Open Implementation and Flexibility in CSCW Toolkits

Paul Dourish

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philisophy
of the
University of London.

Department of Computer Science
University College London

June 1996

This distribution copy has been repaginated to reduce its size. The page numbers are not the same as in the submitted version. Please do not cite anything by page number.

Abstract

The design of Computer-Supported Cooperative Work (CSCW) systems involves a variety of disciplinary approaches, drawing as much on sociological and psychological perspectives on group and individual activity as on technical approaches to designing distributed systems. Traditionally, these have been applied independently—the technical approaches focussing on design criteria and implementation strategies, the social approaches focussing on the analysis of working activity with or without technological support.

However, the disciplines are more strongly related than this suggests. Technical strategies—such as the mechanisms for data replication, distribution and coordination—have a significant impact on the forms of interaction in which users can engage, and therefore on how their work proceeds. Consequently, the findings of sociological and psychological investigations of collaborative working have direct impact for how we go about designing collaborative systems.

In support of this relationship, this thesis concentrates on the provision of flexibility in CSCW systems, and, in particular, in toolkits from which they are generated. Flexibility is key to supporting many characteristics of group behaviour detailed by observational investigations—the improvised nature of work and activity, individual and group tailoring, customisation and re-purposing, changing group membership and activity over the course of a collaboration, and so forth.

Based on an analysis of current CSCW toolkits, and on the interaction between user behaviour and system design, I will demonstrate that, as in many other areas of system development, traditional notions of abstraction in system design mitigate against the design of open, flexible systems. "Open Implementation" is an emerging approach based on the systematic and principled exposure of mechanism in system components, "opening up" abstractions to examination and manipulation. Concentrating particularly on distributed data management and concurrency, I will show how these ideas can be exploited to provide an open and customisable framework enabling programmers and end-users to tailor toolkit structures to the needs of applications and domains.

Abstract .

Dedication

The character of Prospero, the magus of Shakespeare's "The Tempest", was reputedly inspired by the Elizabethan scientist, alchemist and Hermetic philosopher, John Dee (1527–1608). The tenets of Hermetic philosophy, while intricate, have a simple basis—that the structure of our earthly world reflects the structure of the heavens. By understanding the earthly world, the alchemist would learn about the world of the spirits; by gaining power and control over the earthly world, he would gain "magical" power and control of the higher dimensions. "As above, so below." So the transmutation of base matter to gold was a metaphor for the transformation of man to god; a transformation which could be achieved by the adept once the correspondence from above to below is understood, since the pattern of man must be the same as the pattern of god.

Since I have spent much of the past few years trying to understand and control the link between what is below and what is above, this thesis is dedicated to John Dee.

Acknowledgements

The work in this thesis has had three homes—the Rank Xerox Research Centre, Cambridge Laboratory (formerly EuroPARC); the Computer Science department at University College, London; and the Xerox Palo Alto Research Center. I couldn't have asked for three more interesting and stimulating places to have spent the last three years, and I owe a great deal to the people who have supported me and my work at each place: Bob Anderson, Graham Button, Wendy Mackay and Allan MacLean at EuroPARC, Jon Crowcroft and Angela Sasse at UCL, and Annette Adler, Austin Henderson and Gregor Kiczales at PARC.

The development of the ideas presented here has been inspired, nurtured and guided by a host of people, only some of whom can be mentioned here. Jon Crowcroft provided guidance, advice, criticism and beer in admirable proportions, tolerating the rather odd path of my work with bemused grace. Annette Adler, Bob Anderson, Graham Button, Wendy Mackay and Lucy Suchman have, wittingly or unwittingly, contributed to my inclination towards the social which underpins this work (although they should not be held responsible). Annette also brought together a group of people who have provided me with much intellectual sustenance over the last year or two; David Levy, Gene McDaniel, Bob Printis, Vijay Saraswat and Brian Smith deserve especial mention. Brian's work has been central to much I've thought and done over the last few years; one day, I hope to understand 10% of it.

Dik Bentley, Jon Crowcroft, Alan Dix, Steve Freeman, Jonathan Grudin, Rachel Jones, Gregor Kiczales and John Lamping have all made valuable contributions to my thinking about the ideas described here, and to the form of their presentation.

Some of my greatest debts are to those who helped me through the deepest moments of inevitable thesis gloom. Lorna Banks, Eevi Beck, Dik Bentley, Matthew Chalmers, Lynn Cherny, Laura Dekker, Bill Gaver, Lorna Goulden, Maxine Gregson, Rory Hamilton, Rachel Hewson, Pernille Marqvardsen, Gillian Ritchie, Ellen Siegel, Lisa Tweedie, and Alex Zbyslaw all helped with email, cynicism, laughter, silliness and beer, and they all deserve better recompense than a few words here. I'll get round to you all in time.

Victoria Bellotti deserves a place on her own as both an inspiring colleague and valued friend. And above all, thanks to Beki Grinter, who's simply more than I deserve.

Thank you all. May you never have to hear about this again.

Table of Contents

Chapter 1: Introduction

1.1 Introductory Remarks

This dissertation examines the issue of flexibility in toolkits for creating multi-user applications. In particular, I will show how a novel software architecture technique—Open Implementation (OI)—can be applied to solve a number of problems inherent in traditional approaches, and I will present a toolkit—Prospero—which embodies and demonstrates these ideas.

The motivation for this work is not simply the technical limitations of a set of software techniques. Certainly, previous work has applied the OI approach to a number of technical areas to solve problems of that sort. However, the support for deeper, more thoroughgoing forms of flexibility in support of collaborative work is also driven by the need to address a split within the disciplinary structure of the field itself.

1.2 The Emergence of CSCW

The field of Computer-Supported Cooperative Work (CSCW) traces its history as an identifiable discipline to a workshop held at MIT in 1984 by Irene Grief and Paul Cashman (who are also responsible for the term). It grew quickly; in 1986, MCC sponsored the first bi-annual international conference, subsequently sponsored by the ACM; and in 1989, a bi-annual European conference began, held in the alternate years. CSCW has spawned sub-disciplines of its own (such as workflow technology, frequently employed within Business Process Reengineering), conferences, journals, prototypes, studies, and jargon.

Technological and organisational trends of the last ten years have served to reinforce the importance of CSCW by strengthening the motivations which gave rise to the field in the first place. First, the rapid dissemination of computer power—CPU performance, graphics processing, etc.—associated with the move from mainframe to distributed personal computing in the early 1980s has continued and shows no signs of stopping; and at the same time, these powerful PCs and workstations are increasingly connected together over local and wide-area networks. Meantime, business trends mean that organisations are increasingly distributed. Not only is a single organisation more likely to be distributed across multiple sites, or perhaps employ individuals working electronically away from a principal site, but also increasing moves towards "outsourcing" and inter-organisation collaboration makes it more likely that work is being performed through the collaborative activity of remotely located individuals. Even within single or co-located organisations, the trend towards automation of tasks around explicit models of business processes places an emphasis within computational technology on collaboration and coordination.

This is the area to which CSCW has addressed itself since the outset—the use of computer-based tools to enhance and facilitate the performance of collaborative work, potentially distributed in space or time. Its scope, then, is very broad, and this is reflected in the wide range of systems which have been designed, built and studied. For example, these have included: shared whiteboard/meeting room systems such as CoLab (Stefik et al., 1987a) and Dolphin (Streitz et al., 1994); workflow/process-based systems such as The Coordinator (Winograd, 1986) and the Milan Conversation System (De Michaelis and Grasso, 1994); authoring systems such as PREP (Neuwirth et al., 1990) and Quilt (Fish et al., 1988); drawing systems such as Commune (Bly and Minneman, 1990) and The Conversation Board (Brink and Gomez, 1992); as well as collaborative Virtual Reality systems, databases, software design environments, and others.

