ABSTRACT
Semiotic Engineering uses semiotic theories to characterize human-computer interaction and support research and development of interactive systems. In order to show the value of Semiotic Engineering in design, we illustrate how semiotic concepts have been used in the analysis and generation of redesign alternatives for a web browser-based program called CoScripter. We also discuss how specific perspectives and expectations about the design process can increase the benefit from Semiotic Engineering in design activities, and describe our future steps in this research.

Categories and Subject Descriptors
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Human Factors.

Keywords
Semiotic engineering, graphical user interface design, UI design methodology, end user programming.

1. INTRODUCTION
Contemporary practitioners of user interface design are familiar with user-centered design, which focuses on how a user interacts with a software program. In this paper, we instead focus on how a software designer communicates with a user through the software's interface. This shift in perspective results in valuable insights that can assist designers in making design tradeoffs. We exemplify this value by analyzing the interface of one specific program, named CoScripter.

CoScripter is a system for sharing "how-to" knowledge about activities in a web browser. It consists of an extension to the Firefox web browser (Figure 1) and an associated wiki website. The extension enables users to record their actions in the browser as scripts: when the ‘record’ feature is turned on, CoScripter records a script of every action performed in Firefox – every click, every command, and any text that is typed. CoScripter can then replay this script, performing all of the clicks, commands, and typing automatically. These recorded scripts are stored on a public wiki, so that all users can replay scripts created by others.

Scripts can be used to automate many tasks, such as filling in forms. Scripts can also be used as a kind of "how-to" knowledge: as CoScripter replays actions, it highlights each action on the screen, turning buttons and links green as it clicks on them, and typing green text into text boxes. That way, users who have never performed a particular task can learn by watching the script as it executes. In Figure 1 we see the instruction for entering the user’s area code (‘650’) into the Area Code: text box. On the web page, green highlighting around the text box and the green text show the user what CoScripter is doing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2008 ACM 1-978-0-0558-141-5…$5.00.
involved in the design problem space [11] and how these bring about solution paths and alternatives that may not arise from user-centered design methods. Concrete illustrations from redesigning the CoScripter sidebar interface language provide support for the points we make.

We begin with a brief overview of Semiotic Engineering. We then present elements of CoScripter's interface design and analyze them in terms of Semiotic Engineering. Next, we take the results of our analysis as input, and show how a designer can apply Semiotic Engineering concepts to generate interface alternatives for CoScripter. Finally, we summarize how the insights gained from a semiotic perspective can be practically useful in HCI design, discuss how specific design perspectives can boost the benefits of our approach, and point to future work in this research project.

2. A BRIEF OVERVIEW OF SEMIOTIC ENGINEERING

A prime goal of semiotic investigation is to understand how people use signs to communicate [6]. Semiotic Engineering provides a semiotic account of human-computer interaction, stressing the fact that designers of interactive software communicate their design vision to users through their systems. They express their message through the signs in the interface – its words, icons, graphical layout, sounds, and widgets (e.g. buttons, links, and dropdown lists). Users then discover and interpret this message as they interact with the system. Since the designer can't personally be present when a user interacts with the software, the signs in the interface, along with their dynamic behavior, are the sole means available to the designer to get the user to understand what the software does, and how to use it.

It is important to realize that although systems' interfaces share a number of interactive patterns (e.g. many systems use pop-up dialog boxes when communicating about opening and saving files), every system has a unique interactive language whose semantics is determined by the system's unique semantic model. This language, which must be learned by users in a process that resembles second-language acquisition, is also delivered through the interface, as an important part of the designer's message to users. In other words, the message contains the very code in which it has been signified (by the designer) and that must be interpreted when these signs co-occur with other signs that show them to be incomplete, incorrect, or simply related to some other sign that we had not been aware of before. For instance, most GUI users know that 'close window' refers to 'close window'. However, some users may take this to mean 'exit program', or even 'cancel / undo action that took me here to (this window)'. Such a reinterpretation may work just fine in some situations, but it may also put the user into trouble in others (for example, the window may disappear without the program terminating and without the action being cancelled).

