This paper appears in ACM Transactions on Computer-Human Interaction, 5(2), 109-155.

Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications

Paul Dourish

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto
CA 94304 USA
dourish@parc.xerox.com

Abstract. Ideally, software toolkits for collaborative applications should provide generic, reusable components, applicable in a wide range of circumstances, which software developers can assemble to produce new applications. However, the nature of CSCW applications and the mechanics of group interaction present a problem. Group interactions are significantly constrained by the structure of the underlying infrastructure, below the level at which toolkits typically offer control.

This article describes the design features of Prospero, a prototype CSCW toolkit designed to be much more flexible than traditional toolkit techniques allow. Prospero uses a metalevel architecture so that application programmers can have control over not only how toolkit components are combined and used, but also over aspects of how they are internally structured and defined. This approach allows programmers to gain access to "internal" aspects of the toolkit's operation that affect how interaction and collaboration proceed. This article explains the metalevel approach and its application to CSCW, introduces two particular metalevel techniques for distributed data management and consistency control, shows how they are realised in Prospero, and illustrates how Prospero can be used to create a range of collaborative applications.

Categories and Subject Descriptors: Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; D.2.2 [**Software Engineering**] Tools and Techniques—*user interfaces*; H.1.2 [**Models and Principles**] User/Machine Systems—*human factors*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*theory and models.*

General terms: Design, Human Factors, Languages.

Additional key words and phrases: metalevel programming, open implementation, data distribution, divergence, consistency control, consistency guarantees, software architecture.

1 INTRODUCTION

In the early days of collaborative system design, the software components for multi-user interaction and collaboration were crafted by hand for each application. Object replication and sharing infrastructures, consistency management mechanisms and multi-user interface widgets were written from scratch to produce experimental systems for investigating the use of computer tools supporting collaborative work.

In this regard, CSCW development is no different from any other area of experimental system design. Being a pioneer means exploring an area where no-one else has been, and so there is nothing left over from previous trips to be reused. Nor is CSCW different from any other area in terms of subsequent development patterns when the needs of a novel application area are better developed. Common features of system design emerge, and developers begin to identify the core system elements and requirements which the application domain defines. As these become better understood, toolkits of reusable, widely applicable components emerge.

A cursory look at the historical development of CSCW applications and systems shows this trend. Most systems described at the first CSCW conference in 1986 (such as those of Greif and Sarin [1986]), or in the earlier workshop in 1984, were designed from scratch; however, by the third ACM conference in 1990, toolkits such as MMConf [Crowley et al., 1990] and Rendezvous [Hill et al., 1994] had emerged offering programmers components which had been found to be generally applicable in a range of situations.

This issue of general applicability is the crux of the design of any toolkit, and will be the primary focus of this article. In particular, we will be concerned with the fact that toolkit designer must furnish the application designer with a set of features regular enough to apply in a range of circumstances, but flexible enough to support the needs of different applications. The problem that motivates the research described in this article tackles is, what technology can be brought to bear on the resolution of this tension?

I will argue that this design issue, common across all toolkits, is particularly problematic in the CSCW domain. I will present the particular techniques used in the design of Prospero, a radically flexible CSCW toolkit, to address these problems. Prospero embodies three particular techniques which will be addressed here. The first, *Open Implementation*, is a general architectural technique which has recently emerged as a solution to the engineering requirement for flexible abstractions in a range of domains, providing metalevel control over implementations. The potential applicability of Open Implementation to CSCW design was originally introduced in a previous article [Dourish, 1995]; this article focuses on how the approach has been realised and applied in the design of Prospero. The second and third techniques, *divergence/synchronisation* and *consistency guarantees*, are particular metalevel techniques developed in Prospero for CSCW applications. These techniques are designed to work well within an Open Implementation framework, but can be applied independently.

The structure of the rest of this article is as follows. Section 2 will outline some issues in CSCW toolkit flexibility. It will present evidence for the particular need for flexibility in CSCW, using studies of collaborative activity to point to the interdependence of usage patterns and system infrastructure. Traditionally, these concerns, the “high level” and the “low level”, are regarded as being relatively independent, and have been approached from this perspective; however, I will argue that the interdependence forces us to take a different approach. Section 3 will introduce the Open Implementation approach which Prospero embodies, and outline the general structure of Prospero’s solution to the problems raised in Section 2. Subsequently, Sections 4 and 5 will discuss divergence/synchronisation and consistency guarantees, the two CSCW-specific techniques used in Prospero. Sections 6 and 7 will show how Prospero is used to construct collaborative applications.

2 FLUID USE AND STATIC INFRASTRUCTURE

The goal of any toolkit design is to provide generic, reusable components that are applicable to a wide range of applications. The value of the toolkit lies in the range of applications that can be supported. However, every application has different requirements. The design tension, then, lies in the creation of components that can be general enough to apply widely, but which can also support the rich set of *specific* needs of those applications.

Along with other system components such as operating systems and network services, toolkits provide elements of application infrastructure—elements that will be deployed by an application in various ways, but which are independent of the specifics of the application domain. So, for example, a user interface toolkit might offer widgets such as pull-down menus and scroll-bars; and a programming language will provide data structures and control structures which can be combined to provide an infinite range of specific configurations. This depends upon a separation between the "low level" features of infrastructure and the "high level" features of an application.

One important feature of CSCW toolkits and applications, however, is that significant interactions emerge between these low and high levels. As will be argued in Section 3, similar interactions can be found in other areas, but the goal of this section is to show that these issues are particularly problematic in collaborative settings, because the static elements of infrastructure frequently interfere with the essentially dynamic elements of what is to be supported—collaborative working.

I will begin, in Section 2.1, by outlining some of the elements of CSCW infrastructure, the fundamental computational elements on which CSCW applications depend. In Section 2.2, I will discuss a number of studies of group working that illustrate how the progress of group work can be systematically undermined by the issue of infrastructure configuration. Section 2.4 will outline how these problems have been addressed in other work.

2.1 Elements of CSCW Infrastructure

There are a number of common elements which arise in the design of CSCW applications, and are therefore areas in which CSCW toolkits can provide support for application developers. In this article, we will focus particularly on data distribution and consistency control.

Data distribution concerns the way in which application data in a collaborative application is distributed across the network nodes participating in that session. Centralised and replicated strategies vary in the number of copies of any data item which are available within the system at any given moment. Replicating data items lowers latency but introduces problems of synchronisation. Static and dynamic strategies vary in their approach to the movement of data items within a network. Moving data around may allow it to be located close to the site of current activity, but introduces problems of finding data items when they are needed.

When multiple copies of a single data item are available at multiple points in a network, then issues of *consistency management* arise. A variety of strategies can be adopted to ensure that data copies are maintained in synchronisation with each other. These might involve interaction mechanisms that ensure that only a single activity can be performed over a data item at any given moment, sequencing mechanisms that ensure consistent execution orders for different actions, or resolution mechanisms that combine potentially simultaneous actions into a unified sequence of logical actions over the data store.

Although this paper will deal largely with these two areas of concern, there are, of course, other areas of CSCW infrastructure which arise in the design of CSCW software toolkits. Examples include user interface management, user interface linkage, interface creation, session management, access control and awareness provision. In each of these areas, the same general concerns arise with the balance between control and flexibility and the wide range of possibilities for the provision of basic functionality.

These areas of concern are presented here as elements of CSCW infrastructure, the underlying technology on which applications rest. What are the consequences of this? Primarily, "infrastructure" here carries two implications. First, it implies that these concerns become invisible to the application programmer, since they are carried out below the level of abstraction at which he or she operates. Applications are developed by manipulating abstract entities such as "shared objects" which encapsulate strategies for distribution, replication, synchronisation and awareness. Second, and relatedly, it implies that the application programmer can have no control over these strategies and mechanisms, since he or she is denied even the terms in which to talk about them. Consequently, then, the particulars of the application domain can play no part in the selection of appropriate design decisions in the infrastructure.

These aspects of infrastructure development are well-known and derive directly from the way in which software systems are developed and deployed. This paper is concerned with particular consequences of this approach for the development of collaborative systems, and in turn with new opportunities for the design of collaboration toolkits which follow from them.

2.2 Flexibility in Cooperative Work

The presentation of different dimensions of flexibility in CSCW design has been driven, so far, by technical demands and technical criteria. However, one reason that these issues are particularly salient in the design of collaborative systems is the flexible working styles observed in collaborative work settings. The issue here is not simply that people engage in collaborative work in widely differing ways; it is that these different working styles carry with them very different implications for the configuration of infrastructure and so, in turn, the variability in working methods calls into question the "encapsulation" of infrastructure design decisions. This section discusses some particular studies to illustrate this point.

Dourish and Bellotti [1992] present findings from a set of laboratory investigations of the use of ShrEdit, a synchronous collaborative text editor. Groups of three collaborators used the system to support their collaboration on an experimental design task. One significant aspect of Dourish and Bellotti's description is the wide variety of ways in which the groups organised their activity. In the absence of any direction about how best to use the tool to support their collaboration and interaction, the groups adopted different working styles to support different forms of collaborative work. For example, some participants divided up the activity between them and then created separate documents representing these different streams of activity. Some used private documents for individual work and shared windows only for group-sanctioned material; and others used a single large shared document as a general collaborative workspace.

Dourish and Bellotti also report on the ways in which individuals would use specific features of the interface to support the fluid coordination of their activities. The system provided interactive shared feedback, showing each user's activity within the document to all users synchronously. This

allowed the collaborators to monitor each other's activities, observe other's progress, gauge their own progress in relation to the progress of the group, opportunistically get involved in the activities of others, and so on. The shared feedback approach was critical not simply to the sort of fine-grained, focussed collaborative activity for which it had been designed, but also for awareness-based coordination in general.

Beck and Bellotti [1993] studied collaborative text processing in a quite different domain, the collaborative authoring of research papers. In the case study that forms the core of their paper, two research collaborators working at different sites separated by thousands of miles work together on a conference paper reporting on joint research, using traditional tools (single-user document editing systems, data networks and telephones) to coordinate their work. One point that Beck and Bellotti stress from this and other studies is the opportunistic way in which individual work is often carried out in these settings. They observe that, even in cases where a division of responsibility for different sections of an emerging document has been decided in advance by the collaborators, this is subject to continual, ongoing and *individual* reconsideration at any given moment. So, one collaborator might well make changes to a part of the document for which she does not have responsibility, perhaps because it is relevant to her own sections, or because she notices an error, or for any of a number of other possible reasons. The significant point here is not simply that these opportunistic activities take place, but that they take place *unproblematically* in the course of the collaboration. They do not, typically, interfere with the organisation of the work or cause the work to be interrupted (indeed, they are often carried out so as to ensure the smooth flow of work). Beck and Bellotti relate their observations to the design of systems which embody and enforce the informal division of labour through access control or process management mechanisms.

In a study of the use of Software Configuration Management (SCM) tools in a production software engineering environment, Grinter [1996] describes how software developers use and abuse the mechanisms provided by the SCM system in the coordination of their own activity. The focus of design attention in the SCM system is primarily the software that it manages, but the focus of the software engineers is their own work. They mediate, then, between a level of description cast in terms of the (software) dependencies between different code modules, and the (social and practical) dependencies between their individual and collective efforts. Grinter details practices which emerge in different settings around the use of one particular SCM tool, and show how the software developers use the tool's visualisation of the dependency structure of the software system as a resource in organising their own activity.

These studies, although carried out in different domains and organised around different sorts of systems, carry some general implications for technological design. They show how individuals and groups creatively deploy the resources at their disposal to accomplish their work in different settings. As groups, settings and circumstances change, so do the ways in which work is conducted and, by implication, so do the ways in which people make use of the technology. The consequence for system design is that we must avoid premature commitments to particular styles of work or to particular processes by which work may be conducted, since these are subject to instantaneous revision and variation. The reason that this is problematic, however, is that any particular technological approach embodies just these sorts of commitments and constraints. Building implementations inherently involves making commitments to styles of work. In other words, the concern is not

simply with the *specific* techniques which have been chosen in any instance, but with the very question of choosing one technique and encoding it in the system.

To an extent, we are used to dealing with this problem—managing the appropriate technological choices for working styles—in the user interface domain. However, this problem of commitment applies not only to the traditionally "high-level" issues of interface design and working activity, but also to the traditionally "low-level" issues of infrastructure support.

A study by Greenberg and Marwood [1994] illustrates just this issue, exploring it in the context of consistency management algorithms for collaborative applications. Consistency management is traditionally provided as infrastructure as described earlier, outside the control of the application programmer. However, Greenberg and Marwood demonstrate a range of significant interface design implications for different consistency designs. For example, not only may locking strategies introduce delays which interfere with the fluid management of parallel tasks, but it can also prevent group members from using text selection as a gestural mechanism accompanying discussion of the document. Consistency management, however, is normally taken to be an infrastructure issue, invisible at the level of the interface and subject primarily to technical rather than usage criteria.

In other words, we must be concerned, in CSCW design, not only with the design of infrastructure and the practices of work at the interface, but with the *interactions between the two*. The requirements for flexibility at each level are inter-dependent. Typical system development practice concentrates on computational techniques or on supporting particular styles of work, but not on the relationship between these two concerns; and that is the topic which is addressed here.

