Towards a Reflective Model of Collaborative Systems

Paul Dourish

Technical Report EPC-1993-114

Rank Xerox Research Centre
Cambridge Laboratory
61 Regent Street
Cambridge CB2 1AB

Tel:+44 1223 341500
Fax:+44 1223 341510

Towards A Reflective Model of Collaborative Systems

Paul Dourish

Rank Xerox EuroPARC, Cambridge, UK
and
Dept. of Computer Science, University College, London
dourish@europarc.xerox.com

Abstract

In recent years, we have come to understand that the design of effective interactive systems is not simply about implementation models and techniques, but also about aspects of the system in use, many of which have come to us from psychology and social science. The issues of work practice, adaptation and evolution which surround interactive systems have become an extremely important area of research.

This paper argues that the reorientation in our view of interactive systems which has resulted from these areas of interest necessitates a similar reorientation in the techniques by which we design and construct interactive systems. 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 towards a new model based on open abstraction which holds great promise in addressing these issues.

Keywords: interactive systems design, computational reflection, metaobject protocol, adaptive systems, CSCW.

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 HCI has moved from being a minor component of software engineering to being the focus of attention for researchers from a variety of disciplines, including psychology and social science. HCI has begun to concern itself not just with the mechanism of the interface, but with a range of related and relevant issues which concern the context in which interactive systems are used and studied. Issues such as customisation of interactive systems, of their evolution within an environment, and of the work practices surrounding their use, have become increasingly important.

Work on customisation, such as that of Maclean *et al* [1990], Mackay [1991] or Trigg *et al* [1987] has pointed to the way in which users adapt systems to their own needs and purposes. It argues that interactive systems designers must appreciate both the flexible nature of the systems they design and the way in which that flexibility will be exploited. Studies of work practice, such as those of Harper *et al* [1991] or Suchman [1987] have emphasised the roles which social and organisational factors contribute to the use of interactive systems, and the way in which that use is firmly situated within particular environments. Finally, studies of adaptation and evolution such as Mackay's [1990] have illustrated the way in which patterns of use evolve over time. These illustrate that the relationship between the evolution of work practice and the evolution of system use is a bidirectional one (in Mackay's terms, they form a *coadaptive phenomenon*).

1.1 A Revised View of Interactive Systems

It is not surprising, then, that these should result in a reorientation of our view of interactive systems, and such a reorientation has been taking place. It moves from an earlier view of systems as fixed, "black-box" artifacts which could be objectively studied and assessed, to a view of "systems-in-use" which acknowledges the influence of these other elements. In the new view, systems are *situated* within particular organisations 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 behaviours emerging around the interactive system, while at the same time the system itself is tailored to particular working styles.

This shift in perspectives, as it has been described here, has been largely in terms of interactive systems as they appear to the user, in particular settings. Indeed, it has been part of a general trend towards *participative* or *user-centered* approaches to system building. However, it raises a number of other concerns, particular for the designers and

implementors of interactive systems. What does this shift in view imply for the systems which we design?

2 Implications for Interactive Systems Design

Grudin [1991] has considered issues in the interaction of this user-centred approach and the systems development process. Here, however, I am more concerned with the implications for the systems themselves. If we generalise some of the components of this shift in perspective on interactive systems, we can separate out two levels of consequences for their design.

2.1 The Cycle of Design

The first of this 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. In more recent models of software design such as that proposed by Booch [1991], an iterative model is adopted, and so the design phase becomes more distributed. However, despite their differences, there is at least one point of fundamental agreement between these models. They both state that, at some point, a product is *delivered* to a user community, and that, for that revision of the software, the design process is now over.

However, this is an assumption which 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, then 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 focussed set of concerns for system developers in terms of the way in which systems are structured, constructed and delivered.

2.2 Creating Evolving Systems

So 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 which is amenable to evolution and adaptation. We can focus on three particular concerns:

1. *Open infrastructures*. With the perspective of systems-in-use, we begin to see delivered systems as not being closed and static, but rather as infrastructures for further specialisation, refinement and end-user design. The system is the structure within which users will apply their changes and customisations, and which will circumscribe the adaptations which users can make as they evolve the system to their own patterns of usage. The system developer is concerned with appropriate openness within the system and the way in which it lends itself to these adaptations. In addition, extending the model of customisation, we must consider the ways in which the system can lend itself to customising *process* (the manipulation of information within an interactive systems) as well as *presentation* (surface-level issues of views and interaction).
2. *Dynamic and reactive systems*. When we think of user behaviour changing over time, then we must consider the ways in which the interactive system will respond to these changes. From this point of view, systems need to be structured so that they can dynamically react to patterns of use and activity. The system's response must be situated in the same sense as is the user's activity.
3. *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. Again, from research on customisation and coadaptivity, we find that this evolution is a process which has its roots in the social aspects of work and which is enabled through the *sharing* of customisations. Important development concerns include the nature of the customisation mechanisms, and the means by which they can be distributed, shared and themselves evolve over time.

