Developing a Reflective Model of Collaborative Systems

PAUL DOURISH
Rank Xerox Research Centre

Recent years have seen a shift in perception of the nature of HCI and interactive systems. As interface work has increasingly become a focus of attention for the social sciences, we have expanded our appreciation of the importance of issues such as work practice, adaptation, and evolution in interactive systems. The reorientation in our view of interactive systems has been accompanied by a call for a new model of design centered around user needs and participation. This article argues that a new process of design is not enough and that the new view necessitates a similar reorientation in the *structure* of the systems we build. It outlines some requirements for systems that support a deeper conception of interaction and argues that the traditional system design techniques are not suited to creating such systems. Finally, using examples from ongoing work in the design of an open toolkit for collaborative applications, it illustrates how the principles of computational reflection and metaobject protocols can lead us toward a new model based on open abstraction that holds great promise in addressing these issues.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; D.2.10 [**Software Engineering**]: Design—*methodologies*; H.1.10 [**Models and Principles**]: General

General Terms: Design

Additional Key Words and Phrases: Collaborative applications, computational reflection, metaobject protocol, open implementations, system architecture.

1. INTRODUCTION

The last ten years or so have seen a remarkable shift in perspectives on the design, evaluation, and use of interactive systems. The field of human–computer interaction (HCI) has moved from being a relatively minor component of software engineering to being the focus of attention for researchers from a variety of disciplines, including psychology and social science. Studies and investigations from these perspectives have led to a gradual evolution in our conception of "the interface" and of computer-based work in general. As a result, HCI has increasingly come to concern itself not just with the *mecha-*

Author's address: Rank Xerox Research Centre, Cambridge Laboratory (EuroPARC), 61 Regent Street, Cambridge, CB2 1AB, UK; and Department of Computer Science, University College, Gower Street, London, WCIE 6BT UK.

nism of the interface, but with a range of related issues concerning the context in which interactive systems are used.

1.1 Studies of Work at the Interface

To ground discussion of this new view of interactive systems development, I will discuss three areas of research that have informed it: (1) the customization of interactive systems, (2) their embedding within a social organization, and (3) the coadaptation of systems and work practices.

1.1.1 *Customization.* Customization and adaptation of computer systems have been studied in a variety of contexts. Trigg et al. [1987] studied adaptation in the Notecards hypertext system. They described four aspects of adaptability that could allow a tool to be used in different application areas by users with different working styles. These were *flexibility* (providing generic, reusable objects and behaviors), *parameterizability* (offering a range of alternative behaviors that users could select), *integrability* (linking with other applications in the environment), and *tailorability* (allowing users to make changes to the system itself). Their work showed how an adaptable system could be applied widely, essentially serving as an infrastructure within which a variety of information management applications could be generated.

MacLean et al. [1990] were also concerned with customization in the Buttons systems. Buttons are graphical on-screen objects that encapsulate behavior; they can be incorporated into on-line documents and sent through electronic mail. Buttons support multiple levels of customization. At the simplest level, their name, placement, or graphical appearance can be customized. More-advanced users can change explicit parameters to customize them for particular needs, and more-radical changes in their behavior can be made by modifying the Lisp code that they contain. Buttons were designed around these levels of tailorability to flatten the learning curve for interactive systems. Establishing a "tailoring culture," in which customizations are encouraged and shared, was also crucial to the investigation.

Sharing customizations (and customization activity) has been a focus of attention for other researchers. Mackay [1990a; 1991] studied the patterns of sharing customizations (in the form of configuration files, either in whole or in part) in technical organizations. She found that a remarkable amount of customization was performed (or shared) in groups, rather than being a private activity, even when the software being customized was single-user. Nardi and Miller [1991] showed that similar collaborative activity goes on in the notionally "single-user" activity of programming and debugging spreadsheets.

Customization is clearly a widespread and important phenomenon. The studies highlight the importance of designing systems with an understanding of how such flexibility can be harnessed and exploited.

1.1.2 *Social Aspects of System Use.* Ethnographic studies of work practice and technology have pointed to the strong social elements present in apparently individual use of technology. Heath and Luff [1991], studying the activities in a control room of London's underground transport system, observed a range of complex practices employed by the controllers. In particular, they showed how controllers would *peripherally monitor* each other's activities in order to coordinate their own actions with those of their colleagues. Heath and Luff further observed that individuals would quietly offer commentaries on their own activities, specifically so that these might be used by colleagues in their own work. Other uses of peripheral monitoring and anticipation in similar settings have been observed by Filippi and Theureau [1993] and Suchman [1992].

Harper et al. [1991] studied a different setting, in this case, the control rooms of air traffic control centers. Again, they observed that apparently individualistic activity is frequently designed to facilitate coordination between individuals. Their investigations pointed to the role that physical artifacts can play in this process. The "flight strips" used by air traffic controllers to record details of flights currently in their managed air space are notionally designed to record information for an individual. However, practices emerge around such artifacts in which they become the focus of coordination activity.

Suchman [1987] discussed these issues more generally, concentrating in particular on the situatedness of technology and the use of technology. Technological interactions are embedded within social and organizational situations, and their influence must be taken into account when attempting to analyze or predict aspects of system interaction. These studies all call into question the extent to which interactions and activity can be predicted from an external viewpoint, and yet such predictions are at the basis of the traditional system development approach.

1.1.3 *Coadaptation of Systems and Practices.* The third aspect, drawing together elements from the previous two, concerns the longer-term mutual evolution of systems and work practices. Mackay [1990b] presented a variety of case studies illustrating this pattern, some of which were discussed above. Perhaps of most interest here is one study on Information Lens [Mackay 1989; Malone et al. 1987]. Information Lens is a system designed to filter and sort electronic mail. Investigations of use revealed that users adapted, or even *subverted,* features of Information Lens to match it to their working styles. This in turn led to a change in the development strategy, which incorporated and developed the mechanisms that the users had created. This is a spiraling process, based on the mutual interaction of system development and patterns of use—in Mackay's terms a *coadaptive* phenomenon.

Echoes of this unanticipated interaction between technology and working practice are to be found in Sproull and Kiesler's [1991] work on the organizational impacts of electronic communication. Such technologies are often construed as "electronic office memos," and subject to the same rules and procedures. However, the experience is typically that aspects of electronic mail—in particular, rapid turnaround and the emergence of self-organizing interest groups—result in a very different pattern of use, often much to the surprise of the organization. As patterns become better established, they affect the further development of the technology, and so the coadaptive spiral continues.

1.2 A Revised View of Interactive Systems

There is a set of general issues underlying the particular concerns raised in all of these studies. Together, they form part of a wider reappraisal of the nature of computer-based work. They encourage us to look beyond widget design in understanding interactive systems. For instance, focusing on the development of the notion of user interface, Grudin [1993] showed how it extends beyond the computer itself, incorporating the wider social and organization issues arising from the use of computer systems. Elsewhere, Grudin [1990] placed these developments within a historical framework of changing concerns in interface design. At the microlevel, these sentiments were echoed by Bowers and Rodden [1993], who found the same factors at work in a very specific setting, the installation of a large network of CSCW systems in a particular observation.

Studies of customization, of the social nature of computer-based work, and of the coevolution of systems and work practices all address issues in the organization of computer-based work practices. They highlight the strong relationship between these practices and the social organization of work generally. Computer-based work is highly socially organized. As a result, these studies place an emphasis on variability in use; they point out the complexity of the relationship between the general pattern of use and the details of particular activities.