The process of constantly generating and revising meanings – called semiosis – has been formally defined by Peirce [10] as a particular kind of logical reasoning, called abduction. To Peirce, abduction is the primary sense making ability that we all share. It permeates human intelligent behavior, from simple common sense reasoning to sophisticated knowledge discovery in science and philosophy. Thus, in this perspective, users will surely generate use meanings that differ from design meanings. And this moves the design target from encoding the correct range of users' meanings to encoding meanings that communicate and achieve goals that are useful and enjoyable to users, in both anticipated and unanticipated situations. The designer's best choice is then to design a system of signs that is easy to learn (the cognitive dictum), but also efficient and effective in communicating meanings back and forth along the designer-system-user path (the semiotic dictum).

To facilitate the users' learning of interface sign systems, designers must cue the interpretations they expect from users by introducing signs that have the potential to trigger consistent abductions in the users' minds. Cuing expected interpretations is what all communicators do in a wide variety of contexts, such as teaching, politics and marketing, to name but a few. Therefore, the semiotic engineering of interface sign systems (or languages, in a computational sense), involves the use of rhetorical strategies that lead to certain cognitive effects, but are not cognitive in essence.

An important strategy in communicating the design vision to users, and concomitantly the interface language in which it is expressed and must be put to use, is elaboration. Elaboration amounts to communicating further details like circumstances, explanations and relations (e.g. analogies and contrast). Another
widely used strategy is redundancy, which amounts to repeating the same meanings (in whole or part) in various related situations. This achieves the cognitive effect of reinforcing certain sense-making patterns that will be useful to get the designer's message across.

Finally, there are dozens of classes and sub-classes of signs that can be manipulated to help communicators achieve their intent. Three classes of signs – symbols, indices and icons – will probably sound familiar to an HCI audience. These terms have lost most of their original theoretical tenets when they crossed the disciplinary boundary between Semiotics and HCI. But because they will play a role in our redesign of CoScripter's sidebar language, we will briefly retrieve some aspects of their original definition by Peirce [10].

This classification applies to how the interpretation of a sign is cued by its representation. Icons are signs whose interpretation is cued by representations that resemble (i.e. that reproduce the perceptual experience caused by the presence of) what they mean. So, for example, a pair of scissors can be classified as an icon in all contexts where it is used to mean 'a folder'. Indices are signs whose interpretation is cued by representations that have a logic, conceptual, causal, or other relation with what they mean. For example, a representation of a 'cut' (which is itself an indexical representation of what this popular editing function really is – it represents this function's meaning by analogy). Symbols, at last, are signs whose interpretation is cued by representations that are associated to certain meanings by convention. Thus, a symbol that by definition means 'close current window'.

3. REDESIGNING COSCRIPTER

In this section we will show how Semiotic Engineering concepts have been used first to analyze the CoScripter sidebar interface language (SIL), and then to design an alternative interface language (SIL*). SIL* has not been implemented in a prototype or tested experimentally. As a result, we only include an analytic evaluation of the communicative qualities of SIL* compared to SIL. This evaluation, nonetheless, is sufficient to highlight the main contributions of Semiotic Engineering in design tasks.

3.1 The Sidebar Interface Language

A redesign of the CoScripter sidebar is motivated by the fact that users have been confused by the current interface language. The first step in the semiotic engineering of an alternative interface is to retrieve the gist of what the designers want to communicate to CoScripter users. The second step is to analyze the sign classes and sign system structures that they are using to convey their message. The third step is to produce alternative classes and sign system structures – based on analytic properties – that should make the sidebar language easier to learn and more efficient and effective in communicating the designers’ intent.

3.1.1 Messages from the Designers

There are two important methods for deriving the designers’ message to users. One, of course, is to ask them to say what the message is and to discuss how they are expressing it through the interface. The other is to inspect the interface for signs that emerge with interaction, to contrast them with the system documentation and help, and reconstruct the designers’ message [5]. In this study we have used both methods, and concluded that the gist of what CoScripter's designers intend to communicate to users can be summarized in 10 messages, where ‘we’ means ‘we, the designers’ and ‘you’ means ‘you, the users’.

a) We don’t have much space, so we will try to communicate as much meaning with as few signs as possible.

b) CoScripter is a system you can use to record and run (play back) Web scripts, and share them with a community of users. The scripts’ content is represented in English-like language to make it easier for you to follow the actions they execute.

c) Scripts can be edited and saved (including saved as another script, with a different name).

d) Scripts can be public (i.e. shared with the whole community of CoScripter’s users) or private (i.e. saved for your usage only). We think you will be willing to share most of your scripts with others.

e) When editing a script, new or not, take advantage of the automatic recording facilities in CoScripter. You can, however, type script commands directly in the scripting language.

f) If automatic recording is not ‘on’, you can switch it ‘on’ and then ‘off’ at any time while editing your scripts.

g) You can run your saved scripts in two modes: ‘non-stop’ or ‘step-by-step’.

h) You can stop recording and playback at any time.

i) You can abandon script recording and editing at any time, and you can also delete saved scripts.

j) More information from designers and from the community of users can be found in CoScripter’s Wiki and Forum.

