Knowledge and Communication Perspectives in Extensible Applications

Simone D.J. Barbosa
Cecília K.V. da Cunha
Sérgio Roberto P. da Silva

Departamento de Informática – PUC-Rio
R. Marquês de São Vicente, 225
Gávea – Rio de Janeiro – RJ – Brasil
[sim,ceciliak,srsilva]@inf.puc-rio.br

Abstract

End-user programming brings forth the opportunity to explore semiotic and communicative aspects of extensible software and computing in general. This paper discusses some of these aspects under the light of Semiotic Engineering [10, 11, 12], adopting a knowledge-based communication-centered approach to software design in order to help users understand and be able to extend their applications to meet their tasks requirements. We do this by carefully designing the application languages users must interact with, and by providing mechanisms that help designers disclose their rationale to users.

Keywords: End-User Programming, Semiotic Engineering, Knowledge Representation, Human–Computer Interaction.

1. INTRODUCTION

There has been a growing need in software industry to provide extensible applications, i.e., applications that users may customize and extend to help them carry out their specific tasks more efficiently. Research in this topic includes the areas of Human-Computer Interaction (HCI), End-User Programming (EUP), and Artificial Intelligence (AI).

Research on end-user programming brings about a some questions, such as: Why should users want to program? Can they learn how to program? If so, why can it be so hard on them? [19].

When users get to know the application well enough to realize that their problems cannot be directly solved, they will feel the need to change the software in order to carry out the tasks that are relevant for an efficient completion of their work. Additionally, many business-specific repetitive tasks that could not be anticipated or were not designed by software designers can and should be automated [5].

Some argue that users are not able to deal with formal languages at all, but according to [21], this is a fallacy. People are used to working with formal languages, as is the case in math, game scoresheets, and even knitting. So, why do users have such a hard time learning how to extend their applications? Nardi [21] argues that the problem lies on 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 [14] between the many languages the user must deal with, namely: interface, programming, and explanation (or documentation) languages. Discontinuity among the languages involved in extensible applications hampers user interaction, by increasing cognitive load on users when they try to recognize equivalent concepts sometimes conveyed in entirely different languages.

Semiotic Engineering [10] provides us with theoretical basis necessary to investigate the semiotic nature of software, which is designed to produce and interpret messages that can be exchanged with users. More specifically, we adopt de Souza's communication-centered architecture for extensible applications [12]. This approach intends to improve users' computer literacy by supporting users' learning processes required for programming, and by disclosing the knowledge the designer has embedded in the application to help users grasp the intended application model. The reported work is part of an ongoing research project aiming at the construction of a fully-functional EUP environment.

Section 2 describes Semiotic Engineering concepts that apply to extensible applications, and then presents an analysis of common problems found in such applications. Section 3 presents the architecture we have adopted, and discusses languages requirements for extensible software. Sections 4 and 5 illustrate our approach by presenting a prototype application and its extension mechanisms. Finally, Section 6 presents our conclusions.

2. A SEMIOTIC APPROACH TO END-USER PROGRAMMING

Many 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 [1, 2, 20], or more specifically, Semiotic Engineering [10, 11].

Treating software as a *communication* artifact can 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 these, the sender selects a set of possible *signs* according to the medium and assigns to them some *signification* hoping that the receiver will *interpret* these 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 thought, feeling, action or another sign) in a triadic schema [24]. 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, unbounded by nature, is called *unlimited semiosis* and brings critical consequences for communication.

The gist of this semiotic background is that when two individuals communicate with each other they negotiate and regulate meanings during 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 conversation, using language to regulate the meaning of language.

When it comes to software applications, this negotiation is frozen at some level by the actual implementation of the designer's interpretant at a given moment. The only resources the user now has for understanding the meaning that designer was trying to convey are the languages embedded in the application by that designer. Following is a discussion about some obstacles end-users must face when extending applications.

2.1. Obstacles to End-User Programming

Users do not start an application by wanting to extend it. At first they need to learn how to use it, and only later on will they realize that it does not do all the things they need it to do. Some applications offer a "record macro" feature, with which it is hoped that users merely need to group some sequentially executed tasks under one "name". But, this may not be enough, since macro recording usually does not allow for any generalizations, iterations, and conditionals. Therefore it is suited only for repeating the same exact task, with the same exact values. In order to make use of generalizations, iterations, and conditionals, users must often manually edit the macro.

