Semiotic engineering principles for evaluating end-user programming environments

Clarisse Sieckenius de Souza
Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente 225
Rio de Janeiro, RJ – Brazil
E-mail: clarisse@inf.puc-rio.br

PUC-Rio Inf MCC10/99 Maio'99

Abstract

End user programming (EUP) environments are difficult to evaluate empirically. Most users don't 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 to 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. They introduce a preliminary theoretical model of EUP and thus facilitate the definition and interpretation of empirical evaluation studies. They also describe some specific semiotic qualifications that more usable and applicable EUP languages could be expected to have.

Keywords: semiotic engineering, end-user programming, evaluation methods, usability, theoretical model

Semiotic engineering principles for evaluating end-user programming environments

Clarisse Sieckenius de Souza
Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente 225
Rio de Janeiro, RJ – Brazil
E-mail: clarisse@inf.puc-rio.br

Abstract

End user programming (EUP) environments are difficult to evaluate empirically. Most users don't 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 to 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. They introduce a preliminary theoretical model of EUP and thus facilitate the definition and interpretation of empirical evaluation studies. They also describe some specific semiotic qualifications that more usable and applicable EUP languages could be expected to have.

Keywords: semiotic engineering, end-user programming, evaluation methods, usability, theoretical model

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 multi-functional applications is high, and users are always demanding enhancements and extensions to successful software tools. The other is the growing 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 nineties (see [Myers'92, Nardi'93]), and the latter has been evolving both in the fields of HCI and Design (see [Fischer'93, Cypher'93c, Nardi'93, DiGiano and Eisenberg'95, Fischer'98]). But the practical needs encountered by users of applications such as CSCW systems [Malone et al.'95], text editors [Cypher'93b], and spreadsheets [Nardi and Miller'90] always point at the tension between requirements related to usability and expressiveness [Repenning and Ambach'97].

In contrast with voluminous research on the usability of software applications from a non-EUP perspective, research about usable EUP environments is scarce [Nardi'93, Traynor and Williams'97]. 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). Yet, another important reason is that EUP environments are typically plainly uninviting for non-programmers, which explains and in a way perpetuates this situation.

The difficulties posed for empirically-based usability evaluation in EUP start with the identification of the *typical subject* in a heterogeneous, scattered, and small community. As a result, heuristic approaches lack significant empirical evidence from which to build a solid body of practical knowledge. However, there is also a theoretical problem 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 in this case, because they (implicitly or explicitly) assume that the semantic model of applications is stable. Despite some preliminary suggestion that user-centered methods could demonstrate the power of tailorable applications [Malone et al.'95], for instance, no theoretical or empirical grounding has yet been fully established to sustain consistent evaluative investigation of EUP environments from a user's point of view.

So, EUP environment designers are currently ill supported in their effort to provide useful tools for software extension and customization tasks. The challenge to usability evaluation in this scenario is that the nature of tasks performed by users is radically different from those performed outside EUP. Namely, EUP tasks are *semiotic design* tasks that introduce new meanings and expressions in the original environment. They are shaped by the users' interpretation of the application, the available *programming* tools, and the *representation* systems the tools act upon.

The ability to turn users from consumers into designers has been repeatedly voiced by researchers like Fischer and his associates [Fischer'93, Repenning'94, Eisenberg and Fischer'94, Fischer'98]. They claim that adding functionality to software will never account for the situated needs of users who are experts in their domains of activity, and must not be prevented from exercising their creativity by technological shortcomings. In their view, software systems must be designed to evolve and reach not only for *usability* at low cognitive and physical costs, but also for *applicability* to relevant situated problems.

This paper takes a semiotic engineering approach [de Souza'93] to EUP tasks and proposes 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* and the *Semiotic Continuum Principle* provide some theoretical underpinning that can serve as a basis for analyzing and explaining specific evaluation issues in EUP environments.

In the next three sections of this paper, 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. The nature of signs and sign systems (or languages) that co-exist in extensible software applications, along with some of the relations that hold between them, set the basis for the *Interpretive Abstraction* and the *Semiotic Continuum* principles, whose effects are demonstrated in concrete situations.

A discussion of the challenges posed by EUP for many evaluation methods, and of how the proposed principles advance the theoretical foundations for EUP evaluation can be found in the last section of this paper. The approach is shown to characterize more precisely Fischer's paradigm of users as designers, to provide a structure for categorizing EUP environments as having qualitatively distinct profiles, and to provide explanations for some of the points made by EUP researchers in *ad hoc* empirical observations.

Examples of usability problems in EUP environments

As pointed out by supporters of the programming by demonstration paradigm [Cypher'93c], the leading motivation for reprogramming parts of an application is having to perform repetitive tasks. So, let us consider a scenario in which a user repetitively performs the following sequence of commands over a wide range of documents:

a) Given a specifically selected cursor position in the document,
b) all occurrences of the word <u>Chapter</u>, **down to the end of the document**, are replaced by the word <u>Section</u>.

In Lotus's WordPro™ (LWP) [Lotus'95], the designer-to-user message implicit in its interface suggests that the task can be easily achieved by the following set of events (see Figure 1):

I. Click on the desired cursor position in the document
II. Type *Ctrl+F* (or select *Find and Replace...* in the *Edit* menu)
III. Type in the word <u>Chapter</u> next to the label *Find* in the Dialogue Box
IV. Type in the word <u>Section</u> next to the label *Replace with* in the Dialogue Box
V. Check that the arrowhead pointing to the right is selected
VI. Click on the button named *Replace All*

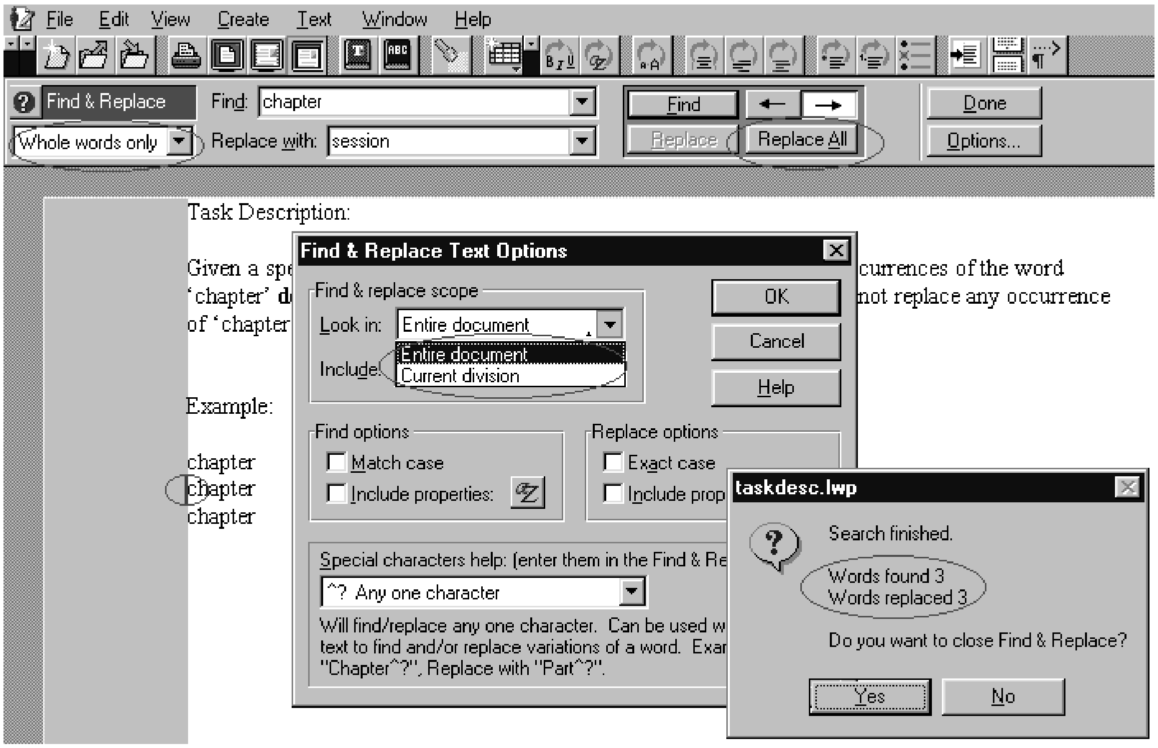

Figure 1: Lotus's WordPro screen shots.

