Communicating Different Perspectives on Extensible Software

Sérgio Roberto P. da Silva Simone D.J. Barbosa Clarisse Sieckenius de Souza
srsilva,sim,clarisse@inf.puc-rio.br

Departamento de Informática PUC-Rio
R. Marquês de S. Vicente 225
Rio de Janeiro,RJ

Communicating Different Perspectives on Extensible Software

Abstract

Actual End-User Programming is as far away from us as is an actual End-User Perception of the rationale beneath software design. One of the major obstacles to achieving this goal is the discontinuity among user interface language, end-user programming language, and documentation language. Objects referred in one of them are not easily identifiable as the same referred to by the others. In this paper, we analyze this situation from a semiotic perspective and propose interconnected environment architecture and language design guidelines that we believe can narrow the gap there is to bridge.

1. Introduction

The main theme of this paper is software that can be extended by end-users, i.e., applications that allow end-users to program. When discussing end-user programming, a few questions come to mind: Why should users want to program? Can they learn how to program? If so, what makes it so hard on them? [Myers '92].

The software industry has been increasing the functionality of general-purpose software, striving to satisfy the majority of users' interests. Unfortunately, this solution to users' problems brings about two major drawbacks: cluttering in the application's interface and consequently increase in the *cognitive load* on users; and the inability to implement a priori solutions to every specific user's problem.

When the user gets to know the application well enough to recognize that his problem cannot be directly solved, he will feel the need to change the software in order to carry out the tasks that are relevant for an efficient completion of his work. Additionally, many business-specific repetitive tasks can and should be automated, that cannot be anticipated by software designers [Cypher '93].

Many applications nowadays offer some means of customization or programming, in order to satisfy such needs, which brings about our second question: can users learn how to program? Many argue that users are not able to deal with formal languages at all, but according to [Nardi '93], this is a fallacy. People are used to working with formal languages, as is the case in maths, game scoresheets, and even knitting.

We now come to our third question: Why do users have such a hard time learning how to extend their applications? Nardi '93 argues that the problem is the programming languages offered to end-users. She claims that the ideal programming language should be task-specific, in order to motivate and interest users so they will want to learn more. In our view, another severe problem is a lack of co-referentiality [Draper '86] between the many languages the user must deal with. This discontinuity between interface, programming and documentation languages in such applications hampers the interaction, because the software conveys similar concepts in entirely different languages.

We thus analyze such problems in a communicative perspective and propose an architecture for a supportive environment that facilitates end-user programming. We also propose design guidelines for an end-user programming language breed that is connected to the architecture.

2. Communication-Centered Applications

A user does not start an application by wanting to extend it. At first he needs to learn how to use it, and only later on will he realize that it does not do all the things he needs it to do. Some applications offer a “record macro” feature, with which it is hoped that the user merely needs to group some sequentially executed tasks under one “name”. But, this is not enough, since macro recording usually does not allow for any abstractions and conditionals. Therefore it is suited only for repeating the same exact task, without changing any “parameters”. For this, the user must edit the macro.

There is an unmeasurable gap between macro recording and editing. For one, the macro code is usually textual, and therefore entirely different from the familiar interface icons and menus the user interacted with in order to record it. There is not a consistent (isomorphic) mapping between the user interface (UI) elements and the end-user programming (EUP) elements, so that the user can bridge the gap by making the appropriate associations. Therefore, instead of focusing on the solution of his problem, the user has to learn and understand a whole new language (which is a new problem).

As mentioned above, most of the problems that arise from end-user programming within an extensible application are due to a discontinuity between its interface and programming environments. These communicative aspects of human–computer interaction can be studied under the light of Computer Semiotics [Andersen ’90, Andersen ’93, Nadin ’88], or more specifically, Semiotic Engineering [de Souza ‘93, de Souza ‘96].

Treating software as a *communication* artifact could bring some new insights into the realm of the duality between the designer and the end-user. Roughly speaking, a communication system is used to deliver a *message* from a *sender* to a *receiver* through a *medium*. A medium may only accept messages in some specific range of forms, so the sender must *code* the message within this range. The code is used to convey the sender’s *intentions* and *meanings* to the receiver. In order to express this, the sender selects a set of possible *signs* in the medium *substance* and attributes them some *signification* hoping that the receiver will *interpret* this signs in the same way.

In order to gain a better understanding of interpretive processes that happen at the edge of a communication system we make use of semiotics, which studies communication and signification systems. In semiotics, a *sign* (i.e. that which represents something) is related to both an *object* (i.e. the thing it stands for) and an *interpretant* (i.e. a feeling, an action or another sign) in a triadic schema [Peirce ’31]. The process of interpreting a sign is called *semiosis*. However, the sign may be interpreted in indefinitely many layers of meaning bringing up a variety of other signs and meanings to mind. This process of generating a chain of meanings is called *unlimited semiosis*, another powerful notion in our context because of its unbounded nature and critical consequences for communication.

The gist of this semiotic background is that when two individuals communicate with each other they negotiate and regulate meanings in conversation, so that unlimited semiosis becomes pragmatically constrained to a territory of mutual understanding. Communication is apparently successful at this level because somehow these people’s interpretants converge to a stable configuration of understanding in both minds. When there are evidences that such convergence has not been met, people engage into using language to regulate the meaning of language—a *metalinguistic* behavior.

2.1. Example of End-User's Problems in Extending Applications

In this section we will, by means of an example, first explore some major problems end-users face when extending an application, and then show how the above concepts can help us understand these problems. To this end we will make use of a well known word processor, Microsoft Word©, and its macro language, WordBasic©.