More significantly, perhaps, CSCW, as a field, has also drawn on a similarly wide range of disciplinary perspectives on its own subject matter. Arising as it did partly out of the field of Human-Computer Interaction (HCI), it is unsurprising that, like HCI, it should draw on cognitive and experimental psychology in addition to computer science and software engineering. However, much of its development and perspective has been driven by the early influence of anthropology and sociology.

1.3 Ethnomethodology and CSCW

A highly significant factor in the development of CSCW was the very early involvement of researchers from anthropology and sociology. Just as cognitive psychology provided the underpinnings of much HCI research in the late 1970s and early 1980s, so sociology has motivated and grounded much work in CSCW since the late 1980s.

For a variety of reasons, the area of sociology most prominent in CSCW has been ethnomethodology. Ethnomethodology arose in the 1960s, through the work of Harold Garfinkel and associates. Its roots lie in a respecification of the objectives of sociology. In particular, it reacts against a view of human behaviour that places social action *within* the frame of social groupings and relationships which is the domain of traditional sociological theory and discourse—categories and their attendant social functions. Ethnomethodology's primary claim is that individuals do *not*, in their day to day behaviour, act according to the rules and relationships which sociological theorising lays down. Quite the opposite. The structures, regularities and patterns of action and behaviour which sociology identifies emerge *out of* the ordinary, everyday action of individuals, working according to their own common-sense understandings of the way the social world works. These common-sense understandings are every bit as valid as those of learned professors of traditional sociology.

From this basic observation, a new picture of social action has arisen. The ethnomethodological view emphasises the way in which social action is not achieved through the execution of pre-conceived plans or models of behaviour, but instead is *improvised* moment-to-moment, according to the particulars of the situation. The sequential structure of behaviour is *locally organised*, and is *situated* in the context of particular settings and times.

Ethnomethodology's concern, since its beginning, has been the organisation of human action and interaction. In 1987, Lucy Suchman published *"Plans and Situated Actions"*, which applied the same techniques and perspectives to the organisation of interaction between humans and technology. In doing so, she opened up significant new areas of investigation both for HCI researchers and ethnomethodologists. In both this and other work (e.g. Suchman, 1983; 1993), Suchman also helped establish the relationship between ethnomethodology and CSCW, and the same approach has been used by a range of researchers studying work in collaborative settings such as stock trading rooms (Heath et al., 1995), print shops (Bowers et al., 1995), and air traffic control rooms (Harper and Hughes, 1993). The ethnomethodological perspective on CSCW systems and design has continually emphasised the flexible and open ways in which working activities are organised, and the distinctions between the formal processes and procedures which are often encoded in CSCW systems and the informal, flexible and opportunistic practices by which work and interaction actually take place. The flexibility required of systems to support the development of these working practices is, then, a key feature of their design.

1.4 The Development of CSCW Toolkits

Against the backdrop of these investigations of collaborative activity, technological development has proceeded apace.

The earliest CSCW systems were built by hand, extending techniques originally developed for single-user interfaces and standard distributed systems to the needs of collaboration. Often, they were designed as extensions of existing systems such as databases (Grief and Sarin, 1988) or hypertext systems (Rein and Ellis, 1991). As both application and implementation experience grew, specifically multi-user aspects of implementation support began to migrate into reusable bodies of code. For explanatory purposes, we can draw an approximate distinction between *execution environments* and *application toolkits*.

Execution environments provide run-time support for multi-user applications. Often, their aim is to allow otherwise single-user applications to function in multi-user arrangements, although they might also support specifically multi-user features. Examples of application environments include commercial systems such as Timbuktu and Shared-X, and research systems such as ABC (Jeffay, 1992). Many early attempts to encapsulate and reuse multi-user functionality from earlier monolithic CSCW systems were based on (or at least, would include) multi-user execution environments.

A number of execution environments were paired with application toolkits. Toolkits provide libraries of functionality for the development of specifically collaborative applications. Multi-user interface widgets, persistent shared data stores and group-replicated objects are examples of abstractions which might be presented in a collaborative application toolkit. As will be discussed in Chapter 2, the design of these various toolkits reflects the needs of the particular application areas which they were destined to support. The generality or flexibility of the toolkit, then, is reflected in the range of applications which can be derived from it.

1.5 Flexibility and Tailorability in HCI

As CSCW development was progressing, questions of openness and customisation of system behaviour had already become an important research issue in HCI generally (see, for example, Trigg et al., 1987; Fischer and Girgensohn, 1990; Mackay, 1990; MacLean et al., 1990). By and large, the focus of these investigations was on *tailorability*—that is, the ways in which applications could be customised to accommodate individual differences between members of a user community. Since this work tends to make customisation available through named, accessible features of the visual language of the interface, research on tailorability tends to focus on applications at a fairly high level, providing controls which either affect largely "surface" features of the interface or control system behaviour at a fairly gross level. More recently, however, these investigations have begun to move into the area of end-user programming, providing greater control over semantic features of the application, which are generally not directly expressed in the interface (Bentley and Dourish, 1995). Notable examples include the work of Nardi (1993) in a variety of domains, Cypher and Smith (1995) with KidSim and Eisenberg (1995) with SchemePaint.

1.6 Flexibility and Tailorability in CSCW Systems and Toolkits

A number of researchers have investigated similar issues of flexibility and tailorability in the domain of multi-user, rather than single-user, interfaces. Greenberg (1991) follows most closely in the tradition of customisation to accommodate individual differences (Chapter 2 will discuss other work in more detail). Arguably, the problem of tailorability *per se* is considerably more pressing in multi-user applications than in single-user ones. The problem of accommodating individual preferences in single-user systems is akin to the problem of accommodating the preferences of different groups in the multi-user case; however, multi-user systems also face the problem of accommodating the different preferences of the individual *members* of each group. However, there is a second, more fundamental, problem which arises when looking at tailorability in multi-user systems; and it arises from the traditional separation between the disciplinary perspectives (loosely, social and computational sciences) which make up the field.

1.6.1 Flexibility from Social and Technical Perspectives

The traditional relationship between these disciplines involves sociological analyses of collaborative work reaching “down” from the day-to-day organisation of work to the collaborative applications which are used to support it, while the systems perspective reaches “up” from the requirements and constraints of the underlying systems towards the applications level, determining the range of functionality which can be provided by those applications. In this arrangement, the perspectives “meet” at the level of applications.

More recent work, though, has come to question the separation this implies—that is, the view that system design requirements and constraints do not have consequences above the level of the application, and that sociological insights do not apply below. As will be discussed in Chapter 3, this notion of the separation of levels of concern is not even entirely appropriate purely from within the systems design perspective. Critically, though, for what is to follow, it is clear that the relationship between the two perspectives is considerably more intricate that this traditional view allows. In general, the *detail of collaborative activity*, as explored by the ethnomethodological perspective, is rooted in the *detail of the technology* available, whether that technology is comprised of everyday physical artifacts (such as those discussed in Heath et al.’s (1995) study of broker dealing rooms) or computational ones (such as those discussed in Grinter’s (1995) study of software engineers).

For instance, consider the issue of conflict management over shared data. Some data object exists in a shared workspace, and is available to a number of users for viewing and editing. A wide range of techniques might be used to achieve this end in any particular implementation. Actions might be serialised over a single copy of the data item; the item might have to be “locked”, or “checked out”; or conflicts over actions applied to multiple copies might be subject to prioritisation or rollback. The particular strategy used is not at issue here. Rather, the critical observation is that the choice of implementation strategy lies *below* the application level; it is invisible—or inexpressible—above it. In terms of the traditional relationship between the perspectives, then, this would imply that the choice of implementation strategy is irrelevant as long as the application functionality is maintained.