2.3 Varieties of Technical Flexibility

What we see when we look at these sorts of studies, then, is the wide variety of ways in which people coordinate their collaborative behaviour around the resources which systems provide for individual and collective activity. These observations result in a reorientation of our view of the goals of system design. Instead of thinking of systems as embodying *mechanisms* of coordination and collaborative activity, we see them instead as *media* for the performance of collective action [Bentley and Dourish, 1995]. We are concerned not simply with the functionality of collaborative applications, but also with the flexibility they provide, allowing users to adapt them to local needs and circumstances and to different ways in which to engage in work.

When we consider the issues of flexibility and malleability in particular collaborative applications, we find that there are a number of different levels at which they can be addressed and, significantly, they are strongly inter-related. The most straight-forward approach is the provision of run-time *customisation*; opportunities for users to adjust parametric controls over the application's behaviour, such as interface coupling modes, access control checks, and so forth. Another technical approach would be to exploit run-time *adaptation* in which the system can re-configure its behaviour according to immediate patterns of use. So, for example, if a particular group of users organises their behaviour in such a way that each deals only with a subset of the shared objects at their disposal, then the system might organise the distribution of those objects so that their access patterns support this division of labour effectively. These forms of flexibility provides users with the opportunity to engage in differing working styles as appropriate to the specific task and circumstances in which they find themselves.

In contrast with this focus on applications, this article is concerned with the problem of flexibility in CSCW *toolkits*. Flexibility in a toolkit used to create collaborative applications can be an even more serious problem than in the design of the toolkits themselves, since the designer of the toolkit is not in control of the applications which will be developed, and so is even further removed from how collaborative work will actually be performed. However, just as we saw before, the implementation decisions that the toolkit programmer makes (such as the replication strategies for shared objects, for example) impose two levels of constraints: first, they impose constraints on application developers, in terms of the sorts of applications that can be developed using the toolkit; and second, they impose constraints on users, in terms of how those applications can be used.

The approach that I will explore here exploits a meta-level run-time architecture in order to allow programmers and program code to become involved in the implementation of the abstractions that the toolkit provides. The primary area to be addressed is the *scope* of the toolkit—the range of applications that can be developed with it. However, the same run-time architecture is also a basis for other forms of flexibility discussed here—for run-time adaptation to the emergent patterns of collaborative work, and for user involvement in how computational structures are exploited to support collaborative action.

2.4 Flexibility in CSCW Toolkits

The first generation of CSCW toolkits, such as MMConf and Rendezvous, focussed particularly on encapsulating common behaviours from a range of collaborative applications in order to make the development of new applications faster and easier. While they clearly aimed at generality across a number of potential applications, they restricted themselves to designs supporting, e.g. synchronous work (both), centralised data (Rendezvous), etc.

The provision of flexibility has been an important concern in the design of a second generation of CSCW toolkits. Systems developed in recent years have provided open structures for the programmer to create a variety of behaviours within the one infrastructure framework. Some examples are given here.

Suite [Dewan and Choudhary, 1992] provides a variety of levels of flexibility, giving the programmer control over a number of design dimensions. Suite adopts an editor-based approach to collaborative activity in a shared workspace, conceptualising clients as editors for structured data. An access control framework [Shen and Dewan, 1992] is used to provide variable control over updates to the shared workspace, and a rich coupling model allows the peer interface clients to be connected in a variety of ways.

GroupKit [Roseman and Greenberg, 1996] provides flexibility through a facility known as *Open Protocols* to allow for the flexible performance of structured interaction between different components of the system [Roseman and Greenberg, 1993]. Open protocols allow developers to encapsulate aspects of the system's behaviour as server state, which can be controlled by policy-free protocols. Clients can then embody different policies so that different behaviours can be exhibited, making the system more flexible and adaptable. For example, different clients could embody policies dealing with floor control management. The server can then be reused by different clients, since it is free of commitments to particular policies or behaviour strategies.

Intermezzo [Edwards, 1996] is a collaborative toolkit providing radical support for flexibility through two mechanisms. The first is that applications can download code into the toolkit, written in a simple interpreted language, which will affect how the toolkit operates.[1] The downloaded code is executed by the toolkit at run-time, and its activation can be controlled by a variety of parameters which the programmer can set so that the system as a whole will respond to particular changes in the world or in the collaborative activity. The second support mechanism is the use of dynamic roles as a context-setting mechanism. Roles are defined by membership predicates rather than by explicit membership listings, and the roles can be used as the basis of a dynamic, secure access control mechanism. The result is that views of the data store are dependent on the dynamic state of the collaboration, allowing for highly flexible and fluid responses to the progress of collaboration.

2.5 Summary

Toolkit flexibility is a serious concern in any domain, because of the degree of separation between toolkit developer and application end-user. The toolkit developer's decisions constrain, first, the design of applications that the toolkit developer cannot know in advance and, second, the interactions of end-users whose collaboration is founded on the run-time infrastructure provided through toolkit features. In collaborative work, given the fluid and flexible nature of group activity, it is a particularly problematic area. We have seen some of the dimensions of this problem, and briefly pointed to some existing solutions embodied in collaborative systems and toolkits. The different approaches that they embody offer different sets of design choices to application programmers, and so open up different regions of the design space.

I am concerned here with the *scope* of a toolkit, or the range of system designs it encompasses, and the work reported here has been directed towards opportunities for radical flexibility in a CSCW toolkit. Earlier systems such as MMConf or Rendezvous were designed to support a range of application domains, but generally focussed on a fixed style of collaboration (synchronous or asynchronous, baton-passing or free-form, etc.) In contrast, and in response to the observed requirements for fluid working style outlined above, Prospero is a prototype collaborative toolkit organised around an explicit architectural model for flexibility. Prospero lies in the second generation of collaborative toolkits. In contrast to other second-generation toolkits, it explores the use of a metalevel architecture in attempting to take a principled approach to the provision of flexibility for programmers and users. The metalevel architecture provides a framework for exploring how application developers can customise the toolkit to support the requirements of specific applications and settings. The next section will introduce the architectural principles behind Prospero's designs, before subsequent sections explore the CSCW-specific solutions that it embodies.

3 OPEN IMPLEMENTATION

The approach adopted in Prospero and explored in this paper draws from ongoing work in software architecture and other areas of system design that face some of the same problems as those outlined above. In particular, Prospero adopts the Open Implementation approach [Kiczales, 1996]. The use of this approach, and the principle of Computational Reflection [Smith, 1984] on which it is based, has been explored elsewhere [Dourish, 1995], and so I will restrict myself to a relatively brief precis here.

1. This approach has much in common with the Open Implementation approach which will be introduced later, although it abandons the abstract metalevel interface of the OI approach in favour of direct access to internals.

Developments in a variety of systems domains point to the emergence of a particular class of problem, in which the traditional "upper layers" of a system have access to detailed information that should be used to configure the behaviour of the lower layers. The problem here lies in the fact that our traditional model of abstraction does not allow for this downward flow of information. Different strategies have emerged as these problems have arisen in different domains. For example, the use of Application Layer Framing and Inter-Layer Processing in network protocol implementation [Clark and Tennenhouse, 1990; Braun and Diot, 1995] or the composable microprotocols of the *x*-kernel [O'Malley and Peterson, 1992] or Horus [van Reneese et al., 1996] are responses to just this problem. In Operating System design, mechanisms for user-level control over page replacement policy [Harty and Cheriton, 1992] or filesystem cache management [Cao et al., 1994] are solutions that have arisen in response to the same class of problems.

3.1 Open Implementation Analysis

The Open Implementation approach provides a general analysis of this class of problems, and a technical solution. The Open Implementation analysis is as follows.

The traditional model of abstraction separates client from implementation by encapsulating the implementation in a black box. Access to this implementation is provided through an abstraction barrier, which provides a view onto implementation functionality in terms of a higher-level abstraction.

In the course of constructing an implementation, the implementor must make a variety of *mapping decisions*—decisions about how aspects of the high-level representation and functionality should be mapped onto the lower-level facilities available. Issues of representation and algorithmic encapsulation are instances of mapping decisions of this sort. In choosing between different ways of making these mapping decisions, the implementor must anticipate the needs of future clients of the implementation. The mapping decisions *support* the abstraction offered at the abstraction barrier, but they are not part of it, since it would be the same abstraction even if the decisions had been made in other ways. Since they are not part of the abstraction, the set of mapping decisions made by an implementor are invisible to the client.

However, any useful implementation (or any useful abstraction) will have more than one client. Each client will perform different activities, making use of the same abstraction. However, since clients have different needs of the implementation, they may have different requirements for the way in which the mapping decisions have been made. For instance, consider an implementation of a simple records processing system. A client that primarily adds and removes large numbers of records would wish that the implementation be optimised for addition and deletion; and so the programmer might hope that the implementor mapped the records onto a linked list rather than a sorted array. However, the programmer of another client, say one that retrieves records, might hope just the opposite, since a sorted array will be considerably faster to search than an unsorted linked list.

These problems inevitably arise whatever actual implementation decisions are made. In general, the more clients make use of an implementation, the more likely it is that there will be this sort of conflict between the needs of one client and the needs of another.

3.2 Reflection and the OI Solution

The Open Implementation solution is based on the principle of *Computational Reflection* [Smith, 1984]. Reflective systems contain representations of their own activity that are "causally connected" to the activity they describe. Self-representation means that systems can examine their own state, structure and configuration by examining the representation; the "causal connection" means that manipulations of the representation will be reflected in changes in the system's behaviour[2]. In programming languages, where this approach was first fully explored, reflection provided a means to make the execution environment part of the language, and to do so coherently at a high level. As a result, reflective languages allow programmers to write portable debuggers and other tools that become involved with the execution behaviour; and extensions to the language syntax and semantics are also possible from within the language itself. The Common Lisp Object System (CLOS) [Gabriel et al., 1991] embodies this sort of mechanism through its *metaobject protocol* [Kiczales et al., 1991], an object-oriented encapsulation of the reflective representation that allows CLOS programs to become involved in the implementation of the language.

Reflection gives us a way to address the problems outlined above in the Open Implementation analysis. The OI solution is to provide, within the implementation, a representation of its own structure and behaviour, along the same lines as the reflective self-representations. In addition to using the functionality of the implementation through the abstraction barrier, clients can also examine and control the implementation by means of this model. This structure sets up two interfaces to the implementation. The *base interface* is the traditional interface, the abstraction barrier of the black box; while the *metalevel interface* allows programs to look inside the black box and make changes. Changes made through the metalevel interface affect how base level programs will execute. In this way, the programmer has the means to *specialise* the implementation to the specific circumstances of each client.

Key features of the metalevel approach, then, is the *level of description* and the *subject matter* of the two sorts of code—base code (written to the base interface) and meta code (written to the metalevel interface). A simple way to conceptualise the distinction is as a separation between specifying *what* is to happen (base code) and specifying *how* it is to happen (meta code). By encoding this separation in an object-oriented framework, metaobject protocols support the isolation of multiple metalevel strategies, reuse of metalevel code and a separation of base and metalevel descriptions [Kiczales, 1992]. The question of subject matter and level of representation is also key to the distinction between "metalevel extensibility" and "hacking the source" or the use of object-oriented frameworks. In the OI approach, metalevel modification is done *in terms of a specific metalevel interface*. That is, it is performed through an appropriate *abstraction* over the implementation. That abstraction may be presented in terms of a set of classes and methods (an object-oriented encoding), but these do not, in themselves, constitute the implementation. (Kiczales and Rodriguez [1990] present an account of an implementation of CLOS that apply demonstrates the distinction.) The metalevel interface is a *designed artifact*, not simply an epiphenomenon of some particular implementation.

Computational reflection was originally developed in the domain of AI systems and AI programming languages; the first fully reflective system was Smith's 3-Lisp [des Rivières and Smith, 1984].

2. In other words, the representation "causes" the behaviour as well as describing it.

Further explorations largely in the domain of programming languages led to the CLOS metaobject protocol and the development of the Open Implementation approach. Subsequently, the approach has been applied to a range of other application domains, such as window systems [Rao, 1991], operating systems [Maeda, 1996], distributed systems [Stroud and Wu, 1995] and databases [Barga and Pu, 1996].

3.3 An Open Implementation Strategy for CSCW

As outlined above, the Open Implementation approach has been applied in a wide range of settings to provide a solution to flexibility problems of the kind described in section 2. It is an inherently flexible technique providing an approach to the architecture of infrastructure components that supports their subsequent specialisation to the needs of particular settings. The research described in this article is an exploration of the use of this technique as a way of mediating between the needs of toolkit developers and application developers in the creation of CSCW applications. Other systems (as described in section 2.4) have explored this tension; Open Implementation provides us with a new way to partition the problem.

Prospero adopts the Open Implementation approach for CSCW toolkit design. As a CSCW toolkit, it provides a set of objects, mechanisms and behaviours that can be combined to create collaborative applications. As an Open Implementation, it provides application programs with the opportunity to become involved in the implementation of the infrastructure that supports them. It sets up a separation between base level programs and metalevel programs by which the toolkit structures can be used and specialised. By doing this, it allows a wider range of applications to be developed, since the commitments (mapping decisions) to particular expected patterns of use, which are made in the course of infrastructure implementation, are now subject to reconsideration and revision.

