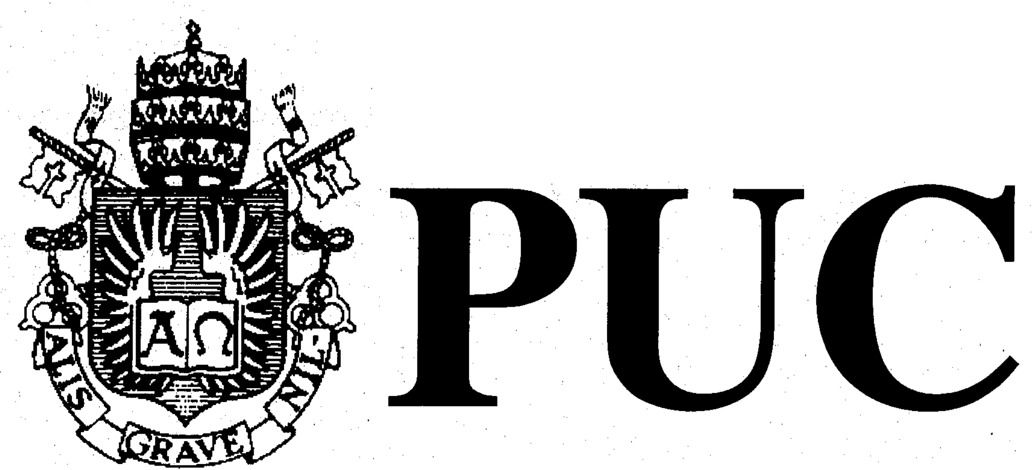

ISSN 0103-9741

Monografias em Ciência da Computação
nº 08/96

The Semiotic Engineering of Concreteness and Abstractness: from User Interface Languages to End User Programming Languages

Clarisse Sieckenius de Souza

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 08/96

Editor: Carlos J. P. Lucena March, 1996

The Semiotic Engineering of Concreteness and Abstractness: from User Interface Languages to End User Programming Languages *

Clarisse Sieckenius de Souza

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

Also presented at the Dagstuhl Seminar on Informatics and Semiotics, Schloss Dagstuhl, February 10-23, 1996.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

The Semiotic Engineering of Concreteness and Abstractness:

From User Interface Languages To End User Programming Languages

Clarisse Sieckenius de Souza
Departamento de Informática — PUC-Rio
R. Marquês de São Vicente 225
22453-900 Rio de Janeiro, RJ - Brazil
email: clarisse@inf.puc-rio.br

PUC-Rio.Inf.MCC08/96 March, 1996

Abstract

Most successful User Interface Languages have been designed observing two important guiding principles: task specificity and direct manipulation of graphic objects. Programming Languages, in their turn, have often been pursuing such goals as general purposeness and efficient symbolic manipulation of linguistic objects. When it comes to End User Programming Languages, features that are apparently in conflict with each other must be combined to allow non-programmers to write extensions to existing applications and to design and implement completely novel applications and programs.

In view of increasingly strong evidence for the need of engaging end users in the programming of software tools, some researchers advocate that the interface language should become a programming language, whereas others remain skeptical about this possibility. We propose that an integrated interface environment should be designed within a unified semiotic framework that accounts for interaction with and specification of computer applications. A brief case study about a successful text editor and its extension language reveals some of the features this unified semiotic framework should have and provides important topics for empirical and theoretical research agendas in the field. Integration of interactive and programming profiles in interface systems design is the object of the Semiotic Engineering of concreteness and abstractness in computer-mediated interpersonal communication through software applications.

Keywords

Semiotic Engineering, User Interface Language Design, End User Programming Language Design, Computer Semiotics

1. Introduction

User Interfaces are metacommunications artifacts. They are designed to convey a message from system designer to system user whose meaning is the answer to two fundamental questions: (1) "What kinds of problems is this application prepared to solve?" and (2) "How can these problems be solved?". This message, however, should not be confused with lower-level messages such as "copy [this]", "delete [that]", or "Save large Clipboard?", that are exchanged between users and systems. The higher-level message from designer to user is the most dominant piece of communication in human-computer interaction (HCI) because it provides the common background against which all lower-level interaction is going to take place. In Figure 1 we try to show four essential aspects of such metacommunications artifacts:

1. The Interface Message is a One-Shot Communicative Act from Designer to User;
2. The Designer's Intended Meaning for the Interface Message and the User's Assigned Meaning for the Interface Message are not the same, though they should be obviously consistent with each other;
3. The Designer's Intended Meaning is an abstraction of the Application's Model;
4. The User's Assigned Meaning is a model of the Application's Usability[1] Model.

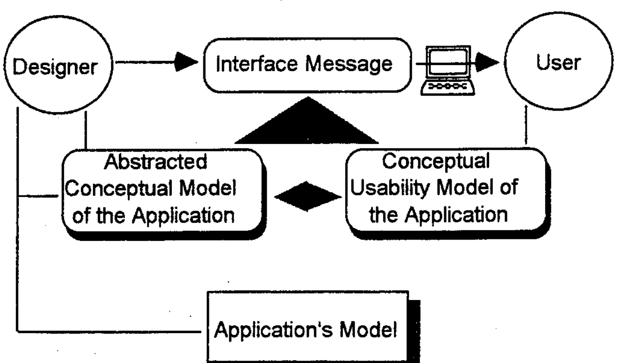

Figure 1: Meanings of the User Interface Message

In a Semiotic Engineering framework [de Souza'93], HCI proper is a kind of "zoom-in" on the arrow that reaches the user in Figure 1, whereas computer-mediated interpersonal communication (CMIC) encompasses all the interactions suggested in Figure 2. The whole interpretation process resulting in a Conceptual Usability Model of the Application is triggered and determined by the interactions supported by the system's messages to users and its corresponding reactions to mouse clicks, keyboard and voice input, data-glove motions, and other device signals.

[1] Usability in this context is taken in Adler & Winograd's sense: the perceived spectrum of possibilities of use associated to a given application as a result of a user's learning and creativity. [Adler & Winograd'92]

From the signs exchanged through I/O screens, users build a cascade of interpretants [Peirce'31] that eventually crystallize into the user's assigned meaning for the interface message — the Application's Usability Model.

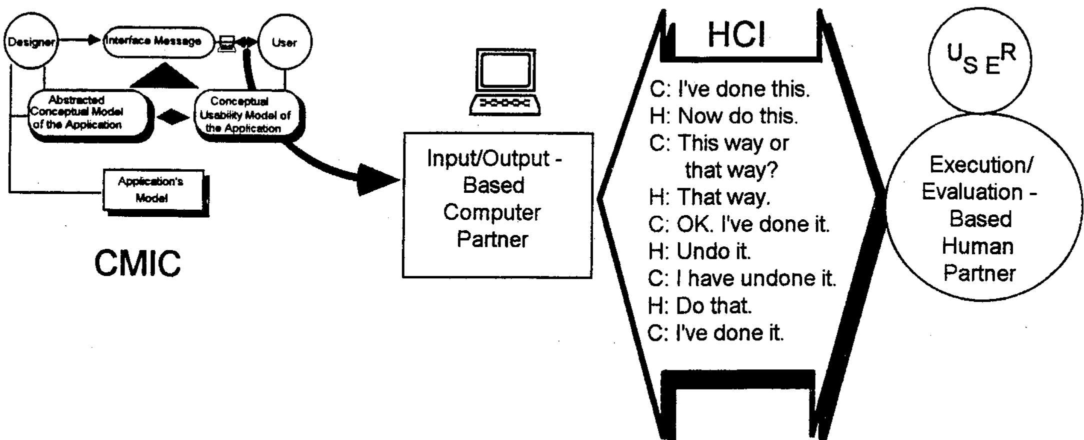

Figure 2: Human-Computer Interaction in a Semiotic Engineering Framework

In this paper we explore a wider spectrum of user interface design issues and extend our Semiotic Engineering approach to the limits of en-user programming languages. As some researchers advocate [Myers, Smith and Horn'92; Nardi'93; Cypher and Canfield Smith'95; Eisenberg'95], users must to be empowered to the point of configuring and writing their own interfaces and applications by means of "a new breed of languages" [Dertouzos'92]. In this updated scenario, User Interfaces must provide users not only with the ability of taking the best out of applications, but also that of customizing, extending, and/or integrating them into novel working environments.

