Interaction Modeling as a Binding Thread in the Software Development Process

Simone Diniz Junqueira Barbosa, Maíra Greco de Paula
Departamento de Informática, PUC-Rio
R. Marquês de São Vicente, 225
Rio de Janeiro, RJ, Brazil – 22453-900
{simone, mgreco}@inf.puc-rio.br

Abstract

This paper proposes the use of an interaction modeling language called MoLIC to graphically represent scenarios as an additional resource in software development. MoLIC brings human-computer interaction (HCI) concerns to software engineering processes. It does so by representing potential user-system interaction paths, which will not only allow software designers to make decisions about the HCI aspects of software, but also provide a blueprint — from the users' point of view — of what needs to be implemented and tested, and how users should perceive it. Using MoLIC as an input artifact, software engineering techniques may be used to decide how to build the software that will make it possible for the represented interaction to happen. As such, MoLIC provides a basis not only for communication and understanding among team members, but also as a concrete resource for software design and development.

1. Introduction

The value of using scenarios in software development has been investigated for almost a decade [2]. In the area of human-computer interaction (HCI), it has been extensively used [3], especially in the preliminary stages of software design.

A few attempts have been made to use scenarios as a representation for bringing HCI aspects to software engineering processes [6]. However, as the collection of natural language scenarios gets larger for a single application, it becomes harder to make consistent decisions about what must be done. In addition, due to frequent ambiguities found in natural language texts, there are too many decisions to be made in moving from scenarios to software specification. These decisions are seldom recorded for future reference or verification of the final product. Unfortunately, more often than not what is developed is very different from what has been represented in the scenarios.

So, in order to maximize the benefits brought about by using scenarios, we need to reduce the gap between scenario construction and software specification. To achieve this, we propose to use an intermediate representation language called MoLIC. MoLIC stands for "Modeling Language for Interaction as Conversation". It is a design language devised under the semiotic engineering theory of human-computer interaction.

2. Semiotic Engineering and Scenarios

MoLIC is rooted in semiotic engineering [5], a theory of HCI which views the user interface as a metamessage sent from designers to users. This message is designed in such a way as to be capable of exchanging messages with users, i.e., allowing human-system interaction. In semiotic engineering, interaction design is viewed as conversation design. This conversation is of a unique kind, however, because the designer is no longer there at the time it is carried out. This means that he/she must anticipate every possible conversation that may occur, and embed in the system his/her "deputy": the designer's spokesman in the system, with whom users will communicate during interaction. We have devised MoLIC, "Modeling Language for Interaction as Conversation", in order to support the creation of this deputy, i.e., to design the whole range of interactions that may take place in the application. Our goal is to support the designer's reflective and decision-making processes about his/her design throughout the design process [11], under a semiotic engineering perspective.

One way to support the creation of the designer-to-user message is to help designers express *what* they want to say, before going into *how* the designer's deputy will say it. This what–how coupling may be achieved by using scenarios [2] together with MoLIC.

Scenarios can be used throughout the development process, starting from the analysis of the domain and the users' tasks and characteristics. A major goal of using scenarios at this stage is to explore or confirm, together with the users, the designers' understanding of the goals and tasks to be supported. By means of scenarios, designers not only learn about the users' vocabulary and

thought processes, but they have a chance to make this knowledge explicit in a language users fully understand, so it can be verified by users themselves. Scenarios are also very useful in uncovering and exploring typical and atypical courses of action in specific circumstances.

It is important to notice that designers should avoid including in the early scenarios references to concrete interface elements, such as text elements and widgets. By avoiding an early commitment and raising user's expectations about the interface solution that will be adopted, both designers and users will be open to explore alternative solutions at later stages.

One of the major advantages of using scenarios in HCI is to provide a detailed setting for better understanding and conceiving contextualized pieces of user–system interaction. Taken from another perspective, however, the contextualized and detailed nature of scenarios may be, to some extent, a drawback in HCI design. When using scenarios alone, designers may have difficulty in grasping a global view of the application as a whole system, and also in understanding the interrelations between the tasks and goals it should support. This limitation hinders the design of a coherent communicative artifact, an essential requirement for the Semiotic Engineering of Human-Computer Interaction.