The starting point for such a design is a generic, specialisable model of CSCW application behaviour. By *generic*, I mean that this model describes, in general terms, a range of strategies that can be or have been adopted in a variety of systems. By *specialisable*, I mean that any particular example can be operationally described as a refinement of the general model. The model, then, is not simply a tool for the analytic description of CSCW architectures and implementations; it can also be used to generate new ones, as the basis of a reflective implementation.

By adopting the Open Implementation approach, we aim to achieve a number of improvements over other approaches to CSCW flexibility. First, we aim to provide application developers with a wide range of potential behaviours, rather than a restricted set of predefined options. Second, we aim to provide a level of control that allows specialisation to the details of individual applications, rather than the reuse of generic components. Third, we aim to make this specialisation available through high-level abstractions at the metalevel interface.

The sections that follow will introduce models of this sort for the areas of data distribution and consistency management. These have been implemented in Prospero, a prototype reflective CSCW toolkit. Prospero is implemented in CLOS. Although CLOS is itself a reflective programming language, reflection in CLOS is not used to support reflection in Prospero. In addition, Prospero itself does not deal with interface management and window systems; the existing application interfaces have been developed using the Garnet user interface toolkit [Myers et al., 1990].

Section 4 will deal with the divergence mechanism for managing distributed or replicated data, and Section 5 will deal with the consistency guarantees mechanism for consistency management. Sections 6 and 7 will describe and illustrate how Prospero is used in developing CSCW applications.

4 DIVERGENCE AND SYNCHRONISATION

Some of the issues and design options in managing distributed data were raised earlier, in the discussion of CSCW infrastructure. Different systems have taken different approaches to the problem. For example, ShrEdit centralises its data, MMConf replicates it, while Rendezvous and GroupKit adopt hybrid approaches.

As was outlined above, one reason for this variation in approach is that the choice of management strategies has strong implications for the interface and for the nature of collaborative interaction in a CSCW system. Collaborative systems differ crucially from other distributed systems in that not only the application, but also the interface, is distributed. The trade-offs between availability, transparency, consistency and responsiveness must be made with this in mind, and so design must be constantly mindful of the way in which application distribution and interface distribution are mutually influential.

A CSCW *toolkit*, of course, is one step removed again from real applications and real use. The goal is to provide a generic framework in which a range of application behaviours can be realised. In adopting the Open Implementation approach, we want to develop this framework so that it can be made widely applicable through metalevel specialisation. In Prospero, data distribution management is achieved through the *divergence/synchronisation model*.

4.1 Inconsistency Avoidance and Streams of Activity

The approach to the divergence model begins with a simple but crucial observation; that most approaches to data management in CSCW deal with *inconsistency avoidance* rather than *consistency management*. Rather than working to make data consistent, they set up barriers to prevent inconsistency arising in the first place. While this approach might work for managing the activity of distributed system components, it is less useful when trying to manage the activity of distributed users, since it sets up barriers to particular styles of working.

The simplest approach to avoiding inconsistency is to avoid simultaneous action over individual data items. This approach attempts to define *single, global stream of activity* over the data space. Various common CSCW idioms embody this model of a single stream of activity. For example, asynchronous access to the workspace uses the distribution of work in time to share the stream between multiple participants. In synchronous tools, floor control policies have the same effect. Locking mechanisms also operate this way, although they divide activity spatially as well as temporally; locks ensure that each data item is subject to a single thread of control, currently available to whomever holds the lock.

Prospero abandons this attempt to construct or create a single stream of activity out of multi-user activity. Instead, it employs a model of *multiple, simultaneous streams of activity* over user data, and then looks to *manage the divergence* between these streams. Divergence occurs when two streams have different views of the system's state or of the data. This could arise through simultaneous execution of conflicting operations; or through a lag in the propagation of compatible operations.

Since this general view does not imply any particular number of parallel streams of activity, it encompasses the traditional views outlined earlier; they correspond to the special case of just one stream. A model based on divergence and multiple streams of activity is the *more general case*; it subsumes attempts to maintain a single thread of control.

4.2 Divergence and Synchronisation

The divergence model operates as follows. First, we view activity in a collaborative system as the progress of *multiple, simultaneous* streams of activity. Second, we view the emergence of inconsistency as *divergence* between these streams' views of user data. In these terms, the problems of distributed data management focus on the *re-synchronisation* of divergent streams of activity. As the collaboration progresses, the streams of activity continually split and merge, diverge and synchronise. At points of synchronisation, they re-establish a common view of the data store; further individual activity will cause them to diverge again, necessitating further synchronisation further down the line.

The divergence/synchronisation model captures aspects of a number of other CSCW approaches. Some particular cases are discussed here.

4.2.1 Divergence and Versioning

Versioning systems maintain a historical record of the versions of some object that have existed over time. They typically allow multiple versions of an object to exist at once, and in some, multiple versions can be simultaneously active. GMD's CoVer [Haake and Haake, 1993] uses a version system to manage the cooperative work. CoVer, however, emphasises the creation and management of parallel versions rather than the subsequent integration of different versions (divergent streams). Munson and Dewan [1994] go further in providing a framework explicitly organised around version merging, but, like Haake and Haake, they primarily emphasise versioning and merging within a context of "asynchronous" work, rather than as a more general approach to distributed data management. So, while a versioning approach can be supported in the divergence model, by diverging at the level of entire documents (or other coarse-grain objects) and synchronising only occasionally, I want to consider the wider use of divergence as a general strategy than simply a versioning model would support.

4.2.2 Divergence and Operational Transformation

Operational transformation is an alternative technique employed in various collaborative systems [Ellis and Gibbs, 1989; Beaudouin-Lafon and Karsenty, 1992; Nichols et al., 1995]. Operational transformation employs a model of multiple streams, and uses a transformation matrix to *transform* records of remote operations before applying them locally, using information about the different contexts in which the operations arose. So, for example, a remote delete operation might be transformed into a null operation locally if the object was simultaneously deleted by the local user; or an insertion might be moved to take account of local activity.

Operational transformation, particularly with its basis in multiple streams, is clearly more similar to the divergence approach than versioning. An operational transformation approach can be exploited within the divergence framework by recording actions, synchronising frequently and performing the transformation as part of the synchronisation process. However, there are two principal differences between divergence and operational transformation. First, just as versioning approaches have typically emphasised *asynchronous* activity, operational transformation has typically emphasised

synchronous; as will be discussed, Prospero's model seeks to encompass both. Second, operational transformation relies upon the transformation matrix to resolve conflicts (easier in the tightly-coupled, synchronous domain); whereas Prospero employs a more general notion of synchronisation that potentially offers a much wider scale of applicability (including user intervention).

4.2.3 Divergence and Replicated Databases

One area of research in which divergence has been considered is replicated database management. In a replicated database, multiple copies of all or part of the database are maintained in parallel, in order to increase availability. Since such a model could also be used as the basis of a collaborative system, the relationship of database techniques to Prospero's approach naturally arises.

The primary difference between the two is that Prospero relies on database-external management of consistency, while database approaches typically attempt to maintain consistency internally to the database. What this means is that, while databases might, for instance, seek to find execution orders for transactions that increase the opportunities for parallel execution, they do this within a model which still attempts to maintain the consistency of the transaction model itself. Prospero allows inconsistency in the data store, on the basis that the degree to which inconsistency is allowed and the means by which it can be resolved are application issues. Traditional database techniques might allow parallelism below the level of the transaction model, but generally still attempt to maintain transaction-level consistency. So while they attempt to deal with the problems that divergence raises, they are not able to directly exploit it to support multi-user activity.

The relationship between Prospero's approach and replicated databases is discussed in considerably more detail later, in the context of the use of consistency guarantees to control divergence (Section 5).

4.3 Capitalising on Divergence

Divergence-based data management in CSCW offers three particular advantages over other techniques. First, it is highly scalable, supporting inter-application communication from periods of milliseconds to periods of weeks or more. Second, it opens up direct CSCW support for an area of application use—one I term *multi-synchronous*—which is supported poorly or not-at-all by existing approaches. Third, it directly supports common patterns of working activity based on observational studies that are at odds with the models embodied in most systems today.

4.3.1 Scalability

Scalability refers to graceful operation across some dimension of system design. In particular, the dimension we are interested in here is the *pace* of interaction [Dix, 1992]; or, more technically, its relationship to the period of synchronisation.

The period of synchronisation is the regularity with which two streams are synchronised, and hence the length of time that two streams will remain divergent. When the period is very small, then synchronisation happens frequently, and therefore the degree of divergence is typically very small before the streams are synchronised and achieve a consistent view of the data store. When individuals use a collaborative system with a very small period of synchronisation, their view of the shared workspace is highly consistent, since synchronisation takes place often relative to their actions. This essentially characterises "real-time" or synchronous groupware, in which users work "simultaneously" in some shared space that communicates the effects of each user's actions to all participants "as they happen". The synchronous element arises from precisely the way in which the

delay between divergence (an action taking place) and synchronisation (the action being propagated to other participants) is small. This is one end of the "pace of interaction" dimension.

At the other end, synchronisation takes place much less frequently in comparison to the actions of the users. There is considerably more divergence, arising from different sorts of activities that take place between synchronisation points. When the period of synchronisation is measured in hours, days or weeks, we approach what is traditionally thought of as "asynchronous" interaction. A (well-worn) example might be the collaborative authoring of an academic paper, in which authors take turns revising drafts of individual sections or of the entire paper over a long period, passing the emerging document between them.

Within the CSCW community, these sorts of asynchronous interactions have generally been seen and presented as being quite different from real-time or synchronous interactions; "synchronous *or* asynchronous" has been a distinction made in both design and analysis. However, by looking at them in terms of *synchronisation* rather than *synchrony*, We can see them as two aspects of the same form of activity, with different *periods* of synchronisation. Being highly scalable across this dimension, the divergence approach provides the basis of a toolkit that generalises across this distinction.

4.3.2 Multi-Synchronous Applications

In fact, we can exploit a divergence-based view of distributed data management to go further than standard "synchronous" and "asynchronous" views of collaboration.

Standard techniques attempt to maintain the illusion of a single stream of activity within the collaborative workspace. We know, however, that groups don't work that way; it's much more common to have a whole range of simultaneous activities, possibly on different levels. Consider the collaboratively-authored paper again. In the absence of restrictions introduced by particular technologies or applications, individuals do not rigorously partition their activity in time, with all activity concentrated in one place at a time; that is, they do not work in the strongly asynchronous style, one at a time, that many collaborative systems embody. A more familiar scenario would see the authors each take a copy of the current draft on paper (or on their portable computers), and work on them in parallel—at home, in the office, on the plane or wherever. Here we have simultaneous work by a number of individuals and subsequent *integration* of those separate activities; neither synchronous, nor asynchronous, but *multi-synchronous* work. This sort of working cannot be supported by traditional asynchronous "baton-passing" approaches, in which there is essentially a single stream of activity that passes back and forth between authors. Multi-synchronous applications extend asynchronous ones by providing synchronisation for parallel streams in disconnected work.

The divergence model, and in particular the notion of multiple, parallel streams of activity, is a natural approach to supporting this familiar pattern of collaborative work. Working activities proceed in parallel (multiple streams of activity), during which time the participants are "disconnected" (divergence occurs); and periodically their individual efforts will be integrated (synchronisation) in order to achieve a consistent state and progress the activity of the group.

Here, we are concerned with the *nature* of synchronisation; this is what allows for flexibility, and will be discussed in more detail subsequently. At this stage, the details of synchronisation in a variety of cases are not of prime importance; examples will be considered in more depth later on. For the moment, however, what's important is to recognise the support for multi-synchronous working within this model of distributed data management.

4.3.3 Supporting Opportunistic Work

However, the use of divergence-based data management techniques is not simply a route to supporting a different style of working; it's also a means to *more naturally* support the other working styles to which CSCW has traditionally addressed itself.

Consider again the style of working described by Beck and Bellotti [1993]. One of their observations was the way in which collaborative authors would opportunistically step outside the pre-agreed bounds of their own activity and engage in unexpected activity. Suchman [1987] has, of course, made similar telling observations about the improvised, opportunistic emergence of sequences of activity.

However, the notions of a single stream of activity and inconsistency avoidance stand directly in the way of these sorts of behaviours, since they require an early commitment to particular patterns of working activity, and then a rigorous adherence to those patterns in the course of carrying out the work. To support the sort of opportunistic working described by Beck and Bellotti, then, our technology must relax rules about exclusion and partitioning; exactly the rules that have been employed to maintain the fiction of the single stream of activity. So the same sorts of mechanisms that were described earlier as supporting multi-synchronous collaboration have, in fact, a wider range of applicability; they support a more naturalistic means of *making asynchronous collaboration work*. Divergence is a direct consequence of these ways of working; and so a model of distributed data management based on a pattern of repeated divergence and synchronisation fits well with support for a wide range of working styles.

Of course, significant consistency issues arise when the system potentially allows users to engage in arbitrary activity in this way. The discussion of consistency guarantees in Section 5 will be motivated by the need to maintain control under these circumstances.

4.4 Divergence and Synchronisation in Prospero

Section 6 will deal with the embodiment of these mechanisms in Prospero and how they can be used to create CSCW applications. A brief sketch is provided here to ground this discussion.