However, the details of the specific implementation strategies carry with them important implications for the nature of the tasks which can be carried out. In an experimental study, Dourish and Bellotti (1992) explored the ways in which groups of collaborating authors organised their activity (both individual and collective) around detailed features of the collaborative interface and the shared workspace control mechanisms, as enabled by specific implementation mechanisms. Similarly, Greenberg and Marwood (1994) examined the relationship between the mechanisms of distributed data management and the functionality of the user interface. As these studies demonstrate, far from being completely independent concerns, the issues of infrastructure flexibility and usage flexibility are closely related.

1.6.2 A Cross-Cutting View of Flexibility

The primary topic of this thesis, then, is the provision of flexibility in CSCW toolkits. Flexibility is investigated from both perspectives. From the systems perspective, the goal is to build toolkits which are more flexible than traditional systems, so that a wider range of applications can be supported. From the usage perspective, the goal is to be able to provide flexibility in forms which support the emergent work practices which social investigations highlight. Running through both of these concerns, though, is the realisation that they are not independent issues. Rather, the relationship between the systems and social perspectives highlights a set of concerns which cut across traditional boundaries between aspects of system design and analysis, and which require a novel approach to both the form and function of those barriers. At the same time as unifying these two views, this approach supports the development of a toolkit which is much more flexible than traditional techniques allow—that is, it extends the “reach” of the toolkit.

The technique that I will draw upon is called Open Implementation (Kiczales, 1996). Open Implementation (OI) has its roots in a re-evaluation of the notion of abstraction in software engineering. The sorts of problems which are encountered in CSCW systems are endemic to systems design, and occur in widely diverse fields. Open Implementations reframe the flexibility problem in terms of the role and use of “abstraction” in current software practice. OI itself has developed from *computational reflection*. Reflection is a mechanism, based on computational self-reference, originally employed in programming language design. By providing systems with an explicit and manipulable model of their own behaviour, reflection introduces opportunities for the system to *introspect* (reason about its own behaviour by examining the model) and *intercede* (control aspects of its behaviour by modifying the model). The OI approach applies this principle more generally to the problems of abstraction, by beginning to "open up" system components and make them amenable to a certain amount of control from above.

1.7 What Is To Come

This, then, is the line of argument which this dissertation will lay out and demonstrate:

1. That the evidence of empirical and naturalistic studies of cooperative work demonstrates that usage issues and system issues are fundamentally linked.
2. That the re-evaluation of abstraction in software engineering, set out by research in Open Implementation, applies to these issues as encountered in CSCW.
3. That OI/reflective principles can be used to design a novel CSCW toolkit.
4. That a toolkit built along these lines yields significant improvements in design (and hence, usage) flexibility.

Following on from the discussion here of the social and technical dimensions of CSCW, Chapter 2 will discuss the approaches taken by a range of CSCW toolkits and systems, looking in particular at their support for flexibility in application design and use. With a clearer view of the problems, then, Chapter 3 will introduce the Open Implementations approach, and its analysis of these problems in terms of the use of abstraction in software design.

This approach leads us towards a solution to the problems of flexibility in CSCW systems. Most importantly, it does this in a way which opens up some of the barriers between traditionally "high" and "low" level concerns—the separation between the disciplines described earlier. Chapter 4 will introduce the use of OI in a CSCW toolkit, Prospero, which attempts to address the problem of flexibility and separation. Dealing especially with the areas of (1) distributed data management and (2) consistency and conflict control, Prospero uses OI techniques to allow applications to become involved in the mechanisms within the toolkit which support them. Application programmers can "reach in" to the toolkit, tailoring and specialising toolkit structures to the particular needs of their applications. Chapters 5 and 6 will discuss, in turn, these two principal areas of attention, and the novel techniques which Prospero introduces to provide application programmers with toolkit control; and finally, Chapter 7 will present some more extended examples to give a clearer overall picture of the toolkit and its uses.

Chapter 2: Flexibility in CSCW Toolkits

2.1 Introduction

The previous chapter emphasised the importance of flexibility in collaboration. CSCW applications are characterised by a diversity of usage patterns, group working styles, interactional requirements and technologies. At the same time, observational and sociological studies of collaborative work highlight the variable, locally-organised nature of the sequential organisation of activity. Flexibility in working styles implies flexibility in CSCW tools, and in turn, in the technological bases for their development.

In this chapter, I will be concerned with issues of flexibility in existing CSCW toolkits. Flexible and adaptable operation is at the heart of toolkit design, since it provides for reuse. I will begin by examining this relationship between flexibility and reuse, from both the system design and HCI perspectives, and then discuss various ways in which flexibility might be embodied in these systems. Following this, in section 2.4, I will present a number of existing CSCW toolkits, discussing their main features and facilities, with particular emphasis on their support for flexibility and on the range of applications which they support. Taking these systems as examples, I will step back in section 2.5 to consider styles and elements of flexibility more generally. In particular, I will draw a distinction between flexible operation through *genericity* and through *extensibility*; and then, on the basis of this analysis, I will highlight problems common to all the toolkits discussed.

2.2 Flexibility and Reuse

As in any other area of software design, early CSCW systems were constructed as monolithic entities, specially crafted to needs of particular applications, user communities and software environments. As experience with these early systems grew, as the scope of applications diversified, and as understandings of system requirements, application characteristics and usage patterns increased, it became both possible and desirable to embody common design elements in toolkits[1].

2.2.1 Software Issues

From a software perspective, one of the primary goals of toolkit design is *reuse*. When embodied in a toolkit, components, algorithms and implementations can be recombined and reused in different applications. Much of the value of the toolkit resides in the *range* of applications it can be used to create. The design of abstractions for the toolkit, then, focuses on the balance between two competing requirements. On the one hand, toolkit design strives to maximise the generality or commonality of the toolkit's component structures, so that they can be used in multiple applications. On the other, it emphasises the flexibility in the ways they can be instantiated

1. Note that, in this discussion, I will use the term "toolkit" to include what might otherwise be called libraries, frameworks, etc.—that is, systems which are used to generate applications, through specialisation, inclusion, run-time support, etc.

and manipulated (so that a wide range of applications can be derived from these components). The flexibility which allows common components to be used in different ways and to different purposes has to be balanced against other criteria such as interoperability, consistency and integration.

These issues, of course, are common to software concerns in the design of toolkits for any domain. Patterson (1991) discusses some of the particular issues which CSCW systems raise in comparison with single-user interactive systems, focussing in particular on a set of issues addressed in the Rendezvous toolkit, which will be examined shortly.

2.2.2 HCI Issues

Flexibility is an important issue in HCI research generally, and a number of researchers have studied issues of customisation, tailorability and adaptation (see, for example Trigg et al., 1987; Maclean et al., 1990; Mackay, 1989, 1990, 1991; Nardi, 1993). HCI's concern with flexibility has been motivated by a desire to match the features and behaviour of interactive tools to the range of working styles and tasks to which they might be applied. The focus of such research, however, has largely been on flexibility within the context of particular applications or particular instances of software systems. In these cases, variability operates across a population of users, employing an application in different circumstances. While some of this work (e.g. Mackay, 1990; Nardi and Miller, 1991) has emphasised the importance of tailoring as a group activity, this tailoring is still largely performed in support of individual needs.

In CSCW applications, the problems of flexibility (or rather, of *inflexibility*) can become more readily apparent—and that much greater—since the "user" of a CSCW system is a *group* of individuals. As a result, variability can occur within a single group, as well as between different groups. This has led to a number of attempts to incorporate intra-group flexibility, variable interface coupling patterns, etc. (e.g. Greenberg, 1991). As in traditional HCI, however, this type of investigation takes a user focus, and concentrates largely on single applications or application areas.