The issue for developers of systems, then, is to develop a set of techniques supporting the construction of software systems which enable the distribution of the design phrase throughout the whole lifecycle of a system, and which, in particular, support these goals of software adaptation and evolution.

3 Reflection and Open Abstraction

Consideration of these issues has followed 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. Customisation can be performed not only by users but by groups as a whole, and even a single group may employ a wide variety of working styles in the course of a collaboration. Factors such as these bring the system developer face-to-face with the issues of reactivity and adaptability. Since I have been primarily concerned with the developments of generic toolkits, which can be used to generate a variety of CSCW applications, a major goal has also been to provide the application developer with sufficient flexibility to create a range of application styles, embodying different models of collaborative work.

The systems approach which I am developing in pursuit of openness and flexibility is based on the principles of *computational reflection* ([Smith, 1982]; [Maes, 1987]) and in particular the *metaobject protocol* [Kiczales *et al*, 1991].

3.1 Computational Reflection and Metaobject Protocols

Computational reflection is the principle that a computational system should embody, within itself, a model of its own behaviour which is *causally connected* to that behaviour. First, this results in systems with the means to examine their own behaviour through examination of the model. Second, such systems can make changes to the model and hence change their own behaviour. Essentially, in addition to the traditional *base-level* computation which concerns the domain of application of the system, the opportunity is provided for *meta-level* computation which concerns the system's own manipulation and execution of base-level concepts.

This principle was originally developed as part of the execution model of 3-Lisp, a reflective dialect of Lisp in which reflection took the form of explicit mechanisms by which a Lisp program could manipulate the structures of its own interpreter [des Rivières and Smith, 1984]. 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*[1] or MOP. The metaobject protocol can be regarded as an embodiment of the reflective model in terms of the classes and methods of object-oriented programming.

The metaobject protocol was developed as part of the Common Lisp Object System (CLOS) definition effort [Bobrow *et al*, 1988]. The CLOS MOP creates a reflective object system, in which the reflective self-representation is defined in object-oriented terms. The model can be changed through standard object-oriented techniques of subclassing and specialisation. In this way, the object system semantics can be adapted by application programmers to support their special needs, either for efficiency on particular platforms, compatibility with other systems, or merely specialised behaviours which enable the development of particular applications.

1. The term *protocol* is here used in the object-oriented sense, defining the set of methods and arguments to which objects of a particular class can respond. The term *metaobject* refers to objects which define the meta-level of the system, that is its self-representation, rather than the base-level domain of computation.

3.2 Using The CLOS MOP

In order to illustrate the use of the CLOS MOP, I will outline an example of the way in which an application programmer can revise design decisions in the implemented language[2].

In creating an implementation of an object-oriented programming language such as CLOS, the language implementor must design a representation for *instances*[3], recording 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 each of the slots of the instance, define that to be the size of an instance of that particular class, and then compile references to instance slots in the code into the appropriately-valued offsets into the instance body. This representation is efficient for many applications which might be built with the programming language, allowing the compiler to generate fast code for slot access.

However, there are certainly examples of applications programs which are not well-served by this representation. For instance, in an AI application, an applications programmer might wish to define a class referring to people. The class might have very many slots (many hundreds or more), referring to various properties of individuals. However, any given instance of that class would define and use only a few. In this case, the simple representation, which allocates memory for slots whether or not they hold values, would be inappropriate. Instead, the application programmer would prefer a mechanism, perhaps based on a lookup table in each instance, which only allocated space when the slot was assigned a value.