Suppose we, as advanced end-users, playing the role of novice designers, want to write a macro that helps someone in the process of program documentation. We would like to change the color of the comments in a computer program text. Our macro specification has two parts:

1. The user input: the end-user's selection of the color for comments.
2. The document modification: the macro code that will modify the document according to the user's choices, looking for expressions starting with " ' " (single quote) outside a string, i.e., not surrounded by double quotes.

First we, as designers, *decide* to implement the "user input" by means of a dialog box containing a drop-down list box for the available colors, from which the end-user selects his choice. Hence, we have as *message senders* decided, among many other possibilities, the kind of *message* our system will pass on to the *message receiver*—the end-user. Briefly, we can paraphrase this message as: "Here are all the colors I can apply to your comments. Please pick ONE up." Based upon our familiarity with some of the application's interface elements, we would like to build the dialog box illustrated in Figure 1:

Figure 1 — Desired dialog box for user input.

Notice that this application-standard color drop-down list box presents a color sample and the corresponding name, for each color. This element is available in the "Format Font" dialog box, and we would like to make it available exactly as presented in that dialog box.

In order to *code* our message we can use an interface construction software, Microsoft Word's Dialog Editor©. The resulting dialog box is shown in Figure 2.

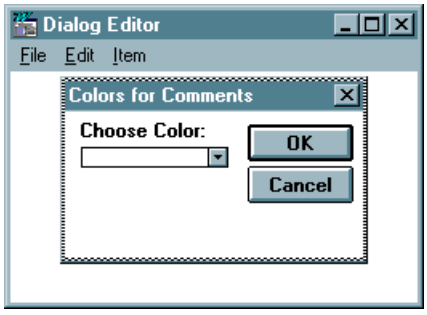

Figure 2 — Dialog editor session to create user input dialog box.

However, this standard drop-down list box—the only one the Dialog Editor allows us to create—has an associated *array of string* values (`DropListBoxColor$()`). There is no way to

provide a color sample, and in order to present a list of color names, we must insert it one by one into the associated array. So much for our friendly, familiar drop-down list box. We notice therefore that:

- There is a serious case of discontinuity between the UIL and the EUPL. There is an element available in the UI, but we can neither use it by making a reference to it nor access it using the EUPL. Besides, there is absolutely no way in the EUPL to code—even from scratch—an equivalent element. And notice we did not even want to change the element in any way; just use it, exactly as it is presented in the UI.

Our next step is to link the dialog box created by the Dialog Editor with the macro code that will execute the required function. Oddly enough, the macro code must include a textual description of the dialog box, instead of a reference to it, which is a full fledged bi-representationality of items with no apparent link between representations. The code corresponding to the above dialog is as follows:

```
Begin Dialog UserDialog 248, 86, "Colors for Comments"
    OKButton 152, 7, 88, 21
    Text 12, 5, 107, 13, "Choose Color:", .Text2
    DropListBox 13, 20, 123, 108, DropListBoxColor$(), .DropListBoxColor
    CancelButton 152, 31, 88, 21
End Dialog
```

- We need to use two entirely different languages—graphical and textual—for representing the same *object*, the dialog box.

Continuing our linking task we must now associate each element in the dialog box to a color data structure or to a function. We would like to associate the color drop-down list box to the list of available colors. But, when looking for “color” on the on-line help provided by the application, the corresponding help text informs us that we have 17 color *numeric codes* available, from 0 to 16. Thus we need to manually associate each element in the string array to the corresponding color:

```
'available colors
DropListBoxColor$(0) = "Auto"
DropListBoxColor$(1) = "Black"
DropListBoxColor$(2) = "Blue"
...
DropListBoxColor$(16) = "Light Gray"
```

- We need to *unfold* a structure that could be treated as a unit in the domain. This unfolding is not imagined a priori by an end-user since for him the UI element is a unit, and not an array of elements. Such difference between the textual and graphical versions of the same element presents a clear semiotic discontinuity.

We can now “show” our dialog box. After some help digging, we come up with the following code fragment:

```
x = Dialog(dlg) ‘this shows the dialog box and waits for an OK or cancel
If x = - 1 Then'the ok button has been pressed
    colorValue = DlgValue("DropListBoxColor")
    ‘here we add the actual functionality
          ‘--> change the text color for comments
Else ‘the cancel button or Esc key has been pressed
    MsgBox "Operation cancelled by user!"
End If
```

Up to now, all our code does is provide lists of color names, and retrieve the chosen value (part 1 of our macro specification). We have not yet programmed the actual changes to the document. At this moment we should proceed to implementing part 2 of the macro specification.

