Interacting with Computers 13 (2001) 467–495
www.elsevier.com/locate/intcom

Semiotic engineering principles for evaluating end-user programming environments

C.S. de Souza[a,*], S.D.J. Barbosa[a,b], S.R.P. da Silva[a,c]

[a]*Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro, R. Marquês de São Vicente, 225, Gávea, Rio de Janeiro, RJ, 22453-900 Brazil*
[b]*Tecgraf/Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro, R. Marquês de São Vicente, 225, Gávea, Rio de Janeiro, RJ, 22453-900 Brazil*
[c]*Departamento de Informática, Universidade Estadual de Maringá Av. Colombo, 5790, Maringá, PR, 87020-900 Brazil*

Received 1 July 1998; revised 21 July 2000; accepted 19 September 2000

Abstract

End user programming (EUP) environments are difficult to evaluate empirically. Most users do not engage in programming, and those who do are often discouraged by the complexity of programming tasks. Often the difficulties arise from the programming languages in which users are expected to express themselves. But there are other difficulties associated with designing extensions and adjustments to artifacts that have been originally designed by others. This paper characterizes EUP as a semiotic design process, and presents two principles that can be used to illustrate the distinctions between the various kinds of techniques and approaches proposed in this field. The principles support a preliminary theoretical model of EUP and should thus facilitate the definition and interpretation of empirical evaluation studies. They also define some specific semiotic qualifications that more usable and applicable EUP languages could be expected to have. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Semiotic engineering; End-user programming; Evaluation methods; Usability; Theoretical model

1. Introduction

End-user programming (EUP) has been gaining importance in software engineering because of at least two identifiable reasons. One is that the cost of building and maintaining multifunctional applications is high, and users are always demanding enhancements and extensions to even the most successful software tools. The other is the growing

* Corresponding author.
E-mail address: clarisse@inf.puc-rio.br (C.S. de Souza).

awareness that users are not always passive software consumers, but that they can play the role of software designers and producers. The former argument in favor of EUP gained popularity in the early 1990s (see Myers, 1992; Nardi, 1993), and the latter has been evolving both in the fields of HCI and Design (Cypher, 1993; Nardi, 1993; DiGiano and Eisenberg, 1995; Fischer, 1998). However, skepticism about EUP is usually encountered when the question whether end users are capable of programming applications or not, given that they do not have any training as programmers, is asked.

The answer is strongly dependent on how usable EUP environments are in the first place. In contrast with voluminous research on the usability of software applications from a non-EUP perspective, research about usable EUP environments is scarce. One of the reasons for this is obviously the fact that, with the exception of advanced spreadsheet users, only a very small percentage of software consumers have ever tried to *program* an application (be it by macro-recording, parameter configuration, or scripting). Another important reason is that EUP environments are typically plainly uninviting for non-programmers, which explains and in a way perpetuates this situation (see Nardi, 1993; Traynor and Williams, 1997).

Difficulties posed for empirically based usability evaluation in EUP start with the identification of the *typical subject* in a heterogeneous, scattered, and small community. Thus, heuristic approaches lack sufficient empirical evidence from which to build a solid body of practical knowledge. But there is also a theoretical difficulty with EUP usability studies, not entirely due to the lack of empirical data. Most of the models and metrics employed in non-EUP studies are inadequate for EUP, because they implicitly or explicitly assume that the **semantic model***[1] of applications is **stable**, that applications do not evolve. Extensible applications do evolve, and the impact of evolution affects the usability of the whole application's environment (i.e. the EUP and the non-EUP portions alike). So, we should be able to identify where stability lies in the overall model of extensible applications before we set out to evaluate it by inspection, observation, experimentation or any other method.

Different techniques have been proposed to help users extend applications. Among them, the most popular are macro-recording, programming by demonstration or by example, and scripting. In this paper, we will be focusing on the latter-scripting, which allows users to generate the widest range of extensions by providing linguistic descriptions of new functionality. It should be noticed that we draw a clear distinction between extending and programming (or re-programming) software. We will be talking only about **extensions**, which are the result of adding new functionality to an existing application without destroying any of its original functionality. We believe that the capacity to modify the existing implementation of functions requires expertise that is beyond the expected ability of end users, and so is the capacity to subtract functions from the implemented model and reestablish the right connections among surrounding (and possibly dependent) functions.

In this perspective, we propose to start by characterizing what a **usable EUP language** is. Language models are typically stable, and we can thus aim at achieving qualitative assessments of language features that may affect the task of extending software as well as

[1] The expressions marked with an asterisk are glossary entries. The glossary is presented after the bibliographical references.

its product and continued evolution. We will argue that this is a preliminary requirement for significant quantitative and qualitative usability measurements in this context, even more fundamental than user-centered methods that could be used to evaluate tailorable applications (see Malone et al., 1995 for instance).

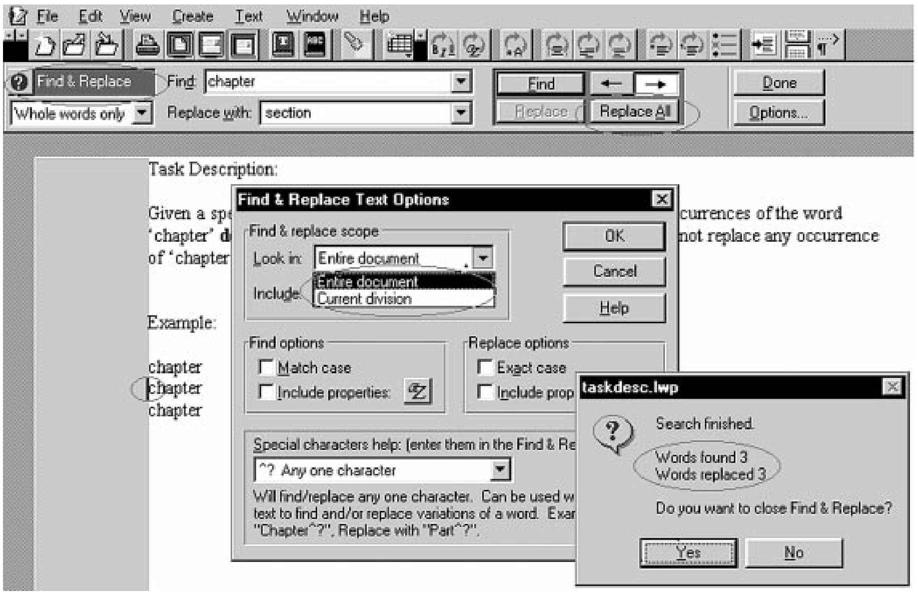
An EUP task is typically a consequence of a user's perception of a lack of needed functionality. This can only happen if the user is convinced that he fully understands the application as it is presented by the user interface language (UIL). The UIL reflects what the application is, and abstracts it for the user. The EUP or scripting task is then carried out by means of a specialized tool, namely an extension language, with which the user builds the necessary extensions to the original application. However, if the extensions are not adequately reflected at the interface, a potential interactive malfunction can jeopardize the product of his efforts. This paper takes a specific semiotic engineering approach (see de Souza, 1993) to evaluate EUP environments. It presents two principles that should hold relative to the user interface language (UIL) and the end-user programming language (EUPL), if designers want to reach for usability and applicability. The *Interpretive Abstraction Principle* aims at assessing how well the UIL abstracts the range of extended functionality in the application. And the *Semiotic Continuum Principle*, in its turn, assesses the obstacles in translating extensions into functional and usable UIL constructs.

These principles provide theoretical underpinnings that can serve as a basis for analyzing and explaining specific evaluation issues in EUP environments. We believe they have the potential not only to advance the state of the art in EUP usability studies, but also in that of computer languages in general, as seen from a semiotic perspective. Representation languages, programming languages, and interactive languages, alike, may be analyzed and evaluated in terms of signification and communication resources that they offer to the different types of users they serve.

In the next four sections, the reader will see how the semiotic dimensions of problems faced by users of some commercial applications can characterize EUP as a representation and communication design activity, and will find the definitions of our two principles. The nature of signs and sign systems (or languages) that coexist in extensible software applications, along with some of the relations that hold between them, demonstrate the effects of the Interpretive Abstraction and the Semiotic Continuum principles in concrete situations. A discussion of the challenges posed by EUP for many evaluation methods and of how the proposed principles address them will support our view that semiotic theory can effectively advance our understanding of the nature of end user programming tasks.

2. Usability problems in EUP environments

Usability problems in EUP environments involve, but are not limited to, challenges in both user interface language design and end-user programming language design. Among usability problems, some are specifically semiotic problems, representation and/or communication breakdowns that prevent users from making satisfactory sense of signs in either language. The sense-making process is based on the users' pattern recognition capacity, on language documentation, on computer literacy and cultural background, or a combination of all.



Find: chapter
Whole words only
Replace with: section

Task Description:

Given a spe… ‘chapter’ d… of ‘chapter… …currences of the word …not replace any occurrence

Example:

chapter
chapter
chapter

Find & Replace Text Options

Find & replace scope
Look in: Entire document
Include: Entire document / Current division

Find options
Match case
Include properties:

Replace options
Exact case
Include prop…

Special characters help: (enter them in the Find & Re…
^? Any one character
Will find/replace any one character. Can be used w… text to find and/or replace variations of a word. Exar… "Chapter^?", Replace with "Part^?".

taskdesc.lwp

Search finished.
Words found 3
Words replaced 3
Do you want to close Find & Replace?

Fig. 1. A sequence of screen dumps in the WordPro™ interface language.

The examples presented in this section have been collected during an expert's inspections, and characterize plausible, if not very probable, sequences of interactive and interpretive steps while using commercially available extensive applications.

Example 1. Let us consider a text-editing scenario in which a user wants to do the following.

Given a specific cursor position in the document, replace *all* occurrences of the word “Chapter”, *down towards the end of the document*, with the word “Section”.

In Lotus WordPro®, the designer-to-user message implicit in its interface suggests that the task can be easily achieved bynthe following set of events (Lotus, 1995), Sketched in the upper portion of Fig. 1.