In order to fill this gap in HCI design, we propose to complement scenarios with MoLIC. MoLIC was conceived to be applied between the initial analysis and the interface specification phases. It has shown to be useful in helping designers grasp a global view of the conversations that comprise the application, and thus design a coherent designer-to-users message, keeping in mind the communicative role of the designer's deputy.

MoLIC may be viewed a first step towards user interface specification. It allows the representation of users' goals and steps to achieve them, as well as the information to be exchanged between users and the designer's deputy at each step. The focus is always on the communicative capacities of both user and designer's deputy: interaction occurs only as a result of the communication between these two agents.

When writing scenarios, the designer should have in mind what the purpose of each scenario is: to investigate certain aspects of a domain or socio-cultural environment, to propose a task structure or an interaction solution, to contrast alternative task structures or interaction solutions, and so on. These purposes should be made explicit when creating scenarios. One way to achieve this is through a set of questions that are expected to be answered by the users' feedback for each scenario.

Another benefit from asking questions based on scenarios is to uncover hidden information, so that the signs[1] (what) and conversations about signs (how to) are made explicit. In semiotic engineering, investigating "how to" means not only supporting users in how to manipulate signs, but also how to understand them in context. In addition, we propose to include a set of questions to investigate the motives underlying each decision the user will be making during interaction (why), in an attempt to capture the design rationale. The answers to these questions will provide valuable information for the designer to proceed. Some of these answers may generate further questions, which in turn may give rise to unanticipated scenarios. Viewed under this perspective, this approach is similar to systematic questioning as illustrated in [4].

As a running example, we will use an application designed using scenarios and MoLIC. It is an extended annotation system for a small research group. Every user may post documents and annotations to documents in order to start discussions about them. Users may belong to one or more groups, according to common interests, level of expertise in certain research areas, or some other user-defined criteria. Users may also evaluate documents and annotations, recommend documents to other users, and change information about their profile.

A sample scenario would be the following:

Carol has just joined the graduate program at UC. In order to get to know the work of her research group, she decides to access the annotation system they use to discuss about their ongoing work and published papers, as well as relevant papers from the international research community. As she enters the system, she realizes she doesn't have a password, and is glad that there is a possibility of logging into the system as a guest [Q1]. She searches for the documents in her area of interest [Q2], views some of the papers being currently discussed and decides to print two of them to read at home [Q3]. She sends a request for a password, so that in the future she will be able to actively participate in the discussions [Q4].

Some of the questions regarding this scenario excerpt might be:

Q1 Who can access the system? (Should the system be available to the general public? If so, which parts, and why?)

Q2 How do users like or need documents to be organized? What is the purpose of document classification?

[1] The most widely used definition of signs in Semiotics is: "anything that stands for something to someone" [10]. We use signs here to stand for the whole range of representations used by designers and users, from conceptual information to user interface elements.

Q3 How can documents be viewed? (Are users online all the time? Should the system have both online and printable versions of documents?)

Q4 Should registration in the system be automatic? Who can register, and how?

From the scenarios, users' goals are identified and may be structured in a hierarchical goal diagram. A user's goal extracted from the sample scenario is, for instance, "searching documents". As we will see in the next section, for each identified goal, a piece of interaction model is built in such a way as to be interrelated with the paths of interaction corresponding to other goals, aiming to form a coherent whole.

3. Using MoLIC in Software Design

MoLIC comprises three interrelated representations: a diagram of users' goals, an ontology of the domain and application signs, and an interaction model.

3.1 User Goals

The first step in using MoLIC is to extract, from the scenarios, the top-level goals users may have when using the application, i.e., goals that are explicitly supported by it. These goals are then organized in a hierarchical diagram, according to some classification the designer finds useful: what are the classes of users that will form this goal; how primary is the goal with respect to the system scope definition (or whether it is just a supporting goal); the frequency with which the goal is expected to be achieved, and so on.

