# PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 19/96

## End-User Programming Environments: the Semiotic Challenges

Clarisse Sieckenius de Souza
Simone Diniz Junqueira Barbosa

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 19/96

Editor: Carlos J. P. Lucena June, 1996

# End-User Programming Environments: the Semiotic Challenges *

Clarisse Sieckenius de Souza

Simone Diniz Junqueira Barbosa

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

**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

# End-User Programming Environments: The Semiotic Challenges

*Clarisse Sieckenius de Souza* *Simone Diniz Junqueira Barbosa*

*Departamento de Informática — PUC-Rio*
*R. Marquês de São Vicente 225*
*22453-900 Rio de Janeiro, RJ - Brasil*
*email: [clarisse,sim]@inf.puc-rio.br*

**PUC-Rio InfMCC19/96 Junho'96**

## Abstract

Empowering users with programming abilities is an issue of increasing importance in the software industry today. A few challenges to be faced are the different trends between interface design and programming language design, the lack of "semantic" embedding in programming languages, and users' fear and lack of motivation to try to program. End-User Programming Languages (EUPL) and User Interface Languages (UIL) are a peculiar piece of communication between system designer and system user, which form a continuum with each other, and which once broken can seriously impact the whole endeavor of end-user programming.

A Semiotic approach to the problem will consider the integration between EUPL and UIL as a dynamic integrative process, like the processes of interpreting, understanding, and learning that users should go through when programming. We will illustrate how the notion of an interpretant can organize and clarify the relationships that hold between a single application's UI and its corresponding EUP Environment. Interpretants are crystallized into a predictable set of variations which can be used to adjust meaning to a very precise configuration of semantic parameters pertinent to a certain application's domain. This approach has the potential to explain and predict, in theoretic terms, the points of breakdown from UI to EUP Environment.

Dar aos usuários a capacidade de programação é um tópico que se tem tornado cada vez mais importante na indústria de software. Alguns dos desafios que se apresentam são as diferentes tendências seguidas pelo design de interfaces e de linguagens de programação, a falta de fundamentação semântica de domínio destas, bem como o medo ou falta de motivação para programar por parte dos próprios usuários. Linguagens de Programação para Usuários Finais (End User Programming Languages — EUPL) e Linguagens de Interface com Usuários (User Interface Languages — UIL) são um tipo especial de mensagem entre designer e usuário de sistemas, formando um contínuo que, uma vez rompido, pode causar um impacto negativo sobre toda a tentativa de programação por parte de usuários finais.

Uma abordagem semiótica para o problema considerará a integração entre EUPL e UIL como um processo dinâmico e integrador, tal como os processos de interpretação, compreensão e aprendizado pelos quais os usuários devem passar quando programam. Ilustraremos como a noção de interpretante pode organizar e esclarecer as relações entre a interface de uma dada aplicação e o correspondente ambiente de programação para usuário final. Os interpretantes são cristalizados em um conjunto previsível de variações que pode ser utilizado para regular significados até atingir-se uma configuração bastante precisa de parâmetros semânticos pertinentes ao domínio da aplicação. Esta abordagem tem o potencial de explicar e prever, em termos teóricos, os pontos de ruptura entre a UIL e a EUPL.

## Keywords

Semiotic Engineering, End-User Programming Environment, Computer Semiotics

## Introduction

End-User Programming environments have been increasingly thought to be a major issue in software industry. The need for ever more sophisticated and adequately tailored applications to serve a growing variety of user communities and technological requirements haunts developers and makes them fear that perhaps only legions of designers and programmers will be able to respond to the current challenges [Myers, Smith & Horn '92]. Consequently, empowering users with programming abilities that allow them to customize, extend, and create applications on their own has become a valuable alternate solution for the impending crisis.

Nevertheless, in spite of its appeal, the design of good End-User Programming environments is still more a matter of an art and talent than the design of good User Interfaces. Firstly, the major trends in interface design have been virtually opposite to those in programming language design. 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. Secondly, even when "linguistic" approaches to design are taken [Gelernter & Jagannathan '90], the programming languages produced may be unwieldy for end-users for lack of "semantic" embedding [Nardi '93]. Thirdly, end users are often scared of or totally unmotivated by the idea of programming. They may find it too difficult, too boring, too laborious, too time-consuming, to even try to build new programs or change the ones they use.

A number of solutions have been proposed over the last years. Programming by Example [Zloof '81], Visual Programming [Shu '88] and Programming by Demonstration [Cypher & Canfield Smith '95] are some of the best known approaches. The importance of fun and pleasure [e.g. Dertouzos '92, Nardi '93] and of learning [e.g. Kay '91, Eisenberg '95, DiGiano & Eisenberg '95], aligned with the need for increased usability [Bowen '89], has led researchers to explore the borderlines between users and programmers [Guzdial, Reppy & Smith '92]. Instantiating the results of such efforts are KidSim [Cypher & Canfield Smith '95], Chart'n'Art [DiGiano & Eisenberg '95], Tinker [Lieberman '93a] and Mondrian [Lieberman '93b].

What none of the above approaches above has explicitly dealt with is the fact that End-User Programming Languages (EUPL), just like User Interface Languages (UIL), are a peculiar piece of communication between system designer and system user [de Souza '93, de Souza '94]. Not only are they tokens of a similar type of communicative artifact, but they actually form a continuum with each other, which once broken can seriously impact the whole endeavor of end-user programming [de Souza '96]. The increasing awareness of the role of computers as medium for an unprecedented kind of communication has invoked Semiotics to contribute and supply some theoretical and methodological background for analysis and design of software in general [Nadin '88, Andersen '90, Andersen '93, Mullet & Sano '95;Nake '94].