It is not surprising, then, that studies such as these should result in a reorientation of our view of the fundamental nature of interactive systems, a reorientation that takes into account this expanded understanding of computer usage. Indeed, such a reorientation has been taking place. It moves away from a view of systems as fixed, "black-box" artifacts that could be objectively studied and assessed, toward a new view of "systems-in-use" which acknowledges the influence of these other elements. In the new view, systems are *situated* within particular organizations and practices; they are *dynamic*, placing greater importance in the study of patterns of use and the cycle of software adoption; and they *evolve*, with specific working practices and behaviors emerging around the interactive system, while at the same time the system itself being tailored to particular working styles. In other words, systems must be able to support the variability exposed by investigations like those discussed above.

This shift in perspective has largely addressed interactive systems *as they appear to the user*, in particular settings. Indeed, it has been part of a general trend toward participative or user-centered approaches to system building. However, it raises other important issues, particularly for the designers and implementors of interactive systems, issues which reach below the interface.

In this article I want to concentrate on the *structure* of the systems we create. I believe that the implications of our reconception of HCI force us to reconsider our notions of system building. This means changes not only to the *process* of design, but also, critically, to the *artifacts* of design. In particular, I will show how fundamental mechanisms that we use in constructing software systems do not support the design of interactive systems that change and evolve. Drawing from work done in other areas of systems development, I will show how the principles of *computational reflection* lead to a more open model of systems development, with revisability as a key element. Using examples from ongoing work in the design of an open CSCW toolkit, I will illustrate how this approach tackles a number of existing flexibility problems.

Our starting point for this investigation, then, is a question, What does the shift in our view of interactive systems imply for the nature of the systems that we design?

2. IMPLICATIONS FOR INTERACTIVE SYSTEMS DESIGN

Our increasing appreciation of the issues discussed above has been accompanied by a call for a form of design that is more sensitive to them, a form of design oriented specifically around user needs and user involvement (e.g., Norman and Draper [1986], Ehn [1988], and Bødker and Grønbæk [1991]). Following on from this, Grudin [1991] highlighted conflicts between the "user-centered" approach to design and the traditional systems development process.

Taking this as a starting point, I want to explore how we can construct systems which support the sorts of practices outlined above. Principally, this involves looking at technical implications of the new approach for interactive systems and then working toward an architectural model that addresses them. So, if we generalize some aspects of the "systems-in-use" perspective, we can identify two sets of consequences for their design, concerned with the cycle of design and creating evolving systems.

2.1 The Cycle of Design

The first of these is a reconsideration of the cycle of design. In the traditional "waterfall" model of software engineering [Royce 1970], the "design" of the system happens at a fixed point, after specification and before implementation. More recent models, such as those of Boehm [1988] or Booch [1991], adopt a more iterative, prototype-based approach. Here, "design" is distributed more evenly through the development process, not concentrated in one place.

Despite their differences, there is at least one point of fundamental agreement among all of these models. They state that, at some point, a product is *delivered* to a user community, at which point, for that revision of the software, the design process is over.

This is an assumption that we must reconsider in the light of the "systems-in-use" model. When we take this perspective, we are forced to ask questions such as, "When does design happen?"; "Who does the design?"; and "When does it stop?" When we look at an interactive system as an evolving artifact in use, it follows that the process of design does *not* end with the delivery of the system to some community of users. Instead, it continues as they use and adapt the system. This leads to a second and more focused set of concerns for system developers, looking at how systems are structured, constructed, and delivered.

2.2 Creating Evolving Systems

The developer of an interactive system must not only be concerned with the traditional issues of system design, but also with the issues of providing a system that is amenable to evolution and adaption. We can focus on three particular aspects of this problem:

Open Infrastructures. From the perspective of systems-in-use, we begin to see delivered systems as not being closed and static, but rather as infrastructures for further specialization, refinement, and end-user design. They provide a framework within which users can change and adapt the basic system to their own patterns of usage. The system developer is concerned with appropriate openness within the system and with ensuring that it lends itself to these adaptations. (The nature of "appropriateness" will be considered in more depth later; for the moment, we can consider "appropriate" as being "lending itself to appropriation.") Additionally, extending the model of customization, we must consider the ways in which the system can lend itself to customization of *function* (the "semantics" of manipulating information in an interactive system) as well as *presentation* (surface-level issues of views and interaction).

Dynamic and Reactive Systems. When we think of user behavior changing over time, then we must consider how the interactive system will support and respond to these changes. From this point of view, systems need to be designed to react dynamically to patterns of use and activity. The systems' response must be situated in the same sense as is the user's activity. Users' needs are highly dependent on many contextual factors, such as patterns of activity and changes in configuration. Similarly, system behavior should react to contexts of use.

Adaptive and Evolving Systems. In addition to the "immediate" view of system reactivity, the developer must also be concerned with the longer-term view of the evolution of the system. Research on customization and coadaptivity shows that this evolution has its roots in the social aspects of work and is enabled, in part, through the sharing of customizations. This implies that it is important to address issues such as the nature of the customization mechanisms, as well as the means by which they can be distributed and shared, and can themselves evolve over time—support for Maclean et al.'s "tailoring culture."

"Tailoring" in this context does not just imply the ability to make changes. It is also crucial that changes and adaptations be separated from core functionality in a principled way, while maintaining the "reach" of tailorability into that core. This separation is crucial if tailorings and adaptations are to be transportable, not just from person to person, but also across software releases. As software product cycles shrink, it is essential that users can rely on this. Without such a separation, a user is unlikely to make the investment of time and effort that tailoring involves, since the effects will be lost with the next software release.

The issue for system designers, then, is to develop a set of techniques for constructing software systems that enable the distribution of the design phase throughout the whole life cycle of a system, and which support software adaptation and evolution. This is a significant departure from the traditional focus of software design, a focus on developing fixed algorithms that manipulate models of the application domain. Instead, it focuses on the way in which such models might be constructed and manipulated—a metalevel problem, one step removed from the application domain itself.

3. REFLECTION AND OPEN ABSTRACTION

This view has developed from ongoing work in the design of systems for Computer-Supported Cooperative Work (CSCW). CSCW systems, by their nature, have very strong requirements for flexibility and openness. Customization may be performed not only by users, but by groups as a whole, and even a single group might employ a wide variety of working styles in the course of their work. These factors bring the system developer face-to-face with the issues of reactivity and adaptability. These problems are magnified for developers of *generic toolkits*. These are used to generate a variety of CSCW applications, which may embody different models of collaboration. As a result, a major goal of my work has also been to provide the application developer with sufficient flexibility to create a range of application styles. Reflecting the shift in design focus outlined above, the emphasis here is on a framework in which mechanisms and interactional styles can be *created*, rather than the traditional approach of providing a selection of mechanisms from which particular components can be *selected*.

The systems approach that I am developing is based on the principles of *computational reflection* [Maes 1987; Smith 1982] and, in particular, the *metaobject protocol* [Kiczales et al. 1991]. This approach is a crucial stepping-stone toward the goals of flexible design which were outlined above. It provides a way of incorporating the flexibility we need not only into the design process, but into the *artifacts of design* themselves; it is fundamentally about systems that are open to explicit change and adaptation. It's worth taking some time, then, to look at the mechanics of reflection, and the way it establishes a link between generic models of system action and the performance of that action.