1. Click on the desired cursor position in the document
2. Type Ctrl + F (or select Find and Replace… in the Edit menu)
3. Type in the word “Chapter” next to the label Find in the Dialog Box
4. Type in the word “Section” next to the label Replace with in the Dialog Box
5. Check that the arrowhead pointing to the right is selected
6. Click on the button named Replace All

Against expectations, these steps do not cause the occurrences of “Chapter” from the current cursor position *onwards* to be replaced by “Section”. Instead, *all* occurrences of “Chapter” are replaced throughout the whole document, either onwards (as expected) or backwards (as not). This can be inferred from the screen dumps in Fig. 1, where more than two instances of the whole (exact) word "Chapter" (without quotes) are reported to have been found. With some further explorations of the interface and online documentation, the user might find that the Replace All action overrides the directional choices signaled by the arrowhead buttons. He could then conclude that once the search mechanism has reached the bottom end of the document, it continues from the top, in a circular mode (*wrapping* in text editing jargon). Therefore, the interface could be taken to suggest functionality (esp. wrapping control) that the application does not really support.

This first example thus highlights an important problem. The UIL suggests the existence of meanings that do not belong in the semantic model of the application (e.g. the direction of search cannot be combined with the Replace All capability). Although we are not into EUP yet, we will see that this apparent inconsistency in the UIL will have major consequences in extension tasks because a fundamental *what you say is what you get* convention is broken, making it difficult for a user to build a robust semantic model of the artifact. In other words, the UIL does not consistently reflect, or abstract, the underlying application.

As pointed out by supporters of the programming by demonstration paradigm (see Cypher, 1993), the leading motivation for EUP is having to perform repetitive tasks. So, let us suppose that this scenario will be frequent enough so that some users would consider generating a macro and directly executing it from the interface. Now some relevant EUP issues associated to Example 1 will become clear.

Example 2. If the user turns on the script-recording tool while performing the aforementioned search-replace interaction (Example 1), the automatically generated script will consist of the following code in LotusScript (see Lotus, 1995):

```
Sub Main
• Application.FindAndReplace.FindString = "Chapter"
• Application.FindAndReplace.ReplaceString = "Section"
• InitFindAndReplace True
• ReplaceAll
• InitFindAndReplace True
End Sub
```

We see that, although the arrowhead button may be marked as pressed during the recording (as in Fig. 1), apparently the script does not capture this parameter (which is a default). However, if the user repeats the recording after pressing the arrowhead **backwards**, a slight difference in the EUP code will become apparent. Again **all** occurrences of "Chapter" are replaced by "Section", but with this explicit change of the default direction of the search, the recorded script will be like this:

```
Sub Main
• Application.FindAndReplace.FindString = "Chapter"
• Application.FindAndReplace.ReplaceString = "Section"
• Application.FindAndReplace.FindForwardDirection = False
• InitFindAndReplace True
• ReplaceAll
• InitFindAndReplace True
End Sub
```

The problem in this case is that the automatically generated EUPL code does not consistently reflect the natural expectations about signs in the UIL sentences produced by the user (e.g. the parameter *forward* for the search is not present in the first script, although the parameter *backward* is present in the second). Although he can probably understand both versions of EUPL code, for instance, it wouldn't be surprising if he equivocally predicted that neither version of code would entail a wrapping effect during the replacement process.

At the heart of this example is the fact that some of the signs intentionally signaled by users during the macro-recording process explicitly appear in the generated EUPL text, whereas other signs that are the semantic counterparts of these do not. Consequently, no systematic predictions can be made about the mapping between interface and end user programming language signs.

Example 3. In Microsoft Word for Windows® (Fig. 2; Microsoft, 1996), we see that the original menu configuration of this application (on the left-hand side of the picture) specifically suggests that users can save their files in HTML format. If they choose to do so, there is a follow-up dialog window (similar to the one in Fig. 3) in which they can specify the desired path and file name for the document being saved. A plausible extension of the application's functionality would be to save files in RTF format (which is interpreted by a number of other software tools). Although the phrasal sign Save as RTF… does not actually exist in that menu, it is certainly a **potential sign*** (see the right-hand side of the picture).

If MS Word® macro recorder is used to produce the code for the new function Save as RTF the generated code in Visual Basic for Applications™ (VBA) will look like this:

```
Sub Save_as_RTF( )
  ActiveDocument.SaveAs                 FileName := "test.rtf",
  FileFormat := wdFormatRTF,             LockComments := False,
  Password := "", AddToRecentFiles := True, WritePassword := "",
  ReadOnlyRecommended := False,    EmbedTrueTypeFonts := False,
  SaveNativePictureFormat := False,     SaveFormsData := False,
  SaveAsAOCELetter := False
End Sub
```

Note that the file name has been captured as a constant value ("test.rtf", in this case), and not as a variable. Aside from the fact that users may not notice this before too late, there is an important inconsistency between UIL and EUPL texts. What is suggested in the UIL dialog is that only the type of the file is being changed (but not its path or name), as can be seen in the following sequence of interactive events that were used to generate the macro above:

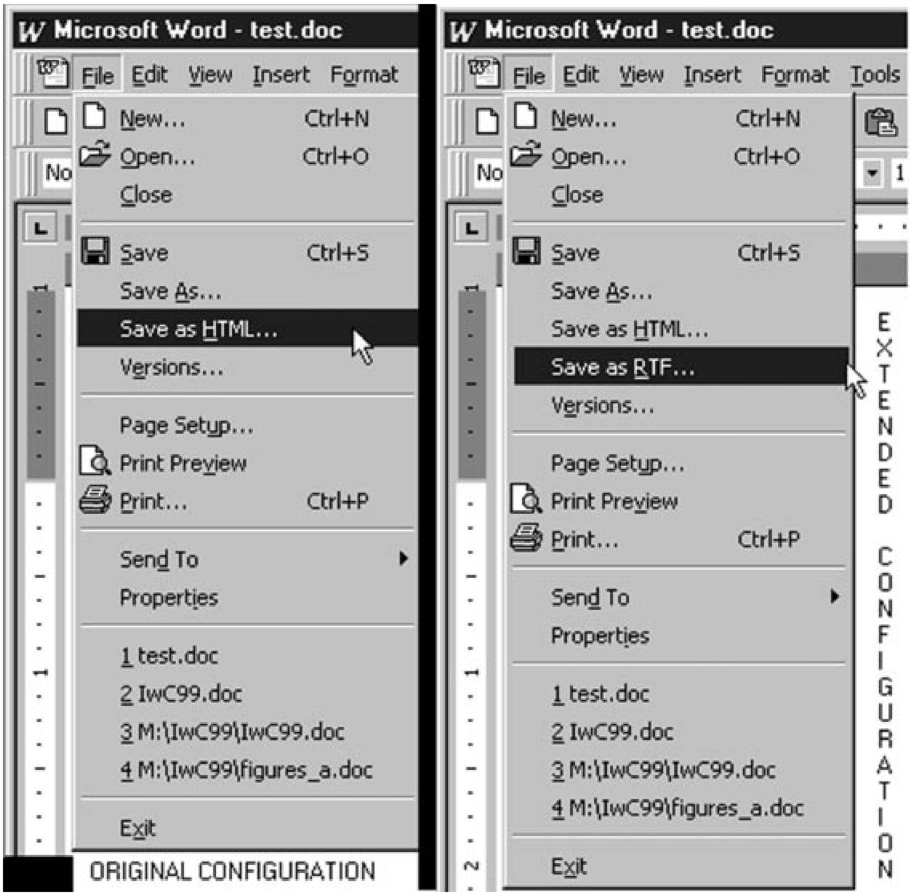
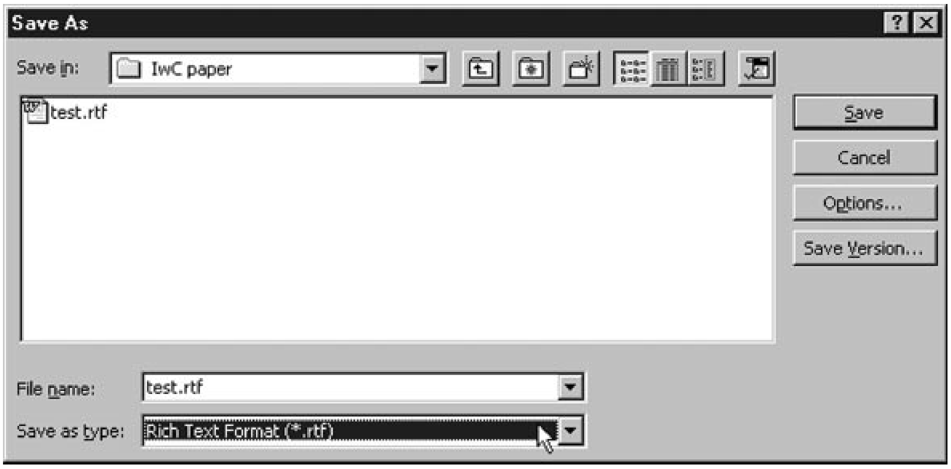


Fig. 2. Extending Microsoft Word for Windows®.

1. Click on File on the menu bar.
2. Position cursor on Save As… on the pull down menu and click.
3. In the dialog box (see Fig. 3), roll Save as type list of options till Rich Text Format (*.rtf) is found.
4. Depress mouse button and click on the Save button.

The user **did not** refer to the file name and path in the above dialog (changes to the file extension are made automatically by the system). Therefore, there is no reason to believe that the macro should capture and crystallize input that the user did not specify. We then see that the text generated by macro recording is **not** an adequate semantic depiction of the interactive discourse as produced and perceived by the user in the UIL. A more acceptable version of the macro code could at least include something like `ActiveDocument.-`



File name: test.rtf

Save as type: Rich Text Format (*.rtf)

Fig. 3. UIL dialog for the Save As RTF macro recording example.