What we propose in this paper is a semiotic approach to what we have named UIL[EUPL (pronounced UIL '*tao*' EUPL) design [de Souza '96]. The sign [, borrowed from ancient Taoist tradition, stands for a dynamic integrative process, never exhausted and always moving, like the processes of interpreting, understanding, and learning that users should go through on the road to becoming programmers. Our main point is to illustrate how the central notion of an interpretant [Peirce '31] can organize and clarify the relationships that hold between a single application's User Interface and its corresponding End-User Programming Environment. We take the example of an existing application (a former version of MS Word for Windows™) and provide analytical evidences of problematic EUPL design in WordBasic™. Then, we show how to improve design through the use of an open interpretant, up to the limits of two primarily discontinuous and independent communication codes used in this application. Finally, we discuss the profile of continuous interdependent communication codes in UILEUPL, stressing the need for integrated design from the beginning and showing evidences of some major constraints for reusability in terms of communicative software artifacts.

## The Semiotic Engineering of End-User Programming Languages

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:

〈 The Interface Message is a One-Shot Communicative Act from Designer to User;

〈 The Designer's **Intended** Meaning and the User's **Assigned** Meaning for the Interface Message are not the same, though they should be obviously consistent with each other;

〈 The Designer's Intended Meaning is an abstraction of the Application's Model;

〈 The User's Assigned Meaning is a model of the Application's Usability[1] Model.

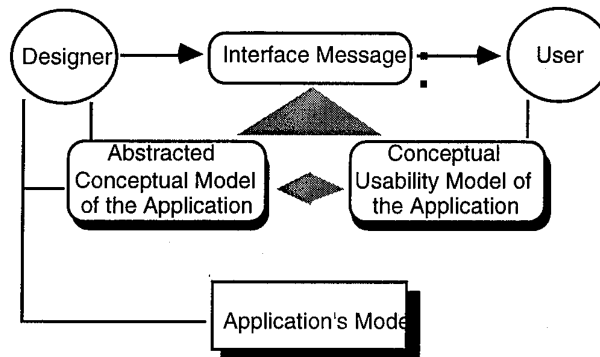
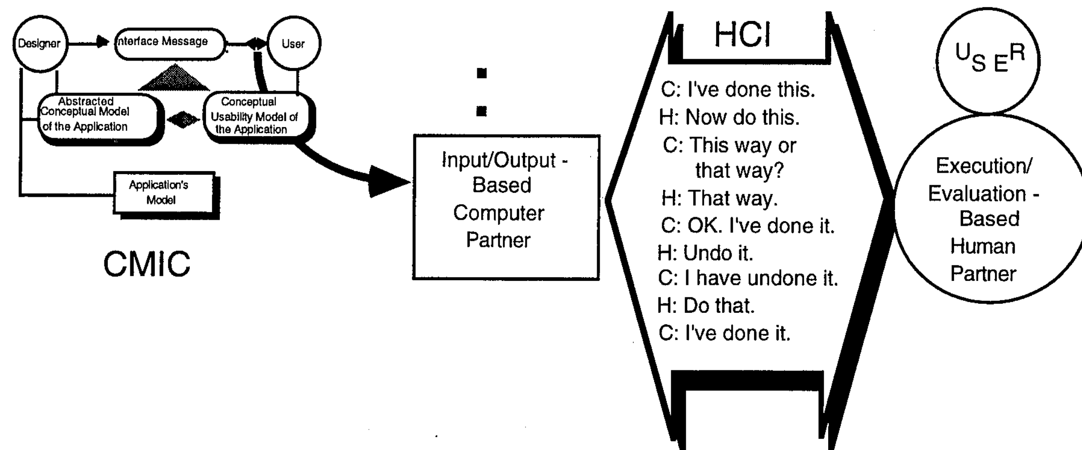



*Figure 1 – Meanings of the User Interface Message*

In a Semiotic Engineering framework [de Souza '93, de Souza '94], 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. 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.




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]

*Figure 2 – Human-Computer Interaction in a Semiotic Engineering Framework*

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. Even more so, 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 3. This leap could certainly represent a turning point in computing, as pointed out by the SIGCHI'90 Working Group Report [Myers, Smith & Horn '92]. However, the leap can only take place if user interface design reflects a new stand in HCI.