In the rest of this section, I will outline the reflective approach, show how it has been developed into the metaobject protocol, and illustrate how such a protocol can be used in a particular case (representation in a programming language). I will show how it can be further generalized into the notion of an *open implementation*, and then return to look at the design issues in interactive systems.

3.1 Computational Reflection and Metaobject Protocols

Computational reflection is the principle that a computational system can embody, within itself, a model of its own behavior (a self-representation) which is *causally connected* to that behavior. Causal connection implies that the representation not only describes, but also controls, the behavior of the system. First, this results in systems that can examine their own behavior through examination of the model; the system can "reason" about its own activity. Second, such systems can make changes to the model and hence, change their own behavior. Essentially, in addition to the traditional "base-level" computation that concerns the system's application domain, reflection enables "metalevel" computation, which concerns the system's own manipulation and execution of base-level concepts.

This principle was originally demonstrated as part of the execution model of 3-Lisp, a reflective dialect of the Lisp programming language. 3-Lisp's reflective facilities were realized by giving the language explicit access to its own interpreter (the program controlling its behavior) [des Rivières and Smith 1984]. By looking at the interpreter structures, 3-Lisp programs could examine their own execution states. For instance, a program could look at the function call sequence recorded in the interpreter's data structures and so ask questions like, "how was this function arrived at?" Furthermore, by making changes to those same structures, programs could alter future behavior; for instance, modifications to the processing of binding structures would allow programs to change the ways in which values were associated with variables. Since the interpreter structures represent a program's execution, providing access gave 3-Lisp programs the facilities to reason about and control their own behavior.

More recently, the principles embodied in 3-Lisp's reflective model have been combined with the techniques of object-oriented programming to yield the metaobject protocol. The metaobject protocol embodies a reflective self-representation in the structures of object-oriented programming. The self-representation in a metaobject protocol-based system is less explicit than that of 3-Lisp. Much of the representation is encoded in the object-oriented structure. Access to the representation, and manipulation of it, is provided through the object system, using the standard techniques of object-oriented programming (subclassing, specialization, overloading, etc.). (An example, to illustrate the mechanics of modifying system behavior in a metaobject protocol, is provided below.)

The first full metaobject protocol (or MOP) was developed within the definition of the Common Lisp Object System (CLOS) [Bobrow et al. 1988; 1993].[1] The CLOS MOP creates a reflective object system, using its own object mechanisms to create an object-oriented representation of its behavior.[2] The reflective model can be changed through standard object-oriented tech-

[1]Since then, metaobject protocols have been incorporated into related languages, such as EuLisp [Bretthauer et al. 1992] and Dylan [Shalit 1992].

[2]In other words, CLOS is not only reflective, but also *metacircular* (defined in itself).

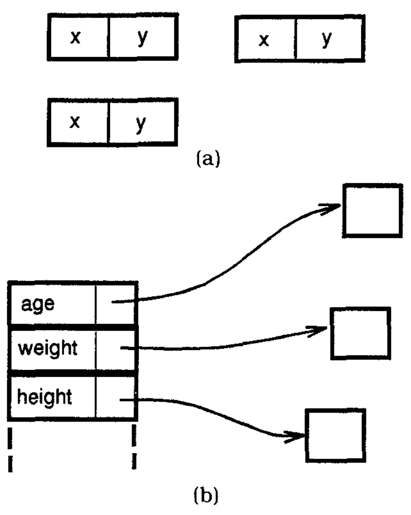

Fig. 1. (a) In a packed representation, each data value is held within the object at a fixed and well-known position. (b) In a sparse representation, each object contains a table that associates assigned slots with pointers to the data they contain.

niques, just like any other object-oriented definition. This allows application programmers to adapt the object system semantics for particular needs, perhaps for efficiency on particular platforms, compatibility with other systems, or specialized behaviors that enable the development of specialized applications.

3.2 Using the CLOS MOP

To explain the CLOS MOP a little more concretely, I will present an example in which an application programmer can revise design decisions in the implemented language. This particular example focuses on the issue of "instance representation" in the implementation, but it is illustrative of the general approach.

In an object-oriented programming language such as CLOS, the implementor must design a representation for instances (objects),[3] which will record such properties as the identity of the instance and the values of its slots. A simple and obvious mechanism might be to allocate enough memory for all of the instance slots, to define that to be the size of an instance of that particular class, and then to compile slot references in programs into the appropriately valued offsets into the instance body.

This is shown in Figure 1(a). The positions of the slots (**x** and **y**) are well known, and so references to **y** can always be compiled as references to the address of the object plus the size of slot **x**. This representation is efficient for many applications which might be built with the programming language, and it lets the compiler generate fast code for slot access.

[3]Throughout I will use CLOS terminology for object-oriented concepts. Individual objects are *instances* of *classes*. Each instance contains a number of defined variables called *slots*. Class-specific behaviors are defined as *methods,* grouped together into *generic functions* that define the patterns of interaction.

However, there are application programs which are not well served by this representation. For instance, consider a knowledge-based application. The application programmer might wish to define a class referring to people. The class might have many slots (hundreds or more), referring to various properties that individuals might have. However, any given instance of that class would perhaps define and use only a few of them. In this case, the simple representation outlined above, which allocates memory for slots *whether or not* they hold values, would be inappropriate. Instead, the application programmer would prefer a mechanism that only allocates space when the slot was assigned a value, perhaps one based on a lookup table for each instance. Figure 1(b) illustrates this alternative representation, where each instance maintains pointers only to the slots which actually have values.

In traditional languages, an implementation's choice of representation scheme is hidden and fixed. Since it is hidden, the programmer becomes aware of it only indirectly, for example, through its effect on performance in particular cases. Since it is fixed, the programmer cannot use high-level information about the program's behavior to inform implementation decisions. Instead, he or she has to write code so that it suits the decisions already made by the implementor, what Kiczales [1992] refers to as "coding between the lines."

In a MOP-based language, however, the application programmer can *revise* the decisions of the language implementor, in this case by changing the representation model used. This is done through traditional object-oriented techniques; in fact, the programming of the CLOS MOP is performed in CLOS. For this example, the mechanisms would be roughly as follows:

(1) There is a class, called standard-class, of which normal user-defined classes are instances. Standard-class is known as the *metaclass* of such classes; a metaclass is the class of a class.

(2) The metaobject protocol defines generic functions for classes, including the functions implementing instance allocation and slot lookup. Methods for these operations specialize in standard-class and, hence, are applied to its instances (normal classes).

(3) A new metaclass, embodying the new instance representation such as the sparse table-driven approach, is defined as a subclass of standard-class. Call this sparse-class.

(4) Since sparse-class is a subclass of standard-class, it inherits the same methods for instance allocation and slot lookup as standard-class. However, we can define new, more specific methods that will override the existing ones and that will apply only to instances of sparse-class.

(5) The applications programmer can now define new methods for allocate-instance (the generic function for creating instance representations) and slot-value-using class (for looking up slot values), specifically for instances of sparse-class. These implement the new, sparse slot representation.

(6) The programmer can now create classes (like **person**) that have sparse-class as their metaclass. These classes will inherit their class-like

behavior from sparse-class and, hence, use the sparse, table-driven instance representation scheme.

So, in this example, the programmer has used knowledge of the requirements of a specific application to revise implementation decisions. The revision was performed by using the system's reflective model of its own behavior. New, specialized behaviors were associated with a subclass of one of the system's internal classes. Since the MOP guarantees the use of generic functions to implement internal behaviors, this approach can be used to redirect the object system's behavior; and since it exploits the polymorphism of object-oriented programming techniques, the new behavior applies only to the objects we specify (those whose metaclass is sparse-class). Without this facility, it would have been necessary to write the application's code in a convoluted way, to live with inefficient performance, or to abandon this particular object system altogether. So there are benefits both to the application developer, who can tailor the system to the needs of a particular application, and to the language implementor, who can design a language with much wider applicability than traditional ones.