3.2 Domain and Application Signs

The next thing to extract from the scenarios are the domain and application signs that are meaningful to users. While these signs are usually treated as data, in semiotic engineering they acquire a new status, going further than establishing the vocabulary to be shared between designer's deputy and users. Defining domain and application *sign systems* help designers uncover users' expectations about how knowledge should be shaped and built. Signs allow designers to establish what the system is all about, and how users will be able to understand and use it.

A widespread example of a simple sign system is the use of ontologies to organize domain and application concepts [1]. A designer should be able to straightforwardly derive an ontology from usage scenarios. However, this is seldom the case. Scenarios are often incomplete and ambiguous, and it is hard to keep the whole set of scenarios consistent and coherent. Thus, in order to build ontologies that define the application signs, designers should explore as many dimensions or classifications of signs as necessary to grasp their full meaning.

As in typical data classification, signs may be grouped according to the kind of information they may have as a value. The most simple classification is probably the one that distinguishes only between textual and numeric values. In order to be useful for HCI design, this classification needs to be complemented with knowledge about the range of values these signs may assume. For instance, a user interface element used for providing unconstrained textual information is different from that for providing a piece of textual information that belongs to a known set of values.

An interesting classification is related to the degree of familiarity to a sign users are expected to have. In this classification, domain signs are those directly transported from the users' world, such as "full name". Application signs, on the other extreme of the scale, are those that were introduced by an application, and have no meaning outside it. Still in this classification, there is an intermediary kind of sign: a transformed sign is derived from an existing sign in the world, but has undergone some kind of transformation when transported to the application. This transformation is often related to an analogy or metaphor.

The reason for this classification to be interesting in HCI is that different kinds of signs may require different kinds of user interface elements to support users. In general, a domain sign would require an explanation only inasmuch there are constraints imposed by the application. For example, the concept of "full name" is clear to users, but a restriction about the number of characters allowed in the corresponding piece of information might not be, and thus need some clarification from the designer's deputy. A transformed sign would require an explanation about the boundaries of the analogy. For example, a folder in the desktop metaphor might require an explanation about its "never" getting full, except when the hard disk which contains it lacks space. At the end of the scale, an application sign may require a complete explanation about its purpose, utility and the operations that can manipulate it. An example might be a sign for "zooming" in a graphics application.

There are of course some signs that can be classified in either group. For example, a password may be thought of as a transported sign, derived from the existing domain sign: signature. In these cases, it is the designer's responsibility to decide, based on the analyzed characteristics of users and their tasks, the amount of support to provide in order to have users fully understand each sign.

It is important to note that users may become familiar with certain signs in one application and then transport this knowledge to another application. When this is done unsuspectingly, however, it may cause unpredictable distortions in the users' conceptual model of the latter application.

There is also the typical input/output classification, which establishes who will be responsible for manipulating the sign at a certain point during interaction: the user or the system (via the designer's deputy). This classification, however, changes during interaction, according to the user's current task, and thus may be considered a task-dependent property of the sign. Task-dependent signs will be explored in the next section, in which we describe MOLIC's interaction notation.

After having established these diverse sign classifications, designers can follow traditional ontology engineering techniques to represent the acquired knowledge.

3.3 Interaction Modeling

From the user-approved scenarios and their corresponding signs, HCI designers have enough elements to build an interaction model and thus shape the computational solution.

When interaction is viewed as conversation, an interaction model should represent the whole range of communicative exchanges that may take place between users and the designer's deputy. In these conversations, designers establish *when* users can "talk about" the signs we extracted from the scenarios. The designer should clearly convey to users *when* they can talk about *what*, and what kinds of *response* to expect from the designer's deputy. Although designers attempt to meet users' needs and preferences as learned during user, task and contextual analyses, designing involves trade offs between solution strategies. As a consequence, users must be informed about the compromises that have been made. For instance, MoLIC allows the representation of different ways to achieve a certain result, criteria to choose one from among them, and of what happens when things go wrong.