The Final Report of the End-User Programming Working Group of SIGCHI'90's Workshop on Languages for Designing User Interfaces [Myers, Smith and Horn'92] starts with projections made in the beginning of this century about the future of telephony in the USA. Predictions were that if growth rates remained stable throughout the years, today mile-long switchboards would be operated in telephone companies by virtually all the female labor of that country. What predictors failed to foresee was a major turning point in the evolution of telecommunications: that users would themselves become operators.

Reporters notice that a similar situation is currently faced by the software industry. Unless users become software generators themselves, wild predictions such as the above will pop up everywhere. Having learned with the history of telephony, software engineers are now starting to look ahead and work on the design of software artifacts that somehow behave as meta-applications, since they allow users to generate other applications by using them.

However, all the mathematical and logic constructs that lie hidden inside software artifacts are unfortunately just a motion away from the friendly surface of user interfaces. One step into the world of programming, and users are overloaded with terminology and concepts that are quite foreign to the average software consumer. As a result, a deep gulf opens between user needs and software affordances — a gulf researchers are starting to try and bridge [Lieberman & Fry'95].

The top level challenge for the "new breed of programming languages" is that users understand what computing is and write programs that help them solve problems or that just let them have a good time [Dertouzos'92]. These new languages should make people move their focus from the value of applications to the value of computing. Thus, computer semioticians have a highly challenging task at hand. They must help software engineers build novel metacommunications artifacts whose message not only contains information about special-purpose domain-dependent computer applications, but also information about computers, computing, and computational worlds in general.

We expect this new scenario to transform drastically our current user interface design process. For instance, consider the model world metaphor [Hutchins, Hollan & Norman'86] behind most direct manipulation interfaces. The very definition of the "world" to be modeled — up to now the application's domain world — will change. The computer world will have to be modeled as well. So, our intended contribution to the debate of such hot issues is only an account of how the ideas we have been working on in recent years can fit into the overall picture. As will be shown in sections ahead, designing user interfaces without an explicit end-user programming perspective in mind may result in major discontinuities within the global environment. Alternatively, a new extended perspective in design may result in quite different segmentation and classification of interface objects and signs, which instead of promoting only the mastery of interactive and task-related skills will also tend to promote computer literacy as a whole.

In order to explore the possibilities of a Semiotic Engineering of integrated User Interface and End-User Programming Languages, we briefly examine the interface profile of a former version of a popular application in personal computing: Microsoft's Word for Windows 2.0 ™, and its extension language, Word Basic™. Selected features of both semiotic systems are analyzed and shown to cause problems for a global unified understanding of how to extend the application for novel needs.

We discuss the reach of our previous approach to designing interfaces within the new boundaries of computer-mediated interpersonal communication, and conclude that extensions in the framework are needed. We also evaluate in very general terms the costs and benefits of a semiotic perspective upon this novel scenario, and propose the addition of some topics in the current research agenda of the field.

2. The Semiotic Engineering of Concreteness and Abstractness: A Case Study

In order to demonstrate some of the relevant issues involved in the integration of User Interface Languages (UIL's) with End-User Programming Languages (EUPL's), we have selected a real-world situation which involves a new kind of "Save As" function in MS Word-for-Windows 2.0™ (WFW). Here is the scenario:

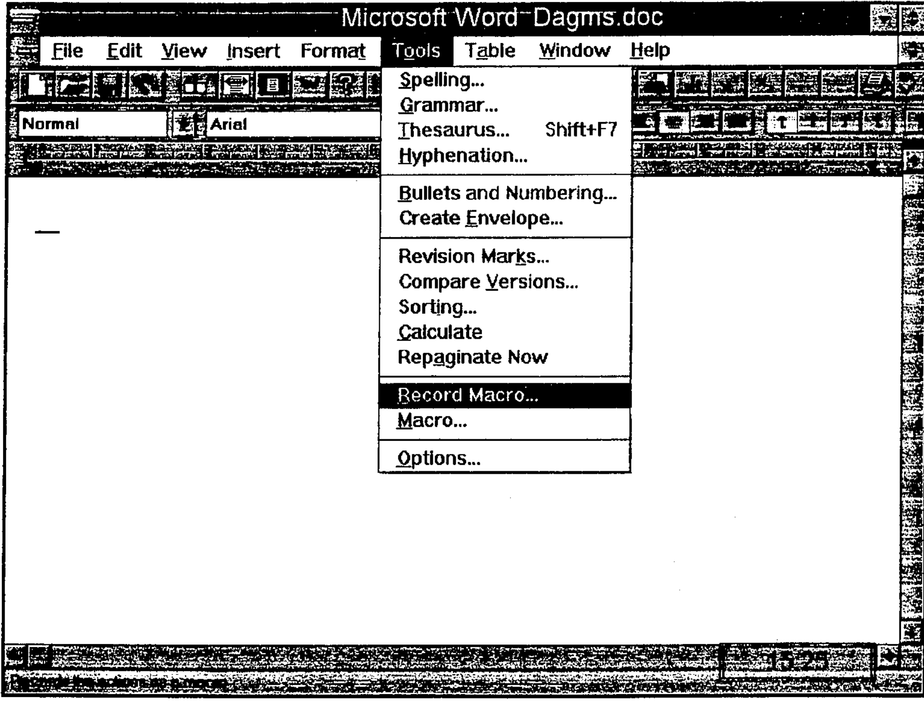

Figure 3: WFW's End User Programming with the RECORD MACRO Tool

This user has a Macintosh computer at her office and a PC at home. She must often bring home texts she has started to edit on her Mac, and do part of the job on the PC. WFW and Word for Macintosh 5.0 (WFM) file formats can be automatically converted into each other, which is perfect as long as no graphic editing is involved. If, for example, she has created a drawing on the Macintosh, she cannot open it for editing on the PC, and vice-versa. However, many times this sort of editing is needed and the work becomes difficult to do.

Of course there is a way around the problem. She can Save WFM files AS WFW, Open these on the PC, Save WFW files AS WFM, Open these on the Mac, and so on. But, there is a lot of version control involved, and the process may become quite laborious. For instance, WFW can save files as WFM but it is the user's responsibility to control file names and extensions, so that the conversion can take place. Neither WFW nor WFM automatically

change file extensions, and the user must step through the whole process of specifying file paths, names and extensions, tediously and repetitively.

As we can see, this is a typical opportunity to use Macros to save time and effort. We will not go into the WFM bit (which behaves differently from WFW), but will take up from the part where the user is working at home and wishing to automate her customized Save_As function. She triggers the Recorder Tool and generate a script for a new Macro called SaveMac, as is seen in Figure 3. The corresponding code automatically generated by this interaction is seen in the first portion of the code (Sub MAIN-Recorded) included in Figure 4.

What the Word Basic code suggests is that SaveMac, upon call, will always save files in drive B under the name of "dagms.doc", which is not what the user wants. Being a programmer, she realizes that the Recorder Tool has treated input data as CONSTANTS, rather than VARIABLES, and picked up the current file's name. She decides to edit the macro code and introduce variables in what she thinks are the appropriate places. The resulting routine is also included in Figure 4 (Sub MAIN-Coded-1).