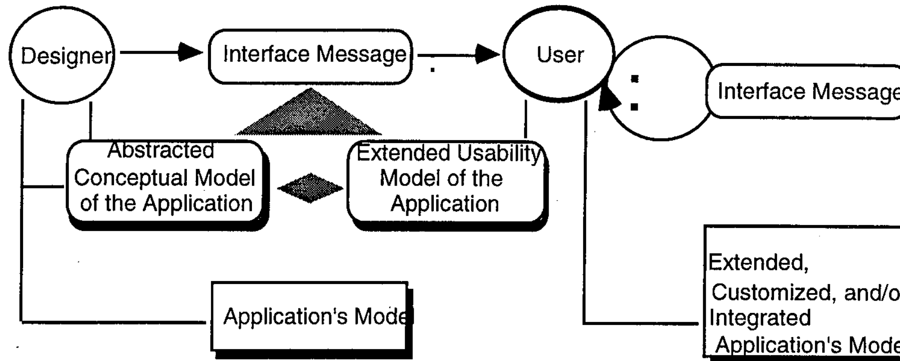



*Figure 3 – The Semiotic Engineering Framework for* UIL[EUPL

One of the most important contributions of Semiotics to HCI and Computer Science as a whole is the notion of *interpretant*, proposed by C.S.Peirce in the turn of this century [Peirce '31]. A sign (i.e. that which represents something) is related to both an Object (i.e. the thing it stands for) and an interpretant (i.e. a feeling, an exertion or another sign) in a triadic schema as shown in Figure 4. The example illustrates that the sign <home> (a word in English) stands for a concrete object existing in space and time relative to an individual's life. However, the sign may be interpreted in indefinitely many layers of meaning bringing up a variety of other signs and meanings to mind. This process of generating a chain of meanings is called *semiosis,* another powerful notion in our context because of its unbounded nature (unlimited semiosis) and critical consequences for communication.

The gist of this semiotic background is that when two individuals communicate with each other they negotiate and regulate meanings in conversation, so that unlimited semiosis becomes pragmatically constrained to a territory of mutual understanding. There is no guarantee (and in fact there is very little chance) that when someone says "How are you?" and someone else replies "Fine, thanks. And you?" the meaning of <fine> is the same for both people. Nevertheless, communication is apparently successful at this level because somehow these people's interpretants converge to a stable configuration of understanding in both minds. When there are evidences that such convergence has not been met, people engage into using language to regulate the meaning of language (metalinguistic behavior). Suppose the preceding dialogue proceeds as follows:

— Are you going home soon?

— Not a chance... I have my job here and I can't afford to leave it behind and go back to Baskerville.

— Oh, no! Sorry. I didn't mean that. I was asking if you were going back to your house in the next couple of hours. I just thought you would like to ride along.

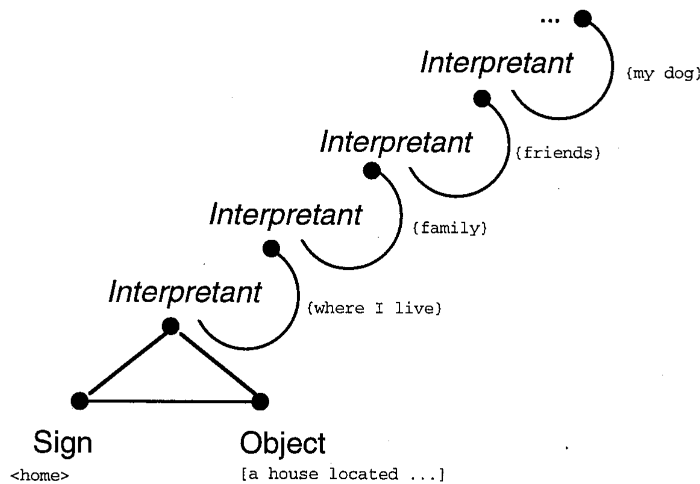



*Figure 4 –Interpretants and Unlimited Semiosis*

Going back to Semiotic Engineering, a similar process takes place at the User Interface level. Interface designers may mean <home> in the more spatial sense, but users may always take it in the emotional sense. The difference is that no direct negotiation takes place (interfaces are one-shot messages), and for as long as the equivocal interpretant lives there lives the interactive trap which can catch users and threat software usability. Some design guidelines may help us choose interface signs that reduce the chances of our being misinterpreted by users (de Souza'93; de Souza'94), but just like in any other communicative situation among humans there are no guarantees.

Stretching the frontiers beyond the territory of the application's functional interface (see snapshots of this territory in Figures 5, 6, 7 or 8 ahead), when it comes to End-User Programming environments the regulation of interpretants is even more crucial because it feeds back semiosis at both levels of operation: Interface and Programming. The disclosure of programming behind interaction [DiGiano & Eisenberg '95], through echoing that in MS Word [Microsoft '94] formatting the font of a character string (Figure 5) amounts to executing a specific command in WordBasic, triggers an abduction process in interpretation, through which from the observed results users try to infer the rules governing the correspondence between interface event and program execution. In this context, MS Word interface should help us understand WordBasic and vice-versa, so that in the long run we can change and extend our MS Word environment by programming the addition of tailored functionalities in the original software.

The Semiotic Engineering of End-User Programming Languages thus requires that a continuum of signs be used in the overall system, so that feedback is not only possible between UIL and EUPL, but that it becomes the very basis for empowering users to perform programming tasks. The links in this continuous chain are the interpretants which are given a physical stance in both languages. Users can actually operate upon such interpretants, in ways that have no precedent in human communication. This singularity, though limiting in immediacy if compared to the negotiation and regulation patterns arising in face-to-face conversation between people, is quite unique in terms of precision. It ultimately allows a user to work on an interpretant to the point that it is perfectly crystallized in a fixed and adequate meaning for a given sign. The meaning is *engineered* in the most genuine sense of the word.

## An Example

MS Word allows for a varying degree of user customization and extension of the original application. Through the <Options> tool accessible from the menu bar, users can configure the interface and application to include only a specific set of tools, to display a customized interface layout, or use a number of default parameters for file handling and formatting, for example. One step ahead, MS Word offers users a <Record Macro> tool with which they can trigger a recording device that collects a fixed sequence of interface events under a unique name and perform the corresponding functions every time the named

option is invoked from then on. Although the user may never realize it (for lack of immediate echoing at the interface level), such macros are WordBasic commands captured in a small program.