By defining CLOS's behavior in terms of the metaobject protocol, the developers of the language had the means to make their language open and adaptable. They avoided a traditional problem within language and toolkit design, a premature commitment forced upon system designers in making implementation decisions which limit the choices open to the later uses of that system. Instead, the system specifies default behaviors, the base-level behaviors of the object system (or whatever); but it also provides the mechanism by which those behaviors can be revised to make them more appropriate in particular circumstances.

The result, of course, is that the designers of a MOP-based system have a much less specific idea of how their system will be used. Through default behaviors, they specify a particular system, which should be generally useful; but through the generic behaviors of the metaobject protocol, they define a framework within which users can create their own customized systems. This is a complex, two-level design task.

3.3 Extending the MOP Approach

The examples of reflective systems given above have concentrated on programming languages (3-Lisp and CLOS). Indeed, the primary use of reflective techniques to date has been to provide flexible semantics for programming languages. However, we have seen that the essence of the reflective approach is closely related to the problems of openness and adaptabilty that were discussed for interactive systems design. So, is it possible that we could adapt reflective techniques for use in other areas?

There are examples that show that we can. One starting point would be Silica [Rao 1991], a reflective window system that forms the basis of the Common Lisp Interface Manager (CLIM). Silica provides a basic window system infrastructure for generating graphical, windowing applications. However, it also provides a set of metalevel abstractions that can be used by application developers to *reach into* the windowing infrastructure and to tailor it to the needs of specific applications, much as we saw with CLOS. In Silica, the metalevel interface is defined in terms of *metaobjects* (windows and window components) and the *contracts* between them. Contracts manage the relationships among system components; they deal with issues such as geometry management, window "stacking," refreshing, and so forth. The metalevel interface is used to avoid the same sorts of problems that motivated the CLOS metaobject protocol, that is, situations in which implementation decisions in the (window system) infrastructure limit its applicability to particular situations.

Silica represents a new departure in our discussion of reflective systems. Unlike 3-Lisp or CLOS, which are general-purpose programming languages. Silica cannot be defined *in its own terms:* a window system does not provide a language that can be used to construct window systems. CLOS and 3-Lisp are *metacircular,* whereas Silica is not. So, Rao's notion of *implementational reflection*, as embodied in Silica, shows how we can apply reflective techniques to a much wider range of application areas then we have seen so far.

Some more recent work, arising from the metaobject protocol experiences, has opened these notions into a more general means of providing a system's clients with control over the abstractions that they use. Kiczales [1992] presented metaobject protocols as one technique which can be used to realize *open implementations,* system implementations which augment traditional abstraction barriers with *modification interfaces*, allowing higher-level users to "reach in" and make appropriate changes. It is an approach which can be applied to a wide range of problems arising in areas where infrastructures must support a wide range of applications. Kiczales also introduced the complementary notion of *open behavior,* in which it is the semantics, rather than the implementation, which are open to change from the higher levels. An example of this, again in the CLOS domain, is PCLOS [Paepcke 1988]. PCLOS exploits metalevel hooks into the mechanisms by which objects are created, accessed, and destroyed, and so creates a *persistent* version of the language which maps objects onto long-term storage in a database. Here, it is the *behavior* of the system that has been extended, rather than the implementation for particular applications or platforms.

These notions are very general. Although they originate in work on programming language design, they point to the application of reflective techniques in a much wider range of software application areas, including the domain of interactive systems.

4. A REFLECTIVE TOOLKIT FOR CSCW DESIGN

The approach to system architecture that I have presented here results from my current work on the design of Prospero, a flexible toolkit for CSCW systems. Individual CSCW applications need to be flexible along various dimensions. First, they must be *statically* flexible, such as in terms of customization to particular individual or group practices or working styles (explored in more detail by Greenberg [1991]). Second, they must be *dynami-*

cally flexible, in response to changes in group behavior in the course of specific collaborations or even specific collaborative sessions. Third, they must be *implementationally* flexible, as infrastructural and interoperative requirements change. At the same time, a toolkit needs to provide developers with sufficient flexibility to generate applications for a wide range of groups, applications, and usage settings.

Just as most programming languages fix the implementations of the language's abstractions, existing CSCW toolkits such as GroupKit [Roseman and Greenberg 1993] or MMConf [Crowley et al. 1990] are forced, through their structure, to impose fixed models of their abstractions, such as distributed data management. This follows directly from the traditional structuring techniques in software development, which hide implementation details behind abstraction barriers, out of reach of the applications developer. There is clearly value to this approach. In particular, it isolates the applications developer from toolkit concerns, focusing attention on those areas specific to the application. The cost is that the range of applications which can be developed within the toolkit is greatly restricted, since the implementation decisions within the toolkit constrain the kinds of interactions which can be supported in applications. The very isolation that toolkits provide prevents the developer from using high-level information about the application to inform lower-level decisions where appropriate.

My current work uses reflective techniques to address these problems. Structuring a CSCW toolkit around a metaobject protocol allows us to tackle two issues: First, as well as providing default behaviors that specify the natural behavior of the system, it gives programmers the opportunity to *specialize* and *refine* the generic framework in the toolkit. This means that the toolkit can be used to provide customized support for particular situations and applications. Second, the self-representation is present in the applications at run time. This allows appropriately written programs to respond dynamically and adaptively as they are used.

This section outlines the way in which the reflective approach is applied in the CSCW domain. Taking a very high-level description of generic application behavior, it shows that, in areas of concern for CSCW application developers, a variety of strategies can be supported within a single metalevel framework.

4.1 Using Reflection in CSCW Design

To apply reflection to the design of a CSCW toolkit, we must "open up" the implementation. This involves specifying the generic behaviors which underlie the system's operation, and the generic entities on which these behaviors act. Providing explicit access to these generic behaviors allows the toolkit user (i.e., the programmer) to specialize them for particular situations. These generic behaviors can be broken down into subprotocols, or specific areas of responsibility.

Clearly, there are a huge number of potential areas of responsibility within the toolkit. One of the major issues in MOP design is the identification of a particular set of concerns that the design should address in order to create a system that is flexible but manageable. We do this by looking at the particular experiences of developers building both applications and toolkits for cooperative systems. We can see a number of issues which are embedded in the design of the infrastructure and yet have strong implications for the kinds of applications which can be supported, areas that are candidates for this approach.

Currently, my work concentrates on three main areas: (1) the management of user data distributed across time and space, (2) mechanisms for managing conflict in user interactions, and (3) control over the linkage between the components of multiple users' interfaces. For each area, the approach is fundamentally the same and involves specifying generic behavior. This is defined in terms of generic function invocations on metaobjects, or explicit representations of the system's behavior. These generic behaviors can be specialized by application developers through incremental modifications to the representations and the actions over them.

Before looking at the use of reflective techniques in these areas, it is necessary to lay down some high-level structure that relates them. A full description of the approach used in Prospero is beyond the scope of this paper, and so a simplified account is presented here; the interested reader is referred to other treatments (e.g., Dourish [1994]).