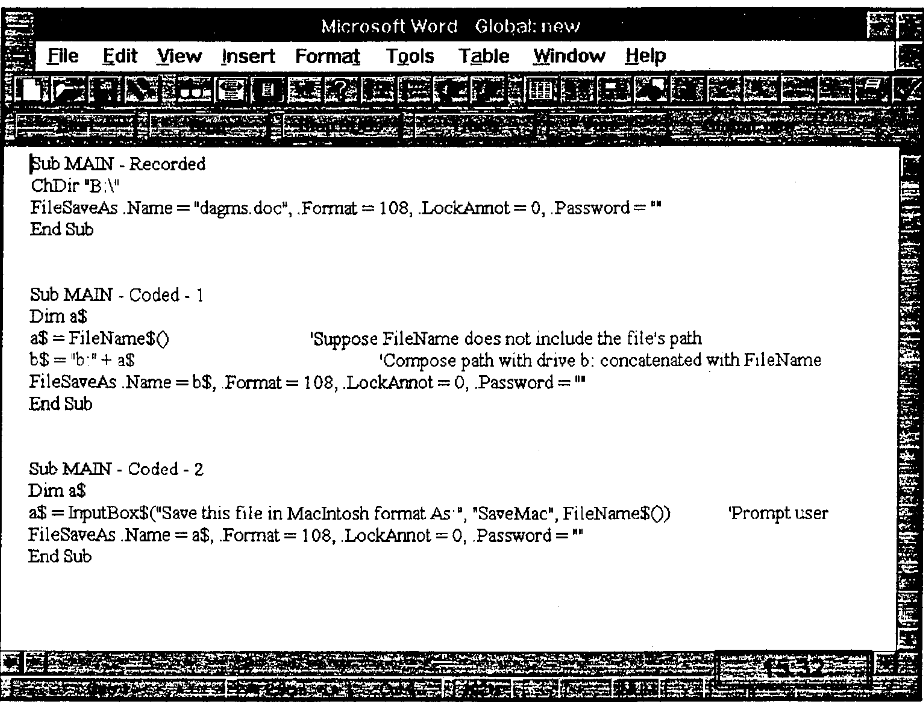

Figure 4: WFW's End User Programming with Word Basic

However, the trial is unsuccessful because function FileName$() returns not only the "name" of the file, but also its complete path from the root directory. (Notice the apparent inconsistency with what was the file's name in the recorded macro.) As a result, the user realizes an interactive input box will be

needed to capture the appropriate file name. Her Word Basic code then looks like the last portion of code in Figure 4 (Sub MAIN-Coded-2).

Our user proceeds to include direct access to the new macro in the File Menu. She also assigns an accelerator key to this option, so that the final execution path takes a minimum of three interactive steps. She is not quite satisfied, though, because she knows one step might have been enough to execute the whole process, if only she had found a function that returned a file name without its complete path.

The point of this long story is to demonstrate that end user programming in Word Basic is still some mileage away for non-programmers. The consequences of this gap for software usability are self evident. The story illustrates most of the issues mentioned in this paper's introduction, and from here we will proceed to a semiotic overview of WFW's graphical user interface and Word Basic's programming environment.

2.1 Word for Windows 2.0 — A Semiotic Overview of the User Interface

A customized configuration of WFW's interface is shown in Figure 3. In it, we can see that the default tool bar has been changed compared to what it looks like when the software is first installed. It now includes some other icons (like the footprints and the pair of glasses) whose meaning only the creator of the customized interface can be expected to know. This peculiar setting is the result of WFW's configuration facilities that allow users to select a set of tools that are tailored to their specific needs. The programming of such new interactions on the tool bar is easily achieved by parameter setting.

The interface is a typical Windows™ style GUI [Microsoft'95], with pulldown menus and lists, buttons, icons, pointers, sliders, a status bar, scroll bars, and a canvas where direct manipulations on text and other objects can be performed. In spite of interference caused by the user's customization of the environment (which is nevertheless part of the interface), we can see that the nature of interface signs is heterogeneous.

A possible high-level interpretant that a user can derive from this WFW's interface message is that text editing is achieved by triggering ACTIONS upon OBJECTS. With the feedback from the global Windows environment, the distinctive status of objects is expressed by numerous pictures (called "icons" in HCI literature) on the tool bar and by such English words as "tools", "table" and "window" on the main menu bar.

Actions upon objects are expressed in the main menu by such words as Edit or Insert, and by tiny oriented arrows on some of the tool bar buttons. Many pulldown

menu options also evoke actions through words and phrasings like Copy, Save As, Arrange All, Exit, and others.

Whereas Object-Action relations are quite clear in the implied syntax of some menu options like "Create Envelope" and "Insert Table", the same is not true when it comes to interpreting some of the graphical signs on the tool bar. A couple of examples should suffice to put this point across.

Insert Table

The action triggered by a click on this "icon" is the insertion of a table. Further manipulations allow users to specify the dimensions of the table they want to insert. Therefore, the complete meaning of this widget is "Insert Table" ("syntagmatic" level), a compound sign of button+drawing. The button accounts for the "action" part ("morphemic" level), and the drawing accounts for the "table" part ("lexical" level). But, the notion of "insert" ("lexical" level for the action involved) must be guessed by the user.

Such guessing is not as easy as it first appears. The generalization about the mutual relationships between the widget's nature (button), its image (drawing of a table), and effect (insert) doesn't work (e.g. printers are not inserted when you click on the printer's button, and neither are folders or diskettes), which means that syntagmatic connections between the meaningful parts of a widget are not the same throughout the interface language. Too many ad hoc inferences about such connections end up by costing an expensive price in terms of cognitive loads upon users. The "Insert Table" widget is a complex graphical sign in which some metonymical process of expression seems to have taken place: the object "table" (graphic expression) stands for the process of "inserting a specific table in text". That is, the expression privileges the result of a whole series of steps (trigger tool, specify object, close interaction) which constitutes the achieved meaning of the widget.

Save File

The action triggered by a click on this "icon" is saving the current file to its current address in memory. Once again, this is a compound sign of button+drawing. The button accounts for the "action" part, but the drawing doesn't seem to directly account for any part of the semantic notion associated to this widget. Taken as an icon, in Peircean terms, the most unequivocal of this drawing's interpretants is a floppy disk. Now, what does a floppy disk have to do with Save the current file?

This image, although depicting a totally familiar object to all of computer users, resorts to quite specialized knowledge about operating systems. Correct associations between a floppy disk and saving a file can only be established if the user knows:

- files are data objects

- data objects are stored in specific memory locations
- disks are memory locations
- file memory locations are expressed as compound strings (path+file name)
- the floppy disk is representing all storage devices

Thus, if a user has this knowledge, the graphical sign connects form to content by a rather tortuous trail. The $3^{1/2}$ floppy disk stands for any existing storage device (CD, cartridge, floppy or hard disk). Besides, the fact that the current file name stands for memory location coordinates that are implied in all "saves" after the file's creation is again something that is often not quite clear to average users, and not consistently signaled in some of the working window components. For instance, as seen in Figure 3 and 4, file names on the window title bar omit the path (memory location) segment.

A user's failure in grasping the right connections deprives the graphical sign from all of its iconic or indexical features. It becomes almost a graphic symbol, with an arbitrary connection between its form and content, something which runs against the tide of much graphical interface design guidelines [Apple'92; Marcus'92; Microsoft'95; Mullet & Sano'95].

Edit Drawing

The action triggered by a click on this "icon" is the opening of a graphical editor. In fact, a call is made to another application (MS Draw™) which the users wield to create drawings and insert them in the current text. The button accounts for a number of actions which combine and result in the widget's meaning.

By examining the image on this button we realize a most complex semiotic process has occurred. What do the geometric objects stand for? They certainly are drawings, but the irony behind it is that none of these geometric shapes appear in MS Draw's tool bar (i.e. they are not drawing tools). Triangles, in particular, can only be drawn and filled with colors or patterns if we use a somewhat strange looking tool: the polyline .