The next and most powerful end-user programming resource is WordBasic itself, a macro programming language which users can wield to introduce novel extensions and enhanced functionality to the software. As its name suggests, WordBasic is an adaptation of MS Basic™ which supplies programmers with a variety of constructs for software engineering. In the following, we'll pursue two instances of end-user programming activities in the MS Word environment to show aspects of semiotic analysis at play.

Firstly, let us suppose a user wants to record a macro that changes certain font characteristics in text. The <Format Font> dialog (Figure 5) is already familiar, so he/she has no problem interacting with its elements.

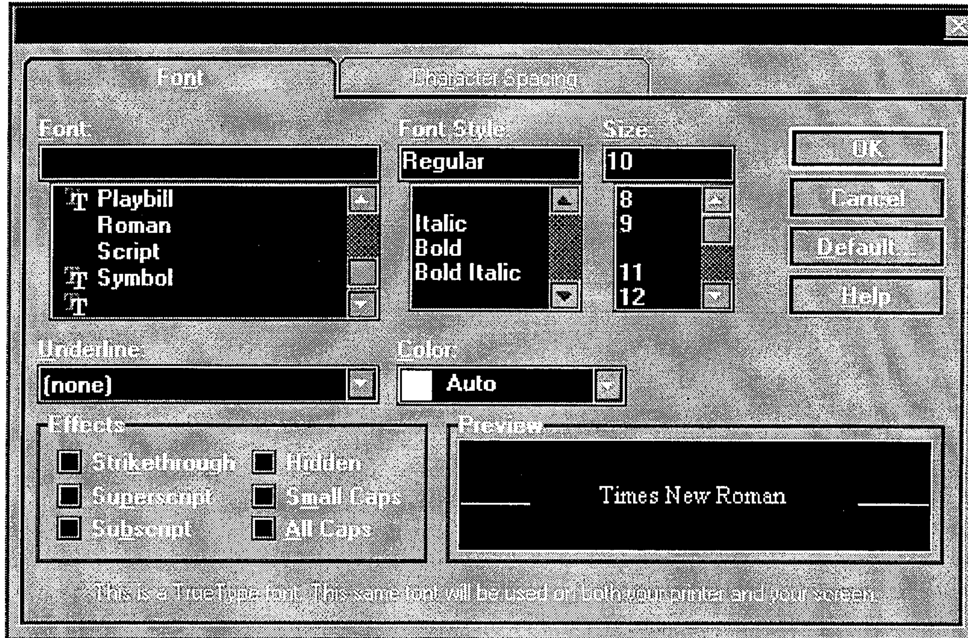



*Figure 5 – Format Font dialog box.*

The recorded macro would contain the following singular statement:

```
FormatFont .Points = "10", .Underline = 0, .Color = 0, .Strikethrough = 0, .Superscript = 0, .Subscript = 0,
.Hidden = 0, .SmallCaps = 0, .AllCaps = 0, .Spacing = "0 pt", .Position = "0 pt", .Kerning = 0, .KerningMin =
"", .Tab = "0", .Font = "Times New Roman", .Bold = 0, .Italic = 0
```

A brief semiotic analysis uncovers some discontinuity problems, due to different levels of articulation between the UIL elements and the EUPL ones. The UIL introduces certain groupings (complex signs) that are absent in the EUPL. For instance, the sign <Regular> in the <Font Style> options list does not appear in the EUPL, being instead mapped onto <.Bold = 0, .Italic = 0>. Of course, abductive inferences might eventually lead users to realize the correct correspondence (and, thus, the correct meaning). But it might not, as well.

Other discontinuities appear in a <Replace> command, for example. Suppose the user makes a mistake while writing a person's name throughout a whole long document, and wants to correct it by using the <Replace> tool. Consider the sample letter fragment below.

> Ms. Mary Jonson has been employed by this company for the past five years, during which time her services as secretary-typist have been eminently satisfactory. She is quick and efficient, punctual, meticulous in her work — in every way a model employee. It is with regret that we accept Ms Jonson's resignation, due to the fact that her family is moving away from this city. We recommend her unhesitatingly to any prospective employer.

The user a spelling mistake was made and wants to replace all occurrences of Jonson with Johnson. By interacting with the <Edit Replace> dialog box (Figure 6) by means of the <Replace All> command, MS Word makes the necessary changes and the UIL returns the number of replacements made, as illustrated in Figure 7.