There is an unmeasurable gap between macro recording and editing. For one, macro code is usually textual, and therefore entirely different from the familiar interface icons, buttons and menus users interacted with in order to record it. Usually there is not a consistent (isomorphic) mapping between the user interface (UI) elements and the end-user programming (EUP) elements, so that users would be able to bridge the gap by making the appropriate associations. Therefore, instead of focusing on the solution to their problems, users have to learn and understand a whole new language. Some obstacles faced by users when trying to extend current so-called extensible applications have been presented elsewhere [4, 7, 9], and will be summarized here:

- There is usually a serious case of discontinuity between the user interface language (UIL) and the end-user programming language (EUPL). This discontinuity may occur in a variety of ways[1]:
  - An element may be available in the UI, but inaccessible through the EUPL;
  - An element may be available in both UI and EUPL, but may behave differently in each language;
  - An element may be available in both UIL and EUPL, but have different representations in each language (for instance, a dialog box may have a graphical representation in the UIL and a textual representation in the EUPL);
  - The level of articulation is different in each language (for instance, an element may be treated as a unit in one language and need to be unfolded into a structure in the other);

[1] We do not consider as a serious discontinuity the absence, in the UIL, of explicit representations of conditionals and iterations, since these constructs would only be provided in the user interface when dealing with visual programming languages. It is expected that EUPLs have a greater expressiveness than UILs.

- Macro recording features are usually limited and inflexible, in that:
  - They usually allow only for the recording of sequences of commands, and not for iterations or conditionals;
  - They record constant values (tokens), and do not allow generalizations (types) to be made.
- The binding between a new EUPL code fragment to a new UI element is frequently random.
  - The mechanism provided to make the connection between these elements is intricate and complex;
  - There is no verification to ensure consistency of the resulting interface.
- The EUPL is inadequate, in that it isn't usually task-specific, as suggested in [21].
- There is often a discontinuity between the Documentation Language (DL) and the EUPL.
  - The on-line help doesn't provide adequate explanation about the EUPL constructs or their usage;
  - Users are usually not prompted to document their code;
  - Some documentation could be automatically inserted (for instance, from macro recording), but it is rarely the case.

These problems can be avoided by adopting a knowledge-based communication-centered approach to software design, in which the languages involved are carefully crafted so as to maintain their co-referentiality [14]. Next section will describe the architecture we are adopting and the requirements for the languages involved in it.

3. COMMUNICATION-CENTERED, KNOWLEDGE-BASED ARCHITECTURE

Software can be seen as meta-communications artifacts. They send and receive messages during interaction with users, but are themselves an achieved one-shot message sent from system designer to system user. This idea is central to Semiotic Engineering [10], a theoreticallybased approach to designing user interface languages and codes. 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 [18].

Moreover, software must now support not only productive user-system interaction, but also end-user programming. This can be achieved by more or less sophisticated configuration settings, macro recordings, or programming in the small techniques [19, 21], as well as by some potentially knowledge-based (KB) programming by demonstration [5] and history-based programming [16]. According to the semiotic perspective, where designers are communicating with users, end-users may now be encouraged to use the same communicative resource to write their own messages (to themselves and/or to others, in a collaborative environment [21]). This situation is depicted in Figure 1, adapted from [9].

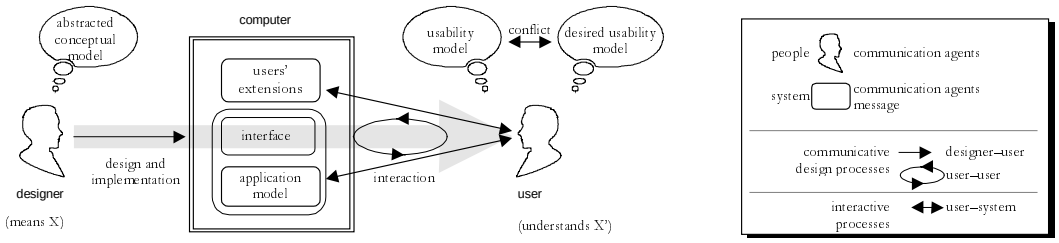


Figure 1 —Semiotic Engineering Framework extended to include End-User Programming

For users to realize that they *need* and *can* re-configure, re-program, or extend software, they first need to understand its meaning, and then understand how new meanings can be added to it. Such laborious processes of interpretation should be facilitated and supported by conscious Semiotic Engineering of computational languages. Thus, users should not only be able to make sense of designers' intents, but also express and achieve their own meanings.