Against the user's expectations, steps (I) through (VI) do not cause all the occurrences of <u>Chapter</u> from the current cursor position onwards to be replaced by <u>Section</u>. Instead, **all** occurrences of <u>Chapter</u>, either onwards (as expected) or backwards (as not), are replaced by <u>Section</u>. With some further explorations in the interface and online documentation, the user may find that the *Replace All* action overrides the directional choices signaled by the arrowhead buttons. The user can 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). So, the interface is perceived to suggest affordances (esp. a wrapping control) that the application does not really provide.

If the user turns on the script-recording tool while performing steps (I) through (VI), the automatically generated script will consist of the following code in LotusScript [Lotus'95]:

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

We see that, although the arrowhead button is marked as pressed during the recording (see Figure 1), apparently the script does not capture this parameter. And, if the user repeats the recording pressing the arrowhead backwards, she will again find out that **all** occurrences of <u>Chapter</u> are replaced by <u>Section</u>. However, with this explicit change of the default direction of the search, the recorded script will consist of a slightly different code:

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

This first scenario highlights two important problems. One is that the UIL suggests affordances that do not really belong in the semantic model of the application (e.g. the direction of search cannot be combined with the *Replace All* capability). The other is that automatically generated EUPL code does not reflect the relevant signs of the UIL sentence produced by the user (e.g. the parameter *forward* for the search is not present in the first script).

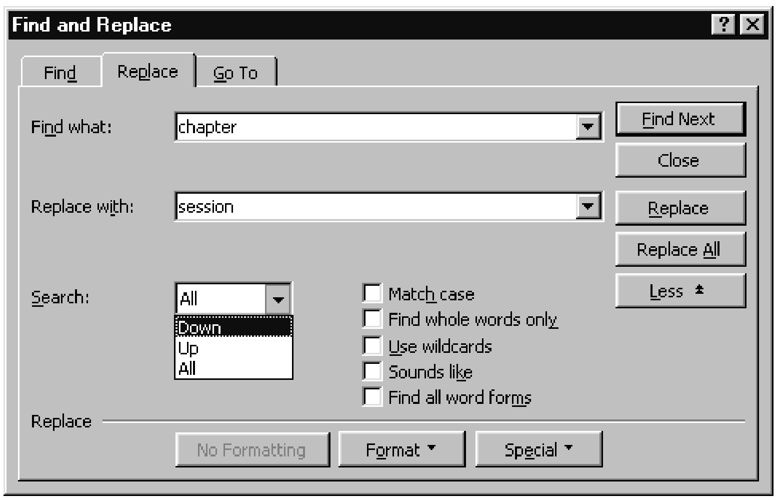

Figure 2: MS Word interface for find and replace tasks

In another application in the same domain, Microsoft's Word™ (MSW) [Microsoft'96], the same task can be carried out in a slightly different way. Find and replace tasks lead users to a dialog window that has two different states. The default one (or short state) does not directly afford directed searches. A button labeled *"More"* (graphically placed where the button labeled *"Less"* is in Figure 2), tells the user that there are other parameters that can impact the realization of this task. So, by pressing this button, the user can see the second state of the dialog window, where more details are provided. In it, the direction of search can be explicitly set.

So, in order to achieve the same find and replace task of the previous scenario, the user now performs the following interface steps:

I. Click on the desired cursor position in the document
II. Type CTRL+H (or select *Replace...* in the *Edit* menu)
III. Type in the word <u>Chapter</u> next to the label *Find what* in the Dialogue Box
IV. Type in the word <u>Section</u> next to the label *Replace with what* in the Dialogue Box
V. Click on More to see further options for the task
VI. Select "Down" in the list next to the Search field
VII. Click on the *Replace All* button
VIII. Read a message saying that the end of the document has been reached and asking if the search should continue from the beginning

IX. Answer *No*

Although there are three additional interactive steps (given the same user strategies for interaction used with LWP), the goal is successfully achieved this time. The recorded corresponding macro in Visual Basic for Applications™ (VBA) consists of the following code:

```
Sub CorrectSections()
  Selection.Find.ClearFormatting
  Selection.Find.Replacement.ClearFormatting
  With Selection.Find
    .Text = "Chapter"
    .Replacement.Text = "Section"
    .Forward = True
    .Wrap = wdFindAsk
    .Format = False
    .MatchCase = False
    .MatchWholeWord = False
    .MatchWildcards = False
    .MatchSoundsLike = False
    .MatchAllWordForms = False
  End With
  Selection.Find.Execute Replace:=wdReplaceAll
End Sub
```

The dialogue that takes place in steps VIII and IX is signaled in the code as *".Wrap = wdFindAsk"*. Although the terms are somewhat cryptic, with a little effort they can be correctly taken to mean "ASK <something having to do with> FIND", and eventually they can be associated to the confirmation that is asked from the user. However, if the user answers *Yes* instead of *No,* the generated code is exactly the same. Consequently, the mapping between UIL signs, on the one hand, and the macro code, on the other, is once again obscure.

A more intuitive value for *.Wrap* parameter would be a choice between True or False, and not *wdFindAsk*. And in fact, if we replace the *wdFindAsk* value with *False*, quite surprisingly the macro not only executes without problem, but also does exactly what the user's intent has been shown to be in (a) and (b), above. In other words, a small change for:

```
Sub CorrectSections()
…
    .Wrap = False
…
End Sub
```

achieves the task without asking for confirmation. The logic of True and False, however, does not lead us too far. If we replace True for False and try to run a macro with:

```
Sub CorrectSections()
…
    .Wrap = True
…
End Sub
```

the outcome is not a circle search of text, but a "Runtime error 4608" message, telling us that there is a *value out of range*. The user is left wondering. What range? What does error 4608 mean?

So, in this second scenario, there are also three main problems. One is that the semantics of UIL signs is confusedly described in the EUPL. It is unintuitive, if we compare the *wdFindAsk* with *True* or *False*, and incomplete, if we take into account that *false* generates executable code, but *true* does not. Another is that a user may wonder what is the role of choosing between **all, up** and **down**, in the dialogue window, if there is no apparent effect of this choice in a *Replace All* situation. And the last one is that while working in the EUP environment, the user is exposed to inexplicable signs that are foreign to the application's domain.

The third kind of problem may also be encountered in LWP, if the user records a script that automatically saves a file on a floppy disk. If the floppy is locked or not inserted, once the script is invoked and triggered, the follow-up message is quite disconcerting (see Figure 3).