Consider a system that operates in terms of generic edit operations applied to shared objects. The most general layer of functionality is provided by the following function:[4]

```
(edit-object object user editop) → state-marker
```

Object is a local reference to a globally shared object in the collaborative workspace, user is a representation of the user performing the operation, and editop is an encoding of the operation being performed. The generic function edit-object applies an edit operation to an object and returns state-marker, which describes the new state. It is implemented in terms of a number of lower-level generic functions:

```
(find-object object) → shobject
(lock-object shobject user editop) → lockid
(apply-edit shobject user editop) → change-marker
(propagate change-marker lockid) → state-marker
```

These functions perform the component operations of edit-object: mapping from local objects, presented within the interface, to object components of the shared workspace; obtaining access to those objects; applying changes; and then propagating those changes more widely and releasing the lock. We use change-markers and state-markers as encapsulations of the state of the system at various points. Change-markers record edits made that have not yet been committed; state-markers checkpoint global status. The model presented by this protocol uses these for synchronization, as it presents a view of

[4]Functions are given here in the format used by the Lisp programming language; the first term is the function name, and subsequent terms name arguments to the function.

edit changes being performed locally; however, as long as it is true to this generic model, implementations may behave differently.

Having set up this general framework, we can now investigate how system variability in the areas of data distribution, conflict management, and interface linkage can be managed.

4.1.1 *Data Distribution.* The issue of data distribution has been a bone of contention within the CSCW implementation community for some time. The term *data distribution* covers mechanisms by which the system manages the user's data storage and manipulation. These data may be replicated or distributed across multiple computers, but the system must present a view of a single, coherent data store. Systems such as MMConf [Crowley et al. 1990] take a *fully replicated* approach in which each participant in a conference has a private copy of the data. Others, such as Rapport [Ahuja et al. 1990], use *centralized* architectures, which concentrate data at a single point in the network. Greenberg et al. [1992] argued in favor of *hybrid* systems that combine these approaches. Each of these solutions—replicated, centralized, or hybrid—makes some trade-off between efficiency and complexity when they are the only approach taken by a particular toolkit.

It seems clear that there can be no solution that is appropriate in every case. Not only are there occasions where any of the centralized, replicated, or hybrid approaches are appropriate, but further, there are times when we might need *others*. For instance, consider *disconnected* systems in which some interfaces involved in the collaboration are not permanently connected to the others, or situations in which network latency is high and intragroup interaction is low. Here it might be useful to adopt *migratory* mechanisms, which allow data objects to move from one node to another in the network. Migration is not a separate strategy in itself; it can be combined with any of the other three basic techniques. Other approaches can be posited that will be particularly appropriate for other situations, and we need to be able to express this variability in the toolkit.

Perhaps more importantly, the data distribution approach adopted by a toolkit or application can have important consequences for the appearance, functionality, and usability of the application. The use of a centralized data store, for instance, can negatively affect the response time of the system; while the use of a replicated approach has implications for the maintenance of data consistency. This is at odds with the traditional view that such factors as data distribution are sufficiently "low-level" that they can be safely encapsulated and hidden behind an abstraction barrier.[5]

Rather than make these decisions up front, we can take the reflective approach. Within a toolkit, we not only provide some default mechanism for managing data within a multiuser system, but we also give access to the mechanism by which data distribution is accomplished. This allows programmers, who may find the default behavior inappropriate in their case (e.g.,

[5]The interaction between the distributed data management and the issue of synchrony of interaction is also critical, but merits a longer investigation than can be presented here.

because of the network topology they are using), to "reach in" to the toolkit and to provide new mechanisms to be used in their applications.

The sample subprotocol outlined above manages data distribution largely through the find-object and propagate mechanisms, which isolate the location and the distribution details from the details of actually making changes to the objects. Using find-object, we can encode new mechanisms for mapping between interface objects and the underlying shared data. For instance, in a centralized system, find-object will always return a pointer to the central object store, and propagate will return the locally changed object to the server. On the other hand, in a fully replicated system, the shared object reference is always local, and more complicated methods on propagate will allow changes to be synchronized appropriately. It is important to note, though, that this approach does not merely provide a switch between these two modes. Instead, it provides a framework in which *new solutions* can be devised. The generality of find-object and propagate allows many alternatives, including hybrid and migratory systems, to be created.

Since the representations are available at run time, rather than simply when the system is defined, other opportunities present themselves. We can amend the data distribution mechanism using a dynamic model, which would allow distribution strategies to be changed in the course of an ongoing collaborative session. This allows an application to adapt to the needs of the group as they arise. For instance, consider two users sharing a "scrawl-style" whiteboard application, connected on the same ethernet segment. Since their connection has fairly low round-trip packet times and high data integrity, the system requirements for data management are fairly minimal; a centralized approach is probably entirely adequate. However, things change if a third user joins their conference from some distance, connected via a much slower dial-up line. In this situation, a centralized approach is no longer appropriate, since the bandwidth of the link to the third user is not sufficient to support a network interaction with a data server for each action at the interface. The system must switch, at run time, from one algorithm to another, from a centralized to a replicated data representation. A reflective approach provides the potential for multiple behaviors within the same generic framework, thus supporting this form of dynamic adaptation. If distribution is associated with an object through a mixin[6] class, then changing the class of the object will result in the dynamic switch to a different behavior.

The reflective approach provides a framework within which new mechanisms can be defined, and the means to attach use mechanisms selectively in different parts of the system. This gives three principal benefits that would not be available with traditional solutions. First, the application developer is no longer constrained by decisions within the toolkit, but is free to adapt the toolkit mechanisms to the needs of the application. Second, the toolkit developer no longer needs to "second guess" the specific needs of the devel-

[6]A *mixin* class is one that can be added to other classes in order to bring some new behavior, orthogonal to that defined in the base classes.

oper or to restrict applicability of the toolkit to a subset of potential applications. Third, using the reflective model at runtime allows applications to respond dynamically to their environments and the requirements of particular situations, all within a single coherent framework. We will see this pattern of benefits repeat itself in other areas where we apply computational reflection to toolkit design.

4.1.2 *Conflict Management.* An important area of concern for collaborative applications is the management, or avoidance, of conflicts within the shared workspace. A conflict might occur, for instance, when two users apply a change to the same object at once. Various techniques have been employed to deal with this sort of situation, including floor control, exclusion, and locking. Some systems, such as ShrEdit [McGuffin and Olson 1992], "lock" regions of the shared workspace, preventing simultaneous updates since only one user can hold a lock on a region at any given time. Others, such as GROVE [Ellis and Gibbs 1989], use an algorithm that "fixes up" conflicts afterward in effect imposing a post hoc serialization on the changes that users make.

The essence of conflict management strategies is that the system be able to provide guarantees that users' changes to the data will not lead to a loss of synchronization or data integrity. Prospero exploits explicit representations of such guarantees; this approach was explored in more detail by Dourish [1994]. In this account, however, I will focus solely on systems providing rapid access to a single thread of control.

Even if we choose a simple approach such as locking, then we have to consider the impact that particular locking mechanisms, defined within the toolkit, might have on higher-level usage issues. For instance, in a collaborative system supporting free-form sketching or brainstorming, the emphasis is probably on unencumbered access to the shared work surface. If each user had to request and relinquish locks on the data or control of the floor explicitly, the overhead would be too high, and the progress of work would be severely disrupted. A looser form of control would be needed. On the other hand, looser control would be inappropriate in systems where data integrity must be rigorously maintained and controlled. In a collaborative software engineering application, or a multiuser CAD system which generates control instructions for a milling machine, data errors due to unchecked conflicts could be potentially disastrous, and a much stronger and more explicit form of locking would be required. Neither approach satisfies the needs of a generic toolkit.