In order to change the color of comments, we would search for single quote and verify if it is not inside a string. In order to search for the first occurrence of this string and modify the text

color (without any other verification), we could use the “record macro” feature, yielding the following code fragment (comments were added later):

```
‘go to start of document
StartOfDocument
‘------------------------------
‘changing the comment text color
‘------------------------------
‘search for single quote
EditFind .Find = "'", .Direction = 0, .MatchCase = 0, .WholeWord = 0, .PatternMatch = 0,
.SoundsLike = 0, .Format = 0, .Wrap = 2, .FindAllWordForms = 0
‘select until the end of the line
EndOfLine 1
‘change the comment text color
FormatFont .Points = "11", .Underline = 0, .Color = 2, .Strikethrough = 0, .Superscript =
0, .Subscript = 0, .Hidden = 0, .SmallCaps = 0, .AllCaps = 0, .Spacing = "0 pt",
.Position = "0 pt", .Kerning = 0, .KerningMin = "", .Tab = "0", .Font = "Book Antiqua",
.Bold = 0, .Italic = 0, .Outline = 0, .Shadow = 0
```

We must now *manually* edit the macro to introduce the *conditions* for code execution. In order to verify if a quote in a certain line is the start of a comment, we must count the number of double quotes to the left of it. If it is odd, then the quote is part of the string, otherwise it is a comment.

```
x = Dialog(dlg)
If x = - 1 Then'the ok button
    colorValue = DlgValue("DropListBoxColor")
    'clear formatting info for search
    EditFindClearFormatting

    'search for single quote
    StartOfDocument
    EditFind .Find = "'", .Direction = 0, .MatchCase = 0, .WholeWord = 0,
    .PatternMatch = 0, .SoundsLike = 0, .Format = 0, .Wrap = 2, .FindAllWordForms= 0
    While EditFindFound() = - 1
          'if not inside string, change text color
          StartOfLine 1
          count = 0
          For i = 1 To Len(selection$())
              If Mid$(selection$(), i, 1) = chr$(34) Then
                  count = count + 1
              End If
          Next i
          If (count Mod 2 = 0) Then
          'yes, it is a comment
              'select until end of line
              EndOfLine 1
              'change text color
              FormatFont .Color = colorValue
              CharRight'position cursor for next find
          Else
              CharRight 2'position cursor for next find
          End If
          RepeatFind
    Wend
Else
    MsgBox "Operation cancelled by user!"
End If
```

💣 The macro recording feature is inflexible in the sense that it only makes it possible to record sequences of commands, forbidding any kind of conditionals or abstractions, i.e., it does not generates types in the EUPL, just tokens.

Now we try to execute the macro. To our surprise, we cannot use the function `DlgValue()` to retrieve values from the dialog box after it has been closed (an execution error occurs). Instead, these functions need to be used within a “dialog function”. We thus need to create another function, and trap the events that change the dialog element values, as follows:

```
Function DlgFunction(identifier$, action, suppvalue)
    Select Case action
    Case 2'choosing a command button or changing the value of a dialog
          Select Case identifier$
              Case "DropListBoxColor"
                  colorValue = DlgValue("DropListBoxColor")
              Case Else
          End Select
    Case Else
```

```
    End Select
End Function
```

At last we must associate this dialog function with the actual dialog, changing the dialog box definition to:

```
Begin Dialog UserDialog 306, 98, "Colors for Comments", .DlgFunction
```

This still does not work. The macro trace and variable lookup reveal that the value of `colorValue` is correctly assigned within `DlgFunction`, but is reset to 0 back in `Sub Main`. After some extensive help digging, we find out that, although variables do not usually need to be declared, they only have local scope, and therefore we must declare `colorValue` as "shared" outside all functions and subs, because its value need to be accessed from within more than one subroutine or function:

```
Dim Shared colorValue
```

- The binding between the UI elements and the macro code selector elements is enormously intricate and complex.

All these last changes we have done to the macro code to get it working make extensive use of software documentation. Also, the documentation structure and hypertext access mechanism make it hard to find the necessary information, because they are not really context-sensitive, but sensitive only to lexical items.

- Therefore, there is also a discontinuity between the documentation language (DL) and the other languages present in the application—UIL and EUPL.

Now we are able to run the macro, and it executes correctly. In order to run the macro in a later moment, we have a few options:

1. We can choose Tools/Macro, select the macro name and choose Run.
2. We can add an interface element (toolbar button, menu item, or a keyboard shortcut) and link it to the macro. For that purpose, we must choose Tools/Customize. To add a toolbar button, we must then choose Toolbars, scroll to the "Macro" category, and drag the macro name from the macros list to the location we want the button to appear, then choose a name and an icon from a pre-defined set. In order to add a menu item or a keyboard shortcut, a similar uncomprehensible sequence of actions is needed.

- In order to link the final macro to the UIL we need to comply with an intricate mechanism that is not natural to any end-user. It belongs to the software interface domain.

Throughout this example, we inserted a few comments in our code. Nevertheless, we did not have to.

- The end-user is not prompted to document his code, making it hard for someone else to understand his program, and even for himself, some time later.

In order to write this sample macro, we have resorted to our previous programming knowledge and to the software documentation. And it took us—advanced programmers—hours to get it up and running.

- It is virtually impossible for an end-user to actually succeed in writing this macro, due to the lack of documentation, his inexperience in programming, and the amount of time required to overcome all the obstacles posed by the environment.

2.2. A Semiotic Analysis

Now we can analyze the generated macro code using a communication approach. In this analysis we will also adopt Gelernter and Jagannathan's approach [Gelernter '90] of viewing

the software as a *machine* with two states: passive—the program text, and active—the running program.

First, the macro program—in its passive state—generated in this example could be viewed as a message that goes from the designer to the computer. This message tells us how to make transformations over a text, in the context of program code. It was codified in a programming language that is understandable by a compiler or interpreter. In particular, it is interesting to remark that this language has syntax and semantics that should also be understood by a human being.

The program text can be viewed and used as a message from the original designer to a maintainer—who is another designer. The message being transmitted is the same as that to the computer, but we will have a problem that has not been mentioned yet: The fact that the interpretation of some representation is always *situated* in its cultural and contextual frame [Suchman '87]. This problem could easily explain why we have so much trouble understanding and maintaining someone else's programs.