`SaveAs FileName := "*.rtf"` instead of `ActiveDocument.SaveAs FileName := "test.rtf"`.

Example 4. Again in Microsoft Word for Windows®, if the user chooses to use directly the macro editor instead of the recorder (supposing that he knows VBA well enough), all the programming power of the language is suddenly within reach. Thus, anything that constitutes a syntactically correct program in VBA is an executable (though possibly not purposeful) macro. Such is the case of:

```
Sub EditedMacro( )
  SaveAsAOCELetter = False
End Sub
```

The `AOCELetter` attribute does not appear in any of the typical Save dialogs in Microsoft Word for Windows (it is a highly specialized parameter valid only in Macintosh computers). Thus, although the minimal code above is syntactically correct, it is pragmatically useless from an interactive perspective. Not only does a PC user ignore the potential sign that could be associated to it in the UIL he is accustomed to, but also the effect of this macro on the document is virtually null in Windows (i.e. running this macro on a PC causes no tangible effect on the document or on the application).

In order to make a useful extension, the environment must guarantee that a **minimum cycle of interaction**[*] is preserved. This minimum cycle consists of three steps: application "says" something to the user; the user "says" something to the application that triggers an action; the application "replies" to the user. If the EUPL has a syntactically distinct structure for **text**[*], or a construct of a higher order than that equivalent to an executable instruction, block or program, then we can always try to associate a pragmatically valid interpretation to text constructs. If an extension does not preserve the minimum cycle of interaction, then the user will be prone to generating executable code that is disconnected from the interface module, one that is non-interactive by nature.

Example 5. This last example taken from Microsoft Word for Windows® refers to the possibility of having users design customized forms to support interactive editing. In these, a set of predefined widgets and/or custom OCX components can be used to achieve the desired interactive effects. This is indeed an advanced case of EUP, but it highlights some of the points we are making about the boundaries between EUP and extending applications, on the one side, and re-programming and upgrading them, on the other.

OCX components are created outside Microsoft Word and VBA by a full-fledged programming language. OCX libraries may be made available to EUP environments, thus allowing users to include one or more of these components in a form. Let us suppose that for some repetitive editing task, a user decides to particularize the general search function of MS Word, so that it now searches for an indicated style of text and returns the pages where the style occurs. The pages are shown iconically in a specific window, and the user may choose to print and/or preview one, some, or all of them. The idea is to support the task of verifying that the selected style is nicely printed throughout the whole hardcopy of the document.

Let us suppose that from an OCX library the user can select a component like the one shown in Fig. 4. It is a simplistic adaptation of the page sorter view found in many slide presentation editors. The user can open and save files, search for a specific style, and choose to print or preview the selected pages for verification of hardcopy quality. The canvas on which the pages are shown functions exactly like the page sorter view in slide presentation editors. It allows users not only to select a number of pages, but also to change the order of pages (by direct manipulation of graphic objects), duplicate, delete and edit selected pages. Likewise, the search function can be specified not only to searching styles of text, but also more generally to searching strings of characters and regular expressions.

What we see in this example is that the component inherits from its source (slide presentation editors) some functions that simply do not make sense in a typical text editing environment. For instance, the direct manipulation of pages (that result in changing their order in the document) is not a desirable feature for the situation we now exemplify. So, on the one hand, using this component brings up the benefits of a direct visual access to the pages where the selected style appears, with the additional information about its frequency (how many pages contain the style) and the distribution (where in the text can the style be found). On the other hand, however, it gives way to interactions that may be totally undesirable in the scope of the task the user is trying to achieve, like changing the linear order of the document pages or deleting them altogether.

The effect of introducing this kind of component that is alien to the original application's design can be clearly disruptive and risky in document editing tasks.

As we have argued earlier, an EUP environment should not be confused with a reprogramming environment. In the former, users should not be able to create new types in the UIL. Creating new types would mean to design different interface language structures, whose consistency with the original model is something typical end users are not prepared

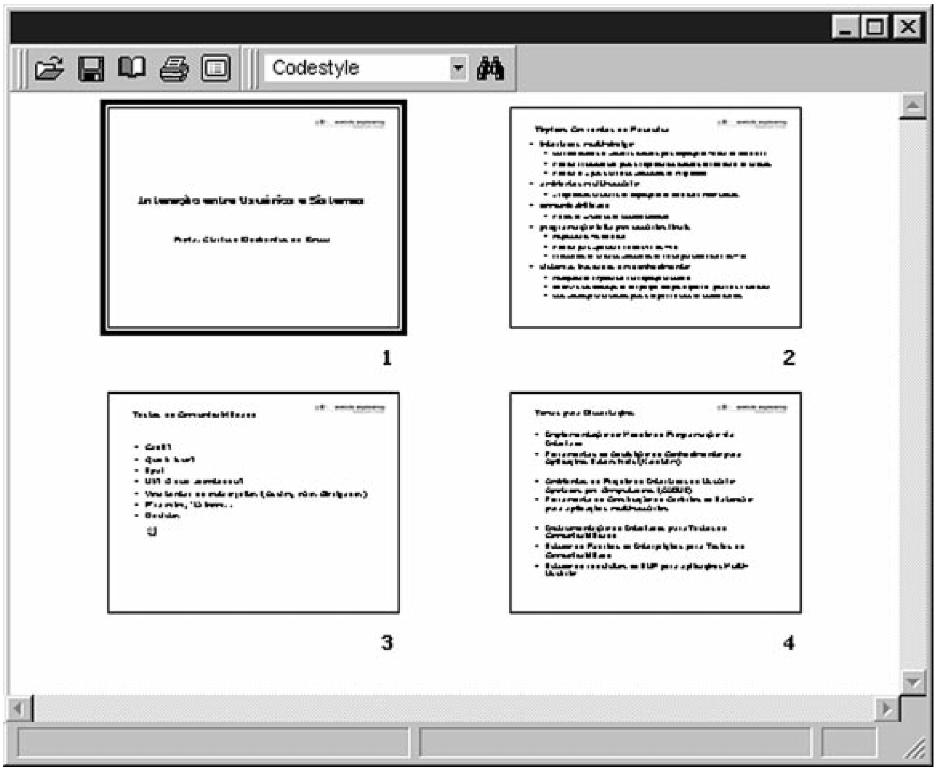


Fig. 4. Use of an OCX component that does not follow the application's interaction patterns.

to be able to verify and maintain. Only trained programmers and interactive software designers can be expected to deal appropriately with the consequences of introducing new functionality and new interactive styles at the same time, so as to avoid major disruptions from existing patterns embedded in the application.

Examples 1–5 illustrate some situations that discourage the vast majority of potential EUP users and perpetuate the separation between users and programmers. At closer examination, commercial scripting and macro languages of the sort we have illustrated are in fact genuine programming languages, supplemented with domain jargon and trimmed short of more elaborate data and control structures. For an end user, they are just as difficult as any programming language (see Cordy, 1992; Nardi, 1993).

The debate about EUP can be summarized by the titles of two papers that appeared in the early 1990s: *The user interface is the language* by Dertouzos (1992) and *Why the interface is not the programming language, and how it can be* by Cordy (1992). Both Dertouzos and Cordy present strong arguments for more powerful interface languages that allow users to extend applications. More intelligent interfaces was the line adopted by many followers of the programming by example (or by demonstration) paradigm (see, e.g. Cypher, 1993; Cypher and Smith, 1995; CACM, 2000). An alternative path was followed by those who tried to introduce users *to the art of programming* in a gentle way (Eisenberg, 1995; DiGiano and Eisenberg, 1995; Malone et al., 1995). Nevertheless, these alternative proposals have not yet entered the market as LotusScript® or Visual Basic for Applications®. AgentSheets®, a more recent product that merges large portions of alternative EUP techniques, is a combination of end-user programmable agents, spreadsheets and Java authoring tools (AgentSheets, 2000), aimed at empowering users to work in the Internet. Among other resources, it offers users a visual programming language specifying small programs.

It is noteworthy that agent technology, derived from artificial intelligence techniques, is compensating for the difficulties of EUP in a number of customization tasks (see CACM, 1994; IUI, 2000). Agents take the initiative of reconfiguring applications based on the user's goals, which can be stated directly by means of interface signs.

Another technique designed to facilitate EUP is visual programming, which in general terms tries to equate the task of building new functionality to that of arranging new configurations of visual elements within a constrained space (see Chang, 1990). This is obtained by means of certain isomorphisms between the user language (programming or interface language) and its executable semantic description language. The effect of visual programming is very close to that of direct manipulation (Shneiderman, 1983), an appealing interactive technique, which conceals the complexities of program specification from users.

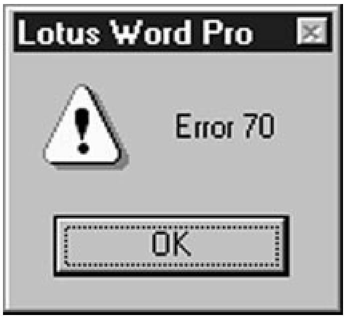
Many programming by demonstration environments actually blend agent and visual programming techniques, evading or at least reducing the usability problems of text-based EUP, presented to users and designers alike.

3. Semiotic considerations

Semiotics* can provide a wealth of insights for the analysis of interactive environments and usability problems that may occur with them (de Souza et al., 2001). The discipline is devoted to investigating the nature of **signs*** and **communication*** (Eco, 1976; Sebeok, 1994), and has been felt by many to bring important contributions to software designers in particular (see Nadin, 1988; Andersen et al., 1993; de Souza, 1993; Mullet and Sano, 1995; Jorna and van Heusden, 1996) and computer scientists in general (see Meystel, 1996; Nadin, 1997). There are numerous (and considerably different) perspectives in this field, but the ideas of Charles Sanders Peirce, who lived most of his life in the 19th century, and had his work published in eight volumes in the first half of the 20th century (Peirce, 1931–1958), have been undoubtfully a common reference for most theorists to-date.