Consequently, the form/content association for the Edit Drawing widget, as was the case with the Save File widget, is not quite straight forward. The geometric objects stand for potential referents very widely distinct from their appearance. To illustrate how wide is the spectrum of graphical objects referred by the circle-square-triangle group it suffices to say that all screen dump images on this paper have been created by the Edit Drawing tool. This is a self-reference feature in the widget. In a way, the expression (graphics) is the content (graphics), an example of what Eco in his Theory of Sign Production [Eco'76] has called ratio difficilis between the expression system type and token. Except for deictic utterances, whenever contents are raised to the level of their own expression, interpreters may have difficulties in segmenting their global perception into the correct meaningful units intended by the utterer.

In Sum...

The few examples above suggest that a considerable amount of arbitrary connections between graphical elements takes place in the design of WFW's interface widgets. Some important syntagmatic relations between verbs and nouns present in pulldown menus, which all contribute to reinforce the overall interactive paradigm of actions upon objects, have no systematic correspondence in the visual part of the interface.

Of course, in the long run, users have no difficulty to memorize the idiosyncrasies of visual widgets and certainly become experts in using the expressive code of the application's interface. The price to pay is affordable for most interface designers and explains why some quite cryptic tool bars do not prevent the commercial success of some popular (though difficult to use) software products.

The problem arises, as we shall see later, when the leap is made from a closed, merely customizable, application to an open, extensible, one. There, where end user programming is possible and EUPL design becomes a crucial interface issue, consistent generalizations, systematic patterns of articulation, and careful segmentation and classification of interface objects play a decisive role in the usability of programming facilities provided to non-programmers. Thus, the price to pay for arbitrariness may become too expensive for interface designers and software developers, themselves.

2.2 Word Basic — A Semiotic Overview of WFW's Extension Language

When we switch from the WFW's text editing screen to the end-user programming screen in Word Basic, a major change in the interface paradigm occurs, although the user may not be immediately aware of it. As can be seen in Figure 4, the general look-and-feel of WFW is still there, with the same menu bar and tool bar. New widgets are shown, however: basically 5 labeled buttons and a message field next to them. A couple of tentative commands show that most menu bar options are disabled, as well as most of the tool bar buttons. The user is in a quite different interface environment.

We will concentrate our analysis on matters the (dis)continuity of signs as the user goes from the editing to programming environment. As pointed out by Nardi, end user programming languages should be task-specific and capitalize on a user's interest and motivation to master the complexities of embedded programming in an effort to extend the usability of a given application [Nardi'93]. Task specificity in our example has a lot to do with semiotic continuity, since WFW users should not be asked to learn a totally new communicative jargon in order to create macros. The objects they want to operate upon, the kinds of operations they conceive of, and all relationships among objects and operations should be expressed within the same semiotic continuum.

For instance, if "files" are expressed in WFM by the word File or by a drawing like , and "save" is expressed by the word Save or by a drawing like , we should be able to write programs in which these signs are used to mean what they mean. Word Basic programs do include words such as File and Save, as can be found in the following instruction:

```
FileSaveAs .Name = a$, .Format = 108, .LockAnnot = 0, .Password = ""
```

However, users cannot include their pictorial counterparts, the and signs, in Word Basic statements. Although the Recorder tool (see Figure 3) generates Word Basic code through manipulations of the WFW interface both in the text editing ("Record Macro") and in the macro editing ("Record Next Command") environments, type-objects like a generic file, a generic paragraph, or a generic character string, cannot be referred to by such icons.

WFW's extension language thus presents serious continuity problems that challenge computer semioticians. As stated by Nardi, "the problem with programming is not programming; it is the languages in which people are asked to program" [Nardi'93, p.40]. She argues that unlike some of the assumptions behind current attempts at stating the problem with extension languages in interactive terms (whose solution is to have users manipulate diagrams and pictures to build programs [Shu'88, Chang'90], or fill out forms which will be input to automatic program generators [Zloof'81]), what is really called for in this area is a "semantic" approach to language design. We will push Nardi's point a little farther and suggest that what is really called for is a "semiotic" approach to language design, integrating semantic and interactive aspects within the same framework. The story at the beginning of this section shows some of the reasons why this might be so:

1. The user realized that the Recorder tool generates code with constants instead of variables;
2. The user could not find how to isolate and manipulate a field (the file extension) of what she thought was WFW's "file" data structure;
3. The user was not totally happy with her final macro because it involved more interactive steps (and quite a lot more of trial and error in Word Basic) than she had wished.

Firstly, the Recorder tool, which alleviates the burden of having to learn a programming language (even if customized to WFW), falls short of dealing with variables in a general and consistent way. Although variables are implicit in macros that, for instance, (1) pick up selected text spans, (2) turn them into boldface characters, (3) render them in green, and then (4) change their character fonts to Times New Roman (see Word Basic Macro in Figure 5), they are not implicitly assumed if the object we want to operate upon is not "selectable". A selected span is an implicit variable whose name is internal to the application.

However, in the absence of previously selected objects, explicit variables must be created and controlled by the programmer (user), as was the case in our story.

```
Sub MAIN
Bold 1
FormatCharacter .Font = "Arial", .Points = "12", .Bold = 1, .Italic = 0, .Strikeout = 0, \
.Hidden = 0, .SmallCaps = 0, .AllCaps = 0, .Underline = 0, .Color = 4, .Position = \
"0 pt", .Spacing = "0 pt"
Font "Times New Roman", 12
End Sub
```

Figure 5: Macro with an implicit Variable

Notice that the user should not be misled by the button "Vars..." in the programming environment. It is usually disabled when the user is writing a macro and, in spite of its caption, it is not meant to help users create variables in their programs: it is actually a debugging tool that can be used to help them watch the values variables take on as execution proceeds.

Secondly, the level of articulation in the File data structure presents an unfortunate discontinuity, this time not with the WFW editing interface but with the operating system on which it runs. Although files do have "names" (see that Word Basic has a field called ".Name" for file objects), these are unarticulated compounds of "path+name+extension" which cannot be broken into morphemes and manipulated independently by specific string functions. In our more global perspective upon software extensibility, this minor detail turns out to present a serious problem for the user. Her misunderstanding was aggravated by the false inferences she drew from her ability to change freely the file extension string in WFW's dialogue box for the "Save_As" tool. Once again, this is another facet of discontinuity in the example: not with the "outside" operating system, however, but with WFW's interactive patterns themselves.

Thirdly, the user's dissatisfaction with the lengthy interaction needed to perform her new function and with the time it took her to figure out how to program it is precisely the gist of our point about a semiotic approach to concreteness and abstraction in an integrated language design. She has been induced into thinking she could wish for short interactions because of communicative patterns present in WFW's. Tool bars provide the best example of the application's designers careful attention to the readiness of functions. Smooth interaction flows quickly from mouse clicks and drag-drops that capitalize on indications about which is the focal object of the action being commanded.

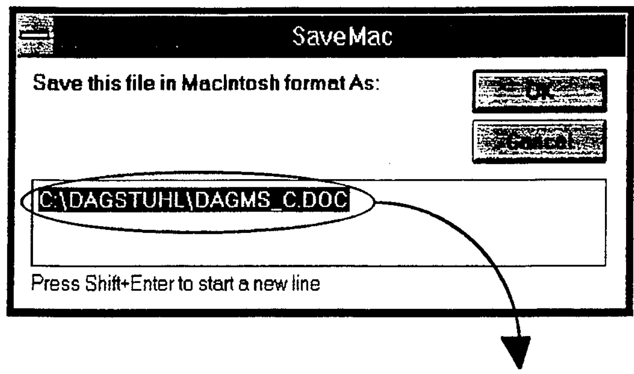

Figure 6: Default name proposed by the interface must be changed

However, as soon as the focal object cannot be identified by application, the interaction gets longer and default values hardly usable. The degradation in the quality of interaction is an important one, as can be seen in Figure 6. The default value (the current complete file path and name) is obviously not the correct one, and the amount of editing required for getting the desired new name is considerable.