Using the MOP, the application programmer can revise the decisions of the language implementor and change the representation model which is used. This is done through traditional object-oriented techniques; in fact, the programming of the CLOS MOP is performed in CLOS. In this case, the mechanism would be roughly as follows:

1. There is a class, called `standard-class`, of which other classes are, by default, instances. `Standard-class` is known as the *metaclass* of such classes.[4]

2. This rather over-worked example is originally due to Gregor Kiczales.

3. In CLOS terminology, individual objects are the *instances* of a class. Each object has a number of *slots*, which are private variables. A *generic function* is a functionally-defined message, for which classes may define specific *methods*, implementing that behaviour for particular parts of the class hierarchy.

4. A *metaclass* is different from a *superclass*. The *superclass* is the class from which a class can inherit structure and behaviour; the *metaclass* is the class of which it is an instance.

2. The metaobject protocol defines *generic functions* for classes, including functions which implement instance allocation and slot lookup. Methods for these operations specialise on `standard-class`, and hence are applied to its instances.
3. A new metaclass, which will embody a new instance representation such as the sparse table-driven approach, is defined as a subclass of `standard-class`. Call this `sparse-class`.
4. By default, `sparse-class` inherits the same methods for performing instance allocation and slot lookup as `standard-class`. However, since `sparse-class` is a subclass of `standard-class`, we can defined new, more specific methods.
5. The application programmer can now defined new methods for the generic functions `allocate-instance` and `slot-value-using-class` specifically for instances of `sparse-class`, which will implement the new, sparse slot representation.

The applications programmer is now in a position to define new classes, such as `person`, which are instances (*i.e.* have as their metaclass) the new `sparse-class` rather than `standard-class`. Since the metaobject protocol defines the generic functions which will be called internally to the object system for operations such as instance allocation and slot lookup, the applications programmer is assured that the newly-defined methods will be used by the object system.

3.3 Designing MOP-Based Open Systems

By defining CLOS's behaviour in terms of the metaobject protocol, the developers of the standard provided a means to make their language *open* and *adaptable*. They avoided a traditional problem within language and toolkit design, that of the *premature commitment* made by system designers in making implementation decisions which limit the choices open to the later users of that system. The system specifies default behaviours, which are the base-level behaviours of in the object system (or whatever); but it also provides the mechanism by which those behaviours 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 behaviours*, they specify a particular system, which should be generally useful; but, through the *generic behaviours* of the metaobject protocol, they define a framework within which users can create their own customised systems.

3.4 Extending the MOP Approach

The examples given above of reflective systems have concentrated on programming languages; and indeed, to date, the primary use of reflective techniques has been in the provision of flexible programming language semantics. However, we have seen that the essence of the reflective approach is closely related to the problems of openness and adaptability which were discussed above in relation to interactive systems design. So, is it possible that we could adapt these techniques for use in other areas?

One starting point for this generalisation would be Silica [Rao, 1991], a reflective window system which forms the basis of the Common Lisp Interface Manager (CLIM). Silica uses reflective techniques in order to allow the developers of windowing applications to makes changes to the window system implementation in order to more appropriately support particular applications. Silica shows the way in which reflection can be applied without the need for the *metacircular* approach exhibited by the definition of CLOS in CLOS, or 3-Lisp in 3-Lisp. Rao's concept of *implementational reflection* allows us to use these techniques in other application areas.

Some more recent work, developing from the metaobject protocol experiences, has opened out these notions into a more general means for providing system users with control over the abstractions which they employ in software systems [Kiczales, 1992]. Kiczales presents metaobject protocols as one technique which can be employed in the creation of *open implementations*—system implementations which augment their traditional abstraction barriers with modification interfaces, allowing higher-level users to "reach in" and make appropriate changes. He also mentions the complementary notion of *open behaviour*, in which it is the semantics, rather than the implementation, which are open to change from the higher levels. These notions are very general; while being based in the work on programming language design they point, like Silica, to the application of reflective techniques in a much wider range of software application areas.

4 A Reflective Toolkit for CSCW Design

The considerations presented here of the relationship between shifts in perspectives in HCI and the need for a reorientation of our model of interactive systems design result from my current work on the design of a flexible toolkit for CSCW systems. Individual CSCW applications need to be flexible along various dimensions: *statically*, such as in terms of customisation to particular individual or group practices or working styles (explored in more detail by Greenberg [Greenberg, 1991]); *dynamically*, in response to changes in group behaviour in the course of specific collaborations or even specific collaborative sessions; and *implementationally*, 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. Existing CSCW toolkits such as Rendezvous [Patterson *et al*, 1990] or MMConf [Crowley *et al*, 1990] are forced, through their structure, to require particular models of, for instance, data distribution, since these components are hidden behind abstraction barriers, out of reach of the applications developer. The goal of my current work is to create a reflective toolkit for the generation of CSCW applications which, in addition to default behaviours specifying the natural behaviour of the system, specifies refinable generic facilities which allow, first, the modification to the toolkit to suit particular application situations, and second, the provision of dynamic facilities within the applications themselves.