The fundamental concept of Peircean semiotics is the **sign***. It differs from the concept of **symbol***, in formal linguistics, because of the definitional requirement that a sign must be *interpreted by a subsequent thought or action* (*apud* Hoopes, 1991). The subsequent thought or action generated by a sign is called its **interpretant***. An important feature of interpretants is that they can, themselves, trigger further interpretants, generating a chain of associations whose links and boundaries cannot be predicted.

Example 6. In order to illustrate this notion and its reach, let us consider running a sample macro recorded in Lotus WordPro®, which is supposed to save the current


Lotus Word Pro

Error 70

Fig. 5. Error message received when there is no floppy in the drive.

document in a floppy disk. If there is no diskette in the floppy drive when the macro is executed, the user gets the error message shown in Fig. 5.

When faced with this message box, a user will most probably generate a chain of interpretations that are associated to each other by causal relations. In this chain there may be signs related to such meanings (or thoughts) as: *something went wrong*, *the script has a bug*, *the file is corrupted*, *the application will crash*, *the diskette is not in the drive*, or *computers never work as we want*. There may also be actions like: ⟨check the diskette⟩, ⟨click the button⟩, ⟨reset the computer⟩, ⟨ask for help⟩.

A significant contribution of semiotic theory to HCI, as illustrated by this case, is to call the attention to the fact that the interpretation of a sign is an associative process, whose depth and length cannot be fully predicted, and not a fixed ideal mental state or configuration. This process has been called **unlimited semiosis***, and has motivated Peirce's theories about pragmatics and the situated nature of the meaning. However, it has also raised the fundamental challenge for all accounts of successful communication: if sign interpretation is unpredictable, how can any two people effectively communicate with each other?

Peirce's answer to this challenge stems from the notion of **abductive reasoning*** (or **abduction***, for short). Abduction is characterized as the process of formulating explanatory hypotheses. Unlike deductive and inductive reasoning processes, abduction involves the introduction of new (i.e. extra-theoretical) information. In abductive reasoning, sign interpretation is equivalent to **sense making***. As soon as a plausible hypothesis is formulated to explain the presence of a sign, in a given situation, it is taken to be at the heart of this sign's meaning[2]. This type of reasoning elegantly explains not only successful communicative exchanges among people, but also the unsuccessful ones. The latter are the result of apparently true faulty hypotheses (or misconceptions), and they may be corrected later through remedial dialog or counterfactual information. It also explains

[2] It follows from this definition that the meaning is always provisional, since, as the situation changes, initial hypotheses may prove to be inadequate. Better ones can then be sought and found, providing new meanings to the same sign.

why we can handle conversation without really bothering to understand the precise meaning of each and every sign we are exposed to.

In Example 6, the user may not really bother to understand what *error* 70 means. He can manage perfectly well with the hypothesis that this is *an internal error code* and generate an interpretant that will eventually include the action of checking the floppy disk drive and concluding that a disk should be inserted. Note that this is neither logic deduction, nor logic induction, but merely a working hypothesis that proves to be satisfactory for the situation at hand. As a consequence, even experienced software users may have a considerably distorted notion of what an application is originally meant to be and of how it actually works. From a user's point of view, the meaning of interface signs is something that systematically generates behavior that successfully leads him to achieve expected goals.

We see that a semiotic notion about the nature of meaning is very different from that proposed by semantic theories that adopt a representational stand (see Eco et al., 1988). It is clearly slanted towards **pragmatics***and the effects of language use on language meaning (see Wittgenstein, 1953; Searle, 1979). Although computation is fully representational when viewed from the perspective of symbol-processing machines, what matters for typical users is what they can actually do with computers, and not how the processing of symbols affects the state of affairs. Many accounts of users' interpretations of interface signs implicitly or explicitly assume a representational stand. Some even propose methods to assess the distance between a targeted meaning and that which the user conceives as he engages in purposeful interaction (Hutchins et al., 1986).

The challenge of HCI, and more critically of EUP, is to reconcile machine meaning and user meaning. In other words, to bring together a machine whose functioning depends on stable representations and a fixed set of transformations on them, and a human being who constantly generates representations that evolve and adapt, according to rules that are often learned in the very process of evolution. In an attempt at this reconciliation, our own work in Semiotic Engineering (de Souza, 1993) focuses on two central ideas:

1. That interactive systems are high-level **messages*** sent from designers to users, about how to send and receive other messages in the system in the process of achieving a certain range of goals.
2. That these designer-to-users messages are **metacommunication artifacts***, performing messages that concretely send and receive a variety of messages to and from the users.

According to this approach, successful user interaction is determined by the users' ability to **make sense** of the messages sent by designers. No representational stance of the meaning assigned to the designers' message is necessarily implied or supposed to exist in the users' minds. The designers' messages are coded in one or more computer languages, according to specific syntactic, semantic, and pragmatic rules. Inasmuch as the user is able to send messages to the application within the communicative setting proposed by designers, we can say that the user gains proficiency in an interactive language, which is equivalent to a process of language acquisition. In addition to the application's documentation, and often in spite of it, the only resources users have at their disposal during this language acquisition process are the user interface tokens and **interactive patterns***.

Although different UILs may share similar interactive patterns, an application's UIL is a **unique language of interaction**. The combination of widgets and labels, the effects of menu selections, the icons and manipulations that a user discovers through the application's interface can never be used to interact with a different application. What can be, and actually is, transferred from one application to another are styles and patterns, which contribute to the expression of software *look-and-feel* or *nativeness* in different platforms, and to the acquisition of *computer literacy* in general. Thus, what is commonly called "interaction languages" (see, for instance, Rheinfrank and Eveson, 1996) are not formally *languages*, but rather a style, or a set of general *patterns* evidenced by recurring tokens* within a range of applications.

Fig. 5 gives us a nutshell example of what we mean by the uniqueness of the interface language. The UIL *text* we see in it involves signs that occur in other Windows® applications (such as the OK button and the message window) or in the English language (such as the expression "error 70" or the exclamation mark). But the compound sign depicted in Fig. 5 only occurs in Lotus WordPro, and is meaningful only within the context of that application. That is to say that the only language to which it belongs is WordPro's UIL. Not even in applications of the same package, such as Lotus 1-2-3®, would it occur, for at least the words in the title bar of the message window would not be the same. However, this pattern of interaction can be found across the whole spectrum of commercially successful MS Windows® applications. In other words, the UIL is strictly unique to the application to which it belongs in spite of all the similarities it holds with UIL's of applications by the same manufacturer and/or applications designed for the same platform with the same toolkit.

After having built a satisfactory semantic model of a unique UIL, users of extensible applications may then proceed to EUP. At this point, not only must they speak the application's language, but also be able to predict and codify new grammatical constructs in it. They must provide precise semantic definitions for these constructs in the programming (or extension) language, since the symbol-processing machine they are extending requires a formal semantic model.

Thus, we narrow our focus on end-user programming languages (EUPL). Just as the UIL of an application is a unique language of interaction, so should the EUPL be a **unique language of extension** (and not only a special-purpose language, as claimed by Nardi, 1993). A special purpose language may be used to represent a problem or class of problems within a specific domain, or across various domains. As such, it is an acceptable programming language. However, because we view EUP as the task of (monotonically) extending existing applications, an EUPL must serve users exclusively within the *single* application it is designed to extend. Thus, it must not include constructs or signs that do not belong to that particular application (see Example 5, in Section 2).

The unique aspect of the extension language is inherited from the uniqueness of the interaction language of the application. Object-oriented approaches are striving to customize the classes made available for extensions by including in the EUP environment only those that are native to the application being extended (e.g. Microsoft® Office 97). This is a major step forward in the direction of more usable EUP facilities, since users would then need to deal only with their linguistic intuitions about the full grammar of **one** isolated UIL-EUPL pair at a time.

Having examined some defining characteristics of both UIL and EUPL, we may now turn to the linguistic extension process itself. A language can be extended in three ways: (1) by the creation of new tokens of existing types; (2) by the creation of new types that will replace existing types in existing patterned structures; and (3) by the creation of new types and structures altogether. In theoretical terms, the third kind of extension is equivalent to modeling the whole discipline of computer programming with Turing machines. Therefore, we should keep this kind of extension to the realm of professionals and use only the first two in EUP tasks.

The effect of creating new **interactive tokens*** of existing **interactive types***, from the UIL perspective, is the same as that of adding a word to the UIL vocabulary. For example, when a user records a macro, and then creates a menu item or toolbar button to trigger it, he is creating a new token of the "menu item" or "toolbar button" types. If an EUP environment limits users to this kind of extension, it is merely abstracting sequences of interactive steps with a new name (or widget token). In spite of its simplicity, this is a most useful resource for automating repetitive tasks.

The effect of creating new types in the second kind of extension is the same as creating analogies from a given model, and it allows for much more sophisticated EUP. If, e.g. in a text editor for instance, we create the notion of "sentence" at the UIL, and apply to it the same formatting operations as those applied to "character", we are introducing a new type ("sentence") by analogy with an existing one ("character"). This is more complex than the former case, because it involves a semantic description of "sentence", which cannot be directly expressed in the UIL.

The combination of these two mechanisms empowers users to extend the application within the boundaries of its domain and the grammatical patterns of its UIL. It is important to notice that, even though users make extensions using an application's EUPL, their goal should be to extend its UIL, otherwise they would be producing extensions that have no effect on the UIL (see Example 4, in Section 2). In typical extensible applications, the UIL is made up of a fixed part and an extensible part (UILx). After one or more extensions are made, the extended UIL includes a fixed part, an extensible part, and the newly created extensions (Fig. 6).