From (a) to (j) we see different categories of content, some of them present in more messages than others. The first category is the system’s purpose – to record and run Web scripts, and share them with a community of users. (b) is an explicit message in this category; (d) and (j) communicate implicit messages in this category, too. The second category is the system’s functions and operations. Messages (c), (f), (g), (h), and (i) communicate just this kind of content. Message (j) also fits in this category, as does the second part of messages (b) and (e), regarding the use of the script language and text. Finally, the third category of messages is the designers’ intent, values and expectations. Messages (a), (d) and (e) express explicitly the designers’ attitude and values, marked by the presence of “we” (in (a) and (d)) and by the word “advantage” (in (e)). In passing, we should note that the first two categories are basically objective, in that they are talking about the object of design, whereas the third is basically subjective, in that it is talking about the subject of design. Compared to user-centered design approaches, this is an important contrast.

There are, of course, other ways to phrase and organize the content expressed in the messages above. For instance, one way would be to constrain message phrasing so that each message conveys just one content unit, and fits into just one of the three proposed categories. In this way, there would be no overloading...
of communicative expression, with more than one unit of content and/or intent per message. However, there are basically two advantages in allowing for such overloading. One is that this kind of phenomenon is pervasive (and arguably unavoidable) in natural communication, and thus it could and should be explored in computer-mediated communication as well. The second is that this helps us address from the very start of the design cycle an important constraint in CoScripter – very little screen space for communicating all 10 messages. Overloading is a possible strategy that can be used to solve communicability problems.

Each message can be analyzed further in terms of traditional semantic categories borrowed from natural languages and used in many representation languages, including programming and interface languages. These are: actor, action, object, and modifier. So, for example, ‘script’ is an object, and there are two modifiers that can be applied to it: ‘public’ and ‘private’. ‘Run’ is an action, to which two other modifiers can be applied: ‘non-stop’ or ‘step-by-step’.

Just as in natural language, however, these categories are sometimes recursive. For example, the action ‘edit’ accepts two modifiers that, unlike in the previous examples, are not atomic concepts (e.g. ‘private’, ‘non-stop’) but complex structures in themselves: ‘by actor=CoScripter action=record object=script’ and ‘by actor=User action=typing object=commands’. Moreover, all communication with CoScripter is determined by context. For instance, ‘stop’ and ‘cancel’ are not valid content elements in contexts where there is no action taking place. Likewise, message contents can be used to achieve different purposes in different contexts: ‘stop recording’ means something different in an execution context from what it means in an explanation context.

So, although CoScripter's communicative domain is highly restricted, it encompasses thoroughly complex communicative structures that must be effectively and efficiently expressed in SIL. Additionally, since screen space is small, communication cannot be verbose, and – as already noted above – single signs in the signification system will often communicate multiple message contents. As a consequence, the design of such signification system (SIL) must be done so that users will be able to learn and express these complex communicative structures as quickly and easily as possible. We can expect the ‘cognitive’ loads associated with the interface language acquisition process to increase as the underlying grammars move from regular to context-free, context-sensitive, and unrestricted [2]. But, especially because non-expert users (to whom CoScripter is targeted, by the way) may not be familiar with the processing constraints of computer languages, users may wish SIL to express far more than easier-to-learn languages (e.g. regular or context-free) can express [13]. In short, lower language acquisition loads run contrary to higher language expressive power, a design tradeoff that can best be formulated and framed when designers think beyond ‘usability’ (highly influenced by cognitive criteria) and into ‘communicability’ (highly influenced by semiotic criteria).

3.1.2 Signs and grammar of CoScripter’s SIL

CoScripter's sidebar appears in three alternative basic forms (see Table 1). Each form has its own sub-variations depending on local context values, as will be discussed later. Signs in SIL can be visual (like 🎧), textual (like ‘Welcome to CoScripter’), or hybrid textual and visual (like 🎧).