4.1 Using Reflection in CSCW Design

In employing the reflective approach in the design of a CSCW toolkit, we must "open up" the implementation by specifying the generic behaviours which underlie the system's operation. Providing explicit access to these generic behaviours allows the toolkit user (*i.e.* the programmer) to specialise them for particular situations. These generic behaviours can be broken down into sub-protocols, or specific areas of responsibility. Currently active areas of investigation include protocols for *data distribution*, *exclusion* and *interface linkage*.

We can study these examples in terms of a particular framework of generic functions which might form part of a metaobject protocol. It should be emphasised that this framework is outlined purely for didactic purposes.

The most generic layer of functionality is provided by the function:
(`edit-object` *object user editop*) $\Longrightarrow$ *state-marker*
This generic function applies an edit operation to an object, and returns a state-marker which describes the new state. It is implemented in terms of a number of lower-level generic functions:

(`find-object` *object*) $\Longrightarrow$ *shobject*
(`lock-object` *shobject user editop*) $\Longrightarrow$ *lockid*
(`apply-edit` *shobject user editop*) $\Longrightarrow$ *change-marker*
(`propagate` *change-marker lockid*) $\Longrightarrow$ *state-marker*

These functions perform the operations which map 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-makers record edits made which have not yet been committed; state-markers checkpoint global status. The model presented by this protocol uses these for synchronisation, as it presents a view of edit changes being performed locally; however, as long as it is true to this generic model, implementations may behave differently.

4.1.1 Data Distribution

Issues of data distribution have been a bone of contention within the CSCW implementation community for some time. Systems such as MMConf have taken the *replicated* approach, in which each participant in a conference has a private copy of the data, while others, such as Rapport [Ahuja *et al*, 1990] have been based on the principle that *centralised* architectures, which concentrate data at a single point in the network, are sufficient. At the same time, Greenberg *et al* [Greenberg *et al*, 1992] have argued in favour of *hybrid* systems which combine these approaches. Each of these approaches makes some trade-off between efficiency and complexity.

It is clear, however, that there can be no solution which is appropriate in every case—not only are there occasions where centralised, replicated or hybrid approaches are appropriate, but further, there are times when we might need *migratory* approaches in which data objects can move from one node to another in the network. Other approaches can be posited which will suit other situations. More importantly, the data distribution approach which is adopted in a toolkit or application can have important consequences for the appearance, functionality and usability of the application. 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 in a system.

Instead, taking the reflective approach, we provide within a toolkit not just some default mechanism by which data can be safely managed within a multi-user system, but also the mechanism by which data distribution is managed. This allows programmers, who may find the default behaviour inappropriate in their case (*e.g.* because of the network topology they are using), to "reach in" and provide new mechanisms which will be used at the toolkit level.

The sample subprotocol outlined above manages data distribution primarily within the `find-object` and `propagate` mechanisms. Using `find-object`, we can encode a variety of mechanisms for mapping between interface representations and underlying shared data. In a centralised system, `find-object` will always return a pointer to the central object store and `propagate` will return the locally changed object to the server. In a fully replicated algorithm, the shared object reference is always local, and more complicated methods on `propagate` will allow changes to be synchronised appropriately. However, this approach does not merely provide a switch between these two modes, but rather 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.

Further, we can extend this to a dynamic model. Consider two users sharing a "scrawl"-style whiteboard application, connected on the same ethernet segment. Round-trip packet times and data integrity requirements are low; a centralised approach to data management will be entirely adequate. However, a third user joins their conference, connected via a much slower dial-up line, from some distance. In this situation, a centralised approach is no longer appropriate; 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 centralised to a replicated data representation. A reflective approach provides the potential for multiple behaviours within the same generic framework, thus supporting this form of dynamic adaptation. If distribution is associated with an object through a mixin[5] class, then changing the class of the object will result in the dynamic switch to a different behaviour.