2.3 A Framework for Flexibility

In their discussion of tailorability facilities in the Notecards hypertext system, Trigg et al. (1987) set out four levels of flexibility:

1. a *flexible* system provides generic objects and behaviours which can be combined and used differently by different users of the system;
2. a *parameterised* system provides objects with a range of alternative behaviours;
3. an *integrable* system can be connected to other system components;
4. a *tailorable* system allows users to change the system itself, add functionality or specialise behaviours.

Drawing on the HCI perspective outlined above, this separation focuses on flexibility and adaptability in applications. The concerns in programmable toolkits differ, although we can fruitfully make use of these distinctions.

Toolkits are designed to be used to create some range of applications. They are all *flexible* in Trigg et al.'s terms. The concern which is addressed here is not their *being flexible*, but the *extent of their flexibility*—that is, the range of applications which they can be used to create. This captures not only the range of application domains to which they can be applied, but also the range of interactional styles which can be supported using toolkit components and mechanisms. So, for my purposes here, flexibility is a relative term, rather than an absolute one. In support of this, the critical aspect which will be addressed here is the fourth level of Trigg et al.'s analysis, *tailorability*; the extent to which toolkit facilities themselves can be adapted and extended. (At the same time, I will also continue to use the term "flexible" as a general one, rather than in the specific sense which Trigg et al. introduce.)

In the analysis which will follow (section 2.5), I will discuss these issues in terms of two particular concepts. G*enericity*, is the extent to which toolkit features are generic, capturing a wide range of behaviours, and the use of this property to support flexible action. *Extensibility*, the extent to which they can be extended and revised to support new behaviours, and the similar way in which this is used.

An orthogonal concern is the "locus of flexibility"—that is, the point within the system where flexibility is handled and exploited. I will distinguish between three points at which flexibility issues come to the fore:

1. *Programmer/toolkit*—flexibility within the toolkit, offered to the programmer so that toolkit structures can be used variably, adapted or augmented (for example, controlling floor control criteria);
2. *Programmer/application*—flexibility exploited in the application under programmer control. This might include, for instance, the use of adaptive algorithms to organise internal behaviours around the patterns of use at a particular time (for example, dynamic data control which adapts to network bandwidth and topology);
3. *User/application*—flexibility in the application under user control. This includes parametric control over application behaviour (for example, control over levels of user interface coupling), as well as the extension of applications with new user-supplied structures or behaviours.

There is a clear relationship between these three forms of flexibility. Flexibility at any one point is helpful in providing further flexibility down-stream. Here, however, I am concerned with the first level, flexibility within the toolkits themselves; the flexibility required to support a diverse range of applications. My focus is on the mechanisms by which such flexibility is realised, how these are specified and used, and with the ways in which they affect the properties (and the range) of applications which can be developed.

Even a brief examination of existing CSCW applications reveals a host of systems and interface issues addressed in different ways, which may each be the locus of particular flexibility concerns. Apart from the obvious differences in application domains, these include the techniques by which data is distributed and managed; mechanisms for joining and leaving collaborative sessions; mechanisms for mediating between individual and group work; means by which individuals become aware of the activity of others; levels of interface coupling; temporal aspects of interpersonal interaction; and so forth. Toolkits, similarly, vary in the extent to which programmers can gain control over these sorts of issues, and can potentially adapt toolkit mechanisms to the need of particular applications. These are the ways in which flexibility is manifest in toolkits themselves, and in which it is made accessible to programmers developing collaborative applications.

2.3.1 Collaboration Transparency and Collaboration Awareness

One approach to collaboration support, particularly represented in early systems, is organised around the need for *collaboration transparency*—that is, the collaborative operation of existing single-user applications, without any modification for explicit collaboration support (Lauwers et al., 1990). A number of CSCW systems provide run-time environments which allow single-user applications to be shared; examples include Rapport (Ahuja et al., 1990) and Shared X (Gust, 1988; Garfinkel et al., 1989). Others, such as MMConf (Crowley et al., 1990) or ABC (Jeffay, 1992), support collaboration transparency as one possible mode of operation.

Typically, such systems interrupt the event flow between application and interface at some point, and insert a multiplexor. This replicates the application's actions, making them available at a number of points on a network, and can potentially accept user input from multiple streams and channel it to the application. Since the underlying applications contain no native understanding of the multiple streams which might now be used to interact with them, application sharing environments might also introduce "floor control" mechanisms to manage input streams explicitly. These regulate users' access to the application input stream, using various policies to determine how access to the application's input stream is to be distributed between the various users.

In a wide range of situations, support for collaboration transparency is critical. Users typically deploy a range of applications in support of a task, and so any of these might need to be made available to a group (at least in some "display" mode, where one user remains in control). However, this approach to application sharing has a number of drawbacks. The first is that the forms of collaborative activity which they can support are typically quite limited, since the interface is not designed to reflect the activity of multiple users. Floor control mechanisms restrict access, or provide a controlled sharing model, to support the single-user approach of the application, at the cost of restricting the group to one-at-a-time action. The second major drawback is that the centralisation implied by collaboration transparency can impose a heavy performance burden. Typically, the collaboration run-time mechanism is sharing access to a single application, running at one point on a network; the cost of replicating display updates to all the participating hosts can be high, especially over long distances or low-bandwidth connections.

In general, though, the collaboration-transparent approach is not of direct concern here; the requirements imposed by collaboration transparency limit the options for incorporating variable or flexible mechanisms within the infrastructure itself. Instead, I will focus primarily on systems supporting collaboration-aware applications.

In the next section, a number of existing CSCW toolkits are presented and briefly described and compared. In the subsequent section, I will step back to consider the issues of flexibility more generally, and consider the ways in which current strategies for incorporating toolkit flexibility are problematic.

2.4 CSCW Toolkits and Approaches

This section presents six CSCW toolkits of various sorts—Rendezvous, GroupKit, MEAD, Suite, Oval and COLA. The systems vary widely in both form and focus. After presenting aspects of the structure and design of each toolkit, I will focus in particular on how they address flexibility issues.

2.4.1 Rendezvous

Rendezvous (Patterson et al., 1990; Hill et al., 1994) is a CSCW toolkit created at Bellcore. Rendezvous is primarily designed for building synchronous, graphical groupware applications. Written in Common Lisp, and running over the X Window System, it has been used to create a number of applications, from drawing tools such as the Conversation Board (Brink and Gomez, 1992) to distributed user interface prototyping environments (Miller et al., 1992).

The mechanism at the heart of Rendezvous is the *Abstraction-Link-View* (ALV) structure (Hill, 1992). *Abstractions* capture shared aspects of application representations. *Views* present individual users with interface representations and interaction mechanisms. Different users may see different representations, or be presented with different interaction modalities to the same abstraction by their different views. Systems of constraints (*Links*) relate Abstractions to Views. Links are responsible for transforming user interface actions in the Views into appropriate modifications to the Abstraction, and similarly for ensuring that Views are appropriately updated when any changes are introduced to the common Abstraction. Rendezvous uses this constraint-based mechanism to manage all communication between Views and the Abstraction; all messages to the Abstraction must be implemented in terms of constrained state-changes.

2.4.1.1 Flexibility in Rendezvous

Rendezvous is, very much, a programmatic toolkit. In other words, the flexibility which it embodies is achieved through the ways in which its mechanisms are built into the base language (Common Lisp) and can be manipulated and combined using the facilities that the programming language offers. This approach works particularly well on top of Lisp as a base language—Lisp has been insightfully described as a language for building little languages. Despite this largely programmatic focus, however, there are also some flexibility issues addressed by aspects of the system's design itself.