As a rule, hybrid signs express actions, some of which have modifiers expressing states like ‘enabled’/’disabled’ or ‘active/inactive’. Although related, these states are not synonymous. For instance, the ‘Run’ button is disabled when there is no script in the sidebar, since there is nothing to run. When a script is present, the ‘Run’ button is enabled, and it is ‘inactive’ when CoScripter is not currently running that script, and ‘active’ when CoScripter is running the script. Given this distinction, both pairs of modifiers must be signified in SIL.

Learning an interface language is facilitated when designers adopt patterns from similar or related languages. Thus, the expression of the modifier pair ‘enabled/disabled’ can be more easily learned if it is encoded using the gray color for ‘disabled’ (and other color(s) for ‘enabled’). There is also a common pattern for expressing the ‘active/inactive’ modifier pair, namely a toggle element. A toggle button for instance is ‘active’ when pressed down, and ‘inactive’ when up.

<table>
<thead>
<tr>
<th>Table 1. CoScripter's Sidebar Interface Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Sidebar Contexts</strong></td>
</tr>
<tr>
<td><strong>Start</strong></td>
</tr>
<tr>
<td><img src="http://example.com/start-icon.png" alt="Start Icon" /></td>
</tr>
<tr>
<td><strong>Edit</strong></td>
</tr>
<tr>
<td><img src="http://example.com/edit-icon.png" alt="Edit Icon" /></td>
</tr>
<tr>
<td><strong>Run</strong></td>
</tr>
<tr>
<td><img src="http://example.com/run-icon.png" alt="Run Icon" /></td>
</tr>
</tbody>
</table>

Because context is such an important dimension in CoScripter’s design, a context-sensitive grammar naturally comes to mind when designing SIL. However, because context-sensitive grammars also impose higher cognitive loads on learners than context-free and regular grammars, the designers of a signification system must be especially attentive to regularity in their underlying grammar. For example, in row 2 of Table 1, designers are communicating to users a number of things:

- That CoScripter is recording the user’s actions in the web browser (the ‘record’ button is toggled down).
- That the recording can be stopped (if the user toggles the ‘record’ button up).


That the designers think it is advantageous to start editing by recording actions (the default state of the ‘record’ button is active, without the user’s telling the system to record anything).

If the user gets all of these messages right, then she is likely to infer (by abduction, as discussed in the previous section) that actions can be stopped or initiated by pressing toggle buttons corresponding to them. However, if she tests this inference when running scripts, she will find out that this is not the case, because the grammatical pattern for running differs from the grammatical pattern for recording. That is, in row 3 of Table 1 we see that there is a ‘stop’ button in the interface. Instead of using a single toggle button, running is instead implemented with one button to start running, and a different button to stop running. Therefore, users must not only learn another rule to express the same communicative type of active/inactive, but they must also learn which rule is used in which context. The absence of a ‘stop’ button in the editing interface facilitates the inference that in order to stop recording, the user must press the ‘record’ toggle button. But the presence of the ‘stop’ button in the execution interface might lead to other kinds of equally valid inferences. For example, the user might suppose that by pressing ‘stop’, she would quit the execution mode and go back to the initial state of the system (why not?). So, it would make perfect sense to her to immediately decide to cancel this activity and clicks on the ‘cancel’ button. This change of mode coincides nicely with the underlying SIL grammar. As a consequence, the user must incur the additional cognitive cost of learning grammar rule exceptions and specificities by trial and error.

**Context** is the most difficult dimension to master in SIL. The user views context in terms of high-level conceptual tasks, which are termed modes. The main modes for the CoScripter user are starting CoScripter, editing a script (by recording or typing), and running a script. The actual program that implements CoScripter has a corresponding context, termed states. Unfortunately, despite being similar, CoScripter’s states are not identical to the user’s modes, and this shift in ontology may give rise to a lot of confusion.

Consider a user who wants to ‘cancel’ her activity in one mode and return to the previous mode. For example, suppose the user starts CoScripter (Table 1 row 1) and then transitions to creating a new script (Table 1 row 2) by clicking the ‘New’ button. The user immediately decides to cancel this activity and clicks on the ‘cancel’ button. This change of mode coincides nicely with CoScripter’s states, and causes a transition back to the start mode (Table 1 row 1).