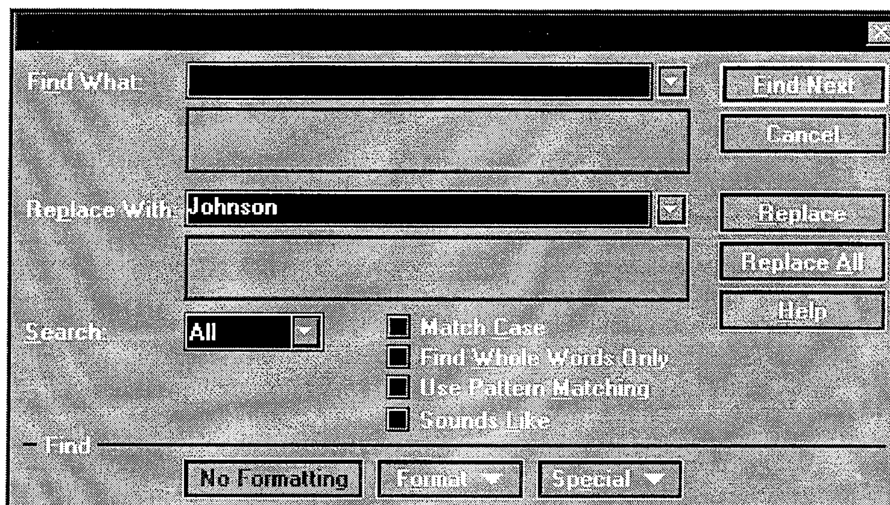



*Figure 6 – Edit Replace dialog box*

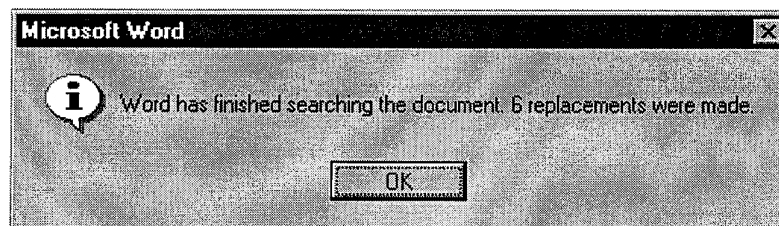



*Figure 7 – Message box displaying number of changes made after a replace command*

If the user wanted to record a macro to execute this command, the resulting recorded statement would be as follows:

```
EditReplace .Find = "Jonson", .Replace = "Johnson", .Direction = 0, .MatchCase = 0, .WholeWord = 0,
.PatternMatch = 0, .SoundsLike = 0, .ReplaceAll, .Format = 0, .Wrap = 1
```

Notice that, once again, we do not have a one-to-one correspondence between UIL and EUPL signs. For instance, the <Search> parameter is mapped to some combination of the <Direction> and <Wrap> parameters. Another problem related to the encapsulation of the iteration structure under the <ReplaceAll> parameter sign will be discussed shortly.

Ideally, we should change the UIL, the EUPL or both. However, supposing that this is not possible or desirable for a number of reasons [Guzdial, Reppy & Smith '92], we should try to help users around the problem, taking certain remedial actions. Our first suggestion is to use the UIL to interpret the EUPL. In other words, to take UIL signs as the *interpretants* of the EUPL.

The Word environment already provides a mechanism to create EUP commands by means of the UIL (macro recording). On the path towards a UIL[EUPL environment, when a user edits a macro, WordBasic syntax can be used to generate the corresponding UIL signs (dialogs). Each application-dependent WordBasic command would include a hypertext anchor which could trigger an interpreter, and thus return the corresponding interpretant (or other UIL sign), with the instantiated parameters.

For instance, by clicking on <<u>FormatFont</u>> on the command text below, the interpreter would show the dialog illustrated in Figure 8.

```
FormatFont .Points = "10", .Underline = 0, .Color = 6, .Strikethrough = 0, .Superscript = 0, .Subscript = 0,
.Hidden = 0, .SmallCaps = 0, .AllCaps = 0, .Spacing = "0 pt", .Position = "0 pt", .Kerning = 0, .KerningMin =
"", .Tab = "0", .Font = "Times New Roman", .Bold = 1, .Italic = 0
```

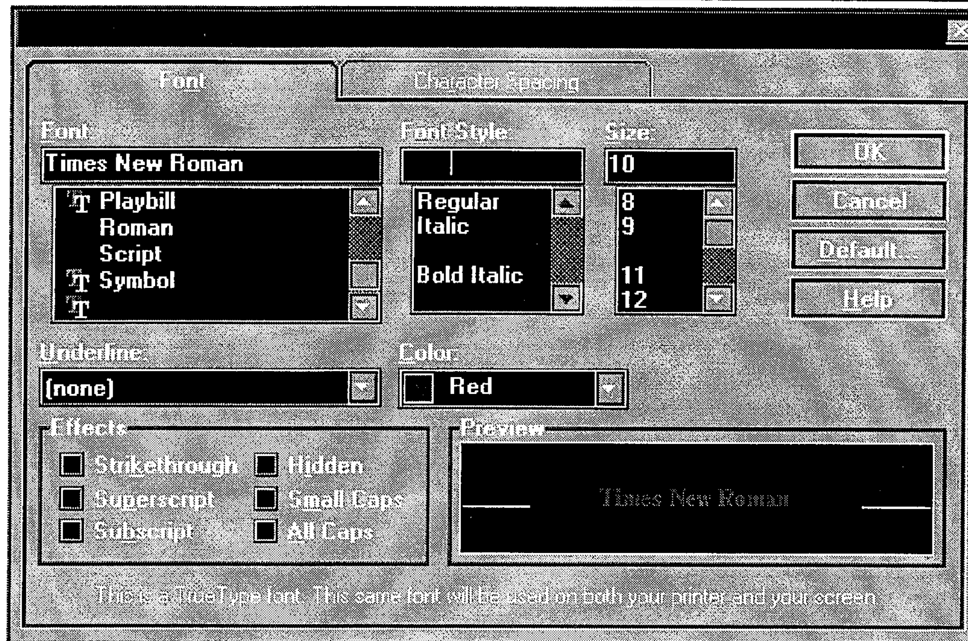



*Figure 8 – Dialog box constructed by the interpreter of a FormatFont command*

An attentive user may become aware of such discontinuities as the use of the UIL sign <red>, supported by the actual rendering of a red little box next to the option and the sample text in the <Preview> area, and the EUPL sign <6> accompanying the parameter <.Color>. Also, he/she may be aware of the problems associated to the expression <Regular> in the UIL, as well as the expression <Bold Italic>, and its sophisticated mapping onto a combination of boolean values assigned to <.Bold> and <.Italic> parameters. All these can be turned into a more difficult puzzle by the fact that such things as <.Kerning>, <.Tab> may never be understood if the user never brings forth the second UIL sentence (Character Spacing) lying behind the default layer (Font).