MoLIC supports the view of interaction as conversation by promoting reflection about how the design decisions made at this step will be conveyed to users through the interface, i.e., how the designers' decisions will affect users in their perception of the interface, in building a usage model compatible with the designers', and in performing the desired actions at the interface. This model has a dual representation: both an abbreviated and an extended diagrammatic views. The goal of the diagrammatic interaction model is to represent all of the potential conversations that may take place between user and system, giving designers an overview of the interactive discourse as a whole.

The interaction model comprises scenes, system processes, and transitions between them. A scene represents a user–deputy conversation about a certain matter or topic, in which it is the user's "turn" to make a decision about where the conversation is going. This conversation may comprise one or more dialogues, and each dialogue is composed of one or more user/deputy utterances, organized in conversational pairs. In other words, a scene represents a certain stage during execution where user–system interaction may take place. In a GUI, for instance, it may be mapped onto a structured interface component, such as window or dialog box, or a page in HTML. In the diagrammatic representation, a scene is represented by a rounded rectangle, whose text describes the topics of the dialogues that may occur in it, from the users' point-of-view (for instance: `Search documents`). Figure 1 illustrates the representation of a scene.

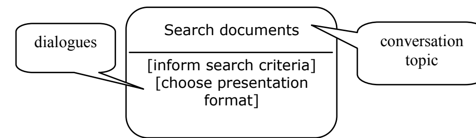

Figure 1. Diagrammatic representation of scene "Search documents".

The signs that make up the topic of dialogues and scenes may also be included in an extended representation. When this is the case, we adopt the following convention: when a sign is uttered by the deputy, i.e., presented to the user, it is represented by the sign name followed by an exclamation mark (e.g. `date!`); whereas a sign about which the user must say something about, i.e., manipulated by the user, is represented by a name followed by an interrogation mark (`login?`, `password?`). Figure 2 illustrates the extended representation of a scene[2].

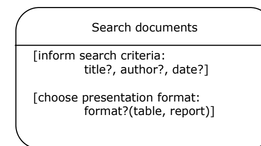

Figure 2. Extended scene representation including signs

A system process is represented by a black rectangle, representing something users do not perceive directly. By doing do, we encourage the careful representation of the deputy's utterances about the result of system processing, as the only means to inform users about what occurs during interaction.

[2] The detailed specification of the extended representation is found in [8]

Transitions represent changes in topic. This may happen due to the user's choice or to a result in system processing. Transitions are represented by labeled arrows. An outgoing transition from a scene represents a user's utterance that causes the transition (represented by a bracketed label, such as u: `[search document]`), whereas an outgoing transition from a process represents the result of the processing as it will be "told" by the designer's deputy (represented by a simple label, such as: `d:document(s) found`). In case there are pre-conditions for the transition to be made, they should come before the transition label, following the keyword `pre:`. Analogously, the `post:` keyword after the label is used to mark a post-condition, or a new situation or application state that affects interaction.

Some scenes may be accessed from any point in the application, i.e., from any other scene. The access to these scenes, named ubiquitous access, is represented by a transition from a grayed scene which contains a number following the letter U, for "ubiquitous". Moreover, there are portions of the interaction that may be reused in various diagrams. Stereotypes are created to represent parameterized interaction diagrams, represented by a rounded rectangle with double borders (such as `View(document)`).

In semiotic engineering, error prevention and handling are an inherent part of the conversation between users and system, and not viewed as an *exception*-handling mechanism. The designer should convey to users not only how to perform their tasks under normal conditions, but also how to avoid or deal with mistaken or unsuccessful situations. Some of these situations may be detected or predicted during interaction modeling. When this is the case, we extend the diagrammatic representation with breakdown tags, classifying the interaction mechanisms for dealing with potential or actual breakdowns in one of the following categories:

Passive prevention (PP): errors that are prevented by documentation or online instructions. For instance, about which users have access to the system, what is the nature of the information expected (and not just "format" of information).

