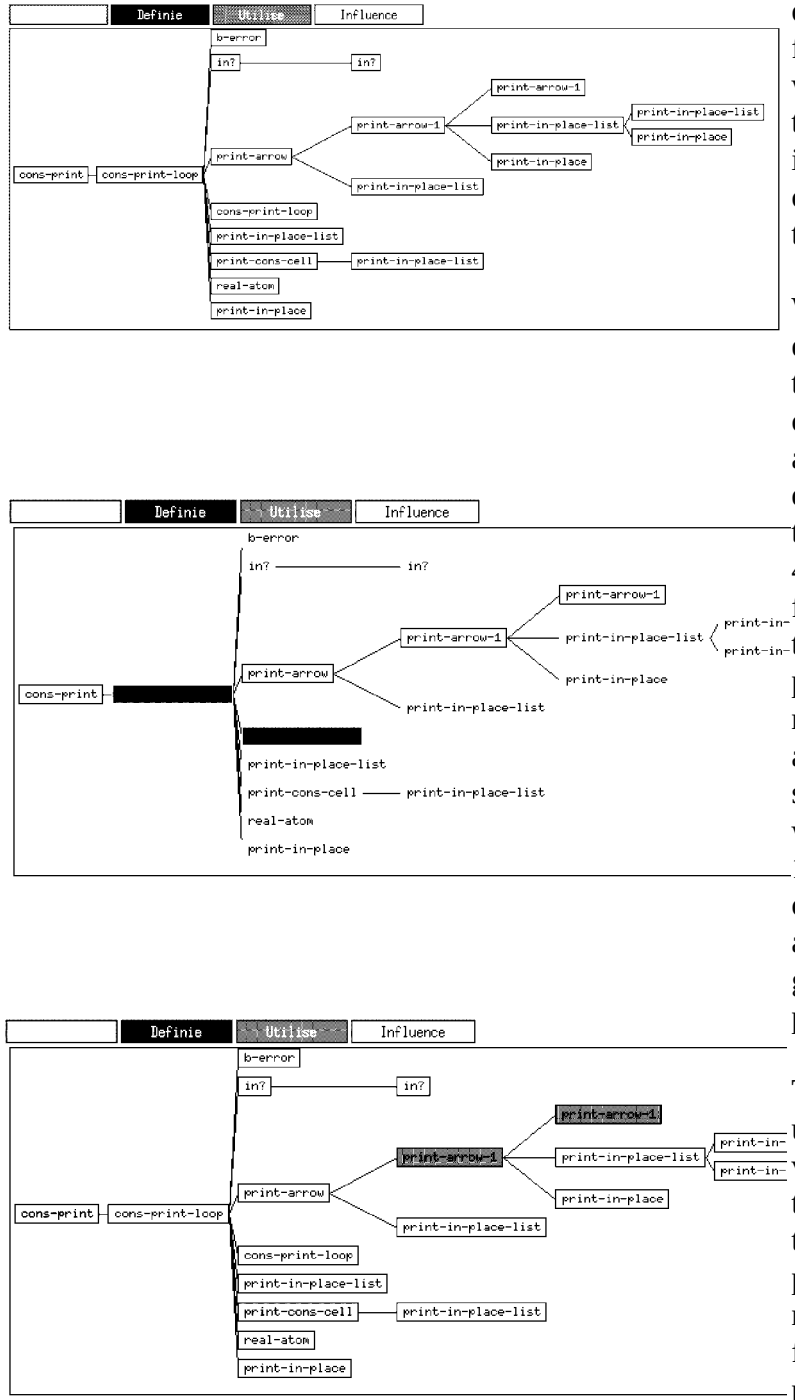


Figure 9 three views of the same program

The third view displayed in Figure 9 superimposes on the same tree the data-flow of a global variable which is modified in the two functions represented in light-gray boxes and consulted in the two functions in dark-gray boxes.

When visually tracing the execution of a program, this same view is used and currently active functions are displayed in inverse-video. An example of such a trace is displayed in Figure 4 (page 10) in the control-flow window. In this case, the variables-window displays the values of the variables of the last currently active function. A view of such an active variables-window is given in Figure 10. There, all the variables of a function having four arguments and using three global variables are displayed.