4.1.1.1 Replication and Centralisation

Rendezvous concentrates on support for real-time (synchronous), graphical, direct-manipulation interfaces, using the constraint mechanism to provide dynamic feedback between an abstraction and multiple views. This structure is reflected in most derived applications described in the research literature (Brink and Gomez, 1992; Brink and Hill, 1993; Hill et al., 1994).

Although the ALV approach presents the abstraction as the central point of coordination, Rendezvous' model is not inherently restricted to centralised designs. For example, one could certainly imagine using the same constraint system which relates Views to Abstractions as a means to synchronise multiple copies of an Abstraction. However, the current Rendezvous implementation uses a centralised model; in fact, the Abstraction object and all View objects exist within a single address space on one machine, using remote X displays to distribute interfaces. The designers comment that they have found a single 64kbps channel sufficient for any of their existing implementations (Hill et al., 1994), although they do not discuss issues of latency over long distances, which are surely problematic. They do, however, discuss the desirability of a distributed implementation which would separate the Views from the Abstraction.

4.1.1.2 Declarative Event Model

Rendezvous' strongly object-oriented architecture supports a powerful programming model. In particular, it allows interfaces to be constructed in a declarative style, in both their appearance and behaviour. Graphic object classes handle the specifics of particular forms of graphical object; and a powerful event model allows behaviours to be associated with objects declaratively, freeing the programmer from the complex spaghetti of callbacks and event-handing code.

This declarative model lends itself to the encapsulation of interface behaviours not unlike Myers' (1990) "interactor" approach. As will be discussed in more detail later, this encapsulation of behaviour in an object-oriented structure is particularly interesting when looking at flexibility issues, since it allows behavioural mechanisms to be specialised and modified independently of graphical and other interface features. It allows application developers to think in terms of extending the toolkit facilities, and then writing their applications in terms of this extended functionality.

2.4.2 GroupKit

GroupKit (Roseman and Greenberg, 1992; Roseman and Greenberg, 1996) is a flexible toolkit for building real-time collaborative applications. Like Rendezvous, it is a programmatic toolkit. The first implementation was in the C++ UI toolkit Interviews, and was informed by both programmer-level and user-level requirements from earlier experiments with monolithic groupware applications, as well as the collaborative window system Share (Greenberg, 1991; Greenberg et al., 1992). The current implementation is written in Tcl/Tk (Ousterhout, 1994).

Unlike Rendezvous, the run-time system is based on a replicated object model. Object replication provides the responsiveness which the system needs to support synchronous graphical applications.

GroupKit has three major components. First, a *run-time system* manages distribution issues and provides transparent support for sharing application objects. Second, an *interface system* (based on transparent overlays) provides multi-user widgets and components for creating user interfaces with collaborative properties (such as support for telepointers). Third, a programming model based on *open protocols* accommodates differences for policy issues such as floor control.

2.4.2.1 Flexibility in GroupKit: Open Protocols

The use of open protocols (Roseman and Greenberg, 1993) is of most relevance to flexibility concerns. Open protocols are designed as a technological mechanism to support personalisable groupware, through which different groups and different users can tailor the appearance and behaviour of collaborative systems.

The open protocols approach begins with the understanding that system behaviour can be controlled and configured through manipulation of state. From this, Roseman and Greenberg separate out the three constituent components of an open protocol system—a *controlled* object, a *controller*, and a *protocol* between them. The controlled object is an abstraction, residing in the (conceptual) server, which makes the protocol available as a mechanism for accessing and changing state; the controller resides in clients of the controlled object server (which are, potentially, also user interface clients of the groupware system). The server imposes no policy on the sequential organisation of state changes, and so clients can embody different policies for manipulation of the information. For instance, floor control can be managed by establishing the "current floor holder" as shared state information in a floor control server, which provides mechanisms for altering that state within the protocol; and the floor control policy (round-robin, request/grant, etc.) can be implemented by clients using the floor control protocol. The server not only does not impose a mechanism, but it is unaware of the mechanism being used, which may have been developed after the server itself was written.

So in general, this mechanism can support not only inter-group differences, but also intra-group policies. Clearly, however, there is a requirement for coordination between the controller objects, so that they operate coherently together. Coordination mechanisms like these are defined outside the open protocol approach. Open protocols simply make the tailoring facilities available; the programmer is then responsible for ensuring that applications function coherently.

2.4.3 MEAD

A long-term project at the University of Lancaster focussed on cooperative interface development in the domain of air traffic control, integrating ethnographic investigations with the system design process (Hughes et al., 1993). MEAD (Bentley et al., 1992; Bentley, 1994) is a multi-user interface prototyping environment developed in this project. The need for tailorability was driven by three concerns:

1. the need for different views and interaction mechanisms over the shared data for the various participants' purposes;
2. the variety of working practices surrounding air traffic control technologies and artifacts;
3. the need to support collaborative development with users through real-time incremental adjustments to running prototypes.

The first two concerns were raised by the domain requirements, particularly as uncovered by the ethnographic material; the third reflects a concern with iterative, collaborative prototyping.

2.4.3.1 Flexibility in MEAD: Tailorable User Display Agents

MEAD supports interface tailoring by introducing User Display Agents which sit between the central information space and individual user interfaces. The User Display Agent is responsible for the management and representation of shared information in each user's display. Each user might have multiple User Display Agents acting collaboratively to manage information across disparate interface components. Communication between the central information store and the display agents is handled via an event mechanism; display agents register interests in particular update events, and a pattern-matching mechanism dynamically relates update events to display agents which should be notified. Communication between display agents and the information store, then, takes place at a high level, defined in terms of domain semantics; the specifics of interface representation are the responsibility of the User Display Agent.

The primary flexibility concern in the User Display Agent model is supporting rapid prototyping and interface construction. This is managed through the manipulation of three classes of User Display properties—selection (objects to be displayed), presentation (forms of graphical representation), and composition (combining the objects into a single display). These attributes are available locally at each User Display site, so that they can be manipulated dynamically, providing a mechanism for user tailorability (albeit, again, largely designed in support of rapid interface prototyping rather than user tailoring). Any changes to the User Display criteria are immediately reflected in the appearance and behaviour of the display. One simplifying factor here is that User Displays are largely independent of each other, operating as separate dynamic visualisations of a shared space; and so, changing aspects of one display will have minimal impact on other Displays (although a single Display might be included in more than one user's "working set").

2.4.4 Suite

Suite is multi-user application toolkit derived from an earlier, single-user application toolkit of the same name (Dewan, 1990; Dewan and Choudhary, 1992). Suite focuses on editor-based interfaces implemented over a shared data model.

Sharing and coordination in Suite is achieved through the linkage between *shared active variables* and *interaction variables*. Essentially, this is the point where multi-user fan-out takes place. Shared active variables represent the shared abstraction whose value is reflected in some aspect of the interface—shared because they are common to all users of an application, and active because, from the point of view of any particular interface, their values may change asynchronously. For each shared active variable, there are a number of interaction variables, which are the sites of local activity for each interface. Interfaces (dialogue managers, in Suite terminology) operate on interaction variables; the toolkit manages the relationship between the interaction variables and their corresponding shared active variables according to a defined set of rules. Rules can be overridden in particular applications to customise sharing behaviour.

2.4.4.1 Flexibility in Suite: Coupling Control

Suite provides a rich model for describing the degrees of interface and value coupling between interfaces (Dewan and Choudhary, 1995). Interaction variables are organised into *coupling sets,* each associated with a set of coupling attributes. *Coupling attributes* control different aspects of interface coupling—whether the value is shared, whether forms of presentation are shared, whether scrolling is shared, and so on. The values of coupling attributes determine the extent of the coupling. *Transmission attributes* control the circumstances under which values or the results of actions are made available to other parts of the system: immediately, incrementally, on commit, after some time interval, etc.