Now consider the example of a user who wants to ‘cancel’ her activity in the run mode and return to the edit mode. The user is editing a script, saves it, runs it, and sees that something went wrong. There is no ‘cancel’ button in the context shown in Table 1 row 3, so she cannot tell the system to ‘cancel’ run mode and return to the previous edit mode. Instead, the interface only offers an ‘edit’ button, whose semantics implies not “returning to” the previous edit mode, but rather "progressing to" a new edit mode. At this point she may already wonder why ‘cancel’ is not uniform, and this introduces the potential for wrong abductions in sense making.

### 3.2 Semiotic Engineering in design activities

In the previous section we showed how semiotic concepts can be used to analyze the cognitive and communicative tradeoffs associated with various design choices in the signification system that conveys the designers’ messages. Can Semiotic Engineering help design a better signification system for CoScripter?

An answer to this question actually involves more than just elaborating a different SIL. It involves choosing the appropriate evaluation criteria that will help designers (i) anticipate that they have found a better solution, and (ii) verify that their anticipation is actually correct. Whereas we can think of analytic criteria that can adequately account for anticipation, only user studies can adequately account for verification. Because the research reported in this paper is still in progress, we will concentrate on anticipation and save verification for further stages in the project.

Going back to the ten messages that designers want to communicate with SIL (cf. (a) to (j) in subsection 3.1.1), we see that SIL communicates different content categories in different ways. **System’s functions and operations** are communicated directly with hybrid signs composed of a visual form and a textual label. Because these functions and operations are associated to specific modifiers and contexts, the signs that express them have different form inferences (e.g. a sign may appear in gray color to express “disabled”, or in full color to express “enabled”) and different distributions of occurrence (e.g. the ‘cancel’ sign co-occurs with the ‘save’ sign, but not with the ‘run’ sign). **Inflection and distribution of occurrence** are used to communicate the designers’ intent, values and expectations. Together, they are components of indexical representations of important meanings. For example, distribution of occurrence is used to express the designers’ concern about optimizing screen space use in SIL. **Visibility and invisibility of representation** (which signal distribution) signify lack of space. Also, inflection is used to express the designers’ belief/expectation that certain default values will call the users’ attention to the advantages of using CoScripter in one way, rather than the other. Here, a mix of iconic and symbolic signs is used to convey the intended message (e.g. button states resemble those in most electronic appliances, and many of their control conventions are adopted in CoScripter).

The use of distribution to resolve design tradeoffs, however, comes at the expense of communicating the **system’s purpose** more directly. For example, instead of communicating all the main purposes of the system simultaneously (recording, running, and sharing scripts), designers of SIL chose to distribute communication across different interactive modes. Modes, however, constitute a cognitive challenge for interface language acquisition. Can Semiotic Engineering improve SIL in view of these givens and findings, and reduce cognitive load in the process?

#### 3.2.1 A new context-sensitive grammar for SIL

In this subsection we present only the sign types and rule types that can improve the designer-to-user meta-communication and facilitate the users’ learning. A full-fledged grammar specification is not only tedious, but also beside the point of this paper. All improvements proposed for SIL are meant to eliminate ambiguities, facilitate learning, and preserve the communication of the designers’ intent, values and expectations.
3.2.2 Using inflection and distribution to express context transitions

Semiotic Engineering’s focus on signification systems and communication processes directs designers to consider their sign choices in terms of what they communicate (and mean) to users, what users may mean by them, and what users take them to communicate back to the system. In redesigning SIL we can first choose sign inflections and sign distribution to be indexical representations of context. Next, by building a consistent set of rules to control the use (and computational interpretation) of such representations, we can produce conventions (or symbolic representations) that mean and communicate the designers’ intent.

As shown in Figure 2, there are two classes of buttons in SIL* (as we call this redesigned language): pulse buttons and toggle buttons. Each class has inflections. Because pulse buttons express instant actions that have no duration over time, ‘active/inactive’ distinctions are unnecessary. Inflection must only account for the ‘enabled/disabled’ distinction. Toggle buttons, however, must communicate ‘active/inactive’ distinctions in order to account for duration aspects of certain actions over time.