Systematic mappings of EUPL instructions onto interpretants expressed in the UIL can allow us to edit macros by changing the meanings of the interpretant itself. Suppose that the user now wants that his/her macro performs a different character formatting action: Blue should be the color of text, and not red. The difficulties of guessing that Blue maps onto <.Color = 2> could be easily overcome by a single selection of the option <Blue> in the interpretant (Dialog Box) to ensure that the original macro is now correctly changed to:

```
FormatFont .Points = "10", .Underline = 0, .Color = 2, .Strikethrough = 0, .Superscript = 0, .Subscript = 0,
.Hidden = 0, .SmallCaps = 0, .AllCaps = 0, .Spacing = "0 pt", .Position = "0 pt", .Kerning = 0, .KerningMin =
"", .Tab = "0", .Font = "Times New Roman", .Bold = 1, .Italic = 0
```

This mechanism goes a long way into helping users understand how interactions correspond to WordBasic code and vice versa. Although it does not eliminate the perceived discontinuities of what should be a UIL[EUPL environment, it remedies some of the undesirable obscurities end users have to decipher in WordBasic macro programming.

According to Myers [Myers '92], end-user programming languages should provide elementary programming structures and resources, such as: variables, conditionals and loops. Our goal now is to see if we can use the same UIL interpretants to provide users with the suggested programming abilities. WordBasic commands do make use of variables, conditionals and loops. However, the signs used to represent them are abysmally far from the UIL signs. As a very straightforward example, take the case of variables. Any macro using a variable should look somewhat like the following:

```
Dim a$
a$ = "c:\"
a$ = a$ + Filename$()
...
```

Nowhere in the whole extension of UIL signs and sentences will a user find anything like <a$> or <Dim>. Worse still, nowhere in the typical programs generated by macro recording will these signs and others like <if...then> (conditionals) or <while...wend> (loops) be found. Take the interesting example of the <replace> command above. The sign <.ReplaceAll> hides a complex programming structure in which loops and conditionals are used. Programmers realize <.ReplaceAll> is a procedure composed by things like:

〈 create loop structure that tests the end-of-file condition

〈 if condition is false replace some string by some other and resume loop

〈 if condition is true then stop loop

But end-users are by no means motivated to think of any constructs of the above. The semiotic challenge is then to provide the adequate links (interpretants) in order to restore the continuum from interface to programming environment. As a first step, we conceive of two sorts of variables: interactively-assigned variables and procedurally-assigned variables. The former are those whose values are set by the user during the macro execution, within dialogs, such as:

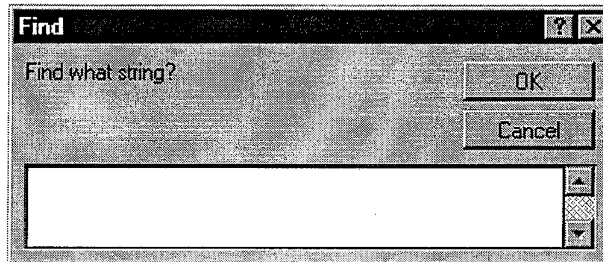



*Figure 9 – Input box requesting the value of a variable*

The latter are those whose values are computed by procedures programmed in an application-independent macro, such as variables I and J in the following WordBasic program:

```
Dim I, J
J = 4
For I = 1 to j
    Beep 48
Next
```

In order to provide a mechanism for defining the interactively-assigned variables, we will extend the interpretant dialog by adding buttons next to each possible variable type, as in Figure 10.