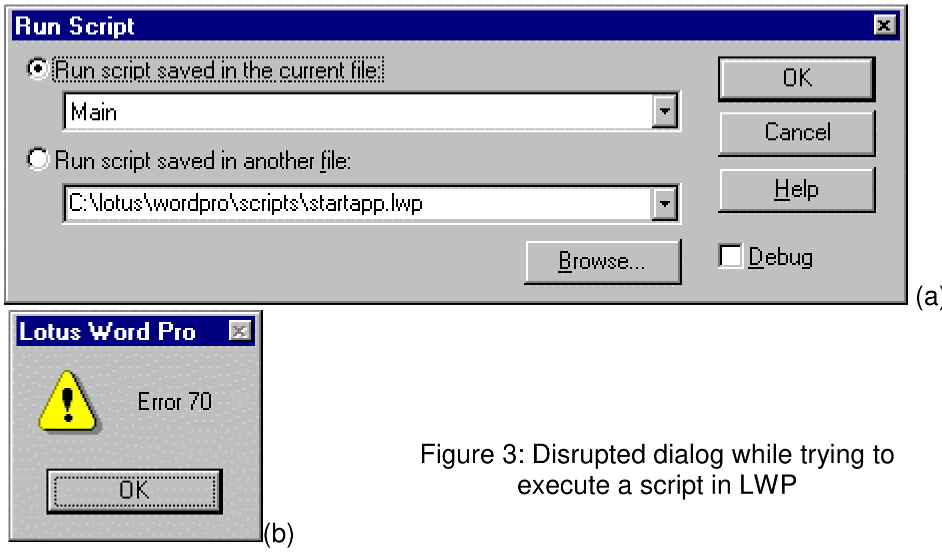

Figure 3: Disrupted dialog while trying to execute a script in LWP

These are significant examples of the kinds of problems that discourage the vast majority of potential EUP users and perpetuate the separation between users and programmers. Commercial scripting and macro languages are genuine programming languages, garnished with domain jargon and trimmed short of more elaborate data and control structures. And they are just as difficult to use as any programming language [Cordy'92, Nardi'93]. Moreover, the correspondence between UIL and EUPL is often inconsistent in spite of the fact that whatever users know about the application is motivated by and constructed within the semiotic (i.e. representational and communicative) systems that are present in the interface language.

The debate about EUP can be superlatively summarized by the titles of two papers that appeared in the early nineties: *The user interface is the language* [Dertouzos'92] and *Why the interface is not the programming language, and how can it be* [Cordy'92]. In both, authors present their arguments for powerful interface languages that allow users to extend applications. This was the line adopted by many followers of the programming by example (or by demonstration) paradigm (see [Cypher'93a, Lieberman'93a, Lieberman'93b, Cypher and Smith'95] for example). The other alternative was followed by those that try to introduce users *'to the art of programming'* in a gentle way (see [Eisenberg'95, DiGiano and Eisenberg'95, and Malone et al.'95] for example).

The Semiotic dimensions of EUP challenges

Semiotics provides a wealth of insights for the analysis of EUP itself and the associated usability problems frequently found in this environment. The discipline is devoted to investigating the nature of signs and communication [Eco'76, Sebeok'94], and has been felt by many to bring important contributions to software designers [Nadin'88, Andersen'90, Andersen et al.'93, Mullet and Sano'95, Jorna and van Heusden'96, Pimenta and Faust'97] and computer scientists

[Meystel'96, Gudwin and Gomide'97]. Our work in Semiotic Engineering [de Souza'93] focuses on two central ideas:

a) Interactive systems are messages sent from designers to users, about how to send and receive other messages in the system.
b) Interactive systems are also *performing* messages that send and receive a range of input/output messages. Thus, they are in fact *metacommunication artifacts*.

It is in EUP environments, however, that semiotics can bring forth innovative insights [Barbosa et al'99] and potentially advance investigations about the usability of extension facilities in software. The analysis of tasks, roles, and resources involved in EUP sheds light on different facets of such environments.

Signs, interpretants, unlimited semiosis and abduction

We draw on theoretical concepts proposed by Peirce, who lived most of his life in the 19th century, and had his work published in eight volumes in the first half of 20th century [Peirce'31-58]. 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 it must be **interpreted by a subsequent thought or action** [Hoopes'91]. The subsequent thought or action generated by a sign is called its **interpretant**. An important feature of interpretants is that they can, themselves, be signs that trigger further interpretants. So for example, when faced with the *OK Button* shown in Figure 3(b), the user actually generates a chain of interpretations that are associated to each other by causal relations. In this chain there may be signs related to such notions as: *"something went wrong", "the script has a bug", "the file is corrupted", "the application will crash", "the diskette is not in the drive", "computers never work as we want"*. And there may also be actions like: <check the diskette>, <click the button>, <reset the computer>, <ask for help>.

The significant contribution of semiotic theory to human-computer interaction (HCI), as illustrated by this case, is that the interpretation of a sign is an associative process, whose depth and length cannot be fully predicted. This process has been called **unlimited semiosis** [Eco'76], 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 unbounded, how can any two people effectively communicate with each other?

The answer to this challenge in peircean semiotics is 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 Peirce's terms, "deduction proves that something must be; induction proves that something actually is; and abduction suggests that something may be" [Peirce'31-58]. Thus, in human communication, successful semiosic processes may converge to a steady point as soon as the interpreter reaches a satisfactory meaning for the sign under interpretation. The reasoning method used to generate it may be deductive, inductive, or abductive in nature. In abductive reasoning situations, sign interpretation is the same as sense making. As soon as a plausible hypothesis is formulated to explain the presence of the sign, it is taken to be at the heart of the sign meaning.

This type of reasoning elegantly explains not only successful communicative exchanges among people, but also unsuccessful ones. The latter result from faulty hypotheses (or misconceptions), that may be further corrected through remedial dialogue or counterfactual information. It also explains why we can handle conversation without really bothering to understand the detailed meaning of each and every sign in it. In HCI scenarios, as seen in the previous section, a user may not really bother to understand what *"error 4608"* means, in precise extensive terms. She may perfectly get by with the hypothesis that this is *"an internal error code related to the value True"* and generate an interpretant with the <restore .wrap=wdFindAsk> action. 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.

Language interpretation and language generation

Semiotic theory has dealt fundamentally with sign interpretation phenomena. But, the adequacy of interpretation must be measured against factors that originate in the sign production process. In other words, the meaning of a sign must be evaluated against the intention of the person (or the entity) that communicates it.

The Theory of Sign Production (TSP) [Eco'76] aims to describe the processes by which a communicating agent chooses how to express some action, idea, sensation or attitude. If signs are linguistic in nature and agents are human, the code used for expression is natural language, and semiotic theory is often replaced by linguistic theories related to discourse analysis [Brown'95]. However, the spectrum of communicative codes used by humans extrapolates natural languages by far, and takes us from linguistics (which deals with natural language phenomena) into semiotics (which deals with all communicative codes, linguistic or not).

An account of computer language interpretation and computer language generation, as actually carried out by end users, would provide EUP with valuable theoretical foundations for semiotic and cognitive research. Norman's Cognitive Engineering [Norman'86] represented a major step in HCI, because it accounted for the communication processes that take place between user and computer, from the radical perspective of users. Hence, sign interpretation and sign generation have been characterized and evaluated in interpretation-centered cognitive terms [Hutchins et al.'86].

Nevertheless, in EUP scenarios, interpretation and generation processes operate on a higher-order coded message. What the user must be able to interpret is the patterned combination of actual and potential signs that appear in HCI. These combinations follow design principles that govern the artifact's behavior and determine the affordances it exhibits. Unless such patterns are correctly interpreted, the user can never generate novel utterances (i.e. extensions or customizations) to be interpreted by the machine.

In the MSW example, we see that the meaning of the sign *"wrap"* in the user's mind is likely to trigger an abductive reasoning path in which *True* and *False* are potential signs standing for total and partial searches, respectively. In Eco's TSP, these two latter signs are characterized as the result of a simple recognition that they are tokens of an established expressive type system for binary choices. However, not only did one of these combinations not work as expected, but also some new invented sign was used (*wdFindAsk*), potentially imposing a heavier abductive labor to human interpreters.