A full-length programming language can be characterized by: data values and structures; command decisions, repetitions and recursions; procedure abstractions and parametrizations [Cordy'92]. End-user programming languages, in their turn, should include at least loops, conditionals and variables [Myers'92]. If we are to empower WFW's users with the ability to write interesting and useful extensions to their text editor, we have to find a way to convey such programming entities in a task-specific language that will be easy and fun to use [Dertouzos'92, Nardi'93].

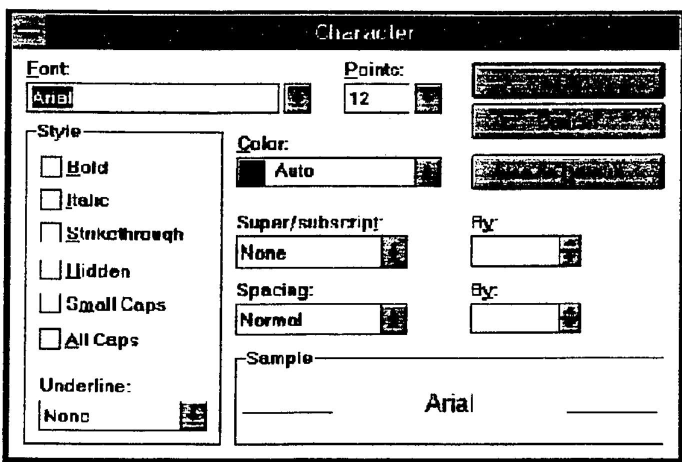

Figure 7: Internal Attributive Articulation of Character Data Objects

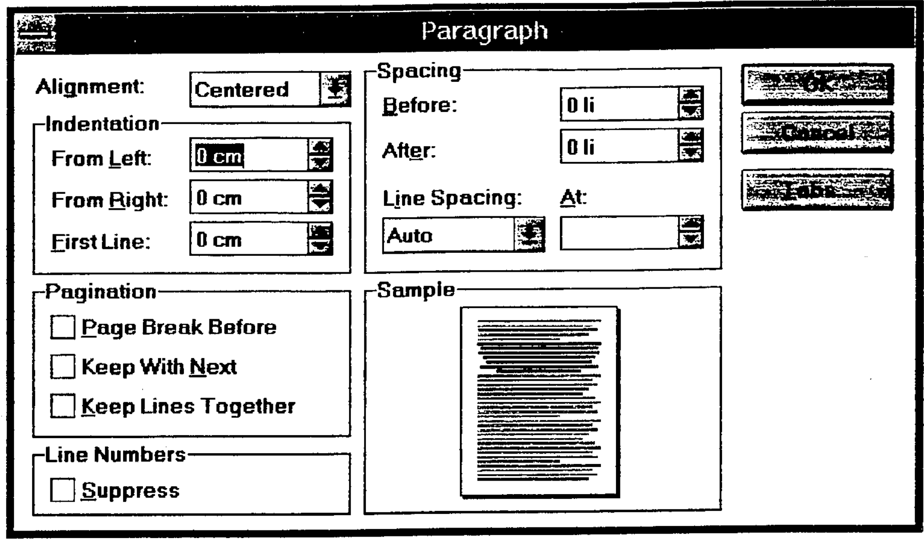

Figure 8: Internal Attributive Articulation of Paragraph Data Objects

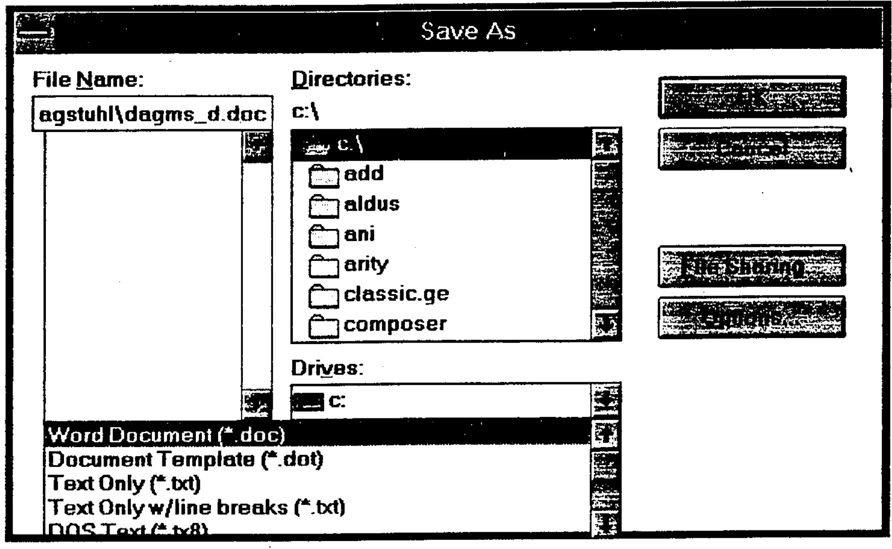

Figure 9: Internal Attributive Articulation of File Data Objects

WFW's data objects, for example, reveal their internal articulation (their attributive "morphology") through dialogue boxes that pop up during routine interaction. Characters, and strings (arrays of characters), have a number of attributes as suggested by the "Format Character" dialogue box (see Figure 7). The same is true for paragraphs, which also reveal their internal data fields through a dialogue box that follows a "Format" command (see Figure 8). Files, however, also have formats, but because they are at the top of the data object hierarchy of this application, they can only be manipulated by operations that involve the operating system itself. There is not "Format File" command in the

menu bar, although what the user is really doing with a "Save-as" is file formatting (see Figure 9).

Data structures in WFW's extension language, therefore, could be organized into records whose fields correspond to the attributes users can set through the interface. In an object-oriented framework, suggested by some of the Word Basic macro code, these record fields could eventually include procedures that apply to the data objects in specialized ways ("FileSaveAs.Name" is precisely an example of a procedure infixed between an object and its attribute). This approach has been extensively discussed in [Andersen'93].

As a result, objects, methods and attributes, appearing in WFW's interface would become complex data structures that the users should easily recognize. In order to illustrate our point, let us take the three objects exemplified above: character-string, paragraph and file. Table 1 shows the kind of articulations they might have.

Object	Methods	Attributes
character-string	format	font, style, color, super/subscript, spacing, points
	delete, copy, paste, select, find, replace	—
paragraph	format	alignment, indentation, pagination, line numbers, spacing, line spacing, tabulation
	delete, copy, paste, select, find, replace	—
file	format	type/extension
	write, read	type/extension, name, (sub)directory, drive, sharing
	close	—

Table 1: Examples of Data Objects, their Methods and Attributes

Table 1 presents examples that we don't claim to be either exhaustive or correct. They are expected to give the reader a flavor of the correspondences that exist and should be maintained between WFW and Word Basic code. The object-oriented paradigm, moreover, has an inheritance mechanism that is not shown in Table 1, but that the user intuitively grasps: for instance, formatting operations on a paragraph affect all of the characters in it.

A noteworthy issue in this framework is that, since the above data object structures are supposed to serve both for text editing and extension programming, the way in which they are expressed in one interface must be consistent with the

way they are presented in the other. What Draper called inter-referential I/O, within the sole limits of a non-extensible application's interface [Draper'86], now becomes a critical feature of a matrix of I/O communication systems. For instance, if files are data objects similar to paragraphs and character-strings, this should be shown in WFW. In its current version, we see that character-strings and paragraphs are grouped together in the interface menu, but files are not. However, as soon as a file is saved as "text with line breaks", all formatting vanishes. Are users expected to understand why this happens in such a way that they can write useful programs involving interactions between paragraph and file formatting methods?

With data structures and methods such as suggested above, the user can write programs equivalent to the ones generated by the Recorded tool. Instructions can be written by recording text editing operations. Variables and constants can be associated to each data structure by special widgets of the programming interface, as is proposed in Figure 10. Control Structures, in their turn, include Myers's small set of conditionals and loops.