The first view of this Figure 10 gives still another view of the control-flow of this same program. This time, instead of just displaying boxes with the names of the different functions composing the program, each of the functions is displayed as a cartoon face. Characteristics such as horizontal and vertical extent, form and position of the mouth, ears and nose, correspond to chosen features of each of the functions. For example, one could choose that the horizontal extent corresponds to the size of the function, the vertical extent to the number of variables used by it, the position of the mouth to the maximum number of interleaved loops, it's form to the number of recursive calls, the distance between the eyes to the

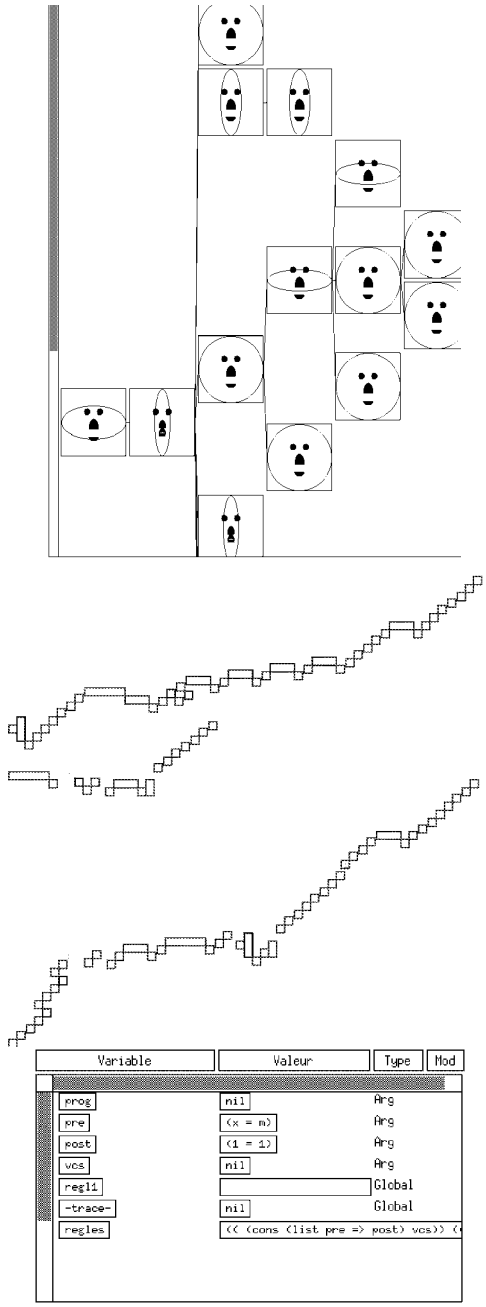


Figure 10 three more views of the same program

number of user-defined functions called and their form to the number of functions calling this one, etc. This kind of representation is used in statistical representations, since it permits, through combinations of easily recognizable features, to give views of complex objects in such a way that the difference between the chosen features is clearly *visible*: it is not necessary to look for a face without eyes, if such a face appears, one immediately "sees" it. xbVLISP offers a special menu, through which one can associate programming features to characteristics of the cartoon face representation.

As a standard control-flow view, this kind of view may also be used for interactive exploration of programs: clicking on one of the faces updates the variables-window accordingly, and, while tracing the execution of the corresponding program, the faces animate, opening or closing their eyes when the corresponding function is activated or exited, opening or closing their mouths when tracing data-flow.

The second view in Figure 10 gives a graphical view of an individual function of the program. This view gives information on complexity measures of the given function and its subcomponents. This kind of view is still highly experimental: one of our current research activities is to explore different ways of viewing. It is difficult to find representations which are easily understandable, easy to manipulate and which also permit interactive selection of subparts as well as their animation during execution. For example, for some time we tried to represent individual functions in the form of houses, with rooms corresponding to sub-parts, elevators for the transmission of data and basement-rooms for stocking data [Ploix91]. Even if, intuitively, such a representation seems to be well suited, experimentation shows that it very quickly becomes overloa-

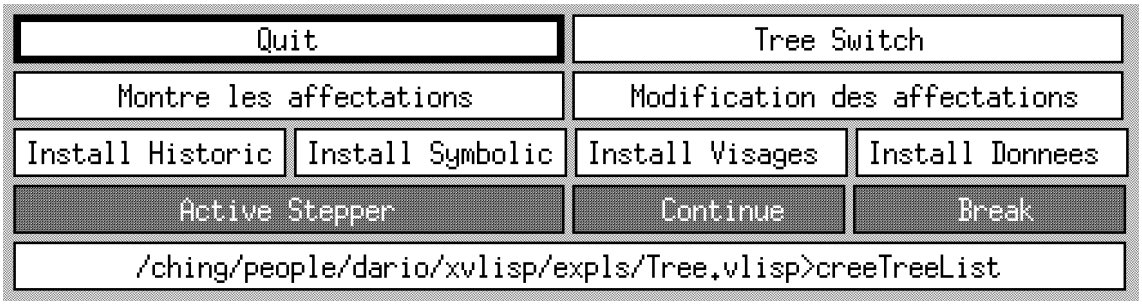


Figure 11 the control-window for interactive exploration

ded and it no longer facilitates understanding of the structure or the behavior of programs. We are currently developping new views of programs in the form of city-maps, with streets connecting different parts of the city. Traffic represents data-flow, individual city-blocks represent individual functions and houses in this block represent the subparts of this function. Zooming in on a individual function, a city-block, renders its representation more readable and easier to manipulate.