The primary focus of this model is on streams of activity; and so these are reified in the implementation. Applications create and name streams for the different threads of activity running through an application (such as for clients, virtual servers, recorders and so forth). Operations on streams allow them to be associated with sessions, to find each other, and so forth.

Individual actions are also reified as command objects [Berlage, 1994], and are added to particular streams. When a user engages in some activity, that action creates a command object which is then added to the stream associated with that user at that moment. When actions are added to specific streams, divergence occurs since the other streams have not yet seen that action.

Streams contain their own mechanisms for synchronisation. Multiple stream types are provided that will synchronise in different ways and under different circumstances. A metaobject protocol is used to control the synchronisation process, as well as to modify the conditions under which streams will be synchronised or the means by which that might be accomplished.

4.5 Divergence and Synchronisation: Summary

Prospero models activity over distributed data in terms of an ongoing cycle of divergence and synchronisation between parallel streams of activity. The divergence cycle is scalable across time; rapid synchronisation yields applications which behave like traditional synchronous systems, while slow synchronisation resembles traditional asynchronous system. At the same time, this pattern of divergence, by allowing inconsistency to arise and then resolving it later, opens up the opportunity for applications supporting "multi-synchronous" (parallel but disconnected) work. This model is general enough to capture a wide range of CSCW application mechanisms, as well as rich enough to act as a language in which to specify new ones. The elements of the divergence model, such as streams, actions and synchronisation conditions, provide the elements of a metaobject protocol by which application programmers can tailor the behaviour of the underlying infrastructure to suit the purposes of particular application scenarios.

Examples of programming with the divergence model will be provided in subsequent sections. First, however, I will introduce the second, complementary mechanism provided in Prospero—consistency guarantees.

5 CONSTRAINING DIVERGENCE: CONSISTENCY GUARANTEES

Prospero's divergence/synchronisation strategy is an *optimistic* one. It presumes that simultaneous actions will probably not result in conflict, but that if conflict *does* occur, it can be sorted out later. Data locking, on the other hand, is a *pessimistic* strategy; it presumes that simultaneous operations are likely to lead to conflict, and so should be prevented.

Pessimistic strategies guarantee that consistency will be maintained, since they prevent the simultaneous action that would lead to inconsistency in the first place. On the other hand, optimistic strategies support more open styles of working, and achieve better interactive performance when the working styles do not lead to inconsistency. Prospero uses an optimistic strategy because the freedom and flexibility it provides is better suited to the needs of collaborative work. The price of this freedom is that the toolkit must provide explicit means to maintain consistency.

With any optimistic strategy, the design problem is the detection and resolution of conflicts as they occur. However, the divergence model *per se* explicitly makes no commitment to either the nature or the extent of the divergence, which would help us bound this problem. The longer two streams of activity remain active but unsynchronised, the greater their potential divergence. In turn, the greater the divergence, the more complex it becomes to resolve conflicts at synchronisation-time. Indeed, the system may *never* be able to resolve two arbitrary streams into a single, coherent view of the data store. Essentially, unconstrained divergence leads to arbitrarily complex synchronisation. Some form of consistency management has to be introduced to resolve this problem.

5.1 Variable Consistency

The first approach used in Prospero is to distinguish between *syntactic* and *semantic* consistency. Semantic consistency is the traditional form employed in distributed and collaborative systems; syntactic consistency can, however, provide a means for supporting ongoing work in the face of potential conflicts.

Semantic consistency guarantees that the data store contains no inconsistencies from the perspective of the application domain. This means that processing can continue and the application can operate

effectively over the data. Syntactic consistency is a weaker form. It guarantees only that there is a consistent structure for the data, but that structure might itself hide semantic inconsistencies. Structural integrity allows certain forms of processing to continue, but there are still inconsistencies from the perspective of the application domain.

Consider the co-authoring case again, as an example. If two users make different changes to the same paragraph, an inconsistency arises, and it can be resolved in various ways. To achieve semantic consistency, the system must ensure that there is only one copy of the paragraph which can be seen by all participants. For minor changes, it may be possible to integrate the changes into a single, unified paragraph. If one of the authors removed a sentence while the other inserted two words into a different sentence, then the system could apply both their changes to yield a semantically consistent result. In the face of major changes, though, this may not be possible. If both users deleted the whole paragraph and wrote a new one, then their changes could not be integrated. In cases like this, most CSCW systems look to some other mechanism to achieve semantic consistency, such as by selecting one paragraph rather than the other as the "winner". The selection might be done by looking at the extent of the changes, by looking at the times when the changes where applied, by using the users' "roles" to decide who has control, or even by choosing arbitrarily[3].

An alternative approach is to make the data syntactically consistent—structurally sound but with unresolved inconsistencies. In this example, one way to achieve syntactic consistency would be to wrap the two paragraphs up in a larger structure that presents them both as alternatives. Prospero calls this approach *aggregation*. This preserves the work of both authors, allows them to continue with their work, but leaves an inconsistency to be resolved later.

Different domains provide different means to support syntactic consistency. For numerical data, for instance, it might involve introducing approximate values with error bounds. Whatever the domain, though, the difference between semantic and syntactic consistency provides an opportunity to defer conflict resolution while continuing to allow work.

Prospero allows programmers to deal in terms of different levels of consistency, and so it can employ this strategy to manage consistency fluidly. As observed, though, the actual realisation of syntactic consistency varies in different settings. The most significant feature of Prospero's consistency model is that it operates in terms of *application semantics*, rather than in terms of a predefined model.

5.2 Using Application Semantics

The key observation behind the variable consistency approach above is that the toolkit components, themselves, are not the final arbiters of "consistency". Consistency is application-relative. With variable consistency, the toolkit can focus on making the data consistent *for the purposes at hand* by capitalising on aspects of the application domain and the circumstances in which the toolkit is being used.

However, while using application-specific synchronisation might *postpone* some of the problems of unbounded divergence, the basic problem of unbounded inconsistency remains with us. The

3. One common alternative, of course, is to presume that the users can sort it out and do nothing at all (the "social protocol" approach). Of course, doing "nothing at all" still involves doing *something*; the question is what level of consistency, if any, is maintained by the system's response to this circumstance.

same basic technique—*exploiting application semantics*—can be applied to this problem. Prospero introduces the notion of application-specific consistency guarantees to control for the divergence process using details of particular circumstances.

This approach has its origins in a generalisation of the locking mechanisms used by many existing CSCW systems.

5.3 Data Locking Approaches

The most widespread traditional mechanism for avoiding inconsistency in CSCW systems is *data locking*. In the course of their work, users (implicitly or explicitly) obtain a "lock" for some or all of the data store. Without a lock, no changes can be made; and since a lock is only granted to one user at a time, inconsistency cannot arise. Asynchronous interaction and floor-control algorithms are special cases of the locking approach that lock the whole data store at once.

As outlined earlier, Greenberg and Marwood [1994] discuss some issues surrounding concurrency control in CSCW systems. In particular, they discuss how the choice of concurrency management strategy can have a significant impact on the styles of interaction that an application can support. For instance, the temporal properties of concurrency control strategies, such as relative execution times of actions over shared data, can interfere with interactional requirements in the interface. Similarly, approaches that apply a *post hoc* serialisation on user actions may introduce unexpected interface behaviours (such as undo-ing actions under the users' feet).

Locking is a very general approach. Different particular implementations vary along many dimensions, such as in how the locks are requested, obtained, granted and relinquished, what kinds of operations require locks, and the granularity of data units controlled by a single lock. In these variations, the central lock-act-release strategy, supporting a pessimistic concurrency model, remains unchanged[4].

As Greenberg and Marwood's analysis points out, the implications for group interaction mean that these sorts of strategies are appropriate in some cases, but not others. While the pessimistic locking model may be appropriate for some cases where data integrity constraints are strong (such as collaborative software development), they impose too high an overhead for many looser or free-form collaborative activities. This makes a strong locking model an inappropriate basis for Prospero, since it aims to capture both these sorts of collaborative application.

5.4 Promises and Guarantees

In an attempt to find a more flexible approach than the strict locking mechanism, and one more attuned to the needs of a CSCW toolkit, our starting point is with a generalisation of the traditional locking process.

5.4.1 Guarantees of Achievable Consistency

We begin by looking at what happens in the lock-act-release cycle. Holding the lock ensures that no other user can act over some piece of data, and so that inconsistency cannot arise. A lock is obtained through a request specifying some part of the data store (possibly all of it) over which the

4. However, Munson and Dewan [1996] have considered rich type-specific lock-table-based concurrency control in coordination with a merging strategy. This is a powerful combination that independently yields similar results to the techniques described here. However, see the discussion later concerning type-specific and application-specific concurrency.

user will act. So, locking is essentially a mechanism by which a system component can obtain a *guarantee of achievable consistency* in return for a *statement of future activity*.

This formulation has a number of interesting implications. First, consider the guarantee. Especially in light of our discussion of variable consistency earlier, there is clearly a range of potential guarantees that could be given. Strong locks are all-or-nothing, but when we generalise to guarantees, we introduce potential variances in the guarantees and in the consistency achievable. The locking authority[5] can determine the level of consistency that can be achieved based on currently-issued guarantees across the system. Second, the statements (or *promises*) of future activity are similarly variable. They may vary in their extent, duration and specificity.

Third, and perhaps most importantly, when we think of this exchange as being less absolute than the strict locking exchange (an absolute guarantee for an absolute promise), then it opens up the opportunity for negotiation; an application may make increasingly restrictive promises in exchange for increasingly strong guarantees of consistency. The promise/guarantee cycle is the basis of the consistency guarantees approach.

5.4.2 Breaking Promises

Flexible promises and guarantees allow better interleaving of activity than would be possible with strong locks. Since the locking authority receives more details of future activity, it can make better decisions about what sorts of actions can be permitted. Since the client applications can accept weaker guarantees, they can proceed where they would otherwise be blocked.

The promise/guarantee mechanism retains the important predictive element of locking. Applications still make "up-front" promises about the activities that they will carry out. However, this predictive aspect interferes with another important criterion for collaborative applications, support for opportunistic action. Predictive strategies restrict opportunistic action (which is by definition unpredictable).

Prospero addresses this problem through a second consistency principle: *a client can break a promise, in which case the locking authority is not held to its guarantee*.

This principle allows client applications to engage in activities that step beyond the bounds of their promise. However, should they do this, then the system may no longer be able to achieve the level of consistency assured by the guarantee. The guarantee, then, is potentially void (although, of course, in specific circumstances the guaranteed consistency may still be achievable).

With this principle in place, the consistency guarantee mechanism provides more direct support for opportunistic working styles. Just as in naturalistic work, stepping outside previously-agreed lines is not impossible; but the mechanism provides more effective guarantees when used cooperatively by both client and server. The system allows for unpredicted action, although users may pay a price; the system may only be able to achieve syntactic consistency later, for example.

By placing this within the consistency mechanism, we allow for the fact that the user need not (often, should not) be exposed to this complexity and unpredictability. The facilities are provided so that they can be appropriately deployed (or not) by an application developer. A developer might

5. These discussions emerge naturally in terms of clients and servers, although in fact Prospero uses a peer-to-peer architecture. I use the term "locking authority" to suggest a "lock server" that may, in fact, be distributed throughout the system.

choose *not* to exploit guarantee negotiation *in a particular application*, where application requirements or usage patterns might make it inappropriate. Safety-critical applications, for instance, would probably not be appropriate places to use this mechanism. In other cases, an application developer might want to warn the user when such a situation was likely to occur, so that an informed decision could be made as appropriate to the particular circumstances. The framework supports these behaviours, but does not require them. The ability to control optimistic concurrency control in terms of semantically defined application actions gives us greater control. For example, the COAST system [Schuckmann et al., 1996] uses optimistic concurrency control, but because their transaction model is automatically derived from application activity, it has no way to understand which sorts of inconsistency are problematic and which are not. Consequently, it errs on the side of safety and implements full ACID[6] properties—even though these may well be considerably stronger properties than are needed (and hence interfere with work when global commits fail).

Essentially, allowing applications to break their promises preserves the advantages of optimistic concurrency control, but combining this with a guarantee mechanism allows, first, a greater awareness of the consequences of those actions, and, second, a richer semantic framework for describing styles of application action.

Consistency guarantees in Prospero provide a way to overcome the problem of unbounded divergence; they curb Prospero's innate optimism. They provide some of the predictable consistency of pessimistic strategies, but in a way that is sensitive to patterns of collaborative work.

5.5 Semantically-Informed Database Management

The variable consistency approach uses knowledge of application semantics to specialise and improve the synchronisation process. In turn, the consistency guarantee mechanism uses knowledge of application semantics—and the semantics of particular operations—to increase the *opportunities* for concurrency and parallel activity.

Some similar approaches are discussed here that use application semantics and specificity in database processing. Barghouti and Kaiser [1991] give a comprehensive survey of advanced concurrency control techniques. Most of these techniques are aimed at improving processing performance while maintaining the illusion of individual activity over the database—clearly, a focus quite different from that of collaboration. However, two aspects of database research are particularly related to the consistency guarantees approach: *semantics-based concurrency* and *application-specific conflict resolution*.