Addressing these problems in a reflective toolkit, we attempt to provide a metalevel interface which defines the generic operations involved in requesting, obtaining, and releasing locks. In the simple protocol outlined at the start of Section 4, we focus on the call to lock-object, and the implicit release-lock called from propagate. As before, the protocol itself does not *embody* a locking policy. Instead, it deals with a procedure by which locks are obtained, and a facility for creating and installing new mechanisms. The generic function specifies that, as well as the object to be locked, the function arguments include the user requesting the lock and the type of operation to be performed. The system can take this information into account when selecting the locking mechanism. So, different locking strategies may apply to different users or activities, and for different sorts of objects within the same system. The implementor can rely on the object system's generic dispatch mechanisms to select dynamically the appropriate locking implementation. So the programmer can not only tailor locking strategies to particular applications, but can also build systems in which the locking mechanisms used rely on specific details of the user or object involved.

The basic mechanism is sufficiently open that a wide range of locking strategies can be defined. Not only will it allow the implementation of standard strong and weak locks, but also multiway locks (held by multiple people at once), tickle locks (which, when idle, may be implicitly reassigned to other users when they perform an operation), and so on. Indeed, we can reproduce schemes such as GROVE's dOPT algorithm, in which explicit locks are not used at all, by making lock-object to construct an appropriate "state vector." This will be distributed by the call to propagate, so that other nodes can use this information to resolve ambiguities arising out of conflicting or misordered operations. In this case, we regard the dOPT state vector as an implicit "lock," in the sense that it is an object that will allow conflict resolution. In other words, the same basic mechanism can be used to encode a form of conflict management that is hardly "lock based" at all.

4.1.3 *Interface Linkage.* One of the most obvious differences among CSCW systems is the level at which they "link" interface features. Linkage determines the level of control that users have over the way their own interfaces appear, without affecting other users of the same collaborative application. The grossest level of linkage is screen replication, as used, for example, in Timbuktu [Farallon 1987]. Screen linkage means that all users see exactly the same thing on their screens. Shared X systems [Garfinkel et al. 1989] link interfaces at the level of *windows;* users share the contents of a window, while their screens may show other, independent applications and window placement can vary from person to person. Many explicit multiuser tools such as ShrEdit are much looser and will replicate only the data.[7] Here, users may have different views of the data and may be provided with individual edit cursors. Within this class of systems, there are further differences in what each user can see of the other's interfaces.

While many systems separate users and isolate their interfaces, research on groups interacting through synchronously shared systems has shown how low-level cues can be used by collaborators to create an awareness of the activity and progress of the group as a whole [Dourish and Bellotti 1992]. Recent work, such as that of Dewan and Choudhary [1991] or Haake and Wilson [1992], has looked at the provision of switchable linkage states, in

[7]Note that our concern here is with replication of interface features, rather than with the underlying data representations discussed earlier. So, in these systems, it is only the data that are guaranteed to be consistently replicated *between interfaces.*

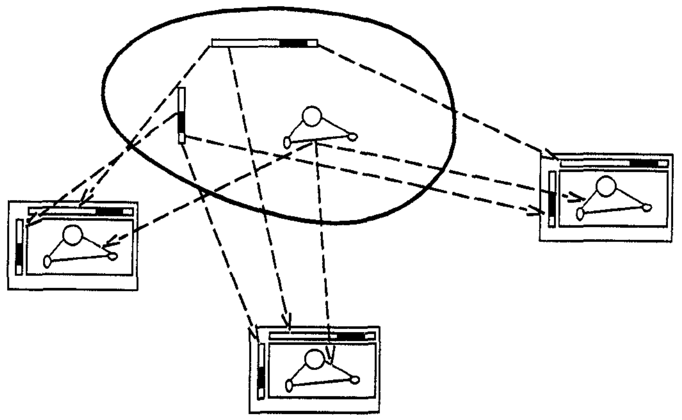

Fig. 2. Note only data objects, but also interface components, can be part of the shared workspace.

which users can choose how much their interfaces will mirror each other's. A similar two-mode switching facility was available in rIBIS [Rein and Ellis 1991].

Once again, we can see the requirements for flexibility within applications and toolkits, and we can see that this flexibility can have a dynamic component. The nondynamic aspect is the same now-familiar toolkit-level problem; that different applications require different linkage strategies, and so a generic toolkit must be able to support a range of linkage options. Dewan's work with Suite or Haake and Wilson's work with SEPIA tackles just this problem, as well as addresses the dynamic problem of switching between these different linkage modes during a collaborative session. However, both systems provide flexible linking through specific "modes," which predefine linkage strategies. This only addresses part of the problem. What if different users or groups require different linkage strategies, or if some situation calls for a strategy that has not been predefined? There are no options while remaining within these frameworks. Although clearly the goal here is to provide flexibility, it is flexibility within the constraints of the set of needs anticipated by the toolkit designer. Although these mechanisms are *parameterized,* they are not *open.*

The approach taken in Prospero is to make aspects of interface components into shared objects, exactly like workspace data objects. This is illustrated in Figure 2. In this way, components such as menus, button states, and cursor positions are subject to the mechanisms outlined earlier for dynamic distribution algorithms. The same mechanisms that maintain consistency between workspace objects can be used to synchronize interfaces.

The result is that the same flexibility that can be applied to distributed data management also applies to interface management. Interface components can be explicitly shared, causing that aspect of interfaces to be linked; or they can be separated and broadcast, allowing each individual control while being able to see others' states; or they can be private, in which case other users cannot see them. In addition, another property they have in common with shared data objects is that they can be moved into and out of the shared workspace in the course of a collaboration. This means that the linked aspects of the users' interfaces can be dynamically controlled and adjusted as the collaboration continues.

5. REFLECTION AND ADAPTIVE COMPUTATION

In this article I have mainly been concerned with the use of computational reflection and related techniques as ways of opening up implementations and providing flexibility. The techniques I have discussed have been developed primarily in the domain of language design. Recently, similar issues have come to prominence in a number of other areas. Various radical solutions have been adopted that go beyond the traditional separation of "mechanism and policy," and often, these address issues very similar to those discussed here within the context of CSCW.

The design of communication protocols on data networks has traditionally been based on a "layered" approach. This form of design is exemplified by the seven-layer ISO protocol stack [Zimmerman 1980]. End-to-end communication requirements are broken down into different areas of responsibility, such as data representation conversion, direct host-to-host communication, and internetwork communication. Each component is encapsulated in a layer, and on a given machine, each layer interacts only with the layers directly above and below it. More recently, however, the need to handle interactive multimedia traffic, as well as issues arising in the design of protocols for gigabit networks, has resulted in a breakdown in this model. In its place, a flatter approach is emerging in which more of the traffic management is controlled *directly by the application,* rather than being hidden in the network software. O'Malley and Peterson [1992] described a model in which the application can compose kernel-internal microprotocols into larger units optimized for their particular requirements. Clark and Tennenhouse [1990] proposed the concept of *Integrated Layer Processing* as a mechanism to avoid the interlayer inefficiencies that emerge when the infrastructure is examined from the point of view of particular end-to-end systems.