Inferring the grammar of a new language

The importance of design patterns in software engineering [Gamma et al.'95] has been extensively demonstrated and explored in recent years. Although HCI patterns play a paramount role in usable computer applications, in EUP their importance is magnified by the complexity of the intrinsic language acquisition task the user must perform.

Many HCI textbooks and design manuals emphasize the interactive patterns that can and at times must be used to ensure adequate usability. Some authors even claim that these patterns constitute a **language** [Rheinfrank and Eveson'96] *per se*. Nonetheless, at a closer examination, some distinctions become evident.

Firstly, a **language**, in formal terms, is a fully coded semiotic system (which includes a vocabulary, a grammar, and a set of semantic interpretation rules), deeply embedded in a (sub)culture (which provides most of its pragmatic rules of interpretation). Thus, the so-called "interactive languages" are not really languages in the same sense, but rather a style (a hypercode), or a set of general patterns (a hypocode) [Eco'76, Leite'98].

Secondly, a computer application's interface does have a language. But this is **a unique language** that can only refer to elements of the application's unique semantic model. 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 (i.e. in a different context). What can be, and actually is, transferred from one application to another are the hypocodes and hypercodes, which contribute to the "nativeness" of software in different platforms, and to "computer literacy" in general.

Consequently, the major (and essential) endeavor of every computer application user is to acquire a <u>new and unique language of interaction.</u> But if, in addition to this, users are expected to do programming tasks and tailor the application to their needs, their competence must extrapolate the ability to interact with the system, and allow them to produce correct linguistic descriptions of the application. In other words, not only must the users speak the application's language, but they must also be able to explain and predict grammatical constructs in it.

The sensitive aspect of this extrapolation is that the language used for designing new meanings is not a general-purpose language, but a single-purpose one. Although this may sound too radical at first sight, especially in view of some general agreement that EUP languages must be special-purpose languages [Nardi'93], a higher level of precision has some advantage for characterizing usability problems.

For instance, a special purpose-language may be taken to mean that it is designed to represent and communicate a set of phenomena of a given domain, or a given class of problems across various domains. However, this approach is precisely the problem with the examples in section 2.

LWP and MSW extension languages are designed to serve not only the text editing applications used in those examples, but also to serve other applications in the same software package (which includes spreadsheets, database, and graphic presentation software in both cases). They thus include constructs and require implementation solutions that entail incomprehensible signs from a single application's perspective.

<u>From designing interaction to designing interaction grammars</u>

The basic challenge in EUP is then to design a single-application language whose grammar can be easily intuited, used in all the directly afforded ways, and (most of all) be abducted and explained by the users. Without an implicit[1] or explicit, formal or informal[2], representation of the operating sign combination and sign interpretation rules of the language, a user cannot be expected to achieve EUP successfully.

Many of the so-called extension languages (e.g. Visual Basic for Applications [Microsoft'96], LotusScript [Lotus'95], Hypercard [Apple'98], and the like) will not meet this challenge. The gap between their own linguistic model and that of any application they can be used to extend is too long for real "end users" to bridge.

Moreover, unless software developers have a highly disciplined use of design patterns, an application must be born extensible. Usable and applicable extensibility can only be added at a

[1] An implicit representation of such rules is, for example, a regular consistent pattern of sign combinations that can be aprehended by users through repeated reinforcing evidence of the combinatorial principle [Mullet and Sano'95].

[2] An informal representation of such rules is, for example, a graphical depiction of patterns and a visual characterization of correct and incorrect arrangement of elements [Chang'90].

later stage (or in the next release of a product) if the extensible part of the UIL[3] can be grammatically described in terms of the EUPL, or if the linguistic instance can be bypassed and inferred by a reasoning subsystem embedded in the application (as with some programming by demonstration techniques [Cypher'93a]).

Finally, although we know that many and varied semiotic systems are drawn upon to communicate meanings in the UIL (e.g. natural language words and expressions, icons, spatial organization, visual and audio cues, among others), the kind of EUP exemplified in section 2 is basically linguistic. Thus, linguistically-based EUP application designers need to project all types of signs they have used in the UIL onto a single integrative linguistic system that serves as a semantic description environment for the original application (and all of its extended versions). Failure to meet this requirement results in breakdowns such as the inneffective arrowhead signs of the first LWP scenario.

Users as designers of interactive tokens

One last relevant point in our semiotic analysis of EUP tasks and environments is that, unlike what seems to be the target of many EUP enthusiasts, users cannot really build other applications with anything less than a programming language. Because every application has its own unique language, and because this language is the front door for EUP, the EUPL itself is unique, too. So, strictly speaking, the EUPL should only generate application-related meanings, but not other applications *per se*.

This is equivalent to generating tokens of actual or potential interactive types that are intrinsic to the conception of the original designer-to-user message. In the LWP save-to-diskette script, the UIL responds consistently (and helpfully) to an attempt of saving a file to a diskette when it is not inserted in the drive. The user receives a message with a number of possible reasons why the action cannot be performed in that situation. This interactive exchange is not expressed as a type in the EUPL. Thus, users are not prompted to produce a token for the error-trapping type of interaction that is present in the UIL.

Semiotic engineering principles for evaluating EUP usability

The following principles are meant to be applicable to *computer languages*. In natural languages, they don't make any sense basically because of the unbounded metalinguistic power of any human language (which allows us to "interface" and "program" NL interpreting agents with the same *code*). Nevertheless, NL has provided us with the radical inspiration for the formulation of these two principles. They are meant to help us say why and when two computer languages can be used to approximate the effect we would achieve in NL if systems could be so programmed and extended.

The Interpretive Abstraction Principle

The *Interpretive Abstraction Principle* relies on two specific notions, defined as follows:

- a <u>LEXICAL SIGN</u> is a meaningful word of a language; and
- a <u>PHRASAL SIGN</u> is a meaningful grammatical construct of a language.

Thus, given any two computer languages Li and Lj, Li **is an interpretive abstraction of** Lj if:

- the semantics of Li can be described in Lj ; and
- a user of Li can make sense of **all** of Li's lexical and phrasal signs by resorting to three fundamental and possibly intersecting sources:
    i. the intrinsic patterns of occurrence of such signs in situated Li *discourse*;
    ii. some extrinsic metalinguistic knowledge about Li; and
    iii. the user's cultural background.

[3] From this point on, the acronym UIL will refer that part of a User Interface Language that is involved in extension and configuration tasks.

The *Interpretive Abstraction Principle* is a qualified form of the *Abstraction Principle* known to all computer specialists. The latter emphasizes the mechanisms by which details are hidden from users, allowing them to focus on relevant aspects of the discourse domain. It 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 thus added to the familiar principle for sake of emphasis and clarification. 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.

In order to explain and illustrate this principle, we can instantiate the variables in it with values drawn from an EUP environment. Thus, the UIL of an extensible application is an interpretive abstraction of some underlying computer language (e.g. the EUPL, the programming languages used by the application's developers, or even the assembly and machine languages) if users can fully understand all signs in it because 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).

Notice that in order to understand the interface language, the user should not need to know 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 abductions that can be made from the summative knowledge base that integrates (i), (ii), and (iii), above. For a concrete example, consider the following evidence.