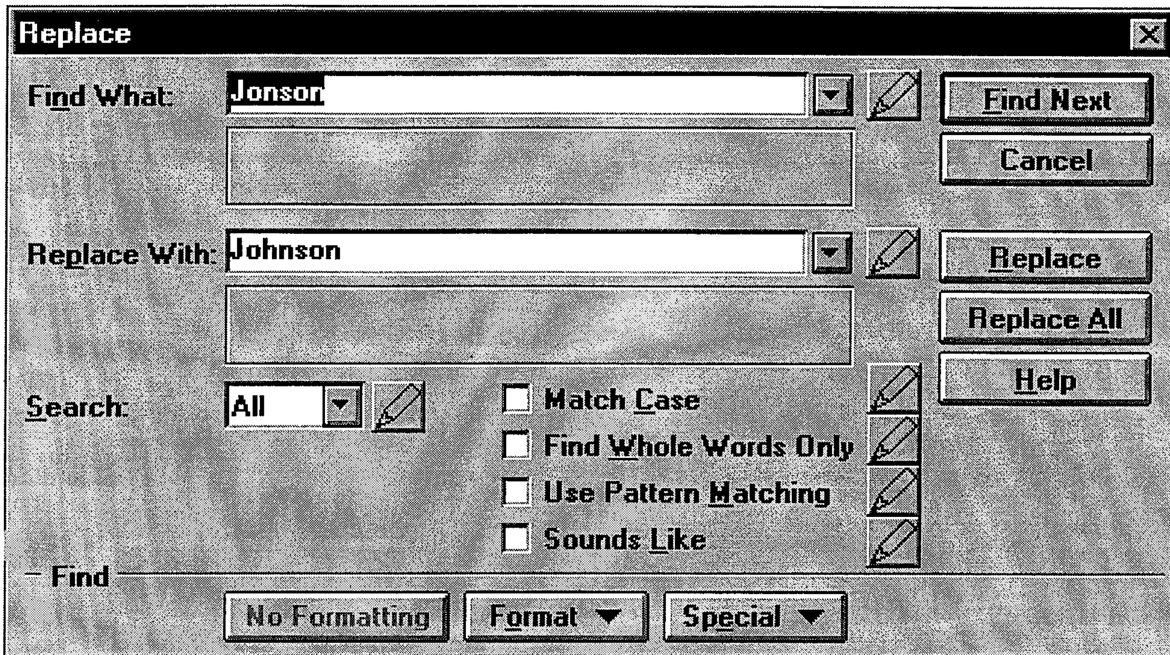



*Figure 10 – Modified dialog box allowing interactive variable definition*

This way the user can define his variables to be assigned at run-time, without having to bother about variable naming, typing, and so on. The corresponding program would automatically include declarations and input box functions, as in:

```
Dim a$
a$ = InputBox$("Replace with what?", "", "")
EditReplace .Find = "Jonson", .Replace = a$, .Direction = 0, .MatchCase = 0, .WholeWord = 0,
.PatternMatch = 0, .SoundsLike = 0, .ReplaceAll, .Format = 0, .Wrap = 1
```

This approach could be somewhat tiresome when the user defines many variables within a single dialog, because it would generate many InputBox$ commands, and consequently many interactions. Another approach would be to present the same dialog box used to generate the command (Figure 10), where the only editable fields would be the requested variables.

Non-interactive variables, however, have to let go of UIL interpretants, by definition. They belong to a more abstract realm of signs - programming language signs - which should be connected to the continuous flow of semiosis via another kind of link. Instead of a UIL interpretant, such signs should be understood based on EUPL interpretants. The anchor mechanism used to call forth the interpretant of WordBasic application-dependent instructions such as FormatFont could be now used to call forth WordBasic application-independent programming structures as in the following:

```
FileSaveAs.Name="sbes'96.doc",.Format=108,.LockAnnot=0,.Password=""
```

A click on FileSaveAs calls forth the UIL dialogue box, whereas a click on Name="sbes'96.doc" calls forth a EUPL text like:

```
Dim a$
a$= FileName$()
```

Another click on FileName$() will disclose internal parameter passing mechanisms existing in function calls. This idea is extensively used by DiGiano [DiGiano & Eisenberg '95, DiGiano '96] to provide learning opportunities users can take or not. Disclosure is, in the author's words, "a gentle introduction to end-user programming". Although no reference is made to semiotics is his work, it is clear that this mechanism restores the chain of successive interpretants arising in semiosis.

The progressive disclosure of EUPL interpretants is also the means by which conditionals and loops are introduced to users. However, unlike what happens with UIL interpretants, for reasons of self-reference, EUPL interpretants cannot be freely altered by users without consequences to the running program itself. Suppose that, just like our user changed the FormatFont macro above, by choosing <Blue> instead of the original <Red> color in the UIL interpretant, he now decided to change <Dim a$> to <Dim MyFile$> in the EUPL interpretant text. The obvious consequence is that the program would not run as before, since a constituent line of its code was altered to include a novel sign it does not know how to handle.

Self-reference is a major semiotic issue in programming [de Souza '96] and it constrains the limits of reversibility in using the interpretants as a common platform for designer and user expression. In the following section we discuss some of the prominent aspects of our proposed approach and highlight what we think of as the main contributions of Semiotic Engineering to end-user programming.

## A Brief Discussion

### What does UILEUPL amount to?

A UIL[EUPL environment is a means to meet extensive software usability and adaptability, as shown in Figure 3 (section 1). It consists of a semiotic continuum that may lead users from User Interface to Programming Environment, in an unbroken chain whose links are interpretants of signs in one language to signs in another language. Such interpretants, unlike typical interpretants derived from human communicative settings, are crystallized into a predictable set of variations which can be used to adjust meaning to a very precise configuration of semantic parameters pertinent to a certain application's domain. The first step of the overall UIL[EUPL semiotic continuum schema is portrayed in Figure 11.

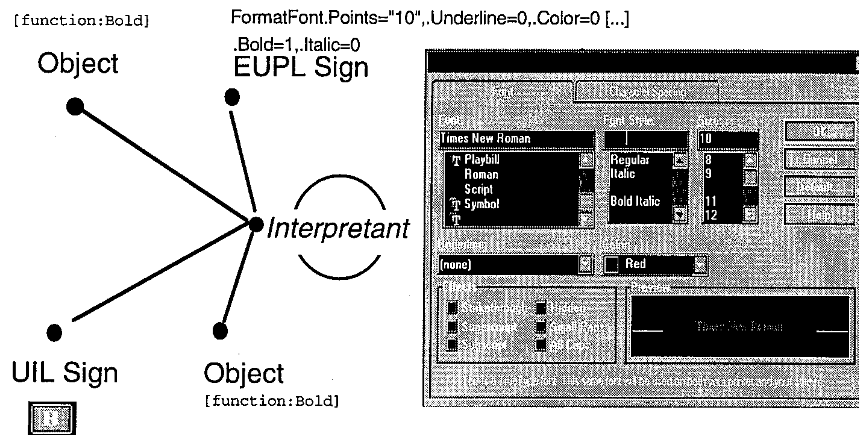



*Figure 11 – Overall UIL[EUPL semiotic continuum schema*

Notice that the correctness of translation is marked by the co-referentiality of the objects involved in communication, whereas the comprehension is marked by the coincidence of interpretants. Further analysis of the schema reveals that by fixing objects and interpretants, our continuum strives to restore a dyadic structure of sign, in which a given sign has just one ideal given meaning. Although this is blatantly false in all-human settings because of unlimited semiosis, it is blatantly true in computer-human settings where, by the very nature of computation, symbolic processing must stop at some point.