The program text is used as a documentation for the *design rationale* that is embedded in the program. It is important to notice that software documentation is not limited to the program text. As a program gets larger it is always necessary to use other languages, such as graphical and natural languages, to express the whole software requirements and structure. Thus, we could say the programming language is a subset of an overall documentation language system, comprised of all the languages used in the software documentation.

Second, the macro program—in its active state—running as a new functionality in the application could also be viewed as a message that goes from the macro designer to the end-user. It represents the designer's ideas on how to solve the problem she has *interpreted* the end-user has. It is trying to *communicate* this designer's ideas to the end-user.

In fact, it generates interactions that guide the end-user into writing an elegant program code and the notion of "elegance" is no more than this designer's interpretation of the word. Hence, the running program makes use of an *interaction language* to transmit its message to the end-user, using the computer as a *medium*. This language is a subset of an *user interface language*, a language that encompasses all the messages that could be transmitted by the software. It is important to observe that the running program is really a *metacommunication* artifact, since it is a message that generates messages to the end-user. In order to understand the application, the end-user must understand this "metamessage".

Moreover, these interactions must be coded by the designer in a programming language for them to be understood by the computer. Since the end-user is the receiver of the interface message, again we have problems of interpretation. Hence we must carefully select the signs composing the message in an attempt to crystallize the chain of interpretants and thus avoid the problem of multiple interpretations. The best way to choose these signs is to get them directly from the end-user's work domain, whenever possible.

There are two main interactions in this sample extension. First, the end-user selects the document—program code—where he wants the comments to be highlighted, and second he selects the colors to be used. If something goes wrong in the second interaction and the end-user needs to maintain the macro code, a great problem arises when he—now a re-designer—tries to identify these interactions in the program.

Any interaction has structural and presentational aspects. Structural aspects define the interaction sequence, and presentational aspects define how it will appear in the user interface. As already said, within WordBasic the interaction's presentational aspects must be inserted

directly inside the textual code, instead of just being linked to its structure. This hampers the end-user's understanding of the program since he is used to viewing the presentation code as part of the interface and not as part of the functional description, which includes the actions to be performed. The real problem is that the presentation language is normally graphical and the user interface language is textual, and therefore we must deal with entirely different codes, between which there is no natural mapping.

The discontinuity between languages is not restricted to the relations between the UIL and the EUPL. It also happens between the EUPL and the DL. This discontinuity is the source of the major difficulties an end-user face when starting to program, since the interpretations he has in one semiotic field cannot be mapped onto the other fields.

Summarizing the main problems that end-users find when trying to extend an application are related to three main areas:

1. The inconsistencies throughout the languages.
2. The failure to communicate the domain objects' representations in the EUPL and in the DL to the end-user through the UIL.
3. The need for the user to keep the co-referentiality of languages when extending the application.

3. UIL☯EUPL: An Integrated Communication Environment

In this section our objective is to present an environment's architecture to help reduce the problems called up previously. A major change in the industry can be expected if designers realize that they (and not the system they write) are communicating with users over the interface, and if users, in their turn, realize they are not getting the machine's or the system's message, but the designer's [Myers, Smith & Horn '92]. The users may then be encouraged to use the same resource to write their own messages (to themselves and/or to others, in a collaborative environment [Nardi '93]). This situation is depicted in Figure 3, adapted from [de Souza & Barbosa '96].

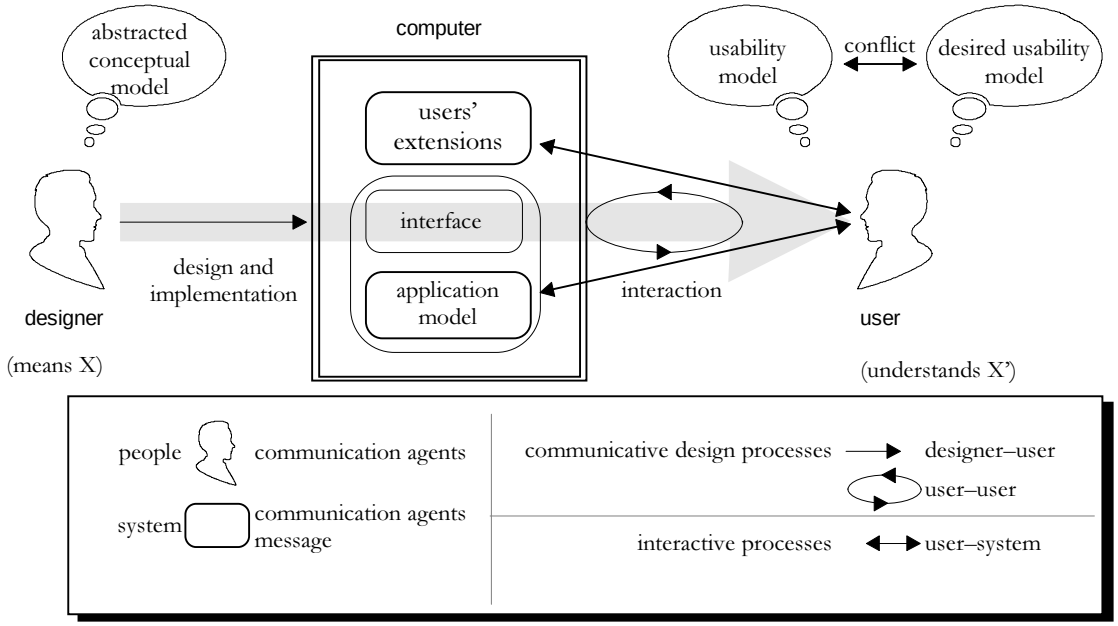

Figure 3 —Semiotic Engineering Framework extended to include End-User Programming

In an environment like that many languages are involved, as shown in the example of section 2.1, namely: user interface, programming and documentation languages. The next subsection will outline an architecture for this environment.