Since attributes are associated with coupling sets, rather than the interface as a whole, different coupling modes can be present within the same system; similarly, different coupling modes may be available between different users. Each user can set coupling attributes privately; a conservative matching algorithm is used to determine the specific coupling parameters which will be used between any two interaction variables. The fan-out between shared active variables and interaction variables is the point at which Suite offers the programmer control over styles of interaction in collaborative applications.

2.4.5 Oval

Oval (Objects, Views, Agents and Links) (Malone et al., 1995) is the latest in a long line of related systems development by Tom Malone and colleagues at MIT. Beginning with Lens (subsequently Information Lens), a rule-based email processing system, Malone and others developed a model of "semi-structured" information processing, with a particular emphasis on end-user customisation and tailoring (Malone et al., 1987; Mackay, 1989). Semi-structured objects and autonomous agents then became the basis of Object Lens, a more general system for supporting collaborative applications; and Object Lens has subsequently developed into Oval.

Oval provides four basic components for which it is named—*Objects*, *Views*, *Agents* and *Links*. *Objects* are "semi-structured"; that is, an object definition provides an untyped, non-exclusive list of fields which an object of that type will contain. *Views* are visualisations of objects and object collections, allowing sets of objects to be clustered and presented in various ways depending on their contents, for instance as tables or graphs. *Agents* are production rules associated with objects which trigger when their conditions—typically, particular values of object fields—are met. *Links* relate objects, allowing the creation of networks, hierarchies, etc.

What Oval provides is essentially a visual programming language, expressed in terms of these objects and derivatives of them created by users. So, an application can be constructed by defining a set of objects, creating agents to trigger behaviours on input or computation, and providing an appropriate set of visualisations.

2.4.5.1 Flexibility in Oval: Supporting Radical Tailorability

A major goal of Oval is the provision of "radical tailorability", a form of end-user programming. By tailorability, Oval's designers mean the ability for end-users to modify and customise applications, creating new systems out of old, and doing this without programming. By "radical", they mean that this tailorability can encompass wide changes; much wider than are traditionally associated with tailoring or customising behaviour.

As an investigation of the scope which Oval provides for variability, it was used in the experimental (re-)implementation of four pre-existing systems. The systems chosen were: The Coordinator, an early workflow system based on a structured conversation model; gIBIS, a graphical hypertext system supporting design argumentation; Lotus Notes, a collaborative database system for information sharing applications; and Information Lens, an email filtering system. The core Oval components were used to re-implement the functionality of these systems, requiring only minimal descent to the systems level, and so demonstrating the value of Oval's "building block" design.

There are some important issues which should be borne in mind when evaluating this claim. While the four systems vary in their domain detail, they're based on the same model of collaboration—asynchronous interaction over messaging systems. This is particularly interesting in the case of the Oval implementation of Notes, since true Notes employs a shared database model, which the Oval implementation does not attempt to replicate. Instead, the Oval model captures the behaviour of an *application* which might be implemented over Lotus Notes (Notes itself is an infrastructure for the creation of collaborative applications). Similarly, the developers explicitly point to "a shared 'live' database" as one feature of gIBIS which they have not attempted to reproduce. At the same time, given its intellectual history, it is not surprising to find that the functionality of Information Lens is easily recaptured in Oval[2].

So, while the reimplementation exercise impressively demonstrates that Oval subsumes the functionality of these disparate systems—and, even more critically, that it makes sufficient variability available *to the user* that they can be implemented without recourse to system modification—we must distinguish the variability of collaborative *applications* from the variability of *patterns of collaboration*. The facilities which Oval provides to users as the means to create and modify applications are largely single-user in nature; sharing and collaboration remain largely outside its immediate domain.

2.4.6 COLA

COLA (Cooperating Objects in Lightweight Activities) (Trevor et al., 1995) is a platform for supporting cooperative work which is derived more from a distributed systems perspective than an HCI perspective. That is, Trevor's primary concern is to extend distributed system understandings and techniques in light of experiences with distributed interfaces, rather than to extend the understandings and techniques of HCI with input from distributed systems. The principal component of the resultant system is a lightweight activity model which provides a context within which objects—perhaps those provided by a separate distributed object platform—can be shared.

The "lightweight" aspect of the system arises from the fact that the system carries with it only minimal semantic implications for the development of applications. So, for instance, the activity model does not introduce any requirements for the temporal ordering of tasks (as would a traditional workflow system, for example). Simi-

2. The developers are at understandable pains avoid confusion between Oval and Information Lens. "Oval is based strongly on the Object Lens system...", they say, "which should not be confused with the more limited Information Lens system" (Malone et al., 1995). However, the intellectual heritage is still strong; Object Lens is described elsewhere as "the 'second generation' of the Information Lens system" (Lai and Malone, 1988).

larly, issues of context are separated from the basic objects themselves through the use of context-particular *object adapters* (Trevor et al., 1994).

2.4.6.1 Flexibility in COLA: The Policy/Mechanism Split

COLA is designed to run in combination with a variety of distributed system infrastructures, as well as in support of a range of applications. In fact, one of its principal motivations is also one of mine—that traditional toolkit approaches embody semantics which restrict the application programmer's freedom. (The implications of this observation will be developed in more detail in Chapter 3.)

COLA's approach, however, differs from that which will be subsequently developed here. COLA's approach is to make a rigorous distinction between *mechanism* and *policy*. This is a common approach across a range of system designs. Mechanism refers to the support for particular actions or system behaviours, while policy refers to the set of decisions about how and when these behaviours will take effect. The X Toolkit, for example, claims to provide the *mechanisms* required to build a scroll bar (objects which track the mouse, constrained action, etc.) but does not provide a scroll bar because that would involve making *policy* decisions (such as which side of the window it should be on, and which mouse keys should operate it). Similarly, in COLA, policy is seen as an application issue—concerning how people are to work together—while the toolkit provides "policy-free mechanisms" which can be employed in a range of ways in support of different applications or application policies.

One aspect of this separation as an approach to toolkit design is that it is typically characterised by a decomposition of toolkit components. As a result, the components which are offered by the toolkit are small, atomic units which will then be combined by the application or client of the toolkit, since the nature of their composition will involve policy decisions. In other words, it is not simply that the client has *control over* policy, but rather that it *creates* policy in the way in which it uses the lower-level components and mechanisms. COLA provides an event mechanism which can be used to overcome some of the issues of coordination (that is, maintaining some correspondence between the states of different components) which this implies.

COLA's lightweight approach results in a considerably more flexible structure than a number of the other systems described here, although there are clearly places where some policy decisions must be made. For example, the event mechanism is synchronous, favouring particular styles of application; and object adaptors are managed by the binder, restricting the ways in which they can be fluidly interchanged within an activity. In general, however, the policy/mechanism split is clearly an effective means to support programmer/toolkit flexibility.

2.4.7 Other Systems

While my primary concern here is on flexibility in toolkits in particular, it is instructive to look at some applications and show how they have been designed for flexible use. These are both suggestive of areas worthy of particular investigation, and illustrative of the purposes to which infrastructure flexibility is put.