### Not all users are, can be, or would ever bother to be programmers

All of this situation does not mean or require that all users could, should, or would wish to become programmers. Everyone is free to choose not to learn how to program, in the first place, and some people may lack the basic intellectual abilities to deal with abstractions required for even the most primary levels of programming literacy. Inspired by previous research in HCI as a whole, we think a designer should support (offer "affordances") and not request (assume "requirements") the development of certain user abilities and capacities. Nevertheless, as suggested in Figure 11, the very existence of a semiotic continuum plays a positive role in allowing for deeper understanding not only of the application, but also

computation itself. In this sense, we follow Alan Kay's views of how computer education could emerge from good software design [Kay '91].

### What does the Semiotic approach actually changes in the overall picture currently in view?

To conclude, we would like to say that the Semiotic Engineering approach does not actually determine the emergence of good EUPL design. Many of the ideas proposed in our example have been implemented in existing EUP prototypes whose authors never thought of Semiotics or, if they did, never acknowledged it. The actual contribution of our work is, in our view, the potential to explain and predict, in theoretic terms, the points of breakdown from User Interface to End-User Programming Environment. Such points may mean much more than cognitive difficulties a user must overcome. In the current scenario faced by software industry, they may well stand for considerable obstacle to ensure and increase minimal software usability.

Probably, one of the most interesting questions underlying this presentation is how reusable EUPL's and EU programs can really be. Compared to general-purpose full-fledged programming languages, UIL[EUPL is clearly lacking in generality. The very notion of interpretants instantiates understanding in time and space, no matter how many times such instances repeatedly occur in real-world applications usage. What it amounts to is that an adequately reengineered UIL[EUPL for MS Word will probably not work for MS Excel or MS Access. However, this leads us to no worse destination that any of the currently available EUPL environments, since the macro language used for Excel is not the same as the one used for Word.

The advantage of a semiotic approach becomes clearer then, if we realize that its major contribution is in the formulation of the problem and of its solution, rather than in the implementation or other concrete instantiation of a given piece of software. The methodological bonuses of pursuing this line of thought, as we have been able to express in previous work, suggest that Semiotics may well furnish an alternative basis of a much needed, and so elusive, theory of software design.

## Acknowledgments

Clarisse Sieckenius de Souza is grateful to CNPq, the agency which has given her a grant to do research about Semiotic Engineering. The authors would also like to thank the Semiotic Engineering research group and graduate students at PUC-Rio for their contributions.

## 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.

[Bowen '89] Bowen, W. (1989) The puny payoff from office computers. In T. Forester (ed.) *Computers in the Human Context*. New York: Basil Blackwell. pp. 267-271.

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

[de Souza '94] de Souza, C.S.(1994) Testing Predictions of Semiotic Engineering in Human-Computer Interaction. *Anais do VII Simpósio Brasileiro de Engenharia de Software* Curitiba, October 26-28, 1994. SBC-CITS-PUC/PR-UFRGS. pp.51-62

[de Souza '96] de Souza, C.S. (1996) *The Semiotic Engineering of Concreteness and Abstractness: from User Interface Languages to End-User Programming Languages.* in Lucena, C.J.P. (Ed.) Monografias em Ciência da Computação. Departamento de Informática. PUC-Rio. Rio de Janeiro. MCC/08/96. 25 p.

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

[DiGiano & Eisenberg '95] DiGiano, C. and Eisenberg, M. (1995) Self-disclosing design tools: A gentle introduction to end-user programming. in *Proceedings of DIS '95.* Ann Arbor, Michigan. August 23-25, 1995. ACM Press.

[DiGiano '96] DiGiano, C. (1996) *A vision of highly-learnable end-user programming languages.* Child's Play '96 Position Paper.

[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.

[Guzdial, Reppy & Smith '92] Guzdial, M.; Reppy, J.; and Canfield Smith, D. Report of the "User/Programmer Distinction" Working Group. In Myers, B. (Ed.) Languages for Developing User Interfaces. Boston. Jones and Bartlett. pp. 343-366

[Kay '91] Kay, A. (1991). Computers, Networks and Education. *Scientific American*. September. pp.138-148

[Lieberman '93a] Lieberman, H. (1993) Tinker: A Programming by Demonstration System for Beginning Programmers. In Cypher, A. et al. (eds.) *Watch What I Do: Programming by Demonstration*. pp. 49-64

[Lieberman '93b] Lieberman, H. (1993) Mondrian: A Teachable Graphical Editor. In Cypher, A. et al. (eds.) *Watch What I Do: Programming by Demonstration*. pp. 341-358

[Microsoft '94] Microsoft Corporation (1994) *Microsoft Word Developer's Kit, Second Edition*. Redmond. Microsoft Press.

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

[Myers '92] Myers, 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 Myers, 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

[Nake '94] Nake, F. (1994). "Human-computer interaction: signs and signals interfacing". *Languages of Design 2*. pp. 193-205

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

[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.

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