3.1. The Environment Architecture

The many languages used in an application, together with the different ways the objects are dealt with in the domain and within such languages, open possibilities for a number of different interpretations over a single object. Possible misinterpretations are increased because, although the end-user probably knows his work domain, the representations of the domain elements and structure are chosen by the designer, and even after an extensive process of requirements elicitation we cannot guarantee their conceptions (interpretants) about the domain to be entirely consistent with each other. But that is just part of the problem.

We have pointed out in [da Silva '96], that end-users always use a specific language in their work domain. The specificity of this language guides and optimizes the communication. Such characteristics are met by *focusing* and *restricting* the possible interpretations an object presents in this domain, i.e., the language used is *situated* in the work domain context.

This clearly suggests that, as long as the message to be transmitted is the *solution* to a problem that is embedded in the end-user's domain, we could certainly motivate the end-user's interpretation if we embed representations for the *domain objects* in the UIL, thus mapping the *domain language* onto the UIL. To be precise, the domain objects should be represented in all the application languages that we believe are needed to the environment architecture, namely: programming, documentation, and user interface languages.

Thus, making the environment architecture, and the languages that comprise it, *domain dependent* is a basic step to helping reduce the end-user's potential interpretation problems. It is also necessary to make the *design rationale* available to the end-user, since it presents the decisions the designer has made when constructing the software.

Another major source of misconceptions, and one of the main obstacles the end-user encounters when trying to program, is presented in [Barbosa et al. '97] and in our example in section 2: the absence of a consistent mapping between the UIL and the EUPL in the current extensible application. Furthermore, since the end-user may frequently need to build his own dialog boxes, he must be able to use the same elements present in the UIL he is used to interact with. Therefore, we must make available to end-users the elements of the *system software domain* he can interact with.

Moreover, we need a clear, direct and easy mechanism to create the bindings between the UIL, EUPL and DL elements. This mechanism must also allow easy access to the UIL elements from within the EUPL code or DL documents and vice-versa, by means of bi-directional links.

On that account, to guarantee that there will be no semiotic discontinuity between the UIL, EUPL and DL elements, not only the actions in the UIL should be mirrored in the EUPL. Everything treated as a unit in the UIL should also be treated as a unit in the EUPL and DL, and they must present the same levels of unfolding (abstraction) in all languages. Also, in order to promote end-users' understanding of a programming language, we should take advantage of the fact that part of the UIL has become familiar to him through the application use.

All of the above indicates that, in order to reduce the problem of different interpretations between the languages, it is necessary to have a mapping between the objects in each of the

languages. Ideally, this mapping would be an *isomorphism* but such ideal is hardly achieved. Therefore, providing a consistent mapping between the UIL signs and the EUPL and DL signs is another element that will help bridge the gap between these languages, and trigger a similar chain of interpretants (semiosis) in the user's mind.

Revisiting our macro example, if there is such a thing as "list of colors" in the UIL, which is usually system-dependent, there should be a corresponding unit in the EUPL. The user should also be able to *use* the colors drop-down list box as present in the UIL. Besides, if in the UIL, when we choose Format/Font, the color listbox indicates the color of the currently selected text, or the default color if there is no selection, the system could analogously set the "initial" value for the color listbox depending on the context where it was activated within the EUPL. All of these mechanisms should be maintained by the environment architecture.

Another common problem with current programming languages is that they only make possible to follow the flow of abstraction in one direction, from less to more abstract. For an EUPL, this process of *folding* must be mirrored with an *unfolding* process that would allow the end-user to visualize the lower level for a given primitive, as shown in [DiGiano & Eisenberg '95]. From another point of view, this process of revealing the designer's interpretants to the user in a variety of languages may be considered a navigation between *perspectives*, an orthogonal process to the folding and unfolding between *abstractions*. These processes are illustrated in Figure 4:

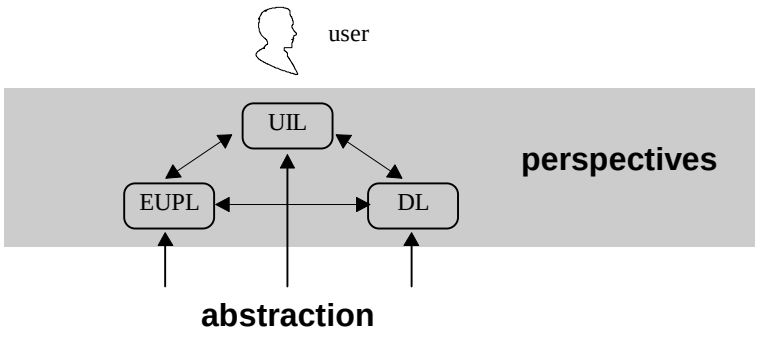

Figure 4 — Traversal through the application languages between perspectives and abstraction levels.

For instance, if we had such an UIL→EUPL mapping, in our macro example above, the end-user would not have difficulties in discovering which EUPL element corresponded to the desired UIL element. And also via the perspective mechanism he could change to the DL, through the UIL→DL mapping, and easily find useful information that could help him correctly use the elements in the first place.