In our Semiotic Engineering perspective, interacting with applications and extending them are tasks that can be paralleled to a notion proposed by Searle in his pragmatic theory of speech acts, the direction of fit. (Searle, 1979). The theory accounts for the fact that speech can not only describe things and refer to them, but also achieve things and change

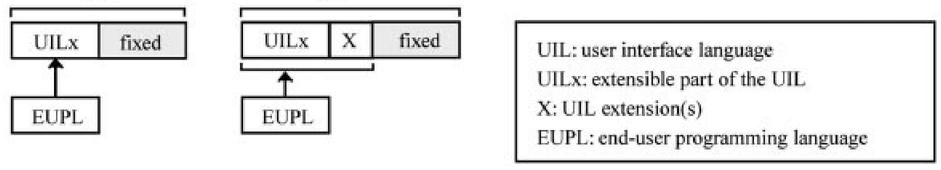

UIL
UILx
fixed
EUPL
UIL
UILx
X
fixed
EUPL
UIL: user interface language
UILx: extensible part of the UIL
X: UIL extension(s)
EUPL: end-user programming language

Fig. 6. Changes in the UIL as a result of an extension.

the state of the world. The former is a case of world-to-word direction of fit, whereas the latter is a case of word-to-world direction of fit. By analogy, while interacting with applications through the UIL, users are abiding to the semantic descriptions encoded in the system, and thus observing a world-to-word direction of fit. When extending applications, users are creating new descriptions that will become effective in the UIL, and are thus observing a word-to-world direction of fit.

4. Interpretive abstraction and semiotic continuum

The two principles we are about to introduce are applicable to computer languages in particular, and artificial languages in general. They don't make any sense if applied to human languages because of the meta-linguistic mechanisms they involve. They are meant to help us say why, when and how two computer languages can be used to approximate the effect we would achieve if systems could be programmed and extended through a natural dialog with the machine. The gist of these principles is that: (a) texts, interactive or descriptive, in computer languages should be taken as abstractions of the computing machinery embedded in applications; and (b) descriptive texts specifying extensions to interactive software should incorporate pragmatic structures that function as markers of interactive adequacy of the descriptions they contain.

4.1. The interpretive abstraction principle

In Example 6 above, we saw that users cannot be expected to grasp the application's semantic model meaning of such UIL signs as “error 70”. The operational semantic definition of this phrase lies in computer codes that are far below the surface of the application's context. In Example 2, Lotus WordPro users anticipate meanings from the UIL dialog signs that are not consistently implemented in the underlying languages (EUPL or other). This contradictions leads to breakdowns of interpretation about how directional searches work and what are the real conditions for wrapping text during a search.

In order to detect the kinds of breakdowns illustrated in these examples, we postulate the *Interpretive Abstraction Principle*. To say that a UIL is an *Interpretive Abstraction* of one or more languages is to say that the meaning of UIL terms and phrases are specified in those languages, and that a user must be able to understand and interact with the UIL without having any knowledge about the underlying languages. In other words, the UIL is an interpretive abstraction of lower-level computer languages (which are necessary to determine the effective semantics of interaction) if users can fully understand all UIL signs by virtue of:

(i) the patterns of signs and sign combinations they encounter while interacting with the application;
(ii) the available explanations about the UIL (as provided by online help and tutorials, screen tips, and documentation in general); and
(iii) their own experience with computers (bringing forth the associated knowledge tapped by interface metaphors, application domain knowledge, and sheer common sense).

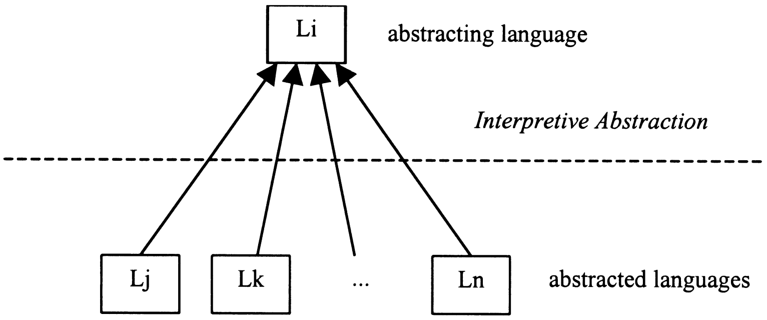


Fig. 7. The interpretive abstraction principle.

In order to understand the interface language, therefore, the user should not need to know anything about the *abstracted* languages. All sign constructs in the *abstracting* language (i.e. the UIL) should make satisfactory sense to the user by virtue of the inferences that can be made from the summative knowledge base that integrates (i), (ii), and (iii), above (Fig. 7). This principle should hold for usable interfaces in general, and not only to EUP environments. In EUP environments, as we will see in the next section, this principle should also hold for the EUPL, and should apply the UIL in a very specific way.

A more formal definition of this principle states that, given any computer languages Li, Lj,…,Ln, Li **is an interpretive abstraction of** languages Lj,…,Ln if:

- the semantics of Li can be described by the union of all sentences in Lj,…,Ln; and
- a user of Li can make sense of **all** signs (lexical* or phrasal*) in Li by resorting to at most three fundamental and possibly intersecting sources:
  (i) the intrinsic patterns of occurrence of such signs in situated Li discourse (i.e. interactive experience or language use);
  (ii) some extrinsic meta-linguistic knowledge about Li (i.e. documentation);
  (iii) the user's own cultural background (i.e. general knowledge and computer literacy).

The *Interpretive Abstraction Principle* is a qualified form of the *Abstraction Principle* known to computer theorists. The latter emphasizes the mechanisms by which details are hidden from users, allowing them to focus on relevant aspects of the discourse domain. The *Abstraction Principle* is primarily employed in programming language design to support the **generation** of clear and concise code, although it evidently increases code legibility in debugging and maintenance tasks. But the *Interpretive Abstraction Principle* is targeted primarily at the **interpretation** of situated discourse produced by computer systems (as they show users some application's output) and by users themselves (as they input commands or data to the system). The **generative** aspects of the *Interpretive Abstraction Principle* are cast in the guise of how users interpret what they are "telling" the system, or namely what they *mean* by what they are saying.

The qualifying term *interpretive* is added to the familiar principle for sake of situating our intent and clarifying specific aspects. In an EUP environment, computer languages all contribute to design tasks. Design tasks are driven by intentions. And all intentions are determined not only by the user's experience with the medium and the domain of activity (items (ii) and (iii) above), but also by the expressiveness of the language in which this experience can be stated (items (i) and (ii) above). So, in fact, it is not only a matter of abstracting the semantic layers beneath syntactic constructs, but also, and maybe more importantly, of abstracting the *purposes* and *effects* that can be achieved as soon as signs are put together. This is the major distinction in our formulation of an abstraction principle when compared to abstraction principles that are generally defined in Computer Science in general.

4.2. The semiotic continuum principle

Example 4 presented a piece of code that could not be translated into any combination of UIL signs, and was, thus, unusable. This clearly illustrates the problems we encounter when the EUPL texts do not distinguish pragmatically inadequate sign combinations from **pragmatically adequate** ones. The syntactically correct macro did not achieve any perceptible effect in the UIL. Thus, it specified something that was not an extension of the application, nor even a perceptible modification (or customization) of an existing UIL sign. A tighter pragmatic coupling between EUPL and UIL is precisely the feature that guarantees the correspondence between well-formed EUPL and UIL texts.

Two languages are semiotically continuous if the pragmatic coupling when translating one language into the other is always preserved. In the case of extensible applications, the extensible part of the UIL (UILx) is semiotically continuous to the EUPL if:

- users are able to make sense of UILx signs and interact with the UILx without any knowledge of the EUPL or of its existence;
- users are able to make sense of the EUPL signs and generate EUPL texts without any knowledge of other lower-level languages, such as programming languages, operating system's API, and the like;
- there is a construct for a pragmatically valid *text* in the EUPL;
- any user who knows the UILx and the EUPL can always translate an arbitrary instance of EUPL text into an actual or potential valid combination of UILx signs.

The *Semiotic Continuum Principle* formally states that, given two computer languages Li and Lj, they are said to be semiotically continuous if (Fig. 8):

(i) Li is an Interpretive Abstraction of Lj (i.e. Lj provides semantic descriptions of Li);
(ii) Lj is itself an Interpretive Abstraction of one or more languages (i.e. the semantics of Lj is defined in other languages);
(iii) Lj can generate instances of **text**, a specific structured syntactic organization of sentences whose meaning incorporates intentional elements (i.e. there is a syntactic marker for pragmatic adequacy of Lj texts in terms of Li);
(iv) any user who knows Li and Lj can **always** translate an arbitrary instance of

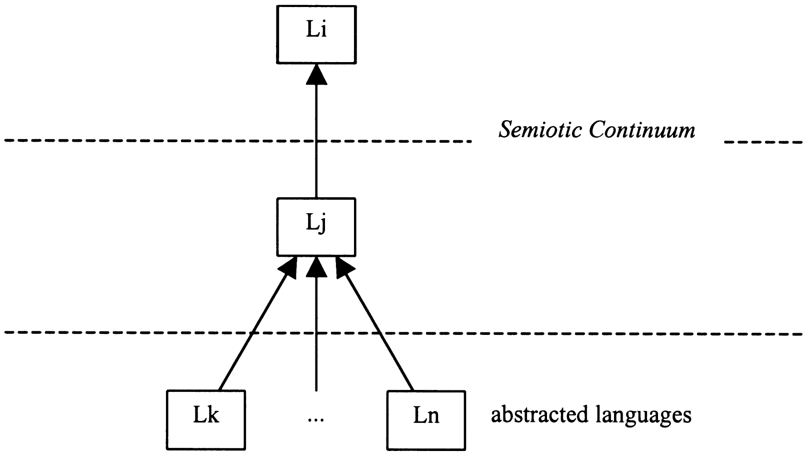


Fig. 8. The semiotic continuum principle.

Lj **text** into actual or potential valid combinations of Li signs (i.e. there are no syntactically correct texts in Lj that cannot be formulated in Li).

Notice the distinction between actual and potential sign. An **actual sign** in a language is an *existing* word or phrase of that language. When we use the Lj for describing actual signs in Li, there is a world-to-word direction of fit (Searle, 1979). On the other hand, a **potential sign** in a language is a *non-existing* word or phrase in that language, which can nevertheless be generated by lexical and/or grammatical extensions that abide to its derivational patterns (i.e. to morphological and grammatical meta-rules). When we use the Lj for generating potential signs in Li, we are following a word-to-world direction of fit.