4.1.2 Exclusion

Exclusion, or locking, is the mechanism by which the system ensures that conflicts are avoided or resolved. A conflict may occur, for instance, when two users apply a change to the same object at once. Different CSCW systems have taken different approaches to this problem. Some, such as ShrEdit [McGuffin and Olson, 1992], lock regions of the shared workspace so that it is impossible for two users to make a change at once. Others, such as GROVE [Ellis and Gibbs, 1989] use an algorithm which "fixes up" conflicts afterwards, in effect imposing a serialisation on edit changes which users make.

Even if we choose a single approach, such as locking rather than serialisation, then we must consider the impact which particular locking mechanisms, defined within the toolkit, might have on higher-level usage issues. For instance, a form of locking which required explicit requests and acknowledgments from the users would be inappropriate in, for instance, free-form sketching or brainstorming tools, as the locking mechanism would interrupt access to the workspace. On the other hand, looser locking mechanisms might fail to provide adequate support for applications which require strong guarantees of data integrity, such as a collaborative CAD application which directly controls milling machinery.

In attacking these problems within the reflective toolkit, we provide a metalevel interface which defines the generic operations involved in requesting, obtaining and releasing locks. In terms of the simple protocol outlined earlier, the principal focus is the `lock-object` call, and the implicit `release-lock` called from `propagate`. Again, the protocol in itself does not embody a locking policy, but rather a procedure by which locks are obtained, and a facility for creating and installing new mechanisms. The generic function specifies that, in addition to the object to be locked, the user requesting the lock and the type of object to be locked are also arguments; using CLOS's facility for multi-methods[6] allows the system to invoke different locking strategies for different users or activities, as well as for different sorts of objects within the same system. The implementor can depend on the object system's *generic dispatch* mechanism to dynamically select the appropriate locking implementation.

This mechanism is sufficiently open that we can define quite different locking strategies from those typically employed. Not only will it allow the implementation of standard strong and weak locks, but also multi-way locks, tickle locks and so on. Indeed, we can reproduce schemes such as the dOPT algorithm used in GROVE, in which explicit locks are not used; instead, the function of lock-object is to construct an appropriate "state vector" which will be distributed by the call to propagate, so that other nodes can use this information to resolve. In this case, we regard the dOPT state vector as an implicit "lock", in the sense that it is an object which will allow conflict resolution.

4.1.3 Interface Linkage

One of the most obvious differences between CSCW systems is the level at which they link interface features. The grossest level of linkage is screen-replication, such as that of Timbuktu [Faralon, 1987]. Shared X systems [Garfinkel *et al*, 1989] replicate at the level of windows, while many explicit multi-user tools such as ShrEdit will replicate the data and provide all users 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, including that of Dewan and Choudhary [1991] or Haake and Wilson [1992], has looked at the provision of switchable linkage states, in which users can choose how much their interfaces will mirror each other's. (A similar two-mode

5. A mixin class is one which is added to another class (through multiple inheritance) in order to provide additional functionality. The word "mixin" derives from the terminology of Symbolics' Flavors system, which was based on an extended analogy with ice cream technology.

6. A standard method is selected on the basis of a single distinguished argument—the recipient, in message-passing terms. A multi-method discriminates on multiple arguments.

switching facility was present in rIBIS [Rein and Ellis, 1991].)

Once again, we can see the requirements for flexibility within applications and toolkits, and that this flexibility can have a dynamic component. The non-dynamic aspect is the primary toolkit-level problem; that different applications require different linkage strategies, and so, if the toolkit is to be applicable to a range of applications, then it must be able to support a range of linkage options. Dewan's work with Suite, or Haake and Wilson's with SEPIA, tackle this problem, as well as looking at the dynamic problem of the need to switch between these different linkage modes in the course of a collaboration or collaborative session. However, they provide flexible linking through specific "modes", which define those components which will be linked at a given time. If different users require different linkage strategies, or a group (or, indeed, applications developer using a toolkit) requires a strategy which has not been predefined, then they have no other recourse. These mechanisms are *parameterised* rather than being *open*.

The approach which is being taken in the reflective toolkit is to make aspects of interface components, such as menus, button states and cursor positions, into classes of objects which, like the data the users are trying to share, can be part of a "shared workspace". This is achieved through a mechanism similar to that outlined earlier for dynamic distribution algorithms, and indeed it can be seen as an example of that use. Defining data distribution methods as specialising on a distributed data mixin class provides the programmer with the ability to dynamically reclass interface facilities as distributed objects. When this happens, the methods which control locking, access, change and propagation, outlined in the subprotocol, also apply to interface objects.