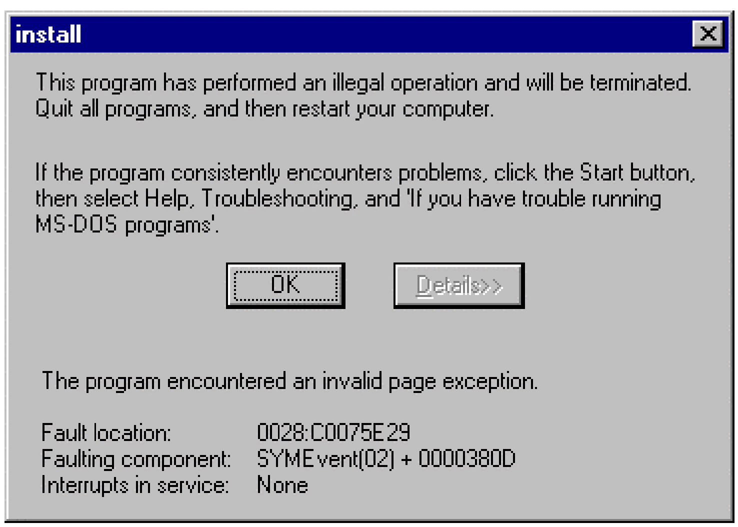

Figure 4: An example of faulty interpretive abstraction

There are many signs and sign constructs in the UIL "text" shown in Figure 4 whose meanings cannot possibly be assigned by users who do not have specialized knowledge of low-level software architecture and operating systems. However, the application whose installation generated this sort of output belongs to the category of "desktop general-purpose program" (namely, a text editor). Nowhere in its online documentation can the user find out a satisfactory explanation for "invalid page exception", for instance. Quite contrarily, "pages" in the user's context have to do with printed documents (and not with the machine's memory locations).

A *satisfactory* interpretation is the result of an abductive reasoning process determined by such factors as the ability to generate an alternative path towards the achievement of the intended goal. The possibility of assigning high-level semantic features like "a problem" or "a bug" is unsatisfactory in this case precisely because it cannot help the user in generating possible hypotheses about why this happened or what must be made differently so that the obstacle is overcome[4].

A more useful error message, redesigned for purposes of demonstrating the effect of observing the *Interpretive Abstraction Principle,* can be seen in Figure 5. Terms and expressions that belong in any online help documentation or tutorial have been used to replace the operating system jargon. The text also includes an explanation of why the user should restart the computer and why everything may occasionally keep on working properly if the user does not do it.

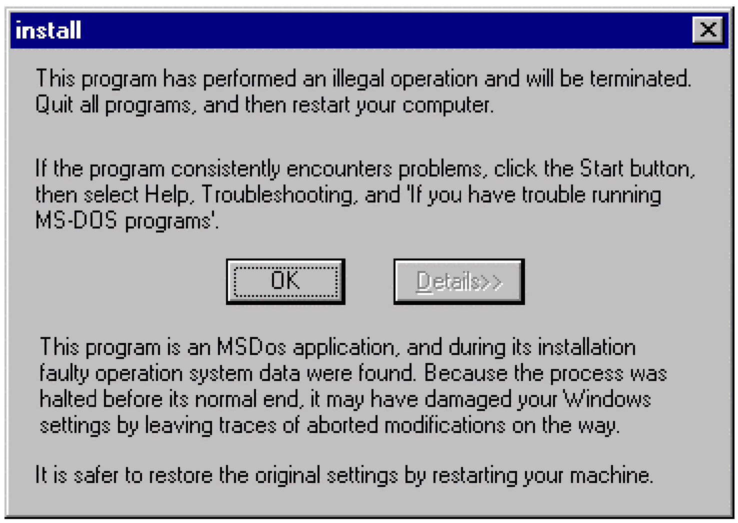

Figure 5: The effect of observing the *Interpretive Abstraction Principle*

This kind of decision-support information is the key to pragmatic adequacy, a feature that the *Semiotic Continuum Principle* will try to capture.

The Semiotic Continuum Principle

The *Semiotic continuum Principle* introduces the last two definitions required in this paper:

- An ACTUAL SIGN in a language is an **existing** word or phrase of that language; and
- A POTENTIAL SIGN in a language is **non-existing** word or phrase in that language, which can however be generated by lexical and/or grammatical extensions that abide to its derivational patterns (i.e. to morphological and grammatical meta-rules).

In the previous section we have seen why the distinction between actual and potential signs, as well as the existence of meta-rules that govern the creation of new words and new syntactic combinations is of crucial importance in EUPL design. For instance, in some programming languages, control structure blocks are lexically marked by the keyword that signals that structure written in reverse other. Thus, an IF block is closed by the word FI, a DO block is ended by OD, and so on. If a new release of one such language includes a CASE control structure, all users (programmers) will intuitively know that the block end will be marked by the word ESAC. This is a

[4] All computer users encounter installation or execution failure messages every once in a while, but only very few experts can assign the correct semantic meaning to them. However, many annecdotes are told about how to generate a *satisfactory* meaning for error messages. Not surprisingly, many people are satisfied with the notion that these messages contain *garbage*, and in practice you just have to press ESC, ENTER , or RESET.

quiet morphological pattern of the language that is quickly picked up by users, and which must be observed in all extensions and/or modifications made to the language.

Syntactic patterns have the same effect. For instance, if parameters are passed in the syntactic form:

```
<procedure_name>'('<parameter_name>{','<parameter_name}*')'
```

it would be surprising and unreasonable if a new release of the programming language passes programs as parameters using the following syntactic structure:

```
<procedure_name>'(goto' <program_name> 'return)'
```

Human pattern-recognition mechanisms are thus the most important resource to be tapped by EUP environment designers, because they are innate to our species. Hence the importance of semiotic design patterns that apply to all communicative systems a computer user encounters in software applications.

So, the *Semiotic Continuum Principle* states that, given two computer languages Li and Lj, they are said to be semiotically continuous if:

- Li is an Interpretive Abstraction of Lj;
- Lj is itself an Interpretive Abstraction of some other language(s);
- Lj has a syntactic instance of **<text>**, as a specific structured organization of sentences (whose meaning incorporates intentional elements); and
- any user who knows Li and Lj can **always** translate an arbitrary instance of Lj **<text>** into Li.

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 or she can **always** translate any arbitrary **text** of the lower-level language into a sign of the upper-level one. This can only be achieved because of a pragmatic component that is projected onto syntactic structures. 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.

From a different perspective, we can see that what the *Semiotic Continuum Principle* is trying to achieve is to rule out EUPL candidates that cannot distinguish pragmatically inadequate sign combinations from pragmatically adequate ones. This tight coupling between EUPL and UIL is precisely the feature that guarantees the correspondence between well-formed EUPL and UIL texts. For a concrete example of this principle's effect, we can show a simple evidence drawn from Microsoft's Word 7.0 environment.

In Figure 6, we see that the original menu configuration of this application (see the left-hand side of the picture) suggests that users can save their files in some specific format, namely HTML. If they choose to do so, there is a follow-up dialog window (not shown in this figure) in which the user can specify the desired path and file name for the current document. A quite plausible extension of the application's functionality would be to save files in RTF format (which is interpreted by a number of other brand 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 a macro recorder is used to produce the code for the new function *Save as RTF…* the generated code can be the following:

```
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
```