Continuity with WFW is achieved by a number of features:

- WFW's commands may be introduced in the user's program by demonstration, through operations performed in a child-window within a global end-user programming environment, in addition to direct typing of statements in Word Basic (or other alternative EUPL);
- Data objects, along with elements of their internal structure, may be created and manipulated by clicking on buttons with icons which are adapted or simply copied from WFW's tool bar icons. Variables to store such objects may be automatically created by direct interface actions such as clicks;
- General control structures such as conditionals and iterations can be introduced in programs by special programming tools in the toolbar. Users may program "if-then-else" and "case" structures, as well as "for" and do-while" structures. Prompts for conditions and actions, as well as for the number of iterations or their escape state may guide novices through programming steps.

A snapshot of the kind of integrated semiotic system that might be devised for both the UIL and the EUPL in a Word for Windows environment is shown in Figure 10. The images used for the Control Structure tools are geometric forms associated to the meaning of conditionals and iterations [de Souza and Ferreira'94]. An "if" statement is a two-fold decision structure we have chosen to represent as a bifurcation of a linear path; a "case" statement is a manifold decision structure represented as a splitting into many optional paths. "For" and "Do-while" statements are represented by a circle (an image that evokes loops, as in expressions like "merry-go-round" or "round-trip"): the latter has a vertical line preceding the loop to represent the test for the condition before the loop, and an alternative path beneath the circle to represent the escape condition.

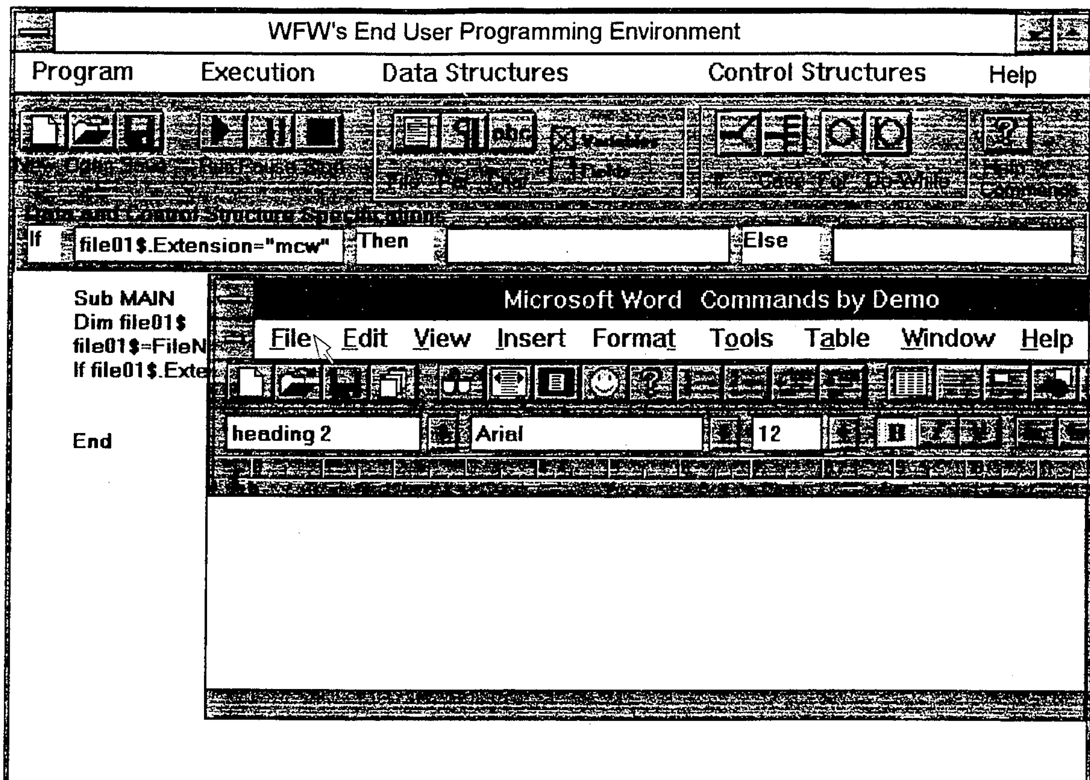

Figure 10: An outline for the End User Programming Interface of WFW

In Sum...

Extension languages associated to computer applications must be carefully designed to meet users needs and allow them to write useful new programs and procedures within the application's environment. Their interest and motivation for doing so is probably in direct proportion to their inclination to play games by the rules [Nardi'93], their talent to learn and use coded languages, and the positive impact of the kinds of extensions the user may program upon his/her work or leisure activities.

Careful end user programming language design involves the conception of an integrated semiotic system that goes as seamlessly as possible from the applications user interface language to its embedded programming environment. Task-specific objects and actions should ideally be accessible through the same code (words and/or images), whereas programming-specific elements (such as variables, conditionals, loops and the like) should be coded in a small, simple and generic system of lexical items and syntactic rules (textual and pictorial) that allow users to grasp the basics of computation.

3. Discussion

Our first attempt at setting the guidelines for Semiotic Engineering [de Souza'93] has approached the issues involved in HCI within the limits of the application's interface. Our major concern was then to contribute to the design of user interface languages (UIL's) from a more static point of view. Applications were considered as achieved artifacts that users could wield to solve novel problems in unpredicted situations, without any changes to the underlying program. However, as HCI moves towards more dynamic settings in which users should be able to become programmers of high-level extensions and adaptations to existing systems and programs, Semiotic Engineering must be revised to accommodate this feature.

The brief analysis reported in the preceding section serves to show that the end user programming language (EUPL) must be harmonized with the UIL. Discontinuities between both languages severely impact the usability of the application's programming facilities, and above everything else prevent designers from getting their full message across. However, the challenge behind an integrated design of UIL and EUPL (UIL☯EUPL) is that whereas UIL draws upon a wealth of task-specific signs (words and images) that an engaged user easily recognizes as long as they are selected from a consistent set of culturally established communication codes, EUPL goes into computer-specific jargon and knowledge. As mentioned before, EUPL draws upon a kind of folk-theory of computer, which is something HCI research has not explicitly tackled to date.

Semiotic Engineering guidelines shortly stated that:

1. Designers should avoid using invented interface signs;
2. Designers should select interface token sign systems from culturally established type systems of communication;
3. Designers should use heteromaterial signs to refer to domain-dependent entities and, as much as possible, use homomaterial signs to refer to and reinforce the expression of computer interface devices (such as mouse pointers, windows, menus, and the like);
4. Designers should always build highly articulated user interface communicative systems.

Nevertheless, as we see, UIL☯EUPL is a much more complex system. The above guidelines apparently do not apply to EUPL as well as they seem to UIL. For instance, the culturally established code for any programming language is a textual type system of which C, Pascal, Smalltalk, Prolog, LISP or Linda may be taken as representative tokens [Gelernter & Jaggannathan'90]. Of course this system is too diverse from the usual GUI type system we find in most current applications interfaces [Marcus'92, Apple'92, Mullet & Sano'95, Microsoft'95]. Integrating both systems is a difficult semiotic problem which has not even been stated to its full extent.

One of the points our guidelines fully fail to address, and that is central to UIL©EUPL, is that of self-referring signs. Some kind of self-reference is always involved in the new integrated language environment we are dealing with inasmuch as users are using some application facility to change the application itself. Decisions about the kind of sign we must use to convey this idea have to face problems like the ones we have mentioned about the Edit Graphic tool in 2.1, above.

One of the complexities of the sign is precisely self-reference. The tool bar where the widget stands is a series of button+drawing interface signs. All are graphical interface signs, created by some sort of graphical editor. But, one of these creatures stands for its "creator": the graphical editor. The collapsing hierarchy of interpretants in this case contributes to blur the distinctions between expression and content, which according to Semiotic Engineering is in principle undesirable rhetoric for one-shot messages.