5.5.1 Semantics-Based Concurrency

Database systems use a transaction model to partition the instruction stream. Transactions provide serialisation (ordered execution) and atomicity (all-or-nothing execution). However, if the system can detect that there is no conflict between two transactions, then it might execute them in parallel or interleaved, without interfering with transactional properties. The interaction-time and response characteristics of database systems are generally such that delays introduced while calculating appropriate serialisation orders for transaction streams will not have a significant impact on interactive performance. However, shared data stores supporting interactive collaborative systems

6. Atomic, Consistent, Isolated and Durable.

require crisp performance, and so it is useful to look at how database research has investigated the opportunities to increase concurrency in transaction execution.

Traditional database systems detect two principal forms of conflict. A *write/write* conflict occurs when two transactions write to the same location in the database. An ordering has to be established for these transactions to retain the model of atomic, serialised execution. A *read/write* conflict occurs when one transaction writes, and the other reads, the same data. Inconsistency can result if the read falls before the write during simultaneous execution. If conflicting transactions are executed concurrently, then the transaction model's serialisation properties may be lost; so conflicting transactions must be executed serially.

However, this is a very expensive way to maintain the transaction model, since the analysis of conflict is very coarse-grained. In the absence of transaction conflicts, the system can guarantee that the transactions can safely be executed in parallel. On the other hand, the presence of a conflict does not imply that inconsistency *will* result. For example, consider a transaction that issues a read request but makes no use of that result in its later computation (or does, but the use is robust to particular changes). It could, quite safely, be executed in parallel with another that writes the same data item. However, those circumstances would signal a *read/write* conflict and the potential concurrency would be lost. More generally (and more practically), transaction concurrency (and hence throughput) could be improved with more detailed access to transaction semantics, or to application semantics.

Approaches of this sort have been explored by a number of researchers. For instance, Herlihy [1990] exploits the semantics of operations over abstract data types to produce validation criteria, which are applied before commit-time to validate transaction schedules. His approach uses predefined sets of conflicting operations, derived from the data type specifications. Looking at the data type operations that transactions execute allows a finer-grained view of potential conflicts, and increases concurrency. Farrag and Oszu [1989] exploit operation semantics by introducing a break-point mechanism into transactions, producing transaction schedules in which semantically-safe transaction interleavings are allowed. Again, the potential for concurrency is increased without disrupting transactional properties.

One potential problem with each of these approaches is that they require *pre-computation* of conflicts, compatibilities and safe partial break-points. This implies that these mechanisms could not be seamlessly integrated into a general-purpose database management system. However, this doesn't pose a problem for using semantically-based techniques in Prospero, since Prospero doesn't need to provide a complete general-purpose service independent of any application. Instead, it provides a framework within which application-specific semantics can be added by application programmers (rather than being known to the system in advance). Particular behaviours are coded in Prospero in full knowledge of the relevant semantic structure of application operations.

The other major distinction between this approach and Prospero's use of consistency guarantees is that Prospero's approach is application-specific rather than type-specific. Type-specific semantic consistency such as that described here provides object-level consistency based on individual operations over those specific objects. However, the consistency guarantees approach provides the application developer with a higher-level, application-specific form of consistency control based not on the semantics of application types, but on how those application types are *used* to achieve

application behaviour. So, while concurrency control based on type semantics can be exploited in collaborative applications (such as in the work of Munson and Dewan [1996]), consistency guarantees are closer in spirit to application-specific approaches than to type-specific ones.

5.5.2 Application-Specific Conflict Resolution

A second approach from database research that is relevant to the consistency guarantees mechanism is the use of application-specific conflict resolution. Application-specific techniques differ from the semantic-based techniques above in that they achieve consistency at the level of the application as a whole rather than the individual object types used to implement it.

The Bayou system, developed at Xerox PARC, provides an example of this approach. Bayou is a replicated database system for mobile computers, which are frequently active but disconnected from their peers. In most systems, disconnection is an unusual state, and the systems can normally be assumed to be connected to each other; but in mobile applications, disconnection is the rule, rather than the exception.

Bayou provides a mechanism by which client applications can become involved in the resolution of database update conflict that can occur with replicated, partially-disconnected databases [Demers et al., 1994]. Bayou write operations can include *mergeprocs*—segments of code that are interpreted within the database system and provide application-specific management of conflicts. For instance, in a meeting scheduling application, a write (carrying a record of a scheduled meeting) might be accompanied with code that would shift the meeting to alternative times if the desired meeting slot is already booked. Mergeprocs provide a means for application specifics to be exploited within the general database framework. Bayou also provides "session guarantees" [Terry et al., 1994] that give applications control over the degree of consistency they require for effective operation in specific circumstances. Clients can trade data consistency for the ability to keep operating in disconnected conditions. Both of these techniques are based on an approach similar to that exploited in Prospero—allowing clients to become involved in how infrastructure support is configured to their particular needs. There has been some experience in applying these approaches to the design of collaborative applications [Edwards et al., 1997], although Bayou's use of a loosely-connected database layer restricts it to asynchronous applications.

More generally, one focus of research, particularly in databases supporting software development or CAD/CAM, has been on variants of the transaction model supporting long-duration and group transactions (e.g. [Kaiser, 1994]). These are variants that exploit a general style of interaction, rather than the specifics of particular applications; however, they do begin to address the needs of inherent collaborative applications.

5.6 Programming with Consistency Guarantees

This section has dealt with the generic model for consistency management using consistency guarantees. Before going on to look at specific examples, we need to consider how consistency guarantees are realised in Prospero.

Prospero makes promises and guarantees available to programmers as the primary interface to the consistency mechanism. As described in the previous section, Prospero uses a command object model to reify user activities within the programming framework. Promises and guarantees need to deal not with specific commands, but with the semantic properties of those actions within the application domain. However, the significant feature of this semantic approach is that Prospero should

itself be neutral with respect to the application domain, and hence to the semantics of actions there. Prospero uses a "semantics-free semantics" model to allow programmers to deal in terms of semantic properties without making specific commitments to the semantic features that may exist in any given domain.

5.6.1 Semantics-Free Semantics

The primary role of the semantic descriptions is to allow coordination between the pre-divergence point (the "promise" phase) and the post-divergence point ("synchronisation"). The efficacy of the approach is dependent on this coordination—actions being described and later recognised—rather than on a detailed, structured semantic account of user-level operations. So, while the properties that we would like to base our descriptions on are *semantic* properties, the descriptions themselves do not have to *have* semantics. What Prospero needs to provide is a way of referring to semantic properties, but not a language of semantics. It's enough to be able to distinguish and recognise semantic property `foo`, without having to give an account of what `foo` *means*.

In other words, Prospero needs to offer a *naming* solution, not a *description* solution. The application programmer names a set of semantic properties relevant to his or her application domain, and coordinates activity in terms of those, but the details of those semantics need not be described. Once they have been named, they can be used as a basis for coordination.

5.6.2 Class-based Encoding

The mechanism that Prospero uses to accomplish this is *class-based encoding*. Particular semantic properties for an application are implemented as classes of command objects. Each instance of a command object represents a particular invocation of that command, along with the relevant parameters and contextual information. Command objects inherit from the classes that represent their semantic properties. Each command object then becomes subject to any methods defined for its particular semantic properties.

The use of explicit command objects is, in itself, a useful mechanism for representing sequences of action and arriving at appropriate mechanisms for resolving conflicts that might arise; but encoding semantic properties in the inheritance structure of the command objects yields two particular benefits when programming with Prospero. First, the mechanism is inherently *extensible*; the application developer can create new semantic properties from existing ones within the same mechanism as she uses to create application structures and objects (i.e. subclassing and specialisation). Second, class-based encoding allows semantically-related behaviours to be defined in a *declarative* style. Prospero programmers declare the consequences of semantic properties through separable methods that apply to classes of semantic property. CLOS's generic dispatch mechanism can be used to ensure that the relevant actions (methods) are applied to specific sorts of actions (semantic classes). This allows the consistency management mechanism to be created declaratively and iteratively, rather than forcing the programmer to encode it in one large, monolithic resolution mechanism (which would then, by its nature, be inaccessible to application programmers and locked within the toolkit).

5.7 Consistency Guarantees: Summary

While the divergence approach introduced in Section 5 allows us to manage activity over distributed data in ways more appropriate to collaborative work, it presents a problem. Potentially unbounded divergence can create situations in which it is difficult or impossible to resolve inconsistency in any useful way.

Consistency guarantees balance this unboundedness by providing a way to constrain divergence. The mechanism generalises the traditional lock-act-release cycle by casting it not in terms of absolute locks, but rather in terms of variable promises of future activity and guarantees of achievable consistency. These promises and guarantees are constructed from terms that are meaningful according to the semantics of the application domain. Exploiting application semantics allows the consistency mechanism to be tailored to the particular needs of specific collaborative applications and settings, and allows programmers to create applications that capitalise upon regularities and patterns in the work being conducted. At the same time, moving away from a basic read/write model to one that is grounded in the application specifics allows new opportunities for parallel work, and so supports multi-synchronous and opportunistic working styles.

6 WRITING APPLICATIONS IN PROSPERO

Divergence, synchronisation and consistency guarantees, as outlined above, provide a framework in which the semantics of applications and their operations can be used to improve concurrency management for collaborative work. Earlier sections have discussed some specific issues in the representation of these mechanisms in the toolkit, and provided some pointers to how programs are developed. This section gives an overview of the programming interface and experience, as an orientation for the examples in Section 7.

Prospero is implemented in CLOS. Network communication between Prospero peers is performed using an RPC layer provided by the underlying Common Lisp implementation (the “wire” package of CMU Common Lisp). Again, Prospero itself provides no user interface functionality; in the prototype applications, the user interfaces have been implemented using Garnet [Myers et al., 1990], although another package could have been used.

6.1 Base-Level Programming: Writing Applications

As outlined earlier, writing programs at the base level means creating collaborative applications that make use of the features and functionality offered by the Prospero toolkit. All applications involve base level programming.

6.1.1 Streams

To write a collaborative application using Prospero, the programmer organises the system as a set of streams of activity. Typically, each stream of activity will be associated with one particular user in a session, although this is not required. Application activity creates action objects, which are then inserted into a particular stream; periodically, the streams are synchronised so that actions in each stream can be exchanged and communicated. Figure 1 shows the functions in the Prospero API that implement this level of functionality.

The form `defaction` creates a new sort of action. Actions are defined by the properties that they exhibit (properties being defined by the form `defproperty`). Actions may have multiple properties. The function `create-action` creates a new action object; its arguments specify the type of action performed and the particular details associated with this particular invocation. The programmer then uses `add-action-to-stream` to associate the action with a particular stream. Prospero provides classes for both “local” and “remote” streams (embodied by the classes `local-stream` and `remote-stream`). Actions can only be added to local streams; remote streams are proxies for streams elsewhere in the system, and are used to specify the details of synchronisation. Two forms

`(defproperty` property-type [parent-type...]`)`
defines new property type based on named parents

`(defaction` action-type [property ...]`)`
defines new action object type

`(create-action` action-type [parameters]`)`
creates an action object of a particular type

`(perform-local-action` action`)`
executes an action object in the local context

`(add-action-to-stream` action stream`)`
associates an action with a stream object

`(synchronise` local-stream remote-stream`)`
synchronises a local stream with a remote one

`(with-local-stream` stream [body ...]`)`
executes body with a local stream binding

`(defpromise` promise-type [parent...]`)`
defines a new promise type based on parents

`(create-promise` promise-type`)`
creates a promise object

`(defguarantee` guarantee-type [parent...]`)`
defnes e a new guarantee type based on parents

`(create-guarantee` guarantee-type`)`
creates a new guarantee object

`(get-guarantee` stream promise`)`
returns guarantee from stream for offered promise

`(redeem-guarantee` stream guarantee`)`
returns a guarantee

`(with-guarantee` stream promise [body ...]`)`
executes body with a guarantee binding

`(require-guarantee` guarantee-type stream promise [body ...]`)`
executes body with a guarantee binding of a particular type

FIGURE 1: Base-level API.

of local stream are predefined and provided by the toolkit. Instances of `bounded-stream` accumulate a certain number of actions before they attempt to synchronise themselves with their peers; instances of `explicit-synch-stream` will not synchronise until synchronisation is explicitly performed by a call to `synchronise`. Both of these stream types are subclasses of the `local-stream` class. Applications can make use of these stream types directly, or create new ones through metalevel customisation.

6.1.2 Promises and Guarantees

Prospero also provides some support for the use of promises and guarantees as the base level; however, since the full benefits of the consistency guarantee approach require the use of semantic information about applications, which constitutes metalevel programming, there is less mechanism of direct use at the base level. However, the basic framework can be used to constrain divergence.

The mechanisms at the base level are primary definitional—that is, mechanisms are provided to define various types of entities, which can subsequently be used to express degrees of metalevel specialisation. These entities, though, are also part of the base level programming interface. We have already seen that `defaction` can be used to define new actions that users can generate; by analogy, `defpromise` defines the promises that can be made. Promises, like actions, can be defined in terms of properties (and also in terms of each other). Conceptually, the distinction is that actions represent specific user actions, while promises are used to describe whole sequences of activity that take place between periods of synchronisation. Promises characterise the sorts of actions that will take place (that is, the properties of the set of actions), but not the specific actions themselves. This provides an opportunity for richer and less restrictive descriptions.