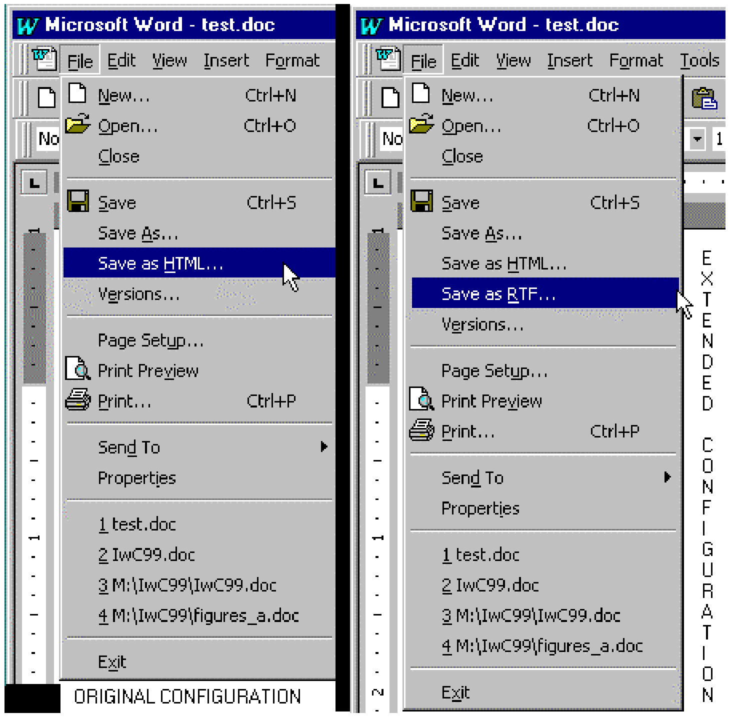

Figure 6: Extending MSWord 7.0

Note that the file name has been captured as a constant value ("test.rtf", in this case), and not a variable. Although macro recorder users may not notice this before too late (when they discover that different files are being recorded under the same name if in RTF format!), there is an important inconsistency between the UIL and the EUPL text. What is suggested by the sequence of steps in the UIL is indeed that only the type of the file is being changed (not its path and name). The above macro was generated by the following sequence of steps:

I. Click on File on the menubar;
II. Position cursor on Save As… on the pulldown menu;
III. In dialogue box, roll Save as type list of formats till Rich Text Format (*.rtf) is found;
IV. Depress mouse button and click on the Save button.

The user **never** provided the file name and path in the above dialogue, so there is no reason to believe that the macro should capture input that the user has not specified. The coded generated by macro recording is **not** an adequate semantic description of the interactive discourse produced by the user at the interface level. An acceptable semantic description of the interactive steps mentioned above would be:

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

The *"*.rtf"* sign in the EUPL code would be semiotically continuous to the signs the user encounters in the interface dialogs. Implementation intricacies of Visual Basic do not allow for this sort of coding, and demand that users really know an enormous lot of the EUPL to be able to program how to capture the name and path of the current file being edited. So, the translation from EUPL into UIL is flawed in this case.

However, other sorts of problems may occur. For instance, if the user chooses to use the macro editor instead of the recorded (supposing that he or she knows the EUPL 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 Word 7.0 (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 or she is accustomed to, but also the effect of this macro on the document may be null[5].

Here, another dimension of the *Semiotic Continuum Principle* can be used to rule out this kind of code. We will suppose that the grammatical rules that operate for the VBA macros above are roughly the following:

Vocabulary (excerpts):

ActiveDocument.SaveAs: task_object, type:file
FileFormat: parameter,type:file,subtype:{wdformatRTF,wdformatDOT,…}
wdformatRtf: value,subtype:wdformatRTF
SaveAsAOCELetter: <parameter>,type:file;subtype:{true,false}
False: value,subtype:false

Generative Rules (simplified excerpts):

<macro> → 'Sub', <label>, '( )' , <body>, 'End Sub'
<body> → <task_object[type:t]>, {<parameter[type:t, subtype:st]>, ':=' ,
<value[subtype:st]>}$^+$
<body> → {<parameter[type:t, subtype:st]>, '=', <value[subtype:st]>}$^+$
<parameters> → <label>, {', '<label>}* | ε

Given the requirement that **texts** in the EUPL must be translatable into UIL actual or potential signs, we should rewrite the above grammatical excerpts so that they reflect, syntactically, the continuum between the two languages (the EUPL as the source, and the UIL as the target). The following set of rules illustrates the kind of phenomena that could be captured:

Generative Rules (simplified excerpts):

<macro_text> → 'Sub', <label>, '( )' , <body>, 'End Sub'
<body> → <task_object[type:t, dialog:d]>,
{<dialog_parameter[type:t, subtype:st, dialog:d]>,
':=' ,
<value[subtype:st, dialog:d]>}$^+$

Basically, the invariant feature **dialog:d** is binding all the syntactic elements of a macro_text together, and representing the cohesion and consistency of the generated code. In Figure 7 we see the corresponding dialog for this example. In it there are a number of widgets that must be marked for co-occurrence in this context. So, the *Save* button would have the same dialog feature

[5] Running this macro on a PC causes no tangible effect on the document or the application.

value *d_save_as* as the *Cancel* button or the *Save in:* pulldown list. Thus, an enhanced new <macro_text> structure code for the Save_as_RTF, given the dialog in Figure 7, could be:

```
Sub Save_as_RTF( )
ActiveDocument.SaveAs FileName:="*.rtf", FileFormat:=wdFormatRTF, _
     SaveVersion:="No_change", Options:="No_change"
End Sub
```

In it, the textual cohesion control feature "dialog" and its constant value *d_save_as* would be specified in the Vocabulary of the language, in the following illustrative way:

Vocabulary (excerpts):

ActiveDocument.SaveAs: task_object, type:file, dialog:d_save_as
FileFormat: dialog_parameter,type:file,subtype:{doc,rtf,dot,txt,…,html}, dialog:d_save_as
*.Rtf: value, subtype:rtf, dialog:d_save_as
wdformatRtf: value, subtype:wdformatRTF, dialog:d_save_as
SaveVersion: <dialog_parameter>,type:file;subtype:{Cahnge,No_change}, dialog:d_save_as
Options: <dialog_parameter>,type:file;subtype:{Change,No_change}, dialog:d_save_as
No_change: value, subtype:no_change, dialog:d_save_as

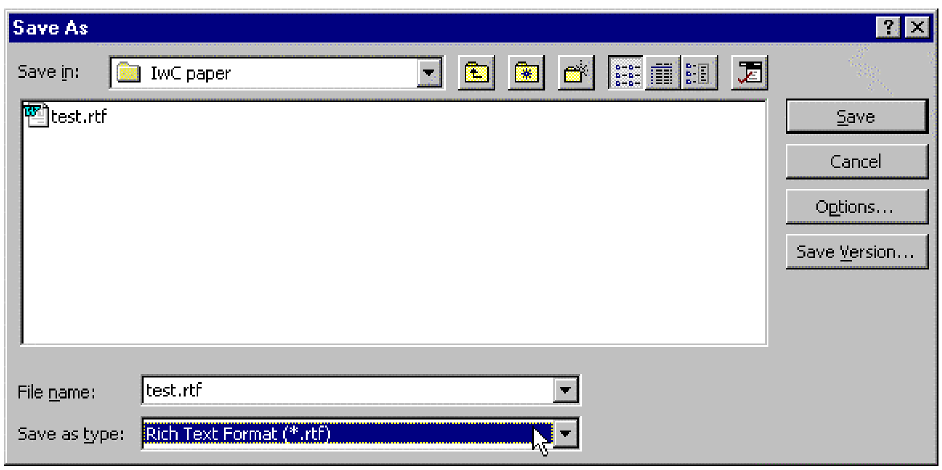

Figure 7: the UIL base for a semiotically continuous EUPL code

The <macro_text> structure code results from the four steps listed at the beginning of this example. Thus, the *"*.rtf"* sign stands for the case where the name of the file is **inferred** by the application, and not typed in by the user. The *"No_change"* sign stands for unchanged settings in subdialogs engaged if the user presses either the Options… button or the Save Version… button on the left. If these buttons are pressed, the <macro_text> structure is extended with the changes that are signaled and achieved there. In addition, the fact that the EUPL is itself an interpretive abstraction of some other language (the application's programming language) allows us to predict that the *"*"* sign can be correctly interpreted by the EUPL, as well as the <macro_text> constructs themselves.