Together with an *explanation mechanism*[1], such abstractions and multiple perspectives tell the end-users about the *designer's rationale* behind each choice she has made when developing the application. These orthogonal processes of navigating through perspectives and folding/unfolding make it possible to create a kind of learning environment [CACM '96], where the end-user could learn how to correctly use a primitive by unfolding it together with its explanation in the context of use, and when this is not enough he could switch his perspective to another language.

This clearly shows that the solution of embedding EUPLs into the application alone is not enough. If the end-user takes the role of a designer, without possessing the designer's kind of knowledge, it would be necessary to make available part of this knowledge from within the

[1] This mechanism will be based in a knowledge application model explained below.

application. The knowledge necessary is not exactly how to program in a given language but how the application was designed, what decisions have been made by the original designer, how the application objects are related, and how the interface objects are linked to the application objects. This knowledge is essential for anyone wanting to correctly understand and modify an application, and even more for an end-user when extending it.

In conclusion, to obtain some continuity between the languages we need to provide a mechanism that maintains an *internal consistency* within the application. This consistency may be achieved by keeping a co-referential [Draper '86] mapping of the domain elements and structure representation throughout those languages. To achieve this mapping we advocate the existence of a knowledge application-specific model (KAM) that will be used as a knowledge base. In it, some environment mechanisms could make operations that will *maintain* and *explain* the decisions and usage the designer has done with the domain objects. This way, by interacting with the application through the end-user interface and an intelligent help system, the user would become acquainted with the designer's interpretants of the domain.

The environment architecture we envisage for truly extensible computer applications is outlined in Figure 5 [de Souza '97]. This architecture views the application as composed of a *runtime consistency checker* which makes use of a *Program and Data Base* (P&DB), where end users' programmed extensions are stored, and an *intelligent reasoning system*, which makes use of a KAM that is absent in current applications. The multimodal multicode interface supports a Graphical User Interface language with direct manipulation and also a Natural Language Question-Answering mechanism (based on the KAM). When extending the application, the end-user will be prompted to update the KAM, through a Knowledge Representation Language (KRL) available through the user interface, and such update mechanism will be responsible for guaranteeing the co-referentiality between the languages.

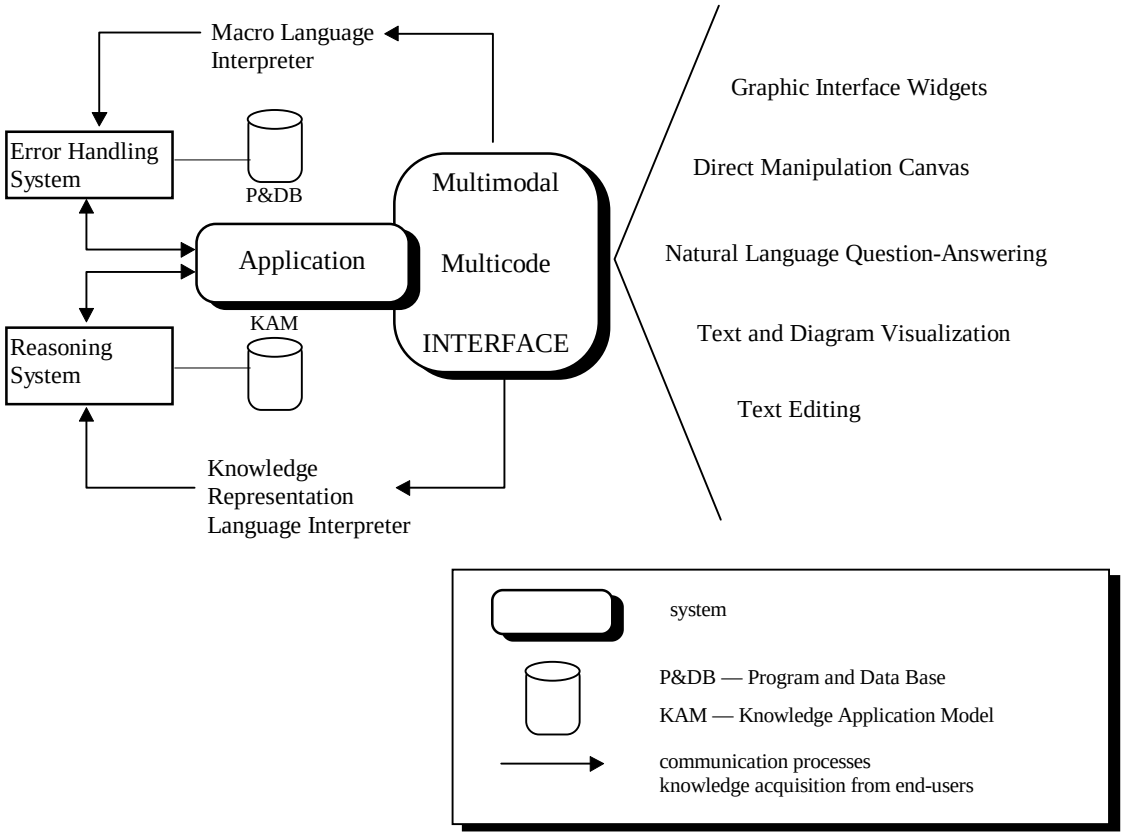

Figure 5 — UIL☯EUPL Communication Architecture

So, in our example, if the application followed this architecture, it would present to the end-user a KAM that, although created by the designer, would reflect the application's usability model as viewed by the end-user (as opposed to the internal application model). Such model will show the already existent objects and relations in a way that the user could understand the restrictions that apply when constructing a new functionality.