2.4.7.1 SEPIA

SEPIA (Streitz et al., 1992; Haake and Wilson, 1992), a collaborative hypertext system developed at GMD, supports a wide range of applications based on the model of collaborative authoring and interaction over hypermedia documents. The basic hypertext system is based on a persistent storage model which provides support for asynchronous collaboration. However, SEPIA is an explicitly collaborative system, and its interface supports a range of collaboration modes supporting different forms of interaction. SEPIA's three modes are *individual*, *loosely-coupled* and *tightly-coupled*. Individual work involves activity on a hypertext node which is currently the focus of attention for only one user. Loosely-coupled and tightly-coupled work both involve simultaneous activity over some node by a number of users, and differ in the ways in which the interface for any one user will reflect the activities of others. When two users are loosely coupled, each is free to scroll around in the document and work separately, although the interface makes them aware of each other's presence and uses an implicit locking mechanism to avoid conflicting changes. When they are tightly coupled, they share the same view of the document (that is, their interfaces are maintained in lock-step), and have a telepointer and audio connection to support their interaction. The transition between individual and loosely-coupled mode happens automatically when two users enter the same node; the further transition to tightly-coupled working can then be controlled by the users.

2.4.7.2 PREP

Early versions of PREP, a collaborative text editor from Carnegie-Mellon University, embodied a strongly asynchronous model of collaboration. More recent versions, however, have offered users parametric control over various aspects of its collaboration support. The basic PREP editor manipulates documents defined as one or more columns, which might be used for the main document text, planning and outline information, reviewers' comments, etc. (Neuwirth et al., 1990). Group interaction is supported in an asynchronous, "draft-passing" mode; however, it provides support for comparing and merging simultaneous copies (Neuwirth et al., 1992). Most recently, user control over aspects of shared interaction have been introduced to explore the utility of various parameters for collaborative authors (Neuwirth et al., 1994). These include control over: the grain size of update blocks; under what conditions updates should be sent to other participants; the speed at which network agents will exchange information; how one is informed of others' changes; and access rights over different parts of the document.

2.5 Genericity and Extensibility

This quick tour through the CSCW toolkits has revealed a host of mechanisms and approaches taken to a host of different design issues—interface linkage, data distribution, access control, consistency management, etc. In addition, the different systems use different mechanisms to allow their facilities to be tailored, controlled or adapted to the variety of applications which they might support. Rather than focussing on the toolkits themselves, this section is concerned with ways to characterise their flexibility.

If flexibility—the extent to which it can be applied in a range of circumstances—is a primary goal in the design of a toolkit, how is this flexibility embodied? For the purposes of this discussion, I will distinguish between two major routes towards flexibility—*genericity* and *extensibility*—and two aspects of toolkit design to which they are applied—*components* and *functions*. This second separation is often more conceptual than practical, but will prove to be a significant distinction in attempting to understand issues of flexibility.

Genericity and extensibility were defined in section 2.3. By "components", I refer to *objects embodied in a system*, and which the system offers for use by clients. Particular components might be expressed in the interface (such as the widgets of user interface toolkits) or might be realised only within the code itself (such as common programming abstractions such as sockets and streams).

By" functions", I refer to *aspects of the functionality of the system* (rather than the value-returning procedures of programming languages). These are the dynamic behaviours in which components engage. Most often, these are directly associated with specific components, and indeed they may be critical to their existence (a scroll-bar, after all, is only a scroll-bar by dint of its functionality). Some systems, however, separate functions into abstract objects in their own right. The Garnet system, for instance, provides abstract "interactor" objects which capture user interface behaviour and can be associated by the programmer with particular aspects of the interface, leading to a declarative style of interface programming (Myers, 1990). For example, rather than setting callbacks for keyboard events and then checking if the keys were pressed while the mouse was over a particular rectangle, the programmer can associate a "text-input-interactor" with the rectangle; the interactor is a concrete object which describes interactive behaviour independent of the graphical form which it might take in any instance. The function has been instantiated as an object. In the terminology in use here, this remains a "function" rather than a "component" because its role is more than structural. In a toolkit, the importance of this separation lies in the opportunity it provides to express the ways in which functions and components can be combined into larger units (like the "rectangle and text interactor" structure), and—most importantly—the means by which associations can be made between them.

Traditionally, *genericity* is the most significant aspect of toolkit flexibility. It captures the extent to which the facilities in the toolkit are "generic", or useful across some range of circumstances or applications. The use of genericity—the identification, generalisation and separation of facilities determined to be common across some number of applications—is perhaps most obviously represented by the design histories of early CSCW toolkits. Since they were often derived from earlier specific applications, their features are often generalised versions of mechanisms created for the applications. Genericity can (indeed, should) apply equally to the components and the functions of the toolkit. Both may be designed to be inherently generic (that is, applying naturally to some range of circumstances), or may use some parameterisation mechanism to achieve wider applicability.

Extensibility is a second route towards the provision of flexibility and increasing the reach of the toolkit. It refers to the opportunity to extend the range of facilities within the toolkit; that is, the extent to which the toolkit allows programmers to create new components and functions which can then be used within the toolkit itself alongside those which were already provided. This may be achieved in a variety of ways, and may be organised around the modification of existing facilities as well as the creation of new ones; but in any of these cases, the toolkit acts as a framework for the creation of new *toolkit* facilities, rather than the creation of new *applications*.

2.5.1 Genericity and Extensibility in CSCW Toolkits

These separations—between genericity and extensibility as techniques, and between components and functions as the objects of their action—create a framework within which we can begin to characterise aspects of flexibility in the toolkits discussed earlier.

It seems slightly ironic that, in pursuit of "radical" tailorability, Oval employs exclusively genericity, the more traditional of the two techniques. (The radical element of Oval's agenda, of course, is that the tailorability is offered to users, rather than programmers; it does not apply to the techniques by which tailorability is accomplished, or to the nature of tailorability itself.) Oval offers four highly general components—objects, views, agents and links—which can be combined in different ways, and are associated with behaviours.

Oval exploits parameterisation to relate its generic components to specific application needs. For some range of behaviours, the parameters are internally determined by the system (such as the types of views which are supported, and the events to which agents can respond). Other parameters are determined by user tailoring; each of the components, for instance, may feature some form of parameterisation based on the names of object slots, which are defined by users as part of the customisation process. This second form of parameterisation is key to the functioning of the system; it's the point of association between components and functions (functions are associated with named slots, which are embedded in objects). However, even though the "language of parameterisation" is, in some ways, being created by the user, this remains parameterisation (and hence a form of genericity) rather than extensibility. Neither new abstractions nor new forms of existing abstractions (such as a new view) can be introduced into the system.

COLA also employs genericity, but to somewhat different ends. While Oval relies on generic semantics—that is, the functions which it supports apply in a wide range of application situations—COLA's genericity is more structural, dealing with the sorts of contextual information which collaborative applications, modelled in terms of activities, might need at different points. The structured information which it makes available carries with it as little semantic information (or constraint) as possible, so that applications are free to use this information in whichever way is appropriate. In some sense, perhaps, the way in which the information can be used in new ways could be seen as a form of extensibility, except that there is nothing to extend. COLA explicitly strives to be policy-free; so application programmers are not *extending* the facilities to include new policies, but rather are *defining* new policies which reside entirely outside the toolkit. As a result, COLA is generic, rather than extensible. COLA itself provides no opportunity for explicit control over its own mechanisms.

MEAD, Rendezvous and Suite begin to open up more explicit control, focussing on particular points of leverage. In MEAD, this is the point of connection between the shared information space and different users' interfaces; in Rendezvous, it is the relationship between specific parts of a shared abstraction and the reflection of this state in users' interfaces. So there is a progression, from MEAD, through Rendezvous, to Suite in the balance of focus between shared interfaces and shared data.