Example 3, regarding the “Save as RTF” macro, provided an interesting illustration of what happens when this principle is not met. The user recorded a saving operation in the UIL, but the generated EUPL code did not mean the same as the UIL interaction sequence. The constant values, hidden or apparent, captured during the recorded dialog do not properly describe the semantics of what the user said in the UIL. According to this principle, it would have been better (though still not enough) if the EUPL had generated a piece of code like the following:

```
Sub Save_as_RTF( )
  ActiveDocument.SaveAs FileFormat := wdFormatRTF
End Sub
```

This text indicates what took place during interaction: the user only changed the file type, and all other values were taken from the current context of use. For the EUPL code to abide fully to the *Semiotic Continuum Principle*, we must add syntactic structures that will account for the pragmatic coupling between UIL and EUPL, namely the structures that reflect the minimum cycle of interaction. The code must then explicitly state:

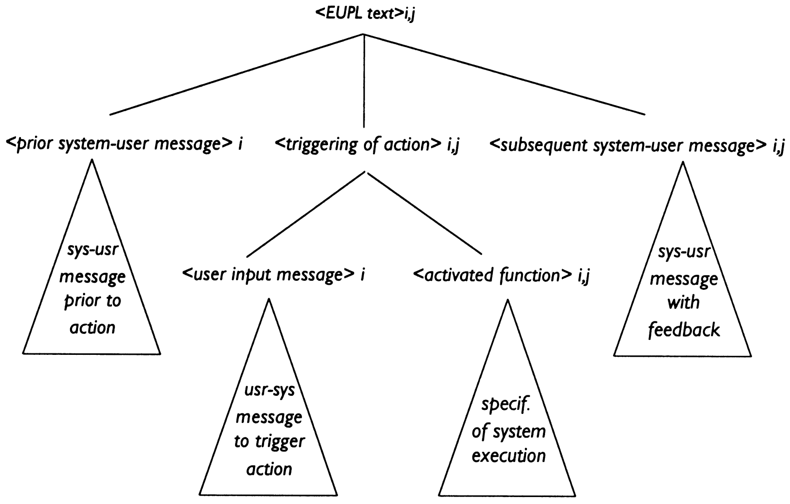

<EUPL text>i,j
<prior system-user message> i
<triggering of action> i,j
<subsequent system-user message> i,j
sys-usr message prior to action
<user input message> i
<activated function> i,j
sys-usr message with feedback
usr-sys message to trigger action
specif. of system execution

Fig. 9. An example of the text structure required by the semiotic continuum principle.

- what the system is telling the user prior to the triggering of the new function (in Example 3, the system should be telling the user, through one of the items in the *File* menu, that he can save files in RTF format);
- what the user can tell the system (keeping with the same example, the user can tell the system to save the current file in RTF format by clicking on the corresponding menu item);
- what the system understands by what it is told (in this case, the macro code line ActiveDocument.SaveAs FileFormat := wdFormatRTF is an appropriate representation);
- what the system responds to the user (if the action is successful, it can provide the standard feedback used in the application to signal the saving of files; in case of failure, the system should respond with an error message the user should include in the **text** structure of this extension's specification).

The general syntactic pattern of a **text** structure in a semiotically continuous EUPL for Example 3 is shown in Fig. 9. We omit the grammatical rules and syntactic substructures that should appear in a real extension language for sake of brevity and the benefit of relevance. An extensive specification of a semiotically continuous EUPL schema is being prepared by da Silva (in preparation). The important elements depicted in Fig. 9 are the three major components of the **text** structure that correspond to the minimum cycle of interaction: the ⟨prior system-user message⟩, the ⟨triggering of action⟩, and the ⟨subsequent system-user message⟩. Contextual parameters shared by these components (represented by the indices adjunct to the component's name) provide for discourse consistency and coherence in this piece of minimal computer–human interaction which we can characterize by the following syntactic rules in indexed BNF style:

$\langle$`EUPL text`$\rangle_{\mathtt{i,j}}$`:`$:=$$\langle$`prior system-user message`$\rangle_{\mathtt{i}}$`,`$\langle$`triggering of action`$\rangle_{\mathtt{i,j}}$`,`$\langle$`subsequent system-user message`$\rangle_{\mathtt{i,j}}$
$\langle$`triggering of action`$\rangle_{\mathtt{i,j}}$`:`$:=$$\langle$`user input message`$\rangle_{\mathtt{i}}$`,`$\langle$`activated function`$\rangle_{\mathtt{i,j}}$

Unlike the *Interpretive Abstraction Principle*, which focuses on the upper-level language, this second principle focuses on the lower-level language. The most important point in the above definition is that if somebody knows both languages, he can **always** translate any arbitrary **text** of the lower-level language into an actual or potential sign of the upper-level one. This can only be achieved because of a pragmatic component that is projected onto syntactic structures, namely the ⟨text⟩ constituent. It can separate signs that express intentions from those that do not, and thus sort out lower-level language texts that should have a translation in the upper-level language from those that should not.

4.3. Evaluating EUP environments using the semiotic engineering principles

Based on the two principles described above, we may summarize the process of evaluating an EUP environment with the diagrammatic representation in Fig. 10.

5. Discussion

Software evaluation methods and techniques can be didactically classified into five groups (see Preece et al., 1994): user observation; questionnaires; experiments; predictive evaluation; and interpretive evaluation. Depending on whether designers are engineering towards a target, comparing design alternatives, verifying conformity to standards, or trying to understand reality, the value of a method or technique will increase or decrease. Given that EUP is itself a design activity whose product must be usable for the user-designer (who is also, typically, the only user of added functionality), important distinctions must be made about this classification.

Starting with the aims of evaluation, engineering towards a target requires a quantitative metrics with which to measure the product's performance. The product of EUP environment design is a tool that can generate a set of extensions to an existing **semiotic system**[*]. Thus, the quantitative metrics required to perform this kind of evaluation should assess this tool (in our case the extension language) with which users express and give effect to new designs. The immediate hurdle on the way to selecting such metrics is that there is no way to isolate and delimit the user's design intent and verify if the tool meets its end. Even if users state, in fluent natural language discourse, what their goal is when trying to extend an application, there is not a way to capture the exact semantics of what is said. Moreover, quantitative metrics to be applied to the tool itself, the scripting language, sound vacuous. Indications about the number of rules or lexical items, about the depth of syntactic structures, about the minimum or maximum length of sentences, all mean close to nothing if taken in isolation. Although we might say that high scores are potential indications of heavier cognitive loads, this would not matter much if rules, lexical items, sentence

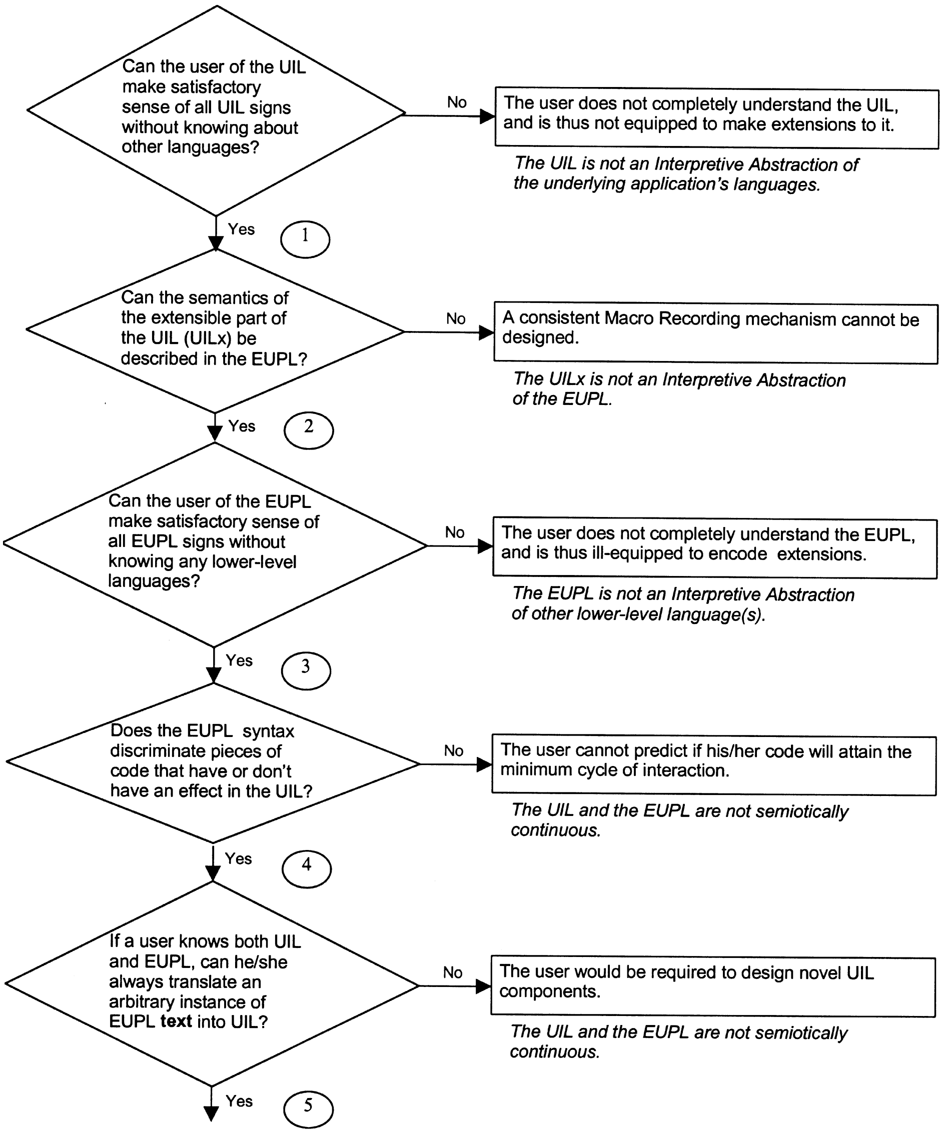

Can the user of the UIL make satisfactory sense of all UIL signs without knowing about other languages?
No
The user does not completely understand the UIL, and is thus not equipped to make extensions to it.
The UIL is not an Interpretive Abstraction of the underlying application's languages.
Yes
1
Can the semantics of the extensible part of the UIL (UILx) be described in the EUPL?
No
A consistent Macro Recording mechanism cannot be designed.
The UILx is not an Interpretive Abstraction of the EUPL.
Yes
2
Can the user of the EUPL make satisfactory sense of all EUPL signs without knowing any lower-level languages?
No
The user does not completely understand the EUPL, and is thus ill-equipped to encode extensions.
The EUPL is not an Interpretive Abstraction of other lower-level language(s).
Yes
3
Does the EUPL syntax discriminate pieces of code that have or don't have an effect in the UIL?
No
The user cannot predict if his/her code will attain the minimum cycle of interaction.
The UIL and the EUPL are not semiotically continuous.
Yes
4
If a user knows both UIL and EUPL, can he/she always translate an arbitrary instance of EUPL text into UIL?
No
The user would be required to design novel UIL components.
The UIL and the EUPL are not semiotically continuous.
Yes
5