![Pulse button inflections and Toggle button inflections](image)

**Figure 2:** Classes of buttons and their inflection in SIL*

Context can also be signified by distribution, as in the original SIL design. In SIL* distribution plays a major role in communicating, indexically, the system’s mode (and the interactive state – the user’s current state of discourse or conversation). Figure 3 illustrates how button inflection and distribution combine to express complex contextual dependencies in SIL*. Note that there are only three signs expressed as toggle buttons: ‘record’, ‘run’ and ‘wiki’. The ‘edit’ sign is a synonym of ‘record’, which will be discussed in the next section. Note, also, that button inflection is being used to communicate a previous state of the system. When the ‘record’ function is active and the user chooses to activate the ‘wiki’ function, the system can preserve the interactive context prior to entering CoScripter’s wiki. This feature is not currently implemented in the system, but the example serves to show how a relatively simple set of signification patterns can account for fairly elaborate human-computer interaction. The ‘home’ button is there to facilitate access to the system’s initial state, where users can choose to ‘record’ a new script, or ‘run’ an existing script. Again, this latter feature is not implemented in the current system. In many interactive contexts, in order to run an existing script users must follow a long and complicated communicative path, through the wiki, with many opportunities for error. SIL* gives faster access to such scripts, and helps prevent wiki navigation errors.

![Five states of SIL*, showing transitions when the user communicates to the system](image)

**Figure 3:** The five states of SIL*, showing transitions when the user communicates to the system.

3.2.3 Some analytic indicators of improvement

The SIL* grammar is presented here with impoverished visual elements, compared to the signs actually used in SIL (see Figure 1 in the Introduction). On the one hand, this helps the reader focus on the crucial semiotic elements of the interface. It does not mean, however, that ‘run’, ‘record’, ‘stop’, and the like cannot or should not be expressed as images, or as a combination of image and text. On the other hand, the introduction of visual signs is in itself a fairly complex Semiotic Engineering task. What images should be used? How should they be rendered in inflected button forms? These issues will not be discussed here for sake of brevity and clarity.

Although SIL* is still a context-sensitive grammar – hence more difficult to learn than a context-free or a regular grammar – it explores inflection and distribution regularities to help users ‘infer’ how to interpret and express communication in context. For example, guessing the effect of communicating ‘Run’ when the user is in the context represented by the top pair of sidebar representations in Figure 3 is not difficult. Neither is it difficult to infer the effect of communicating ‘Wiki’ when the user is viewing a sidebar like the one at the bottom of Figure 3. Note that the redesign alternative builds a convention for how indexical signs are used in the interface. Note also that if we add images, it is likely that visual representations will introduce redundancy into this communication and reinforce the design messages.

In its current form, nonetheless, SIL* sorts out some of the confusion between mode and state, mentioned in 3.1, but it requires the implementation of additional semantics compared to SIL. Therefore, designers must trade off superior semiotic and cognitive quality, against increased programming costs.

Another analytic indicator of improvement in SIL* is the introduction of a visual grammar – a conventional sign (or symbol) – in the sidebar layout. There are constantly three different segments in the sidebar, separated by ‘|’. The leftmost segment constantly communicates the two main operations in CoScripter: record and run scripts. Both are toggle buttons, constantly enabled to signal a switch of context. The ‘record’ button is aliased to ‘edit’ in the context of ‘run’. This choice breaks the visual and lexical regularity of other signs in SIL*. However, this is a conscious design choice, an index intentionally introduced to tease users into wondering whether there is more than ‘recording’ involved in building and refining scripts. And
With respect to these analytic criteria, we can state that SIL*’s design is analytically superior to SIL’s in terms of communicability and usability. However, this analysis is decidedly not empirical. While only user studies can empirically verify that SIL* actually improves the whole process of computer-mediated communication taking place in HCI, Semiotic Engineering adds a valuable tool for the design process by enabling designers to conclude analytically that one design has a simpler and more consistent signification system than another.

4. CONCLUSION

This paper reports on work in progress, and we are focusing exclusively on a specific component of the CoScripter interface – the sidebar. The whole interface actually involves other languages, such as the scripting language, the wiki interface language, the browser interface language (especially in playback mode, when recorded actions are signified in the browser’s interface), and natural language (present in a considerable extent of web material and throughout the CoScripter wiki and users’ forum). Nevertheless, even in this constrained context, we demonstrate Semiotic Engineering concepts and show how they can concretely impact design decisions.