Active prevention (AP): errors that will be actively prevented by the system. For instance, tasks that will be unavailable in certain situations. In the interface specification, this may be mapped to making widgets disabled depending on the application status or preventing the user to type in letters or symbols in numerical fields, and so on.

Supported prevention (SP): situations which the system detects as being potential errors, but whose decision is left to the user. For instance, in the user interface, they may be realized as confirmation messages, such as "File already exists. Overwrite?")

Error capture (EC): errors that are identified by the system and that should be notified to users, but for which there is no possible remedial action. For instance, when a file is corrupted.

Supported error handling (EH): errors that should be corrected by the user, with system support. For instance, presenting an error message and an opportunity for the user to correct the error.

Figure 3 presents a diagrammatic representation of the interaction model for the goal "search documents".

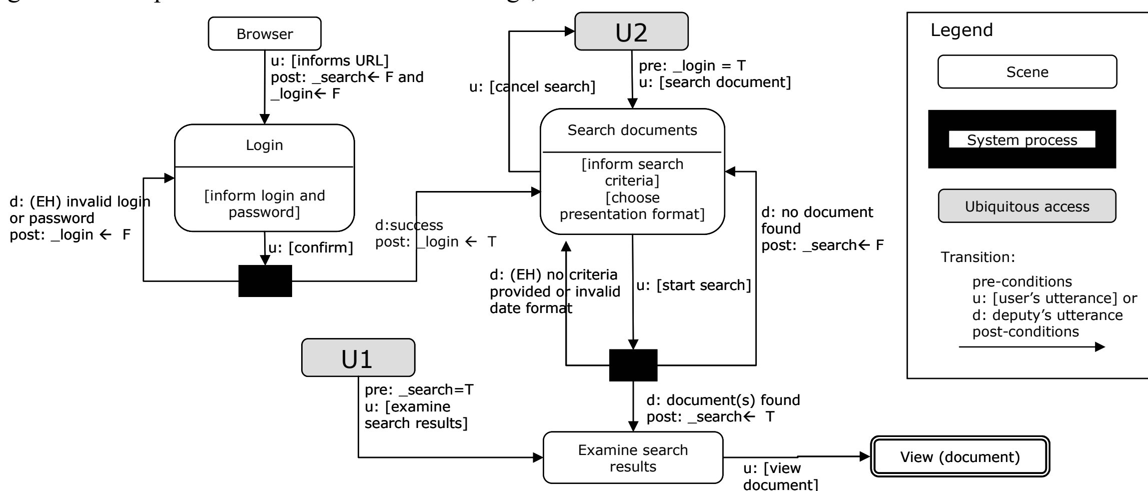

Figure 3. Sample abbreviated interaction diagram.

In order to illustrate the benefits of using MoLIC, we now exemplify two kinds of decisions made based on the diagrammatic interaction model, in two distinct situations, both related to searching:

1. Negative search result. When no document is found in a search task, the initial designer's solution was to present a message informing that no document was found, and then take the user back to the Search documents scene, where he/she could try to perform another search (Figure 4). This solution, though common in Windows applications, was inadequate in the Web, where the cost-per-click is higher. The adopted solution then was to return directly to the Search documents scene (Figure 3), which should then be modified (during execution) to reflect the source of the incoming transition: 1) a user utterance requesting a search; or 2) a deputy's response, indicating that no document was found.

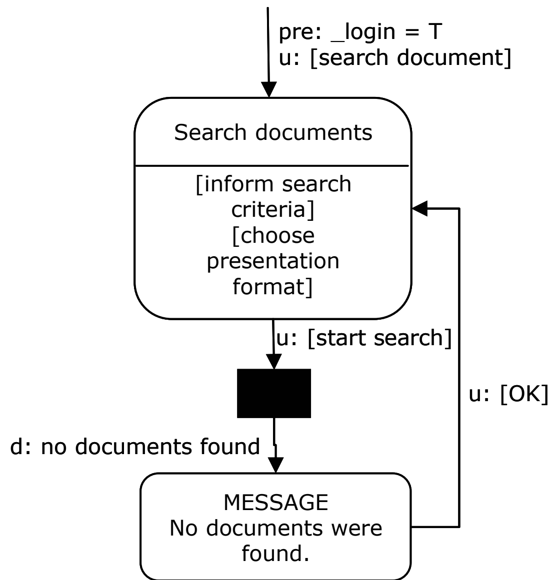

Figure 4. Alternative negative search result (no documents were found).