Designers always predict and prescribe communication form and content. However, users are seldom (if ever) deliberately made aware of the arbitrary rules and models that underlie applications. Our goal is to disclose these rules and models to facilitate end-users' understanding and extension of applications.

3.1. The Architecture

Our communication-centered architecture is one which exposes the existence of arbitrary but learnable computer codes, and optionally unfolds [13] successive levels of their grammatical and semantic design rules, so that users can potentially wield such linguistic resources to build new discourse, by adapting and programming applications.

Along with each extensible application, a rich learning environment must be provided to users. Important components thereof are multimodal multicode interfaces, explanation knowledge-based-systems, situated help and documentation modules, as well as textual and graphic programming environments. Embedded explanation modules should help designers adopt a communication perspective as if they were engaged in documentary bookwriting or filmmaking, and cared to get their message across.

The basic architecture we envisage for truly extensible computer applications, from an end-user's point of view, is presented in Figure 2. At first sight, the adopted architecture may appear overwhelmingly complex and undesirable from a cost/benefit perspective. Nevertheless, at a closer look, we can realize that a number of popular office automation packages already offer users most of the proposed components. On the user interface side, the only lacking modes are the natural language (NL) question-answering and the visualization facilities. MS Word for Windows®, for example, includes the other three alternatives and a macro language interpreter of its own.

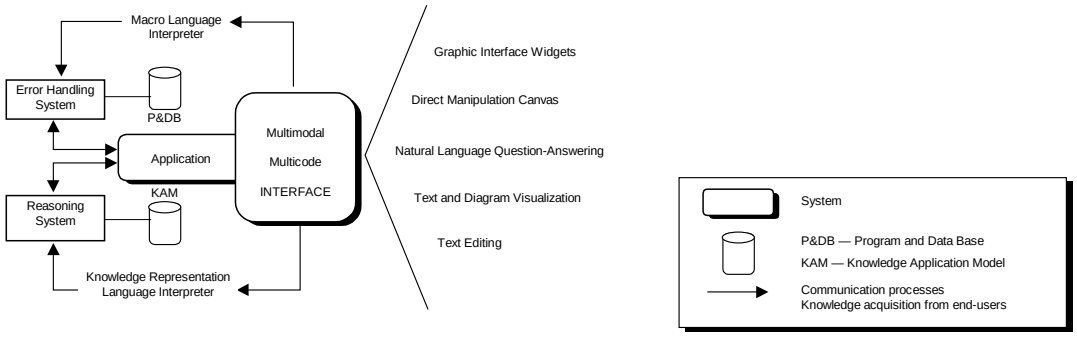


Figure 2 – A Communication-Centered Architecture

The missing end of existing extensible applications is the knowledge base. In the lower left side of Figure 2, we clearly see an embedded intelligent system (reasoner and KAM). In the upper side, we see a runtime consistency checker and a Program and DataBase (P&DB). This latter component is the repository of end users' programmed extensions, which are added to the application in separate modular form.

Our architecture incorporates the following components:

(1) A domain ontology that represents the intrinsic nature of the domain's objects and their static relations.

(2) A domain model that represents the interactions between the domain's objects, over time and for a particular end. It could contain some, or all, elements of the domain ontology.

(3) An application model that represents the interactions between some domain objects and the system objects, necessary for the implementation of the application's functionality and its interface[2].

(4) An extensive user interface language (UIL) that must include extension mechanisms such as the use of metaphoric operators and that must reflect (1), (2), and (3).

(5) An end-user programming language that is semiotically continuous with the UIL.

(6) A knowledge representation language that must give semantic support for UIL and EUPL and that makes it possible to generate coherent NL explanations for end users.

The application's design knowledge base is represented in Figure 2 by the KAM, which is composed of the domain ontology and the domain model, and by the P&DB, which includes the application model. The reasoning and error handling subsystems will be composed, among others, of mechanisms to support: the co-referentiality between the UIL and EUPL, the EUPL disambiguation, the concepts' categorization movement, and metaphorical mappings that could be used at the UIL level.

We will not enter into further detail in this paper about items (1), (2), and (3). We would just like to state that the domain ontology and the domain and application models must have a dynamic nature. They will be defined by the software designer and may be extended by end-users, in a controlled manner.