The Semiotic Engineering approach to redesigning SIL is not meant to be taken as a design method. There are two main reasons for this. First, new design methods require extensive and laborious field research from which we can derive the recurrent conceptual and procedural steps that constitute the method’s original contribution to design practice, and that distinguish it from other existing design methods. Second, this theory can clearly be used in combination with a well-known design approach and associated methods – Schön’s reflection in action [11].

Schön’s approach stresses the fact that every design is a unique problem, whose first solution step is actually to name and frame the elements that, according to the designer’s interpretation, are part of the unique situation that they are dealing with. This view is in sharp contrast to other design traditions in which, for example, design problems are seen as instances of general problem types that can be solved by a systematic application of pre-established problem-solving procedures [12].

When we look at the theoretical roots of Semiotic Engineering, it is clear that viewing meaning as a constantly evolving sign-producing interpretive process is at odds with the idea that design methods can be used to ensure that the product will be interpreted in one way or another. Thus, Semiotic Engineering should not be used to predict how users will interpret the designers’ message. In fact, there is no theory that can do this.

The advantage of using Semiotic Engineering in combination with reflection in action is that both emphasize the role and the value of knowledge generation in the design process. Schön believes that the most important requirement in design is what he calls ‘an epistemology of practice’. By this he means a set of practical tools to help designers produce and evaluate design knowledge (‘episteme’ in Greek) that continuously arises and intervenes in the design process. Semiotic analysis and concepts help designers organize this knowledge according to communicative and interpretive categories, and to explicitly formulate what they mean by (i.e. how they name and how they frame) design
elements in an attempt to bring up certain features and effects when the design product is used.

Because Semiotic Engineering has a clear-cut characterization of HCI (as a designer-to-user communication about how to communicate with a system in order to achieve a certain range of effects) and provides an ontology for analyzing the elements involved in this process (e.g., sign classes, signification systems, and communication processes) it can boost the advantages of adopting Schön’s perspective in design. Conversely, Semiotic Engineering has a great impedance with design methods that are expected to predict how users will react to the presence of certain signs in the interface, and with those that seek to generate universally applicable solutions.

Nevertheless, even in the absence of predictive statements, Semiotic Engineering can be used to improve design choices in important ways. In section 3.2 we have shown how knowledge about sign classes and properties of signification systems helps designers understand and (re)formulate the role and the function of interface languages as communication tools that are crucially important to achieve even the most traditional usability goals (like ease of learning and ease of use, for example).

In talking about Semiotic Engineering, we have mentioned certain communicative strategies, such as elaboration and redundancy, that communicators can effectively use to bring about the desired interpretations in the interpreters’ minds. Throughout the paper we have provided instances of the use of redundancy in designing and redesigning SIL, but not of elaboration. Elaboration requires further communication, and given the spatial constraints of CoScripter, elaboration is a major challenge for the designers. SIL's hybrid signs (a composite of visual and textual material), as well as the tool tips that users view when they position the mouse on certain interface elements, are instances of elaboration. Signs of a given signification system are being elaborated (explained or described) by signs of another. The signs in the first system are invitations to the user to explore the interface and learn how to use it. However, this is only a timid attempt at meta-communication – communication about communication. So, one of the items in our future work agenda is to explore the design space for elaboration messages, in the form of online help, explanatory question-answering, parallel signification systems, and the like.

Another important step is to include other CoScripter interface languages in our analysis and redesign efforts. Among such languages the main ones are: the scripting language, in which recorded steps are codified in the sidebar, and which users can edit to introduce or modify certain steps in execution; and the playback preview language, in which CoScripter signals the interactive steps that are being performed in the browser’s interface as users run a recorded script. Together, the scripting language, the playback language, the sidebar interface language and even natural language (used to elaborate on the messages conveyed by all of the other languages) must be consistent and cohesive with each other. Such is the complex semiotic engineering challenge of a relatively small system like CoScripter, which we have used to demonstrate and improve this new theory of HCI.

5. ACKNOWLEDGMENTS

The authors thank Laura Haas for establishing the Visiting Researcher Program at IBM Almaden Research Center, which provided the means to carry on the studies reported in this paper. Clarisse de Souza thanks CNPq for sustained support of her research, and her graduate students at PUC-Rio for providing numerous examples of interface challenges in CoScripter.

6. REFERENCES