In the area of operating systems, much functionality that has usually been in the domain of the system itself is being opened up to external control. One of the most obvious examples is Mach's external pager [Rashid et al. 1987], which allows user programs to involve themselves in aspects of the virtual memory system's operation. Similarly, Anderson et al. [1992] described "scheduler activations" as a means to avoid trade-offs in the implementation of threads, which are traditionally a completely opaque abstraction over an implementation based either in the operating system kernel or in a user library. Scheduler activations provide a finer grain of control and are explicitly designed to allow application-specific customization.

This trend is repeated in many other areas too, such as interprocessor communication [Felten 1992] or even microprocessor design [Athanas and

Silverman 1993]. The same principle is at work in all of these examples. They are all based on an understanding that traditional closed abstractions are not always appropriate for high-level systems design in general, and in particular for the design of infrastructural components. The various solutions are oriented around a *downward flow of information,* from the higher levels (applications) to the lower levels (toolkits and infrastructures), in order to support better interaction between the two. This downward flow, from application requirements to the details of system support, mirrors the problems that Prospero addresses with reflective techniques.

Reflection achieves this by opening up the underlying implementation and allowing the applications programmer to explore alternative implementations and behaviors within the metalevel framework. This corresponds with what Kay [1993] characterized as *late-binding systems*, those in which design and implementation decisions that affect observable behavior are delayed until they can be resolved with as much context as possible. While the techniques discussed here have been derived from work on language design, late-binding is useful and important in interactive systems, especially in CSCW systems where contextual factors play such a large part in the interaction.

Prospero is a toolkit for CSCW applications under development, based on these principles. It concentrates primarily on issues in data distribution, conflict management, and interface linkage. The examples in this article have been drawn from this work in progress. The system provides default behaviors which can be used to construct applications in the usual way, where they are appropriate. More importantly, though, it also provides a metalevel framework that can be used to revise implementation decisions, to extend the structure to cover new areas, and to make the toolkit more appropriate for a range of applications.

6. SUMMARY

The primary focus of this article has been on models of implementing interactive systems. I have argued that recent years have seen a fundamental reorientation in our view of interactive systems and their use. In turn, this forces a reorientation in our view of system design and structure. In particular, appreciation of the need for (and use of) customization facilities, the role of work practice and situation in system use, and the coadaptive nature of system use and user behavior lead us to a model of systems design which emphasizes openness, dynamic behavior, and evolution of systems and practices.

The move away from static systems leads us to reconsider the architectures that underlie interactive systems. It is not enough simply to change the process of design; instead, we need to change the nature of the artifacts themselves. By drawing on the principles and techniques of computational reflection, derived originally from research into programming language semantics, I have outlined a model of interactive system design which is oriented specifically toward these new goals of flexibility and adaptation. In particular, this model is currently being used as the implementational basis of a toolkit for CSCW design, and I have outlined how this toolkit tackles a number of current problems in CSCW toolkits which must be used in a wide range of different circumstances and situations.

This work is currently ongoing. It is hoped that the reflective toolkit for CSCW can provide insights into the general application of notions of open implementation and behavior to a range of current problems in interactive system design.

ACKNOWLEDGMENTS

The ideas expressed in this article would never have seen the light of day without the contributions and encouragement of many people. I would particularly like to thank Hal Abelson, Bob Anderson, Victoria Bellotti, Danny Bobrow, Jon Crowcroft, Gregor Kiczales, and Wendy Mackay for fruitful and enlightening discussions. I am very grateful to Annette Adler, Jon Crowcroft, Laura Dekker, Marge Eldridge, Lorna Goulden, Jonathan Grudin, Gillian Ritchie, Lisa Tweedie, and the anonymous reviewers for careful readings and valuable comments on earlier drafts of this article.

REFERENCES

AHUJA, S., ENSOR, J., AND LUCCO, S. 1990. A comparison of application sharing mechanisms in real-time desktop conferencing systems. In *Proceedings Conference on Office Information Systems COIS 90* (Boston, Mass., Apr.). ACM, New York, 238–248.

ANDERSON, T., BERSHAD, B., LAZOWSKA, E., AND LEVY, H. 1992. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10,* 1 (Feb.), 53–79.

ATHANAS, P. AND SILVERMAN, H. 1993. Processor reconfiguration through instruction-set metamorphosis. *IEEE Comput.* (Mar.), 11–18.

BOBROW, D., DEMICHIEL, L., GABRIEL, R., KEENE, S., KICZALES, G., AND MOON, D. 1988. Common Lisp object system specification. X3J13 Doc. 88-002R, June.

BOBROW, D., GABRIEL, R., AND WHITE, J. L. 1993. CLOS in context: The shape of the design space. In *Object-Oriented Programming: The CLOS Perspective,* A. Paepcke, Ed. MIT Press, Cambridge, Mass.

BØDKER, S. AND GRØNBÆK, K., 1991. Cooperative prototyping: Users and designers in mutual activity. *Int. J. Man-Machine Stud. 34,* 3, 453–479.

BOEHM, B. 1988. A spiral model of software development and enhancement. *IEEE Comput.* (May), 61–72.

BOOCH, G. 1991. *Object Oriented Design.* Benjamin/Cummings, Redwood City, Calif.

BOWERS, J. AND RODDEN, T. 1993. Exploding the interface: Experiences of a CSCW network. In *Proceedings InterCHI 93* (Amsterdam, The Netherlands, Apr.). ACM, New York, 255–262.

BRETTHAUER, H., DAVIS, H., KOPP, J., AND PLAYFORD, K. 1992. Balancing the EuLisp metaobject protocol. In *Proceedings IMSA 92 Workshop on Reflection and Metalevel Architectures* (Tokyo, Japan, Nov.).

CLARK, D. AND TENNENHOUSE, D. 1990. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Commun. Rev. 20,* 4, 200–208.

CROWLEY, T., MILAZZO, P., BAKER, E., FORSDICK, H., AND TOMLINSON, R. 1990. MMConf: An infrastructure for building shared multimedia applications. In *Proceedings ACM Conference on Computer-Supported Cooperative Work CSCW 90* (Los Angeles, Calif., Oct.). ACM, New York.

DES RIVIÈRES, J. AND SMITH, B. 1984. The implementation of procedurally reflective languages. Tech. Rep. ISL-4, Xerox PARC, Palo Alto, Calif. June.

DEWAN, P. AND CHOUDHARY, R. 1991. Flexible user interface coupling in a collaborative system. In *Proceedings of the ACM Conference on Human Factors in Computing Systems CHI 91* (New Orleans, La., Apr.). ACM, New York.

DOURISH, P. 1994. A divergence-based model of synchrony and distribution in collaborative systems. EuroPARC Tech. Rep. EPC-92-102. Rank Xerox EuroPARC, Cambridge, U.K.

DOURISH, P. AND BELLOTTI, V. 1992. Awareness and coordination in shared workspaces. In *Proceedings ACM Conference on Computer-Supported Cooperative Work CSCW 92* (Toronto, Canada, Nov.). ACM New York.

EHN, P. 1988. *Work-Oriented Design of Computer Artifacts.* Arbetslivscentrum, Stockholm, Sweden.

ELLIS, C. AND GIBBS, S. 1989. Concurrency control in groupware systems. In *Proceedings ACM Conference on Management of Data SIGMOD 89* (Seattle, Wash.). ACM, New York.

FARALLON COMPUTING. 1987. *Timbuktu: The Next Best Thing to Being There.* Farallon Computing.

FELTEN, E. 1992. The case for application-specific communication protocols. Tech. Rep. TR-02-03-11, Dept. of Computer Science, Univ. of Washington, Seattle, Wash.