The overall communication management is achieved by means of integrated programming-explaining activities. When users engage into introducing a new functionality to an existing application, explanations are elicited from them in schematic form. Thus, not only are new items introduced in the P&DB, but also the corresponding explanatory

[2] Components (1), (2), and (3) were present under a different guise in [23]'s discussion about his ecological approach to modeling human-computer interaction.

items are added to the KAM. This feature requires that the interface, programming, and knowledge representation languages be co-referential and maintain consistency among each other.

Co-referentiality between input and output languages has been previously proposed in the framework of user-centered systems design [14]. Our leap here is to extend linguistic co-referentiality from the realm of interfaces to that of applications. Of course, requiring users to explain their programming to the system is preceded by requiring applications' designers and programmers to do the same. The whole point of communication-centered programming is to tell people what we as programmers have in mind and increase the chances that we are understood, and that our product becomes really useable, useful and extensible.

In an extensible environment, many languages are involved, namely: user interface, programming and documentation languages. Next section will describe some of the requirements these languages must meet in order to achieve our communicative goals.

3.2. The Environment Languages

The many languages used in an extensible 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 end-users probably know their 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 [6], that end-users always use a specific language in their work domain [see also 21]. The specificity of this language guides and optimizes 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 end-users' interpretations if we embed representations for the *domain objects* in the UIL, thus mapping the *domain language* onto the UIL. To be precise, domain objects should be represented in all of the application languages that we believe are needed to the environment architecture, namely: user interface, programming, and documentation languages.

Thus, the languages that comprise the environment should be made *domain-dependent*, to help reduce end-users' potential interpretation problems. It is also necessary to make the *design rationale* available to end-users, since it presents the decisions the designer has made when constructing the software.

As stated before, a major source of misconceptions is the absence of a consistent mapping between the UIL and the EUPL in the current extensible applications. Furthermore, since end-users may frequently need to build their own dialog boxes, they must be able to use the same elements present in the UIL they are used to interacting with. Therefore, we must make available to end-users, as they extend an application, the elements of the *system software domain* they 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, but 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 them through application use.

All of the above indicates that, in order to reduce the problem of different interpretations caused by the existence of multiple languages, it is necessary to have a mapping between objects in each of these languages. Ideally, this mapping would be *isomorphic*, but such ideal is hardly achieved. Nevertheless, providing a consistent mapping between the UIL signs and the EUPL and DL signs will surely help bridge the gap between these languages, and help users reach a mental application model similar to the designer's.

Another common problem with current programming languages is that they only make it 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 end-users to visualize the lower level for a given primitive, as shown in [13]. 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*.

For instance, if we had such an UIL→EUPL mapping, end-users would not have difficulties in discovering which EUPL element corresponded to the desired UIL element. And also via the perspective mechanism they could switch to the DL, through the UIL→DL mapping, and easily find useful information that could help them correctly use the elements in the first place.

Together with an *explanation mechanism*, such abstractions and multiple perspectives tell end-users about the *designer's rationale* behind each choice the designer 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, where end-users 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 they could switch their perspective to another language.

This discussion clearly shows that the solution of embedding EUPLs into the application alone is not enough. If end-users take the role of a designer, without possessing the designer's kind of knowledge, it would be necessary to make available part of his or her knowledge from within the application. The knowledge necessary is not only 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 end-users, who have little or no programming experience.

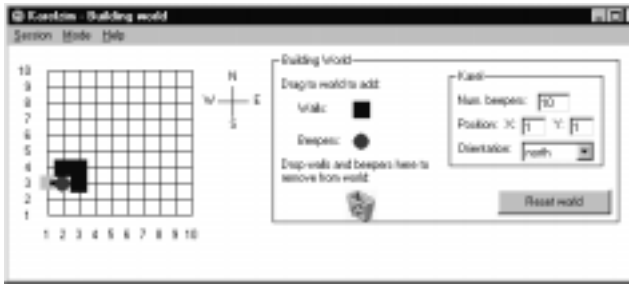
In conclusion, to obtain some continuity between the languages we need to provide a mechanism that maintains an *internal consistency* within the application. This consistency may be achieved by keeping a co-referential [14] mapping of the domain elements and structure representation throughout those languages. To achieve this mapping we advocate the existence of a knowledge application model (KAM) that will be used as a knowledge base. In it, some environment mechanisms could perform operations that will *maintain* and *explain* the decisions and usage the designer has made about 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, through learning and communicative processes.