At the base level, two classes of consistency guarantee are provided, called `full-guarantee` and `null-guarantee`. Essentially, these resemble, respectively, full locks and no locks at all (although, of course, unlike locks, they do not have to be acquired, nor respected). In other words, the significance of guarantees is not in the use of predefined ones like these, but that the offer a language in which to define new sorts of guarantees (and since those new sorts will be rooted in the application semantics, they cannot be defined in advance.)

Figure 1 also shows the API for manipulating the guarantees mechanism at the base level. Guarantees are made in terms of promises, which are defined with `defproperty` and created with `create-property`, just like actions. The function `get-guarantee` requests a guarantee from a stream for a particular promise, and `redeem-guarantee` is used to redeem it later; so calls to these two functions delimit a sequence of activity carried out under the guarantee. Calling `get-guarantee` will cause Prospero to send a promise to a stream, which will evaluate the promise, comparing it to other guarantees that have already been issued, and return the best guarantee that can be given under the circumstances. The nature of that comparison will be examined in more detail in a moment, but does not form part of the base level interface. Later, once a sequence of activities have been carried out under a particular guarantee, the guarantee can be returned using `redeem-guarantee`.

For convenience, the form `with-guarantee` can be used as a shorthand for a common idiomatic structure; it requests a guarantee, executes a body of code under that guarantee, and then redeems it. Its partner, `require-guarantee`, ensures that a particular sort of guarantee be achieved before the code is executed.

6.2 Metalevel Programming: Customising the Toolkit

Base level programming combines the features of the toolkit to create collaborative programming. Metalevel programming augments and specialises those features to provide customised support for particular applications. Metalevel programming allows the toolkit to provide more efficient support for particular applications, to support applications that would otherwise not be supported, or to allow forms of collaborative work that cannot be accommodated in the base level framework.

The metalevel is implemented using a *metaobject protocol* [Kiczales et al., 1991]. A metaobject protocol is an object-oriented encoding of the reflective link between application and implementation internals. A view of the implementation is described in object-oriented terms, and then provided to the application for examination and revision. The metaobject protocol defines not simply the toolkit's internal structures, but also a view into *how it uses* those structures to achieve its functionality. This then allows the application programmer to inspect behaviours and interpose code, and so become involved in the infrastructure implementation. In other words, while the metaobjects define the internal structures of the toolkit, the protocol reveals how these interact to achieve visible behaviours.

Customisation is achieved by "inserting" code into the implementation layer. Other reflective systems such as 3-Lisp have created customised ways of doing this "level-shifting", but in a metaobject protocol it can be achieved through subclassing and specialisation. In Prospero, we provide the means to insert programmer code into the components of the toolkit dealing with the creation of objects, the manipulation of streams and synchronisation, as well as the evaluation of promises and granting of guarantees. Creating new sorts of streams and guarantees provides us with a means to do this through standard object-oriented techniques, as well as providing us with control over the scope of changes (by associating them only with these new classes).

Metalevel programming in Prospero consists in identifying those behaviours in the toolkit that should be adjusted, and then creating new streams and classes with new behaviours for these components. Control, then, is achieved through the protocol: the sequence of generic functions through which the toolkit's internal behaviours are coordinated.

`(propagate-action-to-stream` action remote-stream ..`)`
sends action object to remote stream

`(check-send-action` action remote-stream`)`
transforms action for transmission

`(check-locally-perform-action` action remote-stream`)`
transforms action for local execution

`(synchronise-remote-action` action`)`
handle an incoming action object

`(compatible-promises` promise1 promise2`)`
determines compatability level of two simultaneous promises

`(grant-guarantee` promise local-stream`)`
grants a guarantee for the local stream

`(guarantee-for-promise` promise-wireform`)`
determines maximal guarantee for promise

FIGURE 2: Generic functions used to control internal behaviour.

The consequence of this approach is that it is difficult to identify an API for the metalevel in quite the same way as we could for the base level. Metalevel programming uses just the same programming constructs as programming at the base level, but it uses them in different ways. Rather than using them to achieve effects directly, it uses them as points of articulation for adjusting internal behaviour. Of course, we can, in a specific piece of code, identify those lines that constitute metacode; lines of code executed in the context of the toolkit rather than that of the application; and we can identify uses of the toolkit structures that constitute metalevel programming. There are also some API calls that are used primarily to adjust the behaviour of the toolkit. For instance, the method `compatible-promises` is used to determine whether two promises are compatible for simultaneous execution. This method is used by Prospero in evaluating promises and making a determination of the level of guarantee that can be offered in response, as part of the remote response to a call to `get-guarantee`, as discussed earlier. So, when an application programmer wishes to specialise the process of granting guarantees, perhaps for newly defined promise types, this is where control can be exercised.

Figure 2 lists some of the major functions that provide points of articulation for modification of internal behaviour (in addition to those listed previously as base-level functions). In order to provide a more detailed exposition of the use of Prospero, the next section illustrates how some simple sample applications can be constructed.

7 APPLICATION EXAMPLES

This section provides some example applications built using Prospero. As far as possible, the examples omit details that are not immediately relevant to the topic at hand. So, for instance, internal data structure representation is not presented, except where it is significant to interaction and synchronisation; similarly, the manipulation of user interface elements, which is typically a significant source of programming verbiage, is also omitted. The examples are written in CLOS, but the discussion will emphasise the structure, which can be applied to other implementations.

Three examples are presented here (further complete examples can be found elsewhere [Dourish, 1996]). The first is a simple shared drawing program, and illustrates the use of actions and streams to create a collaborative application. The second example is a hypertext database editor, and illustrates the use of consistency guarantees to manage more complex interaction. The third example is an extension of the hypertext database editor to accommodate opportunistic work.

```
1: (defun polyline-editor ()
2:   (let ((window (create-window “Polyline Editor”)))
3:      (add-to-window (create-interactor ‘polyline-interactor
4:                                        :where window
5:                                        :when ‘mouse-down
6:                                        :finish-fn #’add-new-polyline)))))
7:
8: (defun add-new-polyline (interactor points)
9:   (add-to-window (create-instance ‘polyline :points points)))
```

FIGURE 3: Simple polyline editor.

```
1: (defaction <polyline-action>)
2:
3: (with-local-stream (make-instance ‘<bounded-stream>)
4:   (let ((window (create-window “Shared Polyline Editor”)))
5:      (add-to-window ...))
6:
7: (defmethod add-new-polyline (interactor points)
8:   (let ((pl-action (create-action <polyline-action> points)))
9:      (perform-local-action pl-action)))
10:
11: (defmethod perform-local-action ((action <polyline-action>))
12:     (add (create-instance ‘polyline (action-params action))
13:          window)))
```

FIGURE 4: Multi-user polyline editing with Prospero base-level support.

Since the primary design goal in Prospero is flexibility in support of a wide range of applicability to specific application settings, it follows that the point to be made with these examples is not that such-and-such an application can be developed, or that such-and-such a technique can be used in creating a collaborative application. For most applications that can be created with Prospero, a similar application could be created with another toolkit such as those discussed earlier. Instead, the point to be illustrated is the *range* of applications that can be developed using this *single* toolkit framework. In other words, while standard approaches might be able to support one application or another, they would typically not support the range of variability found in across the set of examples presented here.

7.1 Example: Shared Drawing

The first example is a simple multi-user whiteboard application with a replicated architecture. Users can create simple “scrawl” strokes (captured as “polylines”, or line segment sequences), and their actions are periodically broadcast to other sites. The functionality is extremely simple; this example is presented mainly to show how collaboration can be added to a single-user application using Prospero’s base-level functionality.

Figure 3 shows the basic structure of the original, single-user application. (Details, particularly of the graphical toolkit, have been omitted where they would distract from the core functionality.) A window is created, and an “interactor” [Myers, 1990] is associated with it, to handle a particular form of user interaction. This particular interaction is configured to run when the user holds down the mouse button; when it is released, the function `add-new-polyline` is called with arguments representing the new object. This function then creates a polyline object and adds it to the display.

```
1: (defmethod perform-local-action :after ((action <action>))
2:   (add-action-to-stream action (current-local-stream)))
3:
4: (defmethod add-action-to-stream ((action <action>) (stream <stream>))
5:   (push action (stream-actions stream))
6:
7: (defmethod add-action-to-stream :after (action (stream <bounded-stream>))
8:   (if (full-p stream)
9:     (synchronise stream (stream-remote stream))))
10:
11: (defmethod synchronise ((stream <bounded-stream>) (stream <remote-stream>))
12:  (dolist ((action (reverse (stream-actions stream)))
13:    (propagate-action-to-stream action remote)
14:  (stream-reset stream))
```

FIGURE 5: Internal Prospero code implementing bounded streams.

Figure 4 shows how this program can be turned into a collaborative application using Prospero. The collaborative application has some minor changes in structure, to accommodate the objects and functions added by Prospero. Lines making use of Prospero features are highlighted by underlining.

Basic graphical interaction is handled with an interactor, just as before. The difference is in how the polyline itself is created. Since Prospero operates in terms of command objects that explicitly represent user actions, this function is now handled indirectly through the creation of an action object. How do we do this? At line 1, a new sort of action is defined for polyline creation[7]. The function `add-new-polyline` at lines 7–9 now creates an action object of this type, corresponding to the user's behaviour, by calling create-action, which creates a new command object encapsulating the particular parameters of this user action (the points of the polyline). Then it "applies" this action by calling the method `perform-local-action`. This is a method defined by Prospero, although it has no functionality for new sorts of objects defined by applications, and so the application programmer has to define some behaviour for performing a `polyline-action`. This is done at lines 11–13 by specialising the generic function specifically for objects of class `<polyline-action>`. This new method overrides other applicable methods; it creates the polyline object and adds it to the shared workspace. All this happens within the context of an instance of `bounded-stream`, defined at line 3.

Why does the application perform its actions indirectly through this action object? The answer is that action objects and the methods defined on them are a point of coordination with Prospero's mechanisms for managing collaboration. Figure 5 shows some of the internal[8] structure of Prospero corresponding to the mechanisms used in this example. At lines 1–2, an "after-method"[9] is defined for the generic function `perform-local-action`. The effect of this is to ensure that, once the local behaviour has been performed (the behaviour provided by the application developer in figure

7. The use of angle brackets is a common notational convention used in CLOS to denote class objects. Objects that Prospero application developers create such as action types, promises and guarantees are generally actually classes, so that they can be used as part of method definitions.

8. In other words, this is the implementation code. Application developers would neither supply or see this. It is shown here for explanation.

9. In addition to normal ("primary") methods, CLOS provides "before-methods" and "after-methods" which are collected and run before and after the primary method for any given generic function. Applications can use them to associate behaviour with generic functions, without replacing the existing functionality.

4), then the method `add-action-to-stream` will be called to associate this action object with the currently active local stream.

Since the application developer set the local stream to be a `bounded-stream`, which is one of Prospero's pre-defined local stream types, the behaviours shown at lines 4–14 in figure 5 apply. Bounded streams accumulate their actions until a threshold (bound) is reached, at which point they are synchronised (lines 8–9). Synchronisation involves going through the accumulated local actions and sending them to peer streams to be incorporated into the action streams there (lines 12–13). The application developer, however, does not have to manage any of this behaviour. Bounded streams carry their own synchronisation behaviour with them; the application developer has simply had to perform actions indirectly through the command objects to get the collaborative action desired.

So, in this case, to turn the application into a collaborative application, the application developer needed only to add a small number of lines of code, to set up the stream and handle application activity through command objects.

7.2 Example: Hypertext Database Editing

The previous example provided a simple illustration of the basic use of actions and streams in Prospero to create a synchronous replicated drawing application. With those basic concepts covered, we can proceed to a more ambitious example, which will illustrate the use of semantic properties and consistency guarantees.

This second example is a hypertext database editor. Nodes are created and edited, and then are linked together to create a hypertext document. We consider the creation and editing of nodes to be content-based activities, and the manipulation of their links and relationships to be structuring activities. We can consider these two classes of action to be independent. Changing structure and changing context can be carried out in parallel without conflict, even though both result in changes to the underlying data store. In a traditional system, there would be no way to distinguish these cases. Any attempt to change stored data would look like a write operation, and since some write operations may result in conflicts, *all* write operations would have to lock out other users. Using consistency guarantees, we can incorporate richer semantic information, achieve a better match between the application and the infrastructure, and increase opportunities for parallel work.

The consistency guarantee mechanism can be used to create a separation between these two forms of access, incorporating this piece of knowledge about the semantic structure of the domain. Figure 6 shows the encoding of these semantic properties in Prospero; metalevel code is highlighted in italics. At this stage the programmer sets up a description of the application domain in terms of the semantic properties of actions (lines 1–4), and of the actions and promises[10] in terms of these properties. The properties will subsequently be used to refine the consistency management mechanism.