Our guideline in this case is that application domain-dependent interface signs should be heteromaterial, whereas computer(software)-related signs should be homomaterial (and not be expressed by analogies with outside world notions and objects). Apparently, although the guideline has been followed in this case, the resulting sign doesn't seem to be as easily interpretable as another one produced under the same conditions: the sign. The reason for the greater complexity of the Edit Graphic widget may lie in the level of abstraction it requires compared to the virtually nonexistent level of abstraction in the Mouse Cursor widget. The geometric objects in the former are raised to the condition of type-objects, whereas in the latter we can find a replica of a uniquely identifiable token-object.

Within the limits of UIL, the Edit Graphic widget defies our guideline about the selection of homomaterial signs because, by necessity, such signs end up by standing for themselves (expression collapses with content) which is a clear case of ratio difficilis (explicitly exemplified by Eco in [Eco'76]) — no established expression system is created by having referents being raised to the position of their own sign, unless we are prepared to go into natural systems where things "mean" what they are.

The problem with UIL©EUPL is that programs do generate other programs, tools do create other tools — software does produce software. It is not a straight forward replicant structure, because although Graphic Editors can and do produce the UIL signs used as their own expression, they can and do produce many other signs. The issue seems to be one of finding which signs should be used to express the notion of a variable, for instance. The low-level association with a fixed storage location for transient values in a program is too far away from task-specific jargon any user can understand. Much better is the notion of a universally

quantified type-like entity (as in the case of Prolog variables, for example) borrowed from the application's domain: character strings, paragraphs, or files. However, such type-like entities are (1) heteromaterial signs (i.e. they contradict our first approach to Semiotic Engineering) and (2) metonymical or metaphorical expressions of their true referents.

The problem with (1) can be easily solved by revising an incipient theoretical approach. The problem with (2), however, has been sensed by other researchers in the field of end-user programming and does not seem to have any promising solution in view. Successful EUPL's like spreadsheet languages, for example, treat cells as variables: but, just how far does the metaphor go? [Myers'92, Nardi'93] The real "message" from designer (in this case the extension language designer) to user is that programming is achieved by manipulating values across space and time in a computing machine [Gelernter & Jaggannathan'90].

Real programming capacity is not a local skill that users should gain unknowingly. Rather, it is a new kind of knowledge and talent any user should consciously acquire, like that of operating telephone sets, VCR's, microwave ovens, cars, and everyday technology in general. Therefore, the first step towards a solution is to pose the question of what knowledge (thus, what message) we want our users to get. The second is how to teach (thus, tell) them what they should learn. And it does seem to be the case that we want to have our users know something quite abstract, although not necessarily difficult, about computation.

Here is one of the hot issues involved in UIL☯EUPL: is there a semiotic system that can blend concreteness (of applications domains) and abstractness (of computing theory) into one "language"? Can this language be easily learned and used? Can this language be useful?

Our brief sketch of a more integrated EUP environment for WFW in 2.2 suggests that data structures of the EUPL should be borrowed from the application domain expressive system used in the UIL. However, such data objects should not be encapsulated entities because users would then be unable to write interesting extensions to software (as shown in our example). Data objects should be articulated structures, in accordance with the attributes and behavior such objects are shown to have in the UIL (the primary designer's message to users). For example, as users choose to format characters in WFW, they discover that this kind of objects have a number of attributes (see Figure 8). Of course, because they can operate upon such attributes in the UIL, they should be able to operate upon them in the EUPL as well. Thus, the data object should be structured in such a way that each attribute is some sort of "field" that the user can access and modify.

In a typically task-oriented UIL like WFW's, similar data structures are scattered throughout different access points. File formats are not found under the Format

menu, but rather under the File (Save, Save_as) menu (see Figure 10). Although it is a good solution for the UIL alone, it is not as good for integrated UIL☯EUPL since important generalizations are totally lost, like the fact that certain data can be interpreted as "[" or as "☯" depending on its "font" attribute, just like certain other data can be interpreted as ASCII characters or realistic rendering of images depending on its "file format" attribute. This particular generalization is perhaps one of the most important pieces of knowledge about computation — that the same finite collection of symbols may take on indefinitely many meanings depending on how we program the machine to react to its presence [Eco'88].

Thus, there is apparently a clash of important dimensions between task-oriented UIL design and what seems to be a use-oriented UIL☯EUPL design. Classification principles that organize signs into encapsulated vs. articulated systems of expression in UIL do not function equally well in the corresponding EUPL. The lines of similarities and contrasts behind user interface language grammars are drawn based on task-related situations, whereas with end user programming languages they are drawn based on reference and discourse situations. The natural consequence is that an integrated language design is exponentially more difficult to achieve.

In face of such design costs, we should ask ourselves about the benefits and gauge the cost/benefit ratio. Although end user programming is undeniably desirable, it is an open question whether an Extended Semiotic Engineering approach to UIL☯EUPL will lead users faster or farther to successful computing compared to competing visual programming, object-oriented programming or logic programming approaches. Whereas Semiotic Engineering has the onus of being an incipient approach which has provided as yet no corpus of fully designed and implemented UIL's, EUPL's, or UIL☯EUPL's, other approaches have the onus of being born within a community that has hardly, if ever, seriously considered communicative problems with programming languages for non-expert end users [Dertouzos'92]. Post-hoc integration with UIL's is as difficult as Word Basic has illustrated in our example.

5. Concluding Remarks

Our conclusions at this point are more a matter of speculation and sense than of thorough testing and analysis. Firstly, the new scenario for end user programming and human-computer interaction as a whole reinforces our belief in that user interfaces are, indeed, metacommunication artifacts that require careful semiotic design. The price to pay for even linguistically correct approaches, in which syntax

and semantics of interface languages are simple and efficient per se, seems to be high. The lack of communicative integration imposes large cognitive loads on users, who have to learn a host of novel constructs and notions as they move from one environment to the other, and eventually threatens software designers with the ghost of unusable (or underutilized) code along with the commercial risks associated with it.

Secondly, because of such semiotic requirements, human-computer interaction does seem to move into the realm of computer-aided interpersonal communication. This idea goes far beyond the most popular views of multimedia and networking systems being a new kind of environment for people to interact with each other, and brings forth the original connections between computing and semiotics perceived by Peirce himself [Peirce'31] and tracked by other researchers over the years [Zemanek'66, Kammersgaard'88, Nadin'88, Eco'88, Andersen'90]. A major change in HCI can be expected if designers realize that they, and not the system they write, are talking to users over the interface and if users, in their turn, realize they are not getting the machine's or the system's message, but the designer's.

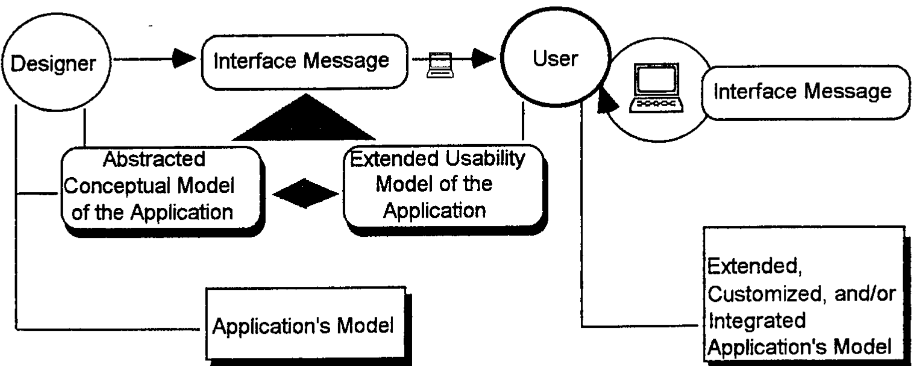

Figure11: The Semiotic Engineering Framework for UIL©EUPL

Thirdly, if users realize that systems are messages (discourse) somebody has written and sent to them, they may be encouraged to use the same resource to write their own messages (to themselves and/or to others, in a collaborative environment [Nardi'93]), as is suggested in Figure 11. This leap could certainly represent a turning point in computing, as pointed out by the SIGCHI'90 Working Group Report we have mentioned in Section 1. However, the leap can only take place if user interface design reflects a new stand in HCI.