2. Alternative access to successful search results. When the search is successful, i.e., when there are one or more documents found, the design solution initially investigated consisted in allowing access to the search result right after the search, or from the visualization of one of the documents found (highlighted in Figure 5). However, designers and users found it useful to keep the list of found documents throughout the session, thus enabling future visits to this list (ubiquitous access to Examine search results, Figure 3). For this latter solution, it was necessary to introduce a _search control sign, which indicates whether there is a valid search result.

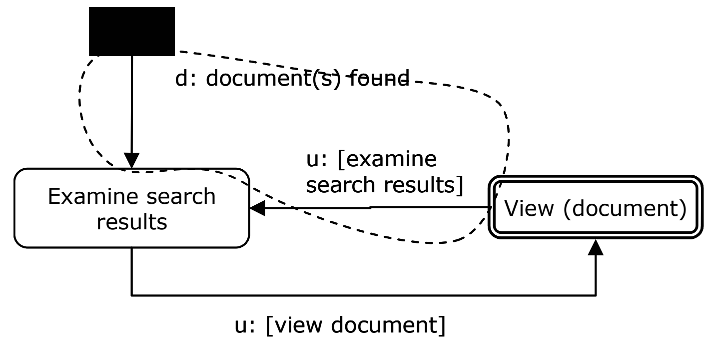

Figure 5. Alternative access strategy to search results.

The interaction model does not present details about interface or technological environment. However, designers must take into account such factors when making design decisions, for the characteristics of the computational environment in which the system will execute may determine whether a certain interaction solution is adequate or not. In other words, the *design decisions* depend on the interface style and technology, but not the *interaction representation*.

The design decisions made during interaction modeling may be presented to users for their appreciation. It may be necessary to generate new scenarios that reflect the proposed solutions, exploring the distinctions between them. For instance, in order to evaluate the users' opinions about the alternatives presented in Figures 4 and 5, scenarios were generated and presented to three users, who then worked with the designers to decide for the solutions depicted in the final interaction model (Figure 3).

4. Discussion

The goal of the work presented in this paper is to bring HCI concerns to software engineering. We argue that software engineering design models may be complemented with representations that favor the domain and task understanding (mostly scenarios), as well as the reflection about alternative solutions and interaction specification (mostly interaction model).

The use of HCI models in interactive systems design is essential for allowing designers to reflect about their solutions from the initial steps of the project on, and not just as a post-implementation evaluation. However, current HCI approaches usually tend to use complex task models which —sometimes awkwardly— intermingle issues of a different nature without calling attention to the fact that many diverse factors are at play which could be better analyzed from different perspectives. This makes it hard to understand and predict the effects of a certain solution. We believe it is best to use simpler representations, each with a well-defined focus: scenarios

focusing on establishing a common ground for mutual understanding designers and users, and MoLIC focusing on the discourse structures being designed.

No method has yet been developed that is capable of efficiently taking advantage of the results from the field of HCI into software development processes. On the one hand, scenarios provide contextual information about the application, from the users' point of view, but they do not map easily into software specification. On the other hand, storyboards and static visual elements (such as dialog boxes, field, and labels) provide concrete specification, but at a level of detail too fine to be of practical use during design or to support designers' decisions about the interactive solution. In general, design decisions made at one phase to augment the software's quality of use are lost due to the lack of an adequate representation at a later phase.