This implies that the standard mechanisms of data distribution used for user data can also apply to these interface components, thus making them accessible to others. 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 other's 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, so that the linked aspects of the users' interfaces can be dynamically controlled and adjusted as the collaboration continues.

4.2 Reflection and Adaptive Computation

The reflective approach is a particular and fairly strong example of a more general trend in a wide variety of computational systems to be able to reconfigure and adapt to particular circumstances. This is perhaps derived in part from a move towards specifying mechanism rather than policy within application frameworks, but goes further in considering aspects of dynamic reorganisation of systems infrastructures. Related approaches occur in areas such as network protocol management (*e.g.* [O'Malley and Peterson, 1992]) interprocessor communication strategies ([Felten, 1992]) and microprocessor design (*e.g.* [Athanas and Silverman, 1993]). Each of these embodies the principle that completely closed abstractions are not always appropriate for high-level systems design; from the perspective of toolkit programming, they attempt to avoid the premature commitment problem by allowing information from the higher (application) levels to influence decisions at the lower (toolkit) level.

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

5 Conclusions

The primary focus of this paper 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) customisation facilities, the role of work practice and situation in system use, and the coadaptive nature of system use and user behaviour lead us to a model of systems design which emphasises openness, dynamic behavior and evolution of systems and practices.

The move away from static systems forces a reconsideration of the architectures which underlie interactive systems. 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 towards 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 behaviour to a range of current problems in interactive system design.

Acknowledgments

The ideas expressed in this paper 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. Victoria Bellotti, Matthew Chalmers, Jon Crowcroft, Steve Freeman and Allan MacLean also made helpful comments on earlier drafts of this paper.

References

[Ahuja *et al*, 1990] S. R. Ahuja, J.R. Ensor and S. E. Lucco, *"A Comparison of Application Sharing Mechanisms In Real-time Desktop Conferencing Systems"*, in Proc. ACM Conference on Office Information Systems COIS'90, Boston, Mass., April 1990.

[Athanas and Silverman, 1993] Peter Athanas and Harvey Silverman, *"Processor Reconfiguration Through Instruction-Set Metamorphosis"*, IEEE Computer, March 1993.

[Bobrow *et al*, 1988] Daniel Bobrow, Linda Demichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales and David Moon, *"Common Lisp Object System Specification"*, X3J13 Document 88-002R, June 1988.

[Booch, 1991] Grady Booch, *"Object Oriented Design"*, Benjamin/Cummings, Redwood City, Ca., 1991.

[Crowley *et al*, 1990] Terry Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick and Raymond Tomlinson, *"MMConf: An Infrastructure for Building Shared Multimedia Applications"*, in Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'90, Los Angeles, October, 1990.

[Dewan and Choudhary, 1991] Prasun Dewan and Rajiv Choudhary, *"Flexible User Interface Coupling In a Collaborative System"*, in Proc. ACM Conf. Human Factors in Computing Systems CHI'91, New Orleans, Louisiana, April 1991.

[Dourish and Bellotti, 1992] Paul Dourish and Victoria Bellotti, *"Awareness and Coordination in Shared Workspaces"*, in Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92, Toronto, Canada, November 1992.

[Ellis and Gibbs, 1989] Clarence Ellis and Simon Gibbs, *"Concurrency Control in Groupware Systems"*, Proc. ACM Conference on Management of Data SIGMOD'89, Seattle, Washington, 1989.

[Farallon, 1987] Farallon Computing, *"Timbuktu: The next best thing to being there"*, 1987.

[Felten, 1992] Edward Felten, *"The Case for Application-Specific Communication Protocols"*, Technical Report TR-02-03-11, Department of Computer Science, University of Washington, Seattle, Washington, 1992.

[Garfinkel *et al*, 1989] Dan Garfinkel, Phil Gust, Mike Lemon and Steve Lowder, *"The SharedX Multi-User Interface User's Guide, Version 2.0"*, Software Technology Lab Report STL-TM-89-07, Hewlett-Packard Laboratories, Palo Alto, Ca., 1989.

[Greenberg, 1991] Saul Greenberg, *"Personalisable Groupware: Accommodating Individual Roles and Group Differences"*, in Proc. European Conference on Computer-Supported Cooperative Work ECSCW'91, Amsterdam, Netherlands, September 1991.

[Greenberg *et al*, 1992] Saul Greenberg, Mark Roseman, Dave Webster and Ralph Bohnet, *"Human and Technical Factors of Distributed Group Drawing Tools"*, Interacting with Computers 4(3), pp. 364–392, 1992.