What xbVLISP offers, is all the tools necessary to rapidly construct and integrate new representations with already existing tools.

Finally, the “history” window gives another standard graphical view of a program. This view shows a tree of the control-flow as does the control-flow window. But rather than giving a general view, this one specifies the most recent execution of the program. This means that the tree is dynamically constructed, parallel to the execution of the program, and gives only those branches which were actually activated. Each node of the tree, corresponding to a function, memorizes the state of the computation, in such a way that one can go back to this state after execution or during a break-point, explore it and eventually restart at this point. Such a view is extremely useful when developing, modifying or maintaining programs, since it permits dynamic error-recovery as well as a post-mortem analysis of the program’s behavior. In Figure 5 (page 11) the window labeled “Historique” shows such a historical view.

Note that all interaction with the graphical components of the system is done through a special environment control window. Such a window is shown in Figure 11. Its first four rows are composed of a set of buttons which allow control of what and how to display. Thus clicking on the left button of the second row opens up a window which displays the currently valid criteria for analogical representations (e.g. the cartoon-faces) and the right button opens-up a sub-menu permitting modification of those same criteria. The other buttons are for installing or removing the “history-view”, the control- and data-flow views or graphical views of data-structures. All these buttons open up submenus, where the user can specify further characteristics of the chosen representation. The last line displays the name of the currently selected function as well as the file in which it is defined.

5. Conclusions

In this paper we presented the current state of the xbVLISP programming environment. This environment combines low level tools, such as an incremental on-line help facili-

ty, a structure editor or a graphical editor for data-structures with high level tools such as incremental construction of program descriptions, automatic error corrections, version control mechanisms and a sophisticated, generalized annotation facility. Those tools are embedded in a unique environment permitting sharing of all information which is either directly given by the programmer or automatically derived by the environment. Interaction with the environment is done through active, programmable views. A view is a graphical representation composed of elements which are reacting to changes during execution. Selecting one of the elements permits either interactive modification or access to more information about those same elements. This permits development of programs through views adapted to different levels of abstraction. In this paper, two simple examples of this abstraction mechanism were given: one showing two different views of a data structure, one showing different views of a control and data flow.

We are convinced that managing large programs can not be done without such possibilities for selective viewing and interactive modification of such views. In our understanding, the activity of programming will increasingly transform from an activity of writing and modifying text to one of combining and modifying pictorial representations. xbVLISP presents an initial prototype for such a graphical program observation, construction and modification environment. Though we have not yet found a truly satisfying pictorial representation, as most of them are still too close to the computational aspects and too far from the domain specific aspects, we have designed an environment where it is easy to develop new active representations and to interface them with the already existing environment.

The next step in our project will be the development and experimentation with the most outer layer of Figure 1: the construction of an interface to a virtual reality laboratory. With such an interface, programming will be transformed into an activity where movements and manipulations of analogical representations result in modifications of the underlying code and data-structures, reminiscent to some descriptions found in such science fiction novels like [Vinge81].

6. References

[ACM83] *ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging*, Pacific Grove, March 1983

[Erman80] L.D. Erman, F. Hayes-Roth, V.R. Lesser, D.R. Reddy, “The Hearsay-II speech understanding system: Integrating knowledge to resolve uncertainty”, *Computational Surveys*, Vol. 12, pp. 213-253, 1980

[Kernigham82] Brian W. Kernighan, *PIC — A Graphics Language for Typesetting, User Manual*, Bell Laboratories, Computing Sciences Technical Report No. 85, Murray Hill, NJ, March 1982

[Lalonde91] Wilf R. Lalonde, John R. Pugh, *Inside Smalltalk, Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991

[Nye90] Adrian Nye, *Xlib Programming Manual*, O'Reilly & Associates, Inc., Sebastopol, CA, 1990

[Ploix91] Damien Ploix, *Représentations Analogiques de Programmes*, Mémoire de Maîtrise, Département Informatique, Université Paris 8, Juillet 1991

[Sendoya92] Ernesto Sendoya G., *Etude et Réalisation d'une Couche à Objets Graphiques pour bVLISP*, Mémoire de DEA, Département Informatique, Université Paris 8, Octobre 1992

[Stefik86] Mark S. Stefik, Daniel G. Bobrow, Kenneth M. Kahn, "Acess-Oriented Programming for a Multipardigm Environment", in: *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*, Volume 2: Software, January 1986, pp. 188-197

[Vinge81] Vernor Vinge, *True Names and other dangers*, Baen Books, N.Y., 1981

[Wertz85] Harald Wertz, "A programming environment for the development of complex systems", in: *Progress in Artificial Intelligence*, editors: Luc Steels and J.A. Campbell, Ellis Horwood Limited, 1985, pp. 204-218

[Wertz87] Harald Wertz, *Automatic Correction and Improvement of Programs*, Ellis Horwood Limited, 1987