In the following subsections we will outline an EUPL based on the needs presented in the preceding analysis.

3.2. The Structure of the Core UIL☯EUPL

Gelernter and Jagganathan propose to view programs as a machine composed mainly of two interlocked and orthogonal components: *space-maps* and *time-maps* [Gelernter '90]. Space-maps are responsible for arranging entities in space and time-maps for arranging them in time. Thus, space-maps describe structures and values to be transformed and time-maps describe the transformations upon those structures.

Such model is very interesting since it resembles very much the way we view objects in real life. The real world objects usually have a composition and a position in the world, which are commonly transformed or manipulated, and consequently having their signification for us changed as well[2].

Another interesting point mentioned in section 3 is the specificity of the languages used by the end-user in a work domain. Such languages create a model, i.e., an ontology for the domain describing all its object types and the relations among them. It is important to notice that,

[2] The position in the world of an object commonly has some *significance* for those who possess that object.

when people work in different domains at the same time, there could be clashes between the signification of a given element, clashes that are commonly resolved by identifying the context in which the element is operating.

Taking into account these remarks we are proposing a new language that combines the UIL and the EUPL in such a way that we could narrow the gap between these two languages the end-user will have to manipulate. Observing the above analysis and the citations of [Dertouzos '90] and [Myers '92] where they mention that the traditional programming commands, such as: decision, repetition, naming, procedure definition and use must be included in a EUPL, we could outline some mechanisms the language must provide:

- mechanisms to define objects' values and types and also to attribute names to these objects so that we could easily manipulate them.
- mechanisms to manipulate the objects, basically altering their value and place in the problem space, i.e., the language has mechanisms to control the order of statement execution in time.
- mechanisms to create procedural abstractions, i.e., it allows end-users to abstract some procedure as a semantic to new lexical element.
- mechanisms to avoid object interpretation clashes within different domains, since end-users use domain-specific languages that may intersect one another.

The last point should be examined carefully. First of all, it is not easy to learn many different languages, one for each different application. To lessen this burden we propose to use a common property of languages following the ideas of space-maps and time-maps. We will initially separate the mechanisms for defining objects and types from the mechanisms that manipulate them. Next, since most of the manipulations are common to almost all sub-languages, we will use a core language for the manipulation of objects and different sub-languages to express the domain model's specificity (object types and their relations). At last, we will define an advanced subset of the language to allow the end-user to define new elements and thus extend the language. So the core language must have the following mechanisms:

- A statement to declare the work domain (lexicon and semantic elements) to be used.
- A statement to declare the names (lexicon elements) of objects, and their types (semantic element), in the space-map.
- A statement to attribute a new value to such objects.
- Statements to control the time-maps over the object's transformations, such as: sequence, selection, and repetition.

Such statements are common to almost any current programming language, so the only thing to be done is to choose them, taking into account the need for easy cognitive assimilation by the end-user. Despite the ease of the language, we must tell the end-user what we understand by a program. For us, a *program* is just a sequence of statements. In fact, the most important function of the core language is to take care of the system's time-maps. A *statement* is a command that changes the state space of the problem in question, and thus any kind of declaration or manipulation is a statement. So, we do not need to impose a strict order between declaration and the other statements (the only constraint is that one object be declared before use, but even this could be relaxed).

As described, the above mechanisms only allow the end-users to operate upon the already defined domain, and not add new elements to it. This set of mechanism will be called the Core

UIL☯EUPL. To allow end-users to extend the domain it will be necessary to use some advanced[3] mechanisms, as will be seen in the next subsection.

3.3. The Structure of the Advanced UIL☯EUPL

Designers should keep in mind the importance of keeping the EUPL as task-specific as possible [Nardi '93]. As such, there is no rule of thumb that will satisfy every application. The domain should drive this design task. What we propose is a *metalanguage* for describing the domain in order to create modules and components. The use of a common core language and domain-specific modules or components has been advocated elsewhere [Eisenberg '95, Dertouzos '92, Smith & Susser '92].

This separation of concepts may be better understood when we take into account the need to maintain the knowledge present in the domain. The introduction of a new procedure in the domain ontology will require that the end-user update the KAM. The proposed advanced statements will not be equal to the statements available in current programming languages. There must be some additional elements to maintain the links with the user interface and the KAM so that the explanation mechanisms could continue to operate well, as will be shown below.

The minimum extension possible is to append a new operation (procedure) to an already defined type of the domain, as it will introduce a new lexicon and a new semantic element into the language. The domain metalanguage would then be used by the end-user to describe new elements in the application's domain. This way the advanced UIL☯EUPL part must have at least the following mechanisms:

- A statement to declare a new procedure.
- A statement to declare a new object type with all its linkages to the interface and explanation mechanisms.
- A mechanism to allow the inheritance from other types.

We must be careful with the pseudo simplicity, since it appears to be present in almost every programming language. One major difference, arising from the explanation linkage, is the need to update the P&DB. An update mechanism should be provided to identify possible clashes that must be resolved before the new element could be accepted in the ontology.

For instance, in our example, when we create the new functionality, we would not only write the procedures and dialogs. We would need to register the new functionality in the P&DB so that the KAM would remain consistent. For that purpose we would use a KRL in which we would define the links between the UIL dialogs, the EUPL procedures and some DL documentation to explain the new functionality.