Suite also achieves flexibility through its concentration on one highly general area, the fan-out between shared active variables and interaction variables (and, relatedly, the coupling between interaction variables). The relationship between the interaction variables of different dialogue managers reflects, primarily, the issues of multiple interface management and coordination; the relationship between an interaction variable and the shared active variable to which it corresponds reflects the split between interface and application. Again, parameterisation is the primary means by which a range of behaviours can be exhibited by a single mechanism (this time focussed on coupling rule sets, so functions rather than components). The Suite toolkit itself sets the terms in which coupling can be controlled; the programmer can manipulate these parameters (or, more accurately, the attributes from which specific parameters are derived by the matching algorithm). However, there is no route by which new parameters can be defined, or new mechanisms created out of existing ones.

Finally, GroupKit provides a set of generic components out of which interfaces might be constructed—essentially, a set of groupware widgets. On the other hand, it also begins to provide a route towards the customisation of behaviour through the open protocols mechanism. Unlike a number of the other approaches presented, open protocols begin to make the relationship between system components and behaviours explicit, so that new mechanisms can be incorporated into the framework created by the system's components and the protocols defined on them. The protocols themselves do not embody behaviours; rather, they embody the sequence of operations out of which behaviours may be created. The result is to allow new behaviours to be incorporated.

2.6 Problems in Toolkit Flexibility

The CSCW toolkits presented here vary considerably in their form of support for collaborative systems and, hence, in the forms of collaborative systems which they can support. However, they use similar sorts of techniques to provide flexible control over their collaboration support mechanisms.

Typically, the behaviour of these systems is modelled in terms of information flows and constraints between predefined elements—the ways in which they can be combined and coordinated, and how information and control passes between them. The transmission attributes in Suite and the sharing parameters in PREP, for instance, control the flows of information between private and shared locations. Flexibility is achieved through control over the information flows—their granularity, frequency and distribution. As techniques for incorporating flexibility into a toolkit, genericity and parameterisation focus on the *range of ways* in which information flow can be regulated, and hence the range of behaviours which can be managed. Extensibility, on the other hand, focuses on the ability to create *new forms* of information available to programmer control, *new components* between which information flows can be defined, or *new control mechanisms* which can be used to regulate the information flows, within the framework which the basic (non-extended) system sets up. However, these approaches leave two particular problems unresolved.

2.6.1 Structure and Semantics

The first is that, since they concentrate on the *structural* aspects of the design—ways of selecting components and plugging them together—they typically make it hard to give the programmer control over the *semantics* of information flow. Semantics, typically, will have to reside in existing components, so that manipulation of system semantics arises *out of* the manipulation of system structure. The basic data model which underlies the structures manipulated is typically unavailable for tailoring. One way to think about this is in terms of the "richness" of the language of tailorability (Bentley and Dourish, 1995). In the case of parameterised control, for example, the "language" in which programmers express their requirements is restricted to the names of predefined modes of operation. Toolkits using this sort of approach (to a greater or lesser degree) do not offer languages in which new semantic behaviours can be expressed, giving them control only over the structural aspects of their application support (how the pre-existing components are to be plugged together).

2.6.2 Extending versus Revising

The second, related problem is that even the more flexible approach, extensibility, typically provides only for the *incorporation* of new mechanisms, rather than the *modification* of existing ones.

In part, this is a question of granularity. With an appropriately fine-grained control, and a framework which allowed behavioural delegation within larger structures, we *could* consider modifying or replacing the behaviour of particular large-scale structures, by extending just those parts of the structure which we might want to change (and hence affecting their behaviour). However, such fine-grained control is rarely available. In general, the extension model is intended to allow the inclusion of new structures, rather than the modification of existing ones.

One approach might simply be to say that we could "modify existing structures" by introducing new ones which act in similar ways, but which include the new behaviours we need. Unfortunately, that "reimplementation" approach is not sufficient. It still gives no control over the *internal* relationships between system components. For example, if a system provided shared text boxes for editing, but used an interaction model unsuited to some particular situation, it's not enough to be able to create a new form of text interactor which behaves more appropriately, if the programmer has no means to apply this to any other *internal* aspects of the system which use those same text units (perhaps for error messages or debugging, or some other internally-originated behaviour). Instead, what's required is the ability to "reach inside" and introduce modifications to the system components, essentially "in place". Again, internal aspects of the system are unavailable for examination and manipulation. Extensibility is limited to those areas of design which are "outside" the toolkit itself.

2.7 Summary

To explore the questions of flexibility which were raised in Chapter 1, this chapter has examined how existing CSCW toolkits have incorporated flexibility into their designs. The primary concern addressed here has been that of flexibility as a relative, rather than an absolute, property. So, I have focussed on questions such as: What range of applications can a toolkit support? What control are programmers offered over toolkit behaviours? How can new behaviours and application needs be expressed in this toolkit?

As has been shown, different toolkits have taken very different approaches to these sorts of questions. I have used the general notions of "genericity" and "extensibility" as a way of discussing the techniques which they demonstrate. The examination presented here has shown that these CSCW toolkits typically embody restricted notions of flexibility, which, while opening up the design space to an extent, give programmers only limited control over the ways in which the toolkit mechanisms will support their applications (and hence, limiting the range of applications which each toolkit can support). In making some set of structures, behaviours and mechanisms available to application programmers, these toolkits also make a set of commitments to particular styles of application and interaction.

These issues are by no means unique to CSCW design, although, as argued in Chapter 1, there are some properties of CSCW systems which bring them more readily to the fore. In many other areas of system design, from HCI to network protocol implementation, from programming language design to operating system architectures, the same issues of flexibility, openness, and the relationship between internal structure and external requirements have arisen. In the next chapter, I will draw on ongoing work in the use of computational reflection and Open Implementations to analyse these problems and present a framework for solutions.

Chapter 3: Computational Reflection and Open Implementation

3.1 Flexibility

In Chapter 2, I showed how a number of CSCW systems and toolkits tackle the problems of generality and flexibility. In the range of components and functions which the toolkits provide (and in the range of facilities they support for combining and composing them), the designers of toolkits and similar application frameworks attempt to encompass a variety of application styles and requirements. Of course, this is not unique to CSCW design, but rather is a motivating factor behind any toolkit, framework, library or language. The goal is to provide reusable components which can be combined again and again to support the needs of some range of applications. The flexibility which the toolkit embodies determines its range of applicability. In this chapter, I will relate the flexibility issues encountered in Chapter 2 to similar issues in other domains. I will draw on recent work on a novel architectural approach—*Open Implementation*—to discuss this problem in terms of the use of "abstraction" in system design, and to propose a solution. Along with the OI approach, I will also introduce a design principle on which it is based (*computational reflection*) and a programming technique used to realise it (the *metaobject protocol*).

Computer systems supporting collaborative work need to be flexible in a number of ways. First, they need to support the variety of working styles which might be adopted by the groups using them. The choice of working style might be affected by the particular individuals in the group, by wider organisational factors, or by the nature of the task which is being performed. I will refer to this as *static* flexibility, since it captures a form of variability between sessions, but not necessarily within them.

In contrast to this, *dynamic* flexibility refers to the requirement that the system be able to match and adapt to changes in the group's interactions over time. In typical settings, groups move fluidly between various styles of work. In a typical meeting, we can observe continual movement between, for instance, formal turn-taking, brainstorming and private activity. Similarly, the constitution of the group itself is subject to change in the course of collaboration. For instance, Beck and Bellotti (1993) report on a questionnaire study of collaborative authoring in which 23% of respondents stated that the membership of their groups changed after the writing stage had begun.

Third is *implementational* flexibility—the need to be able to respond to a variety of implementation environments, which may themselves change over time. For instance, the technical requirements of supporting collaborative activity over local-area and wide-area networks, with different latencies and response characteristics, are quite different. Not only might a single tool be used in these various circumstances, but the circumstances of use, and therefore the technical requirements, might change *in the course of