FILIPPI, G. AND THEUREAU, J. 1993. Analyzing cooperative work in an urban traffic control room for the design of a coordination support system. In *Proceedings 3rd European Conference on Computer Supported Cooperative Work ECSCW 93* (Milano, Italy, Sept.). Kluwer Academic, Amsterdam, 171–186.

GARFINKEL, D., GUST, P., LEMON, M., AND LOWDER, S. 1989. The SharedX multi-user interface user's guide, version 2.0. Software Technology Lab Rep. STL-TM-89-07, Hewlett-Packard Laboratories, Palo Alto, Calif.

GREENBERG, S. 1991. Personalisable groupware: Accommodating individual roles and group differences. In *Proceedings European Conference on Computer-Supported Cooperative Work ECSCW 91* (Amsterdam, The Netherlands, Sept.). ACM, New York.

GREENBERG, S., ROSEMAN, M., WEBSTER, D., AND BOHNET, R. 1992. Human and technical factors of distributed group drawing tools. *Interacting Comput. 4,* 3, 364–392.

GRUDIN, J. 1993. Interface: An evolving concept. *Commun. ACM 36,* 4 (Apr.), 110–119.

GRUDIN, J. 1991. Obstacles to user involvement in software product development, with implications for CSCW. *Int. J. Man-Machine Stud. 34,* 3 (Mar.), 435–452.

GRUDIN, J. 1990. The computer reaches out: The historical continuity of interface design. In *Proceedings ACM Conference on Human Factors in Computing Systems CHI 90* (Seattle, Wash., Apr.). ACM, New York.

HAAKE, J. AND WILSON, B. 1992. Supporting collaborative writing of hyperdocuments. In *Proceedings ACM Conference on Computer-Supported Cooperative Work CSCW 92* (Toronto, Canada, Nov.). ACM, New York.

HARPER, R., HUGHES, J., AND SHAPIRO, D. 1991. Harmonious working and CSCW: Computer technology and air traffic control. In *Studies in Computer Supported Cooperative Work.* J. Bowers and S. Benford, Eds. North-Holland, Amsterdam, 225–234.

HEATH, C. AND LUFF, P. 1991. Collaborative activity and technological design: Task coordination in London underground control rooms. In *Proceedings European Conference on Computer-Supported Cooperative Work ECSCW 91* (Amsterdam, The Netherlands, Sept.). Kluwer Academic, Amsterdam, 65–80.

KAY, A. 1993. The early history of Smalltalk. In *Proceedings ACM Conference History of Programming Languages HOPL-II. SIGPLAN Not. 28,* 3 (Mar.).

KICZALES, G. 1992. Towards a new model of abstraction in software engineering. In *Proceedings IMSA 92 Workshop on Reflection and Metalevel Architectures.* (Tokyo, Japan, Nov. 4–7).

KICZALES, G., DES RIVIÈRES, J., AND BOBROW, D. 1991. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Mass.

MACKAY, W. 1991. Triggers and barriers to customising software. In *Proceedings ACM Conference on Human Factors in Computing Systems CHI 91* (New Orleans, La., Apr.). ACM, New York.

MACKAY, W. 1990a. Patterns of sharing customisable software. In *Proceedings ACM Conference on Computer-Supported Cooperative Work CSCWW 90* (Los Angeles, Calif., Oct.). ACM, New York.

MACKAY, W. 1990b. Users and customisable software: A co-adaptive phenomenon. Ph.D. thesis, Sloan School of Management, MIT, Cambridge, Mass.

MACKAY, W. 1989. How do experienced Information Lens users use rules? In *Proceedings ACM Conference on Human Factors in Computing Systems CHI 89* (Austin, Tex.). ACM, New York.

MACLEAN, A. CARTER, K. MORAN, T., AND LOVSTRAND, L. 1990. User tailorable systems: Pressing the issues with Buttons. In *Proceedings ACM Conference on Human Factors in Computing Systems CHI 90* (Seattle, Wash., Apr.). ACM, New York.

MAES, P. 1987. Computational reflection. Tech. Rep. 87.2, Artificial Intelligence Lab. Vrije Univ., Brussels, Belgium.

MALONE, T., CROWSTON, K., RAO., R., ROSENBLITT, D., AND CARD, S. 1987. Semi-structured messages are surprisingly useful for computer-supported coordination. *ACM Trans. Off. Inf. Syst. 5,* 2, 115–131.

MCGUFFIN, L. AND OLSON, G. 1992. ShrEdit: A shared electronic workspace. CSMIL Tech. Rep., Cognitive Science and Machine Intelligence Lab., Univ. of Michigan, Ann Arbor.

NARDI, B. AND MILLER, J. 1991. Twinkling lights and nested loops: Distributed problem solving and spreadsheet development. In *Computer-Supported Cooperative Work and Groupware,* S. Greenberg, Ed. Academic Press, New York.

NORMAN, D. AND DRAPER, S. (EDS.) 1986. *User-Centered Systems Design.* Lawrence Erlbaum Associates, Hillsdale, N.J.

O'MALLEY S. AND PETERSON, L. 1992. A dynamic network architecture. *ACM Trans. Comput. Syst. 10,* 2 (May).

PAEPCKE, A. 1988. PCLOS: A flexible implementation of CLOS persistance. In *Proceedings European Conference on Object-Oriented Programming ECOOP 88.* Springer-Verlag, New York.

RAO, R. 1991. Implementational reflection in Silica. In *Proceedings European Conference on Object-Oriented Programming ECOOP 91* (Geneva, Switzerland). Springer-Verlag, New York.

RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOKOSKY, W., AND CHEW, J. 1987. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings Conference Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Oct.).

REIN, G. AND ELLIS, C. 1991. rIBIS: A real-time group hypertext system. *Int. J. Man-Mach. Stud. 34,* 3 (Mar.), 349–367.

ROSEMAN, M. AND GREENBERG, S. 1993. GroupKit: A groupware toolkit for building real-time conferencing applications. In *Proccedings ACM Conference on Computer-Supported Cooperative Work CSCW 92* (Toronto, Canada, Nov.). ACM, New York.

ROYCE, W. W. 1970. Managing the development of large software systems. In *Proceedings WESTCON* (Calif.).

SHALIT, A. 1992. *Dylan: An Object-Oriented Dynamic Language.* Apple Computer, Cupertino, Calif.

SMITH, B. 1982. Reflection and semantics in a procedural language. Rep. MIT-TR-272, Lab. for Computer Science, MIT, Cambridge, Mass.

SPROULL, L. AND KIESLER, S. 1991. *Connections: New Ways of Working in the Networked Organization.* MIT Press, Cambridge, Mass.

SUCHMAN, L. 1992. Technologies of accountability: Of lizards and aeroplanes. In *Technology ın Working Order: Studies of Work, Interactıon and Technology,* G. Button, Ed. Routledge, London, U.K., 113–126.

SUCHMAN, L. 1987. *Plans and Situated Actions.* Cambridge University Press, Cambridge, U.K.

TRIGG, R., MORAN, T., AND HALASZ, F. 1987. Adaptability and tailorability in NoteCards. In *Proceedings Interact 87* (Stuttgart, Germany). IFIP, 723–728.

ZIMMERMAN, H. 1980. OSI reference model—The ISO model of architecture for open systems interconnection. *IEEE Trans. Commun. 28,* 4 (Apr.), 425–432.

Received January 1994; revised October 1994; accepted October 1994