We believe MoLIC is a representation suitable for bringing HCI aspects to software engineering design. It may be used to represent scenarios in a semi-formal way, composing a blueprint of the system, from the user's point of view. This blueprint can then be used in later software design phases as input to what must be implemented and tested, interaction-wise. MoLIC also makes it easier to define an application's storyboards. Roughly speaking, each scene in MoLIC is a good candidate for a complex user interface element such as window or dialog box. The information needed to decide which widgets should be included in the storyboard comes from both the ontology of signs and the interaction model, which represent the nature of the information and the way it is to be presented or manipulated, respectively.

As we have said, interaction design in the semiotic engineering perspective is concerned with building a coherent and cohesive message, focusing on the users' understanding of the designers' conception of the application. We believe that, if the designer's assumptions and design decisions are conveyed successfully, with few communicative breakdowns or misunderstandings, i.e., through a well-designed user interface, users will not only make better use of the application, but also have more resources to use it creatively when unanticipated needs arise.

A decisive factor in conceiving MoLIC was to keep it as independent of specific user interface issues and technological platform as possible. This consideration not only facilitates the (re)use of models in various stages of software design and development, but also avoids that decisions about the user interface are forced to be made prematurely, making it harder for designers to explore different alternative solutions.

MoLIC has been used in the design and redesign of interactive systems in different environments. It has allowed a deeper and more organized understanding of the domain, and promoted the reflection about alternative design solutions, offering better support for the design team's decision making processes, before proceeding to the interface specification, storyboarding and implementation. Although reflection occurs when any design model is used, existing models generally allow the representation of solutions at too low an abstraction level, where important aspects of the solution are omitted, such as the representation of alternative or failure courses of action, and the communicative exchanges that take place between user and designer's deputy. In particular, this representation of "conversation" allows designers to perceive more clearly the effect of their design decisions, and thus makes it easier to determine which solution is adequate for the situation at hand.

We are currently investigating the integration of MoLIC with system specification models used in software engineering, aiming at augmenting HCI design quality without causing a negative impact on the software development cycle. Also underway there is a study about the use of MoLIC for building HCI patterns [9]; and another one for modelling synchronous multi-user environments.

5. References

[1] CACM (2002) Ontology: different ways of representing the same concept. Communications of the ACM, Volume 45, Issue 2 (February 2002).

[2] Carroll, J. M. (ed) (1995). Scenario-based design: envisioning work and technology in system development, New York, Wiley.

[3] Carroll, J. M. (ed) (2000) Making use: Scenario-Based Design of Human-Computer Interactions. The MIT Press. Cambridge, MA.

[4] Carroll, J.M.; Mack, R.L.; Robertson, S.P.; Rosson, M.B. (1994). "Binding Objects to Scenarios of Use", International Journal of Human-Computer Studies 41:243-276.

[5] de Souza, C.S., Barbosa, S.D.J., da Silva, S.R.P. (2001) "Semiotic Engineering Principles for Evaluating End-user Programming Environments", Interacting with Computers, 13-4, pp. 467-495. Elsevier.

[6] Imaz, M. & Benyon, D. (1999) How Stories Capture Interactions. Proceedings of IFIP TC.13 International Conference on Human-Computer Interaction, Interact'99. pp. 321–328.

[7] Norman, D. e Draper, S. (eds., 1986) User Centered System Design. Hillsdale, NJ. Lawrence Erlbaum.

[8] Paula, M.G. (2003) "Projeto da Interação Humano-Computador Baseado em Modelos Fundamentados na Engenharia Semiótica: Construção de um Modelo de

Interação" (in Portuguese). Masters Dissertation. Informatics Department, PUC-Rio, Brasil. March, 2003.

[9] Paula, M.G. & Barbosa, S.D.J. "Bringing Interaction Specifications to HCI Design Patterns". CHI 2003 *Workshop on Perspectives on HCI Patterns: Concepts and Tools*. Fort Lauderdale, Florida, April, 2003.

[10] Peirce, C.S. (1931) Collected Papers. Cambridge, Ma. Harvard University Press. (excerpted in Buchler, Justus, ed., Philosophical Writings of Peirce, New York: Dover, 1955)

[11] Schön, D. (1983) The Reflective Practitioner: How Professionals Think in Action, New York, Basic Books.