[Grudin, 1991] Jonathan Grudin, *"Obstacles to User Involvement in Software Product Development, with Implications for CSCW"*, Intl. Journal of Man-Machine Studies, 34, pp. 435–452, 1991.

[Haake and Wilson, 1992] Jorg Haake and Brian Wilson, *"Supporting Collaborative Writing of Hyperdocuments"*, in Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92, Toronto, Canada, November 1992.

[Harper *et al*, 1991] Richard Harper, John Hughes and Dan Shapiro, *"Harmonious Working and CSCW: Computer Technology and Air Traffic Control"*, in Bowers and Benford (Eds.) ''Studies in Computer Supported Cooperative Work'', North-Holland, Amsterdam, 1991.

[Kay, 1993] Alan Kay, *"The Early History of Smalltalk"*, in Proc. ACM Conf. History of Programming Languages HOPL-II, Cambridge, Mass, published in SIGPLAN Notices 28(3), March, 1993.

[Kiczales *et al*, 1991] Gregor Kiczales, Jim des Rivières and Danny Bobrow, "The Art of the Metaobject Protocol", MIT Press, Cambridge, Mass., 1991.

[Kizcales, 1992] Gregor Kiczales, *"Towards a New Model of Abstraction in Software Enginering"*, in Proc. IMSA'92 Workshop on Reflection and Metalevel Architectures, Tokyo, Japan, Nov. 4–7, 1992.

[Mackay, 1990] Wendy Mackay, *"Users and Customisable Software: A Co-Adaptive Phenomenon"*, PhD thesis, Sloan School of Management, MIT, Cambridge, Mass., 1990.

[Mackay, 1991] Wendy Mackay, *"Triggers and Barriers to Customising Software"*, in Proc. ACM Conference on Human Factors in Computing Systems CHI'91, New Orleans, Louisiana, April 1991.

[McGuffin and Olson, 1992] Lola McGuffin and Gary Olson, *"ShrEdit: A Shared Electronic Workspace"*, CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, 1992.

[MacLean *et al*, 1990] Allan MacLean, Kathleen Carter, Thomas Moran and Lennart Lovstrand, *"User-Tailorable Systems: Pressing the Issues with Buttons"*, in Proc. ACM Conference on Human Factors in Computing Systems CHI'90, Seattle, Washington, April 1990.

[Maes, 1987] Pattie Maes, *"Computational Reflection"*, Technical Report 87.2, Artificial Intelligence Laboratory, Vrije Universiteit, Brussels, Belgium, 1987.

[O'Malley and Peterson, 1992] Sean O'Malley and Larry Peterson, *"A Dynamic Network Architecture"*, ACM Transactions on Computer Systems, 10(2), May 1992.

[Patterson *et al*, 1990] John Patterson, Ralph Hill, Stephen Rohall and Scott Meeks, *"Rendezvous: An Architecture for Synchronous Multi-User Applications"*, in Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'90, Los Angeles, Ca., October 1990.

[Rao, 1991] Ramana Rao, *"Implementational Reflection in Silica"*, in Proc. European Conference on Object-Oriented Programming ECOOP'91, Geneva, Swizerland, 1991.

[Rein and Ellis, 1991] Gail Rein and Clarence Ellis, *"rIBIS: A Real-Time Group Hypertext System"*, Intl. Journal of Man-Machine Studies, 34, pp. 349–367, 1991.

[des Rivières and Smith, 1984] Jim des Rivières and Brian Smith, *"The Implementation of Procedurally Reflective Languages"*, Technical Report ISL-4, Xerox PARC, Palo Alto, California, June 1984.

[Royce, 1970] W. W. Royce, *"Managing the Development of Large Software Systems"*, Proc. WESTCON, Ca., 1970.

[Smith, 1982] Brian Smith, *"Reflection and Semantics in a Procedural Language"*, Report MIT-TR-272, MIT Laboratory for Computer Science, Cambridge, Mass., 1982.

[Suchman, 1987] Lucy Suchman, *"Plans and Situated Actions"*, Cambridge University Press, Cambridge, UK, 1987.

[Trigg *et al*, 1987] Randy Trigg, Tom Moran and Frank Halasz, *"Adaptability and Tailorability in NoteCards"*, in Proc. Interact'87, Stuttgart, Germany, 1987.