4. HUMAN-COMPUTER INTERACTION AND KNOWLEDGE-REPRESENTATION IN END-USER PROGRAMMING

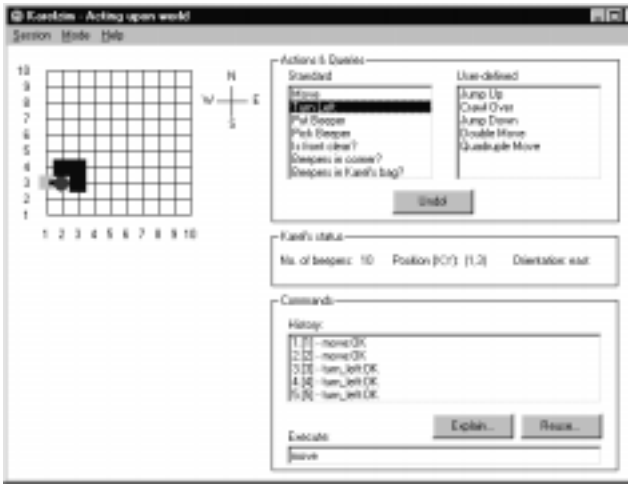
This section illustrates how our communication-centered perspective impacts design, by means of a demo extensible application inspired on Pattis et al.'s "Karel, the Robot – A Gentle Application to the Art of Programming" [22], whose aim is to teach novices how to program. We have built a Visual Basic® Interface to Karel's World (see Figure 3), and provided interaction for Karel's minimal operating capabilities about moving, turning left, picking up and putting down beepers, in a gridded (limited) space where walls can block its passage. We have also opened the opportunity for extensions by allowing Karel to learn new things.

Within our application, we want to convey to users not only the designer's arbitrary decisions, but the notion that they now can, as users, make their own decisions about the application and express their own interpretations and intentions.

At the User Interface level, the new perspective in design becomes evident in the layouts shown in Figure 3. The "building window" in Figure 3a allows users to deploy walls and beepers in Karel's world via direct manipulation of graphic objects. Since the idea is to program (i.e. to tell, in linguistic mode) Karel to evolve in this space, direct manipulation is halted when it comes to commanding the robot to move and change the world (Figure 3b). For this, users have the more traditional options of menu selection and button pressing. Another way to help users evolve from interacting with to extending the application is provided by disclosing the command syntax below the command history, so it will be easier for them to recognize the EUP statements corresponding to their user actions in the UI.



(a)



(b)

Figure 3 (a) Snapshot of world building window. (b) Snapshot of world screen.

The users' extensions is shown next to the primitive actions and queries. The simplest extension would be DOUBLE MOVE, generated by some macro recording mechanism upon two consecutive MOVE actions.

It is interesting to note the differences between extensions made by end-users and experienced programmers. While the former can introduce "naïve" and inefficient computational solutions, the latter may work out complex algorithms to minimize memory and processing requirements. It is important to allow end-users to follow the path of least cognitive effort when extending their applications.

We, as designers, decided that Karel is *not* an intelligent agent. As such, it may answer questions about the current state of the world, but not about the *rationale* behind its actions. Notice the "Actions & Queries" section in Figure 3b. It presents questions that may be addressed to Karel, i.e., situational questions that the agent may answer.

When it comes to questions that only the designer may answer, typically "why" and "what" questions, we must invoke an explanation module, by clicking the "Explain…" button in Figure 3b, which reveals the explanation window (Figure 4) at the step we want to investigate. For instance, the question "why did move fail?" is addressed to the designer, since the agent has no knowledge about the rationale behind its actions.

Note that knowledge in this scenario is used by the *designer*, not by the agent. Karel is *not* an intelligent agent. We could endow Karel with virtually no knowledge and have it shutdown if it is ordered to move forward and bumps into a wall, as first suggested by Pattis. However, in our program, we have decided to test for disastrous actions before they are performed. It is the programmer, and not Karel, who stops the agent short of shutting down and uses a KB explanation system to reflect his or her decision that Karel is a physical model which does not traverse walls or world boundaries.

Figure 4 – Snapshot of explanation window.