Fig. 10. Evaluation steps of an extensible application's languages based on the proposed semiotic principles.

structures, and the like are nearly the same as those found in the user's native language grammar.

The meaning of a new feature is in the mind of the end user, who extends the application under *pragmatic conditions* often unanticipated by the original application's designer. Consequently, such metrics as semantic and articulatory distances proposed by Hutchins et al. (1986) only make sense relative to the *product* of EUP, but not to the *process*. Other metrics like error rates and duration of the task also require that the goal and context of the design task be controlled, otherwise the identification of dependent and independent variables may become impossible.

A comparison between concurrent EUP design alternatives would require an underlying model of EUP, against which the variations in performance could be explained and justified. But, since EUP itself is little understood, models are still incipient and implicit in various research initiatives that strive to develop EUP applications in which experiments can be run. With the result of experiments, more will be understood, and models will certainly emerge. In this context, the conformity to standards is definitely a premature aim for EUP evaluation.

But if evaluation is carried out to understand reality, then EUP environments not only can but actually should be extensively evaluated. Although there is a considerable amount of research about EUP techniques and approaches, a characterization of what EUP is and what cognitive tasks the users perform while they are making changes to an original application has not reached the front stage in HCI.

For instance, most of the methods and techniques for evaluating non-EUP applications are inadequate for EUP because they assume (implicitly or explicitly) that the application's model is stable and that the application's designer is the one in charge of creating and communicating this model. They fail to show that the original application's designer must also design the meta-language for his design principles in such a way that a user can understand it and use it while performing extension tasks. Myers' *Natural Programming Project* (Myers, 2000) is a key initiative towards evaluating the usability of programming languages. His findings will certainly shed light on fundamental aspects of computer language use, from which EUP research will benefit greatly.

Preece and her colleagues propose that observing users, applying questionnaires and interviewing, as well as performing interpretive analysis, are alternative ways to evaluate product usability. One of the characteristics of observing phenomena is that inevitably the observer is impregnated with what he wants to observe. Nevertheless, logging software usage, using video and audio tapes, registering verbal protocols, all of these techniques collect data that can be revisited when more solid theoretical tenets are known or proposed. Asking for the user's opinion about selected aspects of EUP or about EUP in general produces a wealth of insights about reality, and can be used to identify the dimensions of end-user design that extensible application designers must, themselves, have in mind. Interpretive evaluation can seemingly help designers grasp the subjective and creative nature of EUP that deals with the unpredictable and the undefined, for which no engineering method in a strict sense can be applied.

The semiotic engineering principles proposed in this paper are one step in the direction of modeling EUP. Although they do not characterize the task in itself, they provide the fundamental elements for analyzing instances of empirically recognizable EUP tasks and support predictions and explanations that can be tested for theoretical consistency.

These principles can characterize sources of problems and inform expected solutions. In Fig. 10, we can see five numbered circles that correspond to five different levels of semiotic quality in linguistically based EUP environments. Each level represents the set of criteria for which the environment was positively evaluated (i.e. the most advanced "yes" obtained as an answer to the proposed question in the examination flow). The flow of

Table 1
A semiotic characterization of EUP environments

Level	Positive features of the environment	Negative aspects and suggested additional features to achieve usable EUP
1	Since the user understands the UIL fully, he can possibly formulate valid extension goals.	But because the semantics of the UILx cannot be expressed in the EUPL, a correspondence between EUPL and UIL cannot be adequately established. To evade this situation, intelligent agents can be designed to capture extension goals expressed in UIL-like language and automatically generate the extension without user mediation.
2	Since the semantics of UILx can be expressed in the EUPL, the user can recognize existing UILx elements as they are expressed in EUPL.	But because the user does not completely understand the EUPL, he cannot generate or produce novel descriptions for elements that still do not (but could) exist in the application. To evade this situation, an intelligent EUPL interpreter could be designed to supervise and assist users in their attempt to devise new functionality for the application.
3	Since the user understands all signs in the EUPL, he can produce valid sentences in EUPL and specify correct executable functions in this language.	But because the user cannot predict if his specifications in EUPL will achieve the minimum cycle of interaction, he is prone to producing extensions that cannot be triggered and used or do not provide feedback. To evade this situation, an intelligent interface agent could be devised to bridge interaction gaps that may persist in the user's code.
4	Since the user can discriminate when EUPL code will meet the minimum cycle of interaction, he can generate executable code that is fully interactive.	But because he could be required to design novel UIL components (or interactive types) for some valid EUPL constructs, this could in fact prevent him from achieving his extension goals. To evade this situation, an intelligent widget editor could be designed to guide the user's steps in creating new interactive types.
5	The user fully understands the UIL and the EUPL and he can always translate arbitrary valid EUPL code into UIL signs (actual and/or potential).	

analysis suggested for evaluating environments allows us to propose a distinction among five different types of EUP environments, which can be seen in Table 1.

Table 1 allows us to classify some programming by demonstration applications as level 1 environments. For example, Metamouse (Maulsby and Witten, 1993) and SmallStar (Halbert, 1993) are instances of programming in the user interface (according to Cypher, 1993, a term coined by Halbert). Barbosa's model of extending software through the use of metaphors and metonymies (Barbosa, 2000), though not a case of programming by demonstration, can also be classified as level 1. All of these applications present an alternative for users to achieve usable extensions without getting involved in actual programming tasks.

We do not know of level 4 or level 5 EUP environments, and thus they are now only theoretical possibilities. Ongoing research by da Silva (forthcoming) is expected to produce a demonstrable EUP environment at these advanced levels of semiotic adequacy. We cannot say, without empirical evidence, that level 5 is sufficient to guarantee EUP usability and applicability. However, it is certainly distinctive of semiotically sophisticated environments. If this is backed by cognitive evidence, we can conjecture that end user programming languages *must* be semiotically continuous with the user interface language of the applications in which they are embedded.

Most commercially available EUP environments can be classified as level 2 or level 3. None of the EUP languages surveyed in this paper are semiotically continuous with the UI languages of their respective applications. The examples we have shown demonstrate the problems users can eventually encounter in these environments. However, intelligent interface agents, toolkit wizards and the like are available in different types of packaging (from academic demos or toy applications to professional CASE tools and implemented autonomous agents). Thus, a combination of EUP environments with these kinds of support tools is not a far-fetched hypothesis.

By theoretical construction, building novel applications in an EUP environment extrapolates the boundaries of semiotically continuous languages. Novel applications are novel languages per se. They are single-purpose languages, and the ability to generate novel and distinct single-purpose languages cannot be achieved by a single-purpose meta-language. Only a programming language can achieve this goal. So, we see that this differentiation leads to a more accurate elaboration of the goals stated by Fischer (1998). If users are expected to be able to design novel applications, then they must be able to program with general-purpose languages. However, if users are expected to be able to design an elaboration of the original software application, then they may find enough resources in end-user programming as qualified by our semiotic engineering principles.

There are, additionally, some important open issues associated to the principles proposed in this paper. They constitute an agenda for further research about EUP usability, applicability, and modeling. One of these has to do with the contents of documentation about computer languages. They play a crucial role in helping users make sense of the languages they use, but they are themselves the result of a design process. In EUP environments, the original design intent conveyed by the application's interface must be addressed by documentation writers, since users need to become proficient in those unique application languages. However, software documentation (see Theodos, 1991; Carey et al., 1992 and design documentation (see Moran and Carroll, 1996) techniques do not allow us to predict the kinds of contents a user will find in this source.

Another open question has to do with the granularity of signs in the UIL and the EUPL. If their granularity is *the same* (though this is an ideal situation), we can think of an isomorphism between interface and end-user programming language. This is apparently a condition that must prevail for visual programming proposals (Chang, 1990). However, if the granularity is not the same, the length and the nature of the reasoning process performed by users may lead to different approaches. For instance, if the reasoning requires only deductive reasoning, we may think of mechanizing the generative steps, and having systems like those proposed in the programming by demonstration paradigm. Nonetheless, if long abductive reasoning is required, mechanization may be too complex or impossible, altogether. Thus, only a human programmer would be able to specify how steps of actions can achieve goals.

Finally, although the ideas presented in this paper are intended to help model EUP on a semiotic basis, and provide principles that can be used to distinguish the various kinds of EUP techniques and approaches, they may also shed light on topics related to computer languages in general. The most obvious connections we can point at are those with error-handling modules embedded in programming language interpreters and compilers, and with computable representation languages. These clearly constitute interesting research topics per se, which we aim to pursue in future work. As the two semiotic engineering principles now stand, they should facilitate the definition and interpretation of empirical studies, and provide some theoretical underpinnings for modeling end user programming activities in more precise terms.