Discussion

Software evaluation methods and techniques have been classified into five groups [Preece et al.'94]: user observation; questionnaires; experiments; interpretive evaluation; and predictive evaluation. Depending on whether designers are engineering towards a target, trying to understand reality, comparing design alternatives, or verifying conformity to standards, 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 the product), 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, as we have seen, a semiotic (communicative and linguistic) system. Thus, the quantitative metrics required to perform this kind of evaluation should assess the languages in which users express and apply 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, without an objective and reliable representation of the goal against which success or failure must be determined, quantitative metrics lose value. 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.

The meaning of a new need and feature is in the mind of the end user that extends the application under pragmatic conditions that have not been anticipated by the original application's designer. Consequently, such metrics as semantic and articulatory distances [Hutchins et al.'86] 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.

But if evaluation is done to understand reality, then EUP environments not only can but also 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.

A comparison between concurrent EUP design alternatives requires an underlying model of EUP, against which the variations in performance can 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. The most adequate one is certainly that of understanding reality, for which some techniques have been proposed. Preece and her colleagues propose that observing users, applying questionnares and interviewing, as well as performing interpretive analysis, are alternative ways to reach for this goal.

One of the characteristics of observing phenomena is that inevitably the observer is impregnated with what he or she wants to observe. Nevertheless, logging software usage, using video and audio tapes, registering verbal protocols, all of these techniques collect data that can not only reinforce and discard hypotheses, but also 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. However, interpretive evaluation methods seem to complement the former with the perspective that EUP is a highly subjective activity (if not totally subjective, considering the creative processes that it strives to support). It can help designers see the nature of EUP as dealing with the unpredictable and the undefined, for which no engineering method in a strict sense can be applied.

Other 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. For instance, popular heuristic evaluation methods deal with design principles (taken from [Preece et al.'94]) such as:

- Use simple and natural dialogue
- Speak the user language
- Minimize the user memory load
- Be consistent
- Provide feedback
- Provide clearly marked exits
- Provide shortcuts
- Provide good error messages
- Prevent errors

Examples presented in this paper show that these principles apply to EUP as well as to non-EUP applications. Nevertheless, they fail to show that the original application's designer must also design the meta-language for his or her design principles in such a way that a user can understand it and use it. So, the above principles must be applied to themselves, and be stated recursively, like in the following few illustrative instances:

- Use simple and natural dialogue about using simple and natural dialogue
- Use simple and natural dialogue about speaking the user language
- Use simple and natural dialogue about providing feedback
- Use simple and natural dialogue about providing clearly marked exits
- Provide feedback about using simple and natural dialogue
- Provide feedback about providing feedback
- Prevent errors when trying to prevent errors

The semiotic engineering principles proposed in this paper are one step in the direction of modeling EUP. The availability of a model, even if preliminary, strengthens the power of evaluation and helps demarcate the limitations of methods, techniques, and models, themselves. We have shown that, in view of some typical EUP situations, these principles can characterize the source of problems and inform the expected solutions.

The *Interpretive Abstractions* and *Semiotic Continuum* principles can also be used to classify different types of languages (or semiotic systems, in more precise terms) that can be encountered in EUP applications. The implicit model in these principles is one in which there are two related languages in every extensible application: the UIL and the EUPL. If the UIL is an interpretive abstraction of the EUPL, and the UIL and EUPL are semiotically continuous, then we have a qualified EUP environment. We cannot say, without empirical evidence, that this qualification is sufficient to guarantee EUP usability and applicability. However, it is certainly distinctive of more and less semiotically complex environments. Semiotically complex ones require from users longer abductive chains in generating successful interpretations for signs. This is a cognitive indication of the complexity of the environment, but a psychological account of EUP falls outside the scope of this paper. If the adequacy of this qualification 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.

Nevertheless, the surveyed EUP languages are not semiotically continuous with the UI languages of their respective applications. The two principles can, in this case, characterize variants of EUPL. In Table I we summarize the different consequences of a semiotic discontinuity between UIL and EUPL, and thus identify four of these variants. Note that there are dependencies between the conditions on the first column. For example, if the EUPL cannot provide a semantic description of the UIL, then it doesn't make sense to try to translate EUPL text into the UIL. Also, without a syntactic marker for text, the last condition is not applicable.

I	**The UIL is not an Interpretive Abstraction of the EUPL.**	• If the semantics of UIL cannot be described in EUPL, then a consistent Macro Recording/Editing mechanism cannot be

		designed and the expression of EUPL texts in the UIL is not guaranteed. • If a user of UIL cannot make satisfactory sense of **all** lexical and phrasal UIL signs based on pattern recognition, UIL documentation and his or her own computer literacy, then the user cannot design extensions to the UIL in the first place.
II	**The EUPL is not itself an Interpretive Abstraction of some other language(s).**	• If a user of EUPL cannot make satisfactory sense of **all** lexical and phrasal EUPL signs based on pattern recognition, EUPL documentation and his or her own computer literacy, then the user cannot express himself or herself in EUPL.
III	**The EUPL does not have a syntactic instance of <text>, as a specific structured organization of sentences.**	• Without a syntactically distinct construct of a higher-order than that equivalent to an executable instruction, block or program, the user is prone to generate executable code that is disconnected from the interface module (and thus, unusable).
IV	**Even if a user knows the UIL and the EUPL, he or she cannot always translate an arbitrary instance of EUPL <text> into UIL.**	• There are texts in EUPL that cannot be expressed by the UIL, and therefore require that the user designs another interface language (not a just a derivation of the existing one).

Table I: A characterization of EUP language types

Table I also supports the conclusion that 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, as we have argued previously. Thus, the ability to generate novel and distinct single-purpose languages cannot be achieved by a single-purpose metalanguage. Only a programming language can achieve this goal. We can see that this differentiation leads to a more accurate elaboration of the goals stated by Fischer, for example [Fischer'98]. 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 an indisputable 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 that unique application language. However, software documentation [Theodos'91, Carey et al.'92] and design documentation [Moran and Carroll'96] 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 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'90]. However, if the granularity is not the same, the length and the nature of the reasoning process 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 goals can be achieved by steps of actions.

In sum, 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. This should facilitate the definition and interpretation of empirical studies, and point at directions in which semiotic engineering should itself be improved to provide more adequate accounts of end user programming activities.

Acknowledgment

The author would like to thank her colleagues Raquel Prates and Simone Barbosa for their substantial contributions to this work. She would also like to thank Tom Carey and two anonymous referees for insightful feedback about the first version of this paper. Last, but not least, the author expresses her gratitude to the members of the Semiotic Engineering Research Group, in PUC-Rio, for their constant enthusiasm and incredible intellectual energy. This research project is partially funded by CNPq, the Brazilian council for the development of science and technology.

References

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

[Andersen et al.'93] Andersen, P.B.; Holmqvist, B.; Jensen, F.F. (eds.) *The Computer as Medium.* Cambridge University Press. 1993.

[Apple'98] Apple Computer. *Hypercard.* Software Documentation. 1998

[Barbosa etal.'99] Barbosa, S.D.J.; da Silva, S.R.P.; de Souza, C.S. *"Extensible Software Applications as a Semiotic Engineering Laboratory"*. To be published in *Working Papers in the Semiotic Sciences*. Legas, Ottawa, Canada. 1999.