Fourthly, the balance between concrete and abstract message elements in an integrated UIL©EUPL environment requires some gradual progression of task-specific domain-dependent interactive languages to more general models of computational solutions to everyday problems. Oberlander and Stenning have provided a most interesting framework in which representation systems can be classified and shown to function as a resource for mastering abstract reasoning

[Oberlander & Stenning'95]. According to these authors, limited-abstraction representation systems, in which people can take certain symbols to stand for higher-order type objects, with undefined features, help them manage the

difficulties of reasoning in highly abstract terms. Along this line, end user programming may well evolve from a limited-abstraction programming language that does not fully implement the idea of Turing Machines, but still lead users to the practical benefits of everyday computing. Among the important features included in this language is efficient type/token distinction and expressibility in program texts, and powerful self-reference mechanisms, which together encourage and facilitate some of the relevant abstractions.

Finally, there are some points that can be added to a research agenda in Semiotic Engineering regarding the integration of user interface languages to end user programming environments. These are the following:

1. Find out if there is a global design orientation that can organize both the UIL and the EUPL semiotic systems into one UIL☯EUPL communicative setting
2. Select minimal data and control structures for EUPL's
3. Identify a general procedure to reuse UIL signs as articulated type-signs that can function as variables
4. Embed the UIL within the EUPL environment, in an apparent reversion of what is commonly believed to happen (embed the EUPL within the UIL)
5. Carry out a host of empirical studies to understand more deeply the semiotic nature of interacting with computers and writing programs, and the hypotheses that emerge from such understanding.

As with the topics included in other research agendas in this area [Myers'92, Nardi'93, Eisenberg'95], the above are expected to contribute to further our understanding about the nature of computing and the kind of quality it may bring to individuals, communities and societies in the future.

Acknowledgements

I would like to thank all my graduate students who volunteered to participate in a work group on Semiotic Engineering last winter and spring. I am especially thankful to Isa Haro Martins, Jair Cavalcanti Leite and Deller James Ferreira for their patient listening and helpful questioning during the first presentation of the ideas contained in this paper. Very special thanks go to Ana Cristina Bicharra Garcia for her enthusiastic support, constant understanding, and immense tolerance to recent changes in our projects priorities due to my total involvement with the issues presented in this paper. I am also grateful to CNPq, the agency which has given me a grant to do research about Semiotic Engineering and to participate in the Dagstuhl Seminar on Informatics and Semiotics. Last, but not least, I'd like to thank Luiz Fernando Gomes Soares for his friendship and help with the high- and low-tech problems of text editing and pretty printing.

6. References

[Adler & Winograd'92] Adler,P. and Winograd,T. (1992) *Usability : Turning Technologies into Tools*. New York. Oxford University Press.

[Andersen'90] Andersen,P.B. (1990) *A Theory of Computer Semiotics*. Cambridge. Cambridge University Press.

[Andersen'93] Andersen,P.B. (1993) A Semiotic Approach to programming. in Andersen, Holmqvist and Jensen (Eds.) *Computers as Media*. Cambridge. Cambridge University Press.

[Apple'92] Apple Computer, Inc. (1992) *Macintosh Human Interface Guidelines*. Reading,Ma. Addison Wesley.

[Chang'90] Chang,S. (1990) *Visual Languages and Visual Programming*. New York. Plenum Press.

[Cordy'92] Cordy,J. (1992) Why the User Interface is NOT the programming language : and how it can be. in Meyers,B. (Ed.) *Languages for Developing User Interfaces*. Boston. Jones and Bartlett. pp. 91-100

[Cypher & Canfield Smith'95] Cypher,A. and Canfield Smith,D. (1995) KidSim: End User Programming of Simulations. *Proceedings of CHI'95*. ACM Press. pp.27-34

[de Souza & Ferreira'94] de Souza,C.S. and Ferreira,D.J. (1994) Especificações Formais para Linguagens Visuais de Programação. *Anais do VII SIBGRAPI*. Curitiba,Pr. SBC/UFPr. pp.181-188

[de Souza'93] de Souza,C.S. (1993) The Semiotic Engineering of user interface languages. **International Journal of Man-Machine Studies**. No.39. pp. 753-773

[Dertouzos'92] Dertouzos,M. (1992) The user interface is the language. in Meyers,B. (Ed.) *Languages for Developing User Interfaces*. Boston. Jones and Bartlett. pp. 21-30

[Draper'86] Draper,S.W. (1986) Display managers as the basis for user machine communication. in Norman and Draper (Eds.) *User Centered System Design*. Hillsdale. Lawrence Erlbaum and Associates. pp. 339-352

[Eco'76] Eco,U. (1976) *A Theory of Semiotics*. Indiana University Press

[Eco'88] Eco,U. (1998) On truth: a fiction. in Eco, Santambrogio and Violi (Eds.) *Meaning and Mental Representations*. Bloomington. Indiana University Press

[Eisenberg'95] Eisenberg,M. (1995) Programmable Applications: Interpreter meets Interface. **SIGCHI Bulletin**. Vol.27, no.2. pp.68-93

[Gelernter & Jagannathan'90] Gelernter,D. and Jagannathan,S. (1990) *Programming Linguistics*. Cambridge,Ma. MIT Press.

[Hutchins, Hollan & Norman'86] Hutchins,E.L.; Hollan,J.D.; and Norman,D.A. (1986) Direct Manipulation Interfaces. in Norman and Draper (Eds.) *User Centered System Design*. Hillsdale. Lawrence Erlbaum and Associates. pp.87-124

[Kammersgaard'88] Kammersgaard,J. (1988) Four different perspectives on human-computer interaction. **International Journal of Man-Machine Studies**. No.28. pp.343-362

[Lieberman & Fry'95] Lieberman,H. and Fry,C. (1995) Bridging the Gulf between Code and Behavior in Programming. *Proceedings of CHI'95*. ACM Press. pp.480-486

[Marcus'92] Marcus,A. (1992) *Graphic Design for Electronic Documents and User Interfaces*. New York. ACM Press.

[Microsoft'95] Microsoft Corporation (1995) *The Windows Interface Guidelines for Software Design*. Redmond. Microsoft Press.

[Mullet & Sano'95] Mullet,K. and Sano, D. (1995) *Designing Visual Interfaces*. Menlo Park. SunSoft Press.

[Myers'92] Meyers,B. (1992) *Languages for Developing User Interfaces*. Boston. Jones and Bartlett.

[Myers, Smith & Horn'92] Myers,B.; Canfield Smith,D.; and Horn,B. (1992) Report of the End User Programming Working Group. in Meyers,B. (Ed.) *Languages for Developing User Interfaces*. Boston. Jones and Bartlett. pp. 343-366

[Nadin'88] Nadin,M. (1988) Interface design: A semiotic paradigm. **Semiotica**. Vol.69. Nos. 3/4. pp. 269-302

[Nardi'93] Nardi,B. (1993) *A Small Matter of Programming*. Cambridge, Ma. MIT Press

[Oberlander & Stenning'95] Stenning,K. and Oberlander,J. (1995) A Cognitive Theory of Graphical and Linguistic Reasoning: Logic and Implementation. **Cognitive Science** Vol.19. No.1. pp.97-140

[Peirce'31] Peirce,C.S. (1931) *Collected Papers*. Cambridge,Ma. Harvard University Press.

[Shu'88] Shu,N. (1988) *Visual Programming.* New York. van Nostrand Reinhold.

[Zemanek'66] Zemanek,H. (1966) Semiotics and Programming Languages. **Communications of the ACM**. Vol.9. No.3. pp. 139-143.

[Zloof'81] Zloof,M. (1981) QBE/OBE: A language for office and business automation. **IEEE Computer**. May-1981. pp. 13-22