The programmer sets up four properties in lines 1–4. These describe the different changes that might introduced into the data store by different user actions. The actions of the application are then given and inherit the relevant properties (6–10). Promises are defined in terms of the sets of expected semantic properties for any given period of divergence. Later on, the system will want to ensure that the properties of promises and the properties of the sequences of action that take place under them line up. Promises can be defined dynamically, but in this case we pre-define the ones that we know

10. In this example, we use only pre-defined guarantees; otherwise, those would also be set at this point.

```
1: (defproperty <append>)
2: (defproperty <structure-change)
3: (defproperty <content-change>)
4: (defproperty <no-change>)
5:
6: (defaction <create-object-action> <append>)
7: (defaction <find-action> <no-change>)
8: (defaction <set-field-action> <content-change>)
9: (defaction <change-field-action> <content-change>)
10: (defaction <add-link-action> <structure-change>)
11:
12: (defpromise <structure-promise> <structure-change> <no-change>)
13: (defpromise <content-promise> <content-change> <no-change>)
14: (defpromise <structure-content-promise> '(<structure-promise> <content-
promise>))
15:
16: (defmethod compatible-promises ((p1 <structure-change>) (p2 <structure-
change>))
17:   nil)
18:
19: (defmethod compatible-promises ((p1 <content-change>) (p2 <content-change))
20:   nil)
```

FIGURE 6: Setting properties, actions and promises for hypertext database editing.

will be used. Note that the `<structure-change-promise>`, which allows for changes to both components of the database, inherits from the `<structure-promise>` and the `<change-promise>`, so that definitions based on those will also apply to this promise, which combines their effects.

Creating these semantic structures provides us with an enriched language (a language of domain semantics) for doing metalevel tailoring. Lines 16–20 show the use of these semantic structures in configuring the behaviour of the consistency control mechanism. These lines inform the toolkit that a promise involving `structure-change` cannot be granted at the same time as another also involving `structure-change`, and that the same exclusion principle holds for promises involving `content-change`. The application developer expresses this by providing new methods for the metalevel generic function `compatible-promises`, which tests whether two promises are compatible. This predicate is used to select the guarantee that will be returned.[11] The user's metacode will be used by the toolkit in resolving requests for promises, based on the application semantic properties defined earlier. In this way, the toolkit's internal behaviour (granting guarantees) has been specialised to accommodate the specific features of this application (parallel updating of structure and content).

11. CLOS's true and false (t and nil) are shorthands for full and null guarantees. By default, promises are compatible; overriding the method for these particular sorts of promises denotes exceptions. CLOS's "generic dispatch" mechanism, which matches generic function calls to specific methods, ensures that the most specific set of comparisons are performed.

```
1: (defun request-guarantee ()
2:   (let ((promise (case *access-mode*
3:                        ((:entry) (create-promise <content-promise>))
4:                        ((:linkage (create-promise <structure-promise>))
5:                        (t (create-promise <structure-content-promise>)))))
6:      (get-guarantee *server-stream* promise)))
7:
8: (defun ui-set-mode (gadget menuitem submenuitem)
9:   (declare (ignore gadget menuitem))
10:  (setq *access-mode* (intern (string-upcase submenuitem)
11:                        (find-package "KEYWORD")))
12:  (if *current-guarantee*
13:          (redeem-guarantee *server-stream* *current-guarantee*))
14:  (setq *current-guarantee* (request-guarantee)))
```

FIGURE 7: Claiming and redeeming guarantees.

Next, the code managing the performance of activities is then surrounded by guards that obtain and resolve guarantees. This is managed through the mechanism in the user interface by which users change between different editing modes, as shown in figure 7. As before, use of Prospero functionality is highlighted by underlining. When the mode is changed through the user interface menu control, the function `ui-set-mode` (line 8) is called; it will request a new guarantee based on a promise constructed from the current edit mode setting (lines 1–6). The call to `get-guarantee` at line 6 causes infrastructure's guarantee mechanism to be invoked, which in turn will exploit the specialisation that the programmer created at lines 16–20 of figure 6 by configuring the promise comparison mechanism. The user interface allows work to proceed, reflecting the current guarantee status in the user interface. When actions are executed for which the guarantee is not valid, the user is informed once through a pop-up menu, and continuously through a status indicator.

Finally, figure 8 show some excerpts from the code that implements the application behaviour. Calls to Prospero behaviour are highlighted by underlining. The methods at lines 1–4 and 6–9 create "after" methods that execute once the application behaviour has run and associate new action objects with the current local stream. The local stream in this example is an instance of `explicit-synch-stream` (rather than `bounded-stream` as in the previous example), supporting a more asynchronous style of working. An explicit-synch-stream is not synchronised with its peers until a specific action of type `<synchronise>` is added to it; this causes synchronisation to take place. In this example, it causes the effects that the user has performed to be checked into the database (in a remote server stream).

```
1: (defmethod add-link :after ((from <record>) (to <record>))
2:   (let* ((from-id (record-id from)) (to-id (record-id to))
3:          (action (create-action <add-link-action> *ident* from-id to-id)))
4:     (add-action-to-stream action (current-local-stream))))
5:
6: (defmethod new-record :after ((record <record>))
7:   (let* ((id (record-id record)) (type (class-name (class-of record)))
8:          (action (create-action <create-object-action> *ident* id type)))
9:     (add-action-to-stream action (current-local-stream))))
10:
11: (defmethod checkin ()
12:  (add-action-to-stream (create-action <synchronise>)
13:     (current-local-stream)))
```

FIGURE 8: Performing actions in the hypertext database.

So, through the use of metalevel specialisation, the toolkit has been extended to accommodate the specific semantic features of this domain. The working style to which this application lends itself would be unavailable to applications created with a toolkit that employed standard, undifferentiated data storage semantics.

7.3 Example: Accommodating Opportunistic Work

The first example showed the use of Prospero's streams mechanism to support synchronous or "real-time" collaborative interaction. In the second example, however, the same basic mechanism—periodically-synchronised streams that accumulate sequences of user actions—was used to support a more asynchronous style of client/server interaction. Suppose, now, that we want to go further, and extend this example to incorporate support for opportunistic work as described earlier. How would we go about this?

The structure of the example was that a user interface control selected a mode of operation, via the function `set-ui-mode`, and that this mode-setting behaviour would control the making of promises and the securing of guarantees. However, this mode is purely advisory; it does not restrict the sort of actions that users can perform. So, actions can still take place that are outside the scope of the promise made (and hence the guarantee granted). Activities like this constitute a broken promise. This is how opportunistic work becomes manifest to the system, in the form of actions that are performed outside the scope of the granted guarantee[12].

The application developer can choose to resolve these situations in different ways, and to place responsibility for handling them at various parts of the system. In this particular case, we will choose to handle broken promises on the server side. Figure 9 shows the server-side code that deals with incoming actions during the synchronisation process. When synchronisation takes place, a stream of actions arrives at the server; the method `synchronise-action` is called to process each one. First, it looks up the stream that performed the action, and the guarantee under which that stream has been acting. A guarantee records the promise that was made for it, and so the action can be compared to the promise that was made (line 5) to determine whether or not it is in agreement with the action. (This comparison is made on the basis of the properties of the action and the properties of the promise; recall that they were both specified in terms of properties.) If they match, then the action can be performed on this side (via the generic function `locally-perform-action`).

```
1: (defmethod synchronise-remote-action (action)
2:   (let* ((stream (action-stream action))
3:          (promise (promise-for-stream stream *local-stream*)))
4:     ;; first check if this action was under a valid promise
5:     (if (guaranteed-action action (guarantee-promise promise))
6:         (locally-perform-action action)
7:       ;; otherwise, try to complete it anyway
8:       (if (acceptable-action action)
9:             (locally-perform-action action)
10:          ;; fail in the last resort
11:          (syntactically-locally-perform-action action)))))
```

FIGURE 9: Supporting opportunistic activity.

12. The term "opportunistic work" implies a certain intention on the user's part. This may not be entirely accurate. For instance, similar circumstances could conceivably result from certain sorts of network partition, server failures, etc.

However, we are interested here in the case where this does not take place. This is where the application developer can choose to allow promises to be broken, and decide how to handle them. In this case, the decision is as follows. First, the application determines whether or not there would be any conflict or error resulting from applying the action even though it is outside the granted promise (line 8, using `acceptable-action`). If not, then the action can be performed anyway, and so it is (line 9). What does it mean for an action to be acceptable in this sense? Clearly, this is an application-specific issue. In this case, for example, one acceptable position is, "an action is compatible with the current state if it results in the current state". Other semantics are possible. Certainly, Prospero does not attempt to encode this. Prospero's role here is to provide the framework in which an application developer can encode application semantics, and to provide enough control over the basic structures such as promises, action, streams and guarantees that the specific application requirements can be met.

The last case is the complicated one, corresponding to the situation in which the user has gone ahead and performed an action outside of the stated promise, and a conflict has resulted from this opportunistic behaviour. This is the "failure" case, although this application developer chooses not to flag a failure, but rather to note the problem and continue. In this case, the application programmer has decided to allow syntactic consistency. The programmer has defined a method `syntactically-locally-perform-action`, which performs the action but maintains only syntactic consistency.

Note that the return value of the function `synchronise-action` is the return value of whichever method was used to perform the action locally, either `locally-perform-action` or `syntactically-locally-perform-action`. The return values of the sequence of invocations to synchronise-action are used to pass information back to the remote stream about the results of synchronisation. This supports cases where the application programmer wishes to distribute the responsibility for handling the consequences of opportunistic action between the streams involved; we make no use of it in this example, however.

The mechanisms shown in these three examples highlight how application semantics can be incorporated into Prospero by making explicit aspects of the domain semantics and encoding these in the Prospero framework. This in turn provides the resources for Prospero to manage data distribution and consistency management in ways appropriate to the specific applications. The examples have shown that way that the application developer building an application in Prospero is not only provided with the means to combine predefined components but is able to reach in and specialise components for particular application settings, enriching the toolkit with application semantics.

8 FLEXIBILITY IN PROSPERO

When we step back from the details, what have the examples in the previous section illustrated?

First, they have shown how the basic components of the toolkit (streams, actions, promises and guarantees) can be combined to create collaborative applications. The first application showed how we can wrap application behaviour in Prospero structures to create a collaborative application; the second illustrated the use of metalevel description to enhance application support by specialising toolkit behaviours to the particular needs of specific applications.

Second, they have shown some of the advantages of the particular structures provided in Prospero. Streams have been used to support both fairly synchronous and fairly asynchronous forms of inter-

action, in the first and second examples respectively. Actions have been used to characterise a range of application behaviour, and guarantees have been used to give different degrees of feedback to users about the likely effects of actions.

Third, they have shown how we can use metalevel description to specialise the toolkit to the needs of specific applications. We showed this through the consistency guarantee mechanism in the second example, although it can also be applied to creating new sorts of stream structures, and we showed using two kinds of predefined streams (created, themselves, through metalevel specialisation).

Fourth, they have shown that the use of this metalevel specialisation allows a single toolkit to support a wider range of applications that would be possible otherwise. Particular applications developed in Prospero may resemble ones that could be generated using other toolkits, but Prospero's novelty is in the range of applications that can be developed from a single framework. So, we can use Prospero to create not only fully synchronous applications, but also loosely synchronous and asynchronous ones; and applications that rely not only on strong locking, but also on application-specific locking or no locking at all. The reflective framework provides for a perspicuous unification of these approaches.

So, we can step back and ask a more general question. How does Prospero address the flexibility problems we encounter in designing CSCW toolkits? There are two ways we can address this question.

One is to consider Prospero, itself, as a CSCW toolkit. Like any toolkit, it provides a particular conceptual model for the structure of applications, and the objects and mechanisms to realise applications organised around this model. Prospero's model is based around streams, actions, promises and guarantees. The choice of these particular mechanisms was motived by two considerations. First, these mechanisms reflect a particular set of requirements for supporting fluid, flexible collaborative work, based on the lessons of investigations of collaboration in real-world and laboratory settings. These four fundamental structures offered by Prospero emphasise the design of applications that avoid temporal or structural constraints, which interfere with the performance of collaborative activity. Second, and more importantly here, they were chosen for their value as *points of articulation* at the meta-level; they provide a conceptual separation for the areas of flexibility and control offered to application developers. Streams and divergence are incorporated into the toolkit not only because they offer a natural mapping onto the patterns of collaborative activity we encounter in real-world settings, but also because they offer a natural way to talk about the wide range of behaviours we might want to accommodate. Similarly, promises and guarantees are introduced precisely because they provide a way of talking not just about what the application does, but also about how the toolkit can go about supporting it, in terms of the application domain. So, the conceptual structure of the toolkit is designed specifically with notions of metalevel extensibility in mind. (Recall the earlier observation about the metalevel interface as a designed artifact rather than an epiphenomenon of the implementation.)

The second way to consider how Prospero addresses the flexibility problems encountered in CSCW toolkit design is in terms of its strategy of providing application developers with metalevel control over the toolkit's internal structure.

Fundamentally, what Prospero does, through the use of Open Implementation, is to *repartition* the design problem. Since the use of Open Implementation allows application developers to revisit internal toolkit design decisions in order to revise them in accordance with the needs of particular applications and settings, it moves the burden of matching toolkit mechanisms with application needs to the application developer. This is a considerable benefit to the application developer, who is not only the person best able to make those decisions anyway, but is also, in conventional settings, the person who is left somehow to identify what those decisions might be and how they can be worked around. Meantime, the toolkit designer can focus attention on how to provide the application developer with an appropriate set of abstractions for both developing applications and controlling the toolkit internals. In other words, both the toolkit developer and the application developer are able to concentrate on those elements of the overall problem that are their particular domain of expertise; yet at the same time the structure of the system allows their two contributions to be brought together more fruitfully in the design of any particular application.