We must be careful with our phrasing when reflecting Karel's and the designer's knowledge. If we let Karel use a 1st person discourse, such as "I cannot traverse walls", users may build an incorrect model of the application by assuming Karel with some intelligence it does not have. For instance, given that it knows it cannot traverse walls, it could be expected to know why not.

In our Semiotic Engineering approach, the designer is a legitimate voice in discourse. The designer's interpretation and programming decisions are embedded in the application, and his or her decision was to endow the robot only with the ability to follow instructions and report whether they could be carried out successfully or not. We have designed

an explainable KB in SWI-Prolog®, to be coupled to Karel's Visual Basic® Interface, where we try to disclose to users the origin of such decisions and interpretations.

This approach is consistent with the idea that the robot is not an intelligent agent. It has some sets of instructions, and it can only assess whether they could be carried out successfully or not, without knowing why it was so. The knowledge behind the actions belongs to the designer, and it is the designer's responsibility to make this information available to users. The robot can answer a few simple questions about the state of the world, such as how many beepers there are in its bag at a given moment. But, if some error should occur, and no answer can be computed by the robot, it is again the designer's turn to explain why.

Therefore, we have one discourse for the robot, stating whether it completely executed an action or not, and another discourse for the designer, where he or she should make available to users the rationale behind the robot's actions and instructions. We intended to make this distinction clear by providing a separate module, the explanation window, where the designer's knowledge can be accessed.

5. EXAMPLE OF AN EXTENSION

In order to illustrate the extension mechanisms supported by our environment, we will describe the creation of new action based on the MOVE primitive. A MOVE instruction causes Karel to advance one step in the direction it is facing, provided that there is no wall or world boundary in front of it. Our new action, let us call it JUMP UP, will teach Karel to climb up a wall. This new action will be coupled with a CRAWL OVER and a JUMP DOWN actions, making up the set of actions necessary for Karel to overcome walls as obstacles.

Just how would users create these new actions? They would probably notice a great resemblance between what they want (JUMP UP) and the primitive action MOVE. They would then choose to view and alter the code corresponding to MOVE. A Prolog version of this code is illustrated below[3]:

```
/*-------------------------------------------------------------
 * move_next_coord(north,OldX,OldY,OldX,NewY)
 *
 * calculates next coordinate for a move action
 *-------------------------------------------------------------*/
move_next_coord(north,OldX,OldY,OldX,NewY) :-
    NewY is OldY + 1.
move_next_coord(west,OldX,OldY,NewX,OldY) :-
    NewX is OldX - 1.
move_next_coord(south,OldX,OldY,OldX,NewY) :-
    NewY is OldY - 1.
move_next_coord(east,OldX,OldY,NewX,OldY) :-
    NewX is OldX + 1.

/*-------------------------------------------------------------
 * move
 *
 * moves Karel
 *-------------------------------------------------------------*/
move :-
    karel(Orientation,coord(OldX,OldY)), /* retrieves current robot position */
    move_next_coord(Orientation,OldX,OldY,NewX,NewY), /* calculates robot's new location */
    check_predicate(in_bounds(NewX,NewY),'Karel would fall off the world.  You could turn it
before trying to move it.',_), /* check for out of bounds situation */
    check_predicate(not(wall(coord(NewX,NewY))),'Karel would hit a wall.  You could turn it
before trying to move it.',_), /* check for inexistence of wall in new location */
    change_karel(Orientation,NewX,NewY),!. /* changes robot position */
```

Provided that users understand the code aided by comments and a high degree of legibility of the extension language, they would then feel the need to invert the pre-condition `not(wall(coord(NewX,NewY)))` to `wall(coord(NewX,NewY))`. This means that Karel can only jump up if it is facing a wall. Since the user is now playing the role of a designer, the "designer's explanation" should also be changed to reflect the code modification. If the previous predicate should fail if 'Karel would hit a wall', now it will fail if 'Karel is not facing a wall'.

Upon finishing the code modification, the application will prompt the user to give a name to the new code, and add it to the user-defined actions list in the user interface. This is a necessary step for end-users to include their rationale into the application KB, so that they or other users may have access to this rationale afterwards. From a user perspective, one overhead is that of connecting user-generated code to user-generated explanations about code. Just

[3] The extension language is not supposed to be Prolog. Part of our research includes the development of a plan-based programming language that resembles the natural language used in recipe and do-it-yourself descriptions [8].

as EUP languages are not C or Pascal, KB representation languages are not Prolog. The need for more adequate programming languages goes hand in hand with the need for more adequate knowledge acquisition languages. This is one of the aims of ongoing research in our current project.