Above all, the principles take user interface languages and end-user programming languages as “user languages” in the first place. Despite the desirable and undesirable similarities with programming languages, the underlying semiotic system in which the semantics of the user interface language is defined can only be usable in EUP if the two principles hold. Even if the programming style of EUP is more declarative than imperative, setting the users free from the need of controlling program flow and manipulating variables, our proposal remains applicable. Therefore, we believe that Semiotics can undoubtfully shed light on the phenomena we have studied, and that it can provide theoretical underpinnings for research about the usability of end-user programming environments.

Acknowledgements

This research project is partially funded by CNPq and FAPERJ. The authors would like to thank Raquel Prates, Tom Carey and Kevin Harrigan for substantial contributions to this work, as well as all the referees who provided much insightful feedback about earlier versions of this paper. They would also like to express their gratitude to John Thompson for the language review, and last, but not least, to the members of the Semiotic Engineering Research Group, in PUC-Rio, for their constant enthusiasm and incredible intellectual energy that give rise to questions that turn research into adventure.

References

AgentSheets, 2000. AgentSheets Home Page on the Internet — http://www.agentsheets.com/home.html (as of March 2000).

Andersen, P.B., Holmqvist, B., Jensen, F.F., 1993. The Computer as Medium. Cambridge University Press, Cambridge.

Barbosa, S.D.J., 2000. Expanding Software through Metaphors and Metonymies. In IUI 2000 — 2000 International Conference on Intelligent User Interfaces, New Orleans, USA, 9–12 January, pp. 13–20.

Brown, G., Yule, G., 1983. Discourse Analysis. Cambridge University Press, New York.

CACM, 1994. Intelligent Agents. Communications of the ACM 37 (7) 18–169.

CACM, 2000. Programming by Example. Communications of the ACM 43 (3), 72–114.

Carey, T., Nonnecke, B., Mitterer, J., 1992. Prospects for active help in online documentation. Proceedings of the ACM Tenth International Conference on Systems Documentation, pp. 289–296.

Chang, S.K., 1990. Visual Languages and Visual Programming. Plenum Press, New York.

Cordy, J., 1992. Why the user interface is NOT the programming language: and how it can be in Myers. In: Myers, B.A. (Ed.). Languages for Developing User Interfaces. Jones and Bartlett, Boston, p. 1992.

Cypher, A., 1993. Watch What I Do: Programming by Demonstration. MIT Press, Cambridge MA.

Cypher, A., Smith, D.C., 1995. KidSim: end user programming of simulations. Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems, Vol. 1, pp. 27–34.

Dertouzos, M., 1992. The user interface is the language. In: Myers, B.A. (Ed.). Languages for Developing User Interfaces. Jones and Bartlett, Boston.

DiGiano, C., Eisenberg, M., 1995. Self-disclosing design tools: a gentle introduction to end-user programming. Proceedings of DIS'95, Ann Arbor, Michigan, 23–25. ACM Press, New York.

Eco, U., 1976. A Theory of Semiotics. Indiana University Press, Bloomington.

Eco, U., Santambrogio, M., Violi, P. (Eds.), 1988. Meaning and Mental Representations Indiana University Press, Bloomington.

Eisenberg, M., 1995. Programmable applications: interpreter meets interface. SIGCHI Bulletin 27 (2).

Fischer, G., (1998). Beyond couch potatoes: from consumers to designers. Proceedings of The Fifth Asia Pacific Computer Human Interaction Conference. IEEE Computer Society, New York, pp. 2–9.

Halbert, D.C., 1993. SmallStar: programming by demonstration in the desktop. In: Cypher, A. (Ed.). Watch What I Do: Programming by Demonstration. MIT Press, Cambridge.

Hoopes, J., 1991. Peirce on Signs. The University of North Carolina Press, Chapel Hill.

Hutchins, E.L., Hollan, J.D., Norman, D.A., 1986. Direct manipulation interfaces. In: Norman, D.A., Draper, S.W. (Eds.). User Centered System Design, New Perspectives on Human-Computer Interaction. , pp. 87–124.

IUI, 2000. 2000 International Conference on Intelligent User Interfaces. New Orleans Louisiana, USA, 9–12 January 2000.

Jorna, R., van Heusden, B., 1996. Semiotics of the user interface. Semiotica 109 (3/4), 237–250.

Lotus, 1995. Smart Suite 96. Lotus Development Corporation. Software Documentation, 1995.

Malone, T.W., Lai, K., Fry, C., 1995. Experiments with oval: a radically tailorable tool for cooperative work. ACM Transactions on Information Systems. 13 (2), 177–205.

Maulsby, D., Witten, I.H., 1993. Metamouse: an instructible agent for programming by demonstration. In: Cypher, A. (Ed.). Watch What I Do: Programming by Demonstration. MIT Press, Cambridge, MA.

Meystel, A.M., 1996. Intelligent systems: a semiotic perspective. International Journal of Intelligent Control Systems 1 (1), 1996.

Microsoft, 1996. Microsoft Office 97. Microsoft Corporation, Software Documentation, 1996.

Moran, T.P., Carroll, J.M., 1996. Design Rationale: Concepts, Techniques, and Use. Lawrence Erlbaum, Mahwa, NJ.

Mullet, K., Sano, D., 1995. Designing Visual Interfaces. Sunsoft Press, Menlo Park.

Myers, B.A., 1992. Languages for Developing User Interfaces. Jones and Bartlett, Boston.

Myers, B.A., 2000. Natural Programming: Project Overview and Proposal. http://www.cs.cmu.edu/(NatProg/projectoverview.html (as of August 2000).

Nadin, M., 1988. Interface design and evaluation-semiotic implications. Hartson, R., Hix, D. (Eds.). Advances in Human-Computer Interaction 2, 45–100.

Nadin, M., 1997. Signs and System, A Semiotic Introduction to Systems Design. Cambridge University Press, Cambridge.

Nardi, B., 1993. A Small Matter of Programming. MIT Press, Cambridge, MA.

Peirce, C.S., 1931–1958. Collected Papers. Harvard University Press, Cambridge, MA.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., Carey, T., 1994. Human–Computer Interaction 1994. Addison-Wesley, Reading, MA.

Rheinfrank, J., Eveson, S., 1996. Design Languages. In: Winograd, T.A. (Ed.). Bringing Design to Software. ACM Press (Addison-Wesley), New York, pp. 63–80.

Searle, J.R., 1979. Expression and Meaning. Cambridge University Press, New York.

Sebeok, T.A., 1994. Signs: An Introduction to Semiotics. University of Toronto Press, Toronto.

Shneiderman, B., 1983. Direct manipulation: a step beyond programming languages. IEEE Computer 16 (8), 57–69.

da Silva, S.R.P., in preparation. Um Modelo Semiótico para EUP. (A Semiotic EUP Model). PhD Thesis, PUC-Rio, Rio de Janeiro.

de Souza, C.S., 1993. The semiotic engineering of user interface languages. International Journal of Man-Machine Studies 39, 753–773.

de Souza, C.S., Barbosa, S.D.J., Prates, R.O., 2001. Report on the CHI2000 workshop on semiotic approaches to user interface design. SIGCHI Bulletin (in press).

Theodos, R.D., 1991. Text in context: writing online documentation for the workplace. Proceedings of the Ninth ACM International Conference on Systems Documentation, pp. 140–148.

Traynor, C., Williams, M.G., 1997. A study of end-user programming for geographic information systems. Empirical Studies of Programmers: Seventh Workshop, pp. 140–156.

Wittgenstein, L.J.J., 1953. Philosophical Investigations. Blackwell (Basil), Oxford.

Abductive reasoning, or abduction: the process of adopting an explanatory hypothesis to assign meaning to a state of affairs

Actual sign: an existing sign* in an application

Communication: the exchange of messages

Interactive patterns: the patterns that emerge through the interaction of a user with the interface of a computational system

Interactive token: a particular instance of an interactive type*

Interactive type: a class of interactive elements. A widget may be considered an interactive type, whereas its particular instances within an application are interactive tokens

Interpretant: see the definition of sign*

Lexical sign: a meaningful word in a given language

Metacommunication artifact: an artifact that conveys its capabilities of communication, i.e. The means by which it can send and receive other messages

Minimum cycle of interaction: the minimum cycle required for the correct interpretation and use of an interactive element. This minimum cycle consists of three steps: application 'says' something to the user; the user 'says' something to the application that triggers an action; the application 'replies' to the user

Phrasal sign: a meaningful grammatical construct in a given language, i.e. An organizing structure of lexical signs*

Potential sign: is a *non-existing* sign* (word or phrase) in a language, which can nevertheless be generated by lexical and/or grammatical extensions that abide to its derivational patterns (i.e. To morphological and grammatical meta-rules)

Pragmatic rules of interpretation: a subset of the symbol interpretation rules that apply to specific situated interactions

Semantic model of an application: the subset of the symbol* interpretation rules and elements that can be invariantly applied to every context of interaction

Semiotics: the study of communication*, representations, their interpretation and usage

Semiotic system: system of signs*: textual, visual, gestural, or other

Sense making: the process of inferring the meaning of a sign*

Sign: 'a sign, or *representamen*, is something which stands to somebody for something in some respect or capacity. It addresses somebody, that is, creates in the mind of that person an equivalent sign, or perhaps a more developed sign. That sign which it creates i call the *interpretant* of the first sign. The sign stands for something, its *object*'. Peirce (1931). Signs are defined not only in a representational dimension, but also in a communicative one

Symbol: minimal representation unit in a formally-defined language. Symbols are defined exclusively in representational dimensions

Text: 'the record of a dynamic process in which language was used as an instrument of communication in a context by a speaker/writer to express meanings and achieve intentions (discourse)'. Brown and yule (1983, p. 26)

Unlimited semiosis: the process of generating the interpretant of a sign*, which may become another sign, and then give rise to a new interpretant, ad infinitum