9 CONCLUSIONS

In any toolkit design, a primary concern of the developer is flexibility—the range of applications that the toolkit can support. For interactive and collaborative systems, the problems are even greater, since the applications themselves must also be flexible enough to accommodate the range of ways in which users and groups engage in their work. The first generation of collaborative toolkits focussed largely on the problems of encapsulating common behaviours to ease application development. A body of recent research has examined the ways in which CSCW toolkits can be designed so as to support a wider range of application and working styles.

The research described in this paper contributes to this work in two ways. First, it presents a novel architectural technique that can be harnessed to create toolkits that are both flexible and expressive. Second, it introduces two particular mechanisms that can be incorporated into the design of CSCW toolkits. The overall goal of a deeper form of toolkit flexibility in support of both a wider range of possible applications and support for flexibility within those applications has been grounded not simply in technological opportunities but in understandings of how group work proceeds. However, where these observational studies make their mark is in the approaches we use and the technologies we develop in trying to provide computational support for collaborative activity.

In this article, I have exploited an architectural approach—Open Implementation—that is being used in other areas of system design to attack these sorts of problems. Open Implementation gives application programmers metalevel control over the implementation on which they depend. It does this by introducing a separation between a base interface, which specifies how the application will use the toolkit infrastructure, and a meta interface, by which the implementation of that infrastructure can be examined and modified. Prospero is a prototype CSCW toolkit based on the Open Implementation approach. Prospero illustrates the application of this technique in the domain of collaborative working, and shows how it can be used to repartition the toolkit design problem to the benefit of both toolkit and application developer.

Prospero also introduces two specific metalevel techniques for the flexible design of collaborative applications. First, the divergence/synchronisation model provides a framework for describing and creating a wide range of access patterns to distributed data, specialised for collaborative work, and scalable across timescales. Second, the consistency guarantees model allows a consistency manage-

ment infrastructure to be specialised according to the semantic properties of the application domain. These approaches can be applied independently of the Open Implementation architectural approach, but they lend themselves particularly to this approach since they provide not only descriptive power but also a means to create new behaviours.

Prospero embodies a new approach to the problems of flexibility in CSCW toolkit design. Rather than attempting to provide a set of maximally-general structures onto which a programmer can map application needs and requirements, it instead deals with that mapping through a metalevel interface, which can be used to specialise the behaviour of the base interface (and hence the toolkit's internal structures). Metalevel programming capitalises upon the programmer's knowledge of the particular domain being supported. Further, the specific mechanisms provided in Prospero for managing collaborative work are ones that operate in terms of the application domain semantics. Streams will synchronise according to semantically meaningful actions in the application domain. Consistency guarantees go further and use explicit encodings of semantic properties to create specialised behaviours.

The design goals in the development of Prospero have been two-fold. The first goal was to develop a toolkit that can be used to conveniently capture a wider range of CSCW applications than is possible using traditional techniques. As illustrated in the examples given here, Prospero has been used to develop applications that are synchronous and ones that are asynchronous; some that are free-for-all and some that use more structured forms of interaction; some that use replicated and some that use centralised data storage; and ones in which synchronisation arises automatically and ones in which it is explicitly controlled by users. Traditional toolkits cannot capture this range of applicability to different situations and application demands. The second goal was to illustrate and develop the use of semantic approaches to the creation of collaborative applications. The variety of forms of collaborative activity that we observe in laboratory and real-world settings are testament to the variety of potential forms of collaboration in different settings. Prospero has taken this observation as a basic design premise and provided technological support for the semantically-based differentiation between domains and applications.

Prospero is a prototype system that provides a proof-of-concept implementation of these ideas. Ongoing development of semantically enriched approaches in which the interaction between infrastructure and use is taken as a primary design directive holds considerable promise for a new range of collaborative applications that fit more seamlessly with patterns of everyday collaborative work.

Acknowledgments

This work was conducted while I was employed at the Xerox Research Centre Europe (Cambridge Lab, formerly EuroPARC) and studying in the Department of Computer Science at University College, London. The first draft of this article was written while I was employed at Apple Research Laboratories.

I would particularly like to thank Richard Bentley, Jon Crowcroft, Prasun Dewan, Beki Grinter, Rachel Jones, John Lamping and Tom Rodden and the anonymous TOCHI reviewers for their insights and contributions to the development of the research and this article.

References

Barga, R. and Pu, C. 1996. Reflection on a Legacy Transaction Processing Monitor. In Proc. *Reflection '96*.

Barghouti, N. and Kaiser, G. 1991. Concurrency Control in Advanced Database Applications. *Computing Surveys*, 23(3), pp. 269–317.

Beaudouin-Lafon, M. and Karsenty, A. 1992. Transparency and Awareness in Real-Time Groupware Systems. In *Proc. ACM Symposium on User Interface Software and Technology UIST'92*. ACM, New York.

Beck, E. and Bellotti, V. 1993. Informed Opportunism as Strategy. In *Proc. Third Eruopean Conference on Computer-Supported Cooperative Work ECSCW'93*. Kluwer, Dordrecht.

Bentley, R. and Dourish, P. 1995. Medium versus Mechanism: Supporting Collaboration through Customisation. In *Proc. Fourth European Conference on Computer-Supported Cooperative Work ECSCW'95*. Kluwer, Dordrecht.

Berlage, T. 1994. A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects. *Transactions on Computer-Human Interaction*, 1(3), 269–294.

Braun, T. and Diot, C. 1995. Protocol Implementation Using Integrated Layer Processing. In *Proc. ACM SIGCOMM'95*. ACM, New York.

Cao, P., Felten, E. and Kai, L. 1994. Implementation and Performance of Application-Controlled File Caching. In *Proc. ACM Symposium on Operating Systems Design and Implementation*, pp. 165–178. ACM, New York.

Clark, D. and Tennenhouse, D. 1990. Architectural Considerations for a New Generation of Protocols. *ACM SIGCOMM Communications Review*, 20(4), 200–208. ACM, New York.

Crowley, T., Milazzo, P., Baker, E., Forsdick, H. and Tomlinson, R. 1990. MMConf: An Infrastructure for Building Shared Multimedia Applications. In *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '90*. ACM, New York.

Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M. and Welch, B. 1994. The Bayou Architecture: Support for Data Sharing Among Mobile Users. In *Proc. IEEE Workshop on Mobile Computing Systems and Applications*. IEEE.

Dewan, P. and Choudhary, R. 1992. A High-Level and Flexible Framework for Implementing Multiuser User Interfaces. *Transactions on Information Systems*, 10(4), 345–380. ACM, New York.

Dix, A. 1992. Pace and Interaction. In *People and Computers VII: Proc. of HCI'92*. Cambridge University press: Cambridge.

Dourish, P. 1995. Developing a Reflective Model of Collaborative Systems. *Transactions on Computer-Human Interaction*, 2(1), 40–63. ACM, New York.

Dourish, P. 1996. *Open Implementation and Flexibility in CSCW Toolkits.* PhD Dissertation. Department of Computer Science, University College, London.

Dourish, P. and Bellotti, V. 1992. Awareness and Coordination in a Shared Workspace. In *Proc. ACM Conf. Computer-Supported Cooperative Work CSCW'92*. ACM, New York.

Edwards, K. 1996. *Coordination Infrastructure in Collaborative Systems*. PhD dissertation. College of Computing, Georgia Institute of Technology, Atlanta, Georgia.

Edwards, K., Mynatt, E., Petersen, K., Spreitzer, M., Terry, D., and Theimer, M. 1997. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proc. ACM Symposium on User Interface Software and Technology UIST'97*. ACM, New York.

Ellis, C. and Gibbs, S. 1989. Concurrency Control in a Groupware System. In *Proc. ACM Conf.Manamagement of Data SIGMOD'89*. ACM, New York.

Farrag, A.A. and Ozsu, M.T. 1989. Using Semantic Knowledge of Transactions to Increase Concurrency. *Transactions on Database Systems*, 14(4), 503–525. ACM, New York.

Gabriel, R., White, J.L. and Bobrow, D. 1991. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9). ACM, New York.

Greenberg, S. and Marwood, D. 1994. Real-time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. In *Proc. ACM Conf Computer Supported Coooperative Work CSCW'94*. ACM, New York.

Greif, I. and Sarin, S. 1986. Data Sharing in Group Work. In *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'86*. ACM, New York.

Grinter, R. 1996. Supporting Articulation Work Using Software Configuration Management Systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 5(4), 447–465. Kluwer, Dordrecht.

Haake, A. and Haake, J. 1993. Take CoVer: Exploiting Version Management in Collaborative Systems. In *Proc. InterCHI'93*. ACM, New York.

Harty, K. and Cheriton, D. 1992. Application-Controlled Physical Memory using External Page-Cache Management. In *Proc. ACM Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS V*. ACM, New York.

Herlihy, M. 1990. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *Transactions on Database Systems*, 15(1), pp. 96–124. ACM, New York.

Hill, R., Brink, T., Rohall, S., Patterson, J. and Wilner, W. 1994. The Rendezvous Architecture and Language for Multi-User Applications. *Transactions on Computer-Human Interaction*, 1(2), pp. 81–125. ACM, New York.

Kaiser, G. 1994. Cooperative Transactions for Multi-User Environments. In Won Kim (ed.), *Modern Database Management: The Object Model, Interoperability and Beyond*. ACM Press, New York.

Kiczales, G. and Rodriguez, L. 1992. Efficient Method Dispatch in PCL. In *Proc. ACM Symposium on Lisp and Functional Programming* (Nice, France). ACM, New York.

Kiczales, G. 1992. Towards a New Model of Abstraction in Software Engineering. In *Proc. International Workshop on New Models for Software Architecture: Reflection and Meta-Level Architecture* (Tokyo, Japan).

Kiczales, G. 1996. Beyond the Black Box: Open Implementation. *IEEE Software*, January, 6–11. IEEE.

Kiczales, G., des Rivières, J. and Bobrow, D. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Mass.

Maeda, C. 1996. A Metaobject Protocol for Accessing File Systems, in *Proc. International Symposium on Object Technologies for Advanced Software ISOTAS'96* .

Munson, J. and Dewan, P. 1994. A Flexible Object Merging Framework. In *Proc. ACM Conf. Computer-Supported Cooperative Work CSCW'94*. ACM, New York.

Munson, J. and Dewan, P. 1996. A Concurrency Control Framework for Collaborative Systems. In *Proc. ACM Conf. Computer-Supported Cooperative Work CSCW'96*. ACM, New York.

Myers, B. 1990. A New Model for Handing Input. *ACM Transactions on Information Systems*, 8(3), pp. 289–320. ACM, New York.

Myers, B., Guise, D., Dannenberg, R., Vander Zanden, B., Kosbie, D., Pervin, E., Micklish, A. and Marchel, P. 1990. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 23(11). IEEE.

Nichols, D., Curtis, P., Dixon, M., and Lamping, J. 1995. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *Proc. ACM Symposium on User Interface Software and Technology UIST'95*. ACM, New York.

O'Malley, S. and Peterson, L. 1992. A Dynamic Network Architecture. *Transactions on Computer Systems*, 10(2). ACM, New York.

Rao, R. 1991. Implementational Reflection in Silica. In *Proc. European Conference on Object-Oriented Programming ECOOP'91*. Springer-Verlag.

van Reneese, R., Birman, K. and Maffeis, S. 1996. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4), 76–83. ACM, New York.

des Rivières, J. and Smith, B. 1984. The Implementation of Procedurally Reflective Languages. In *Proc. ACM Conference on Lisp and Functional Programming*, 331–347. ACM, New York.

Roseman, M. and Greenberg, S. 1993. Building Flexible Groupware Through Open Protocols. In *Proc. ACM Conference on Organisational Computing Systems COOCS'93*. ACM, New York.

Roseman, M. and Greenberg, S. 1996. Building Real-Time Groupware with GroupKit, a Groupware Toolkit. *Transactions on Computer-Human Interaction*, 3(1). ACM, New York.

Schuckmann, C., Kirchner, L., Schümmer, J. and Haake, J. 1996. Designing Object-Oriented Synchronous Groupware with COAST. In *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'96*. ACM, New York.

Shen, H. and Dewan, P. 1992. Access Control for Collaborative Environments. In *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92*. ACM, New York.

Smith, B. 1984. Reflection and Semantics in LISP. In *Proc. ACM Symposium on Principles of Programming Languages POPL*. ACM, New York.

Stroud, R. and Wu, Z. 1995. Using Metaobject Protocols to Implement Atomic Data Types. In *Proc. European Conference on Object-Oriented Programming ECOOP'95*. Springer-Verlag.

Suchman, L. 1987. *Plans and Situated Actions.* Cambridge University Press, Cambridge.

Terry, D., Demers, A., Petersen, K., Sprietzer, M., Theimer, M. and Welch, B. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. International Conference on Parallel and Distributed Information Systems*.