[Brown'95] Brown,G. *Speakers, listeners and communication*. Cambridge. Cambridge University Press. 1995

[Carey et al.'92] Carey,T.;Nonnecke,B.; and Mitterer, J. *"Prospects for Active Help in OnLine Documentation"*. In Proceedings of the *ACM Tenth International Conference on Systems Documentation* 1992 p.289-296

[Chang'90] Chang,S.K. *Visual languages and visual programming*. New York. Plenum Press.1990

[Cordy'92] Cordy, J. "*Why the user interface is NOT the programming language: and how it can be"*. in Myers,B.A. (Ed.) *Languages for developing user interfaces*. Boston. Jones and Bartlett Publishers. 1992

[Cypher and Smith'95] Cypher, A. and Smith, D.C. *"KidSim: End User Programming of Simulations"*. In Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems 1995 v.1 p.27-34

[Cypher'93a] Cypher, A. *"Eager: Programming Repetitive Tasks by Demonstration"*. In Cypher, A. (ed.) *Watch What I Do: Programming by Demonstration*.

[Cypher'93b] Cypher, A. *"The Practical Use of Macro Recording: A Case Study Customizing Interfaces"*. In Proceedings of the East-West International Conference on Human-Computer Interaction: Proceedings of the EWHCI'93 1993 v.2 p.203-210

[Cypher'93c] Cypher, A. (ed.) *Watch What I Do: Programming by Demonstration*. The MIT Press. Cambridge MA. 1993.

[de Souza'93] de Souza, C.S. *"The Semiotic Engineering of User Interface Languages"*. *International Journal of Man-Machine Studies*. No. 39. pp. 753-773. 1993.

[Dertouzos'92] Dertouzos,M. *"The user interface is the language"*. in Myers,B.A. (Ed.) *Languages for developing user interfaces*. Boston. Jones and Bartlett Publishers. 1992

[DiGiano and Eisenberg'95] DiGiano, C. e Eisenberg, M. *"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. 1995.

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

[Eisenberg and Fischer'94] Eisenberg, M. and Fischer,G. *"Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance".* In Proceedings of CHI'95 ACM Conference on Human Factors in Computing. ACM Press. Pp. 431-437

[Eisenberg'95] Eisenberg, M. "Programmable Applications: Interpreter Meets Interface". *SIGCHI Bulletin*. Apr. Vol. 27(2), ACM Press. 1995.

[Fischer'93] Fischer, G. *"Beyond Human Computer Interaction: Designing Useful and Usable Computational Environments"*. In Proceedings of the HCI'93 Conference on People and Computers VIII 1993 p.17-31

[Fischer'98] Fischer, G. *"Beyond 'Couch Potatoes': From Consumers to Designers"*. In Proceedings of The 5th Asia Pacific Computer Human Interaction Conference. IEEE Computer Society. 1998. pp.2-9.

[Gamma et et al.'95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

[Gudwin and Gomide'97] Gudwin, R.R. and Gomide, F.A.C. *"An Approach to Computational Semiotics*". In Proceedings of ISAS'97. NIST. Gaithersburg, Md. pp. 467-470.

[Hoopes'91] Hoopes, J. (ed.) *Peirce on Signs*. The University of North Carolina Press. Chapel Hill. 1991.

[Hutchins et al.'86] Hutchins, E.L., Hollan, J.D. and Norman, D.A. *"Direct Manipulation Interfaces*. In D.A.Norman, S.W.Draper (eds.) *User Centered System Design, New Perspectives on Human-Computer Interaction.* Lawrence Erlbaum Associates. Hillsdale, NJ. 1986. pp. 87-124

[Jorna and van Heusden'96] Jorna, R. & van Heusden, B. *"Semiotics of the user interface"*, in *Semiotica* 109 - 3/4, 237-250, 1996.

[Leite'98] Leite, J.C. *"Modelos e Formalismos para a Engenharia Semiótica de Interfaces de Usuário."* Ph.D. Dissertation. Departamento de Informática. PUC-Rio. Rio de Janeiro. October, 1998.

[Lieberman'93a] Lieberman, H. *"Mondrian: A Teachable Graphical Editor"*. In Cypher, A. et al. (eds.) *Watch What I Do: Programming by Demonstration*.

[Lieberman'93b] Lieberman, H. *"Tinker: A Programming by Demonstration System for Beginning Programmers"*. In Cypher, A. et al. (eds.) *Watch What I Do: Programming by Demonstration*.

[Lotus'95] Lotus Development Corporation®. *Smart Suite 96*. Software Documentation. 1995

[Malone et al.'95] Malone, T.W., Lai, K. and Fry, C. *"Experiments with Oval: A Radically Tailorable Tool for Cooperative Work". In ACM Transactions on Information Systems*. Vol. 13, No. 2, April 1995. pp.177-205.

[Meystel'96] Meystel, A.M. *"Intelligent Systems: A Semiotic Perspective". International Journal of Intelligent Control Systems. Vol.1, No. 1. March 1996.*

[Microsoft'96] Microsoft Corporation. *Microsoft® Office 97.* Software Documentation. 1996.

[Moran and Carroll'96] Moran,T.P. and Carroll,J.M *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates. Mahwa,NJ. 1996

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

[Myers'92] Myers, B.A. *Languages for Developing User Interfaces*. London. Jones and Bartlett Publishers, Inc. Boston. 1992.

[Nadin'88] Nadin, M. *"Interface Design and Evaluation – Semiotic Implications"*. In Hartson, R. e Hix, D. (eds.), *Advances in Human-Computer Interaction*, Volume 2, 45-100. 1988.

[Nardi and Miller'90] Nardi, B. and Miller, J. R. *"The Spreadsheet Interface: A Basis for End User Programming Applications and Case Studies"*. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction* 1990 p.977-983

[Nardi'93] Nardi, B. *A Small Matter of Programming*. The MIT Press. Cambridge, MA. 1993.

[Norman'86] Norman, D.A. *Cognitive Engineering*. In Norman, D.A. e Draper, S. (eds.) *User-Centered Systems Design*. Lawrence Erlbaum and Associates. Hillsdale, NJ. pp.31-61. 1986.

[Peirce'31-58] Peirce, C.S. *Collected Papers*. Cambridge, Ma. Harvard University Press. 1931.

[Pimenta and Faust'97] Pimenta, M. & Faust, R. 1997. *"Eliciting Interactive Systems Requirements in a Language Centred User-Designer Collaboration: a semiotic approach"*, in SIGCHI Bulletin, v.29, n.1, 1997.

[Preece et al.'94] Preece, J.; Rogers, Y.; Sharp, H.; Benyon, D.; Holland, S.; and Carey, T. *Human-Computer Interaction* 1994 Reading, Mass. Addison-Wesley Publishing

[Repenning and Ambach'97] Repenning, A. and Ambach, J. *"The Agentsheets Behavior Exchange: Supporting Social Behavior Processing"* DEMONSTRATIONS: "Programming with Less Programming". In Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems 1997 v.2 p.26-27

[Repenning'94] Repenning, A. *"Programming Substrates to Create Interactive Learning Environments".* In Journal of Interactive Learning Environments. Special Issue on End-User Environments. 1994. No. 4. Pp. 45-74

[Rheinfrank and Eveson'96] Rheinfrank, J. and Eveson, S. *"Design Languages"*. In T.A. Winograd (ed.) *Bringing Design to Software*. New York. ACM Press (Addison-Wesley). 1996. pp.63-80.

[Sebeok'94] Sebeok,T.A. *Signs: An introduction to Semiotics*. Toronto. University of Toronto Press. 1994

[Theodos'91] Theodos, R.D. *"Text in Context: Writing Online Documentation for the Workplace".* In *Proceedings* of the Ninth ACM International Conference on Systems Documentation 1991 p.140-148

[Traynor and William'97] Traynor, C. and Williams, M.G. *"A Study of End-User Programming for Geographic Information Systems"*. In *Empirical Studies of Programmers: Seventh Workshop* . 1997 p.140-156