Another important point is that the language only allows the end-user to construct new elements from already existing ones. Although this restriction may seem very inflexible, it is necessary in order to avoid the creation of a totally new ontology for which the end-user does not have enough knowledge. This is exactly the cutting edge between the designer and the end-user as a programmer.

[3] Advanced from the end-user's point of view.

4. Conclusions

A semiotic approach to the design of extensible applications allows us to analyze the problem of communicating the designer's solution to—what she supposes is—the user's problems, and the tools the designer made available to the user to solve idiosyncratic problems left unsolved. We have seen this communication involves three different languages, which all have in common the representation of domain objects or elements.

We have shown it is essential to design extensible applications keeping the *co-referentiality* among different languages, with respect to the subsets made available to the end-user. This co-referentiality may be achieved by providing, within all those languages, corresponding representations for the domain elements, at *similar abstraction levels*. Although human–computer communication is very limited, provided there is no negotiation and regulation as occurs in human-human conversation, such mapping is one step forward to the convergence of the *interpretants* for a given *object* in the designer's and the end-user's minds. The coherent mapping among those languages becomes thus the very basis for empowering users to perform programming tasks in a familiar application, to solve their problems in a familiar domain.

In the designer's endeavor to achieve the desired co-referentiality, she must carefully choose adequate representations that will lead the end-user to the same meaning, independently of the language he is dealing with at the moment. Besides, apart from the *explanation mechanism*, the variety of representations itself may be used to help the user understand a concept, illustrated under different *perspectives* and contexts.

We have proposed some guidelines for a new language that combines the UIL and the EUPL in order to narrow the gap between these languages the end-user must deal with. We have shown the importance of embedding in this language mechanisms to declare domain-specific elements.

The same mechanisms the designer has used to create the bindings between the languages must be made available to the end-user, so that he can maintain these bindings when extending the application.

Therefore, when an application is designed taking into consideration the semiotic continuum between its languages, the gap the end-user must traverse in order to extend the application is much narrower.

We have addressed several issues related to end-user programming languages and environments. Our final goal is to formalize a methodology for end-user programming environment design, including the programming, documentation, and interface languages. In order to achieve our purpose, we have outlined research topics that may be addressed separately, but keeping them situated within the larger context. These topics include:

- The development of a knowledge modeling language (KML) that could be used to model the design decisions taken during the application construction as well as to express relations among the object types.
- The development of an explanation mechanism that could make use of the knowledge application model created with the KML.
- The development of a linkage mechanism that allows linking the object types semantic structure to their interface representation.
- The definition of the EUPL for the mechanisms cited above.

- The investigation on how to create mechanisms that allow the end-user to make use of metaphors and analogies when creating his programs.

These are some of the possibilities open to investigation following the preceding analysis.

5. Bibliography

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

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

[Barbosa et al. '97] BARBOSA, S.D.J.; CARA, M.P.; CEREJA,J.R.; CUNHA, C.K.V.; DE SOUZA, C.S. (1997) Interactive Aspects in Switching between User Interface Language and End-User Programming Environment: A Case Study. To appear in *Proceedings of WOMH'97*. São Carlos, Brazil.

[CACM '96] Communication of the ACM Special Section on Learn-Centered Design. *CACM*. Apr. Vol. 39(2). ACM Press.

[Cypher '93] CYPHER, A. *ET AL*. (EDS.) (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, Ma. MIT Press.

[da Silva '96] DA SILVA, S.R.P. (1996) *Guidelines for an UIL☯EUPL for word processors.* SERG Technical Report. PUC-Rio.

[de Souza & Barbosa '96] DE SOUZA, C.S. AND BARBOSA, S.D.J. (1996) *End-User Programming Environments: The Semiotic Challenges*. Technical Report. PUC-RIO MCC 19/96.

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

[de Souza '96] DE SOUZA, C.S. (1996). *The Semiotic Engineering of Concretness and Abstractness: From User Interface Languages to End User Programming Languages*. Dagstuhl Seminar on Informatics and Semiotics. Schloss Dagstuhl, February 16-23.

[de Souza '97] DE SOUZA, C.S. (1997). *Supporting End-User Programming with Explanatory Discourse*. Submitted to ISAS'97 .

[Dertouzos '90] DERTOUZOS, M.L. (1990) Redefining Tomorrow's User Interface. In *Human Factors in Computing Systems*, page 1. Proceedings of SIGCHI'90. Seattle.

[Dertouzos '92] DERTOUZOS, M.L. (1992) The User Interface is *The* Language. In Myers, B. (Ed.) *Languages for Developing User Interfaces*. Boston. Jones and Bartlett. pp. 31-56.

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

[Draper '86] DRAPER, S.W. (1986) Display managers as the basis for user machine communication. In Norman and Draper (eds.) *User Centered System Design*. Hillsdale. Lawrence Erlbaum and Associates.

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

[Gelernter '90] GELERNTER, D. AND JAGANNATHAN, S. (1990) *Programming Linguistics*. Cambridge, Ma. The MIT Press.

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

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

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

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

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

[Smith & Susser ’92] SMITH, D.C.; AND SUSSER, J. (1992) A Component Architecture for Personal Computer Software. In Myers, B. (Ed.) *Languages for Developing User Interfaces*. Boston. Jones and Bartlett. pp. 31-56.

[Suchman ’87] SUCHMAN, L. (1987) Plans and Situated Actions. Cambridge, Ma. Cambridge University Press.