6. CONCLUSION

The need for end-user programming comes hand in hand with the need to let users learn more about applications and programming. We have suggested how knowledge-based systems may support this learning process by adopting a communicative approach where users are made aware of the designer's interpretation of a given domain, its objects and tasks, as well as his or her encoded arbitrary decisions in building and implementing a computer model of the domain.

A Semiotic Engineering approach to the design of extensible applications allows us to analyze the problem of communicating not only the designer's solution to—what he or she supposes is—the users' problems, but also the tools he or she made available to users 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 the importance of keeping the *co-referentiality* among these languages.

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 end-users' minds. The coherent mapping among those languages becomes thus the very basis for empowering users to perform programming tasks in a familiar application, and thus solve their problems in a familiar domain. The same mechanisms the designer has used to create the bindings between the languages must be made available to end-users, so that they too can maintain these bindings when extending the application.

This leads us to adopt a broader semiotic perspective on Software Engineering. Interface messages can be interpreted in a limited chain of signs that turn into other signs and eventually crystallize into output. Whichever sign is present in this chain was once part of the designer's *interpretant* derived from the application's domain. This knowledge-based communicative approach helps users understand and modify extensible applications, by expressing their intentions and rationale in writing software. The linguistic overhead is that of designing co-referential programming and knowledge representation languages that users can access from the application's interface.

REFERENCES

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

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

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

[4] 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. In *Proceedings of WOMH'97*. São Carlos, Brazil.

[5] Cypher, A. (eds.) *Watch What I Do: Programming by Demonstration*. MIT Press. Cambridge, Ma. 1993.

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

[7] da Silva, S.R.P.; Barbosa, S.D.J.; de Souza, C.S. (1997) *Communicating Different Perspectives on Extensible Software*. In Lucena, C.J.P. (ed.) Monografias em Ciência da Computação. Departamento de Informática. PUC-RioInf MCC 23/97. Rio de Janeiro.

[8] da Silva, S.R.P.; de Souza, C.S.; Ierusalimschy, R. (1997) *A Communicative Approach to End-User Programming Languages*. In Lucena, C.J.P. (ed.) Monografias em Ciência da Computação. Departamento de Informática. PUC-RioInf MCC 47/97. Rio de Janeiro.

[9] de Souza, C.S. and Barbosa, S.D.J. (1996) *End-User Programming Environments: The Semiotic Challenges*. In Lucena, C.J.P. (ed.) Monografias em Ciência da Computação. Departamento de Informática. . PUC-RioInf MCC 19/96. Rio de Janeiro.

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

[11] 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.

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

[13] 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.
[14] Draper, S.W. (1986) Display managers as the basis for user machine communication. In Norman and Draper (eds.) *User Centered System Design*. Lawrence Erlbaum and Associates. Hillsdale, NJ.
[15] Gelernter, D. and Jagannathan, S. (1990) *Programming Linguistics*. Cambridge, Ma. The MIT Press.
[16] Kurlander, D. & Feiner, S. "A History-Based Macro by Example System" in *Watch What I Do: Programming by Demonstration* edited by Allen Cypher.MIT Press. Cambridge, Ma. 1993.
[17] Microsoft Corporation (1995) *The Windows Interface Guidelines for Software Design*. Redmond. Microsoft Press.
[18] 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.
[19] Myers, B.A. (Ed.) (1992). *Languages for Developing User Interfaces*. Jones and Bartlett Publications. Boston.
[20] 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.
[21] Nardi, B. (1993) *A Small Matter of Programming*. Cambridge, Ma. The MIT Press
[22] Pattis,R.E.; Roberts,J.; and Stehlik,M. (1995) *Karel the Robot: A Gentle Introduction to the Art of Programming*. New York, N.Y. John Wiley and Sons.
[23] Payne. S.J. Interface Problems and Interface Resources. In Carroll, J. (ed.) *Designing Interactions: Psychology at the Human-Computer Interface*. Cambridge University Press. Cambridge, Ma.
[24] Peirce, C.S. (1931) *Collected Papers*. Cambridge, Ma. Harvard University Press. (excerpted in Buchler, Justus, ed., Philosophical Writings of Peirce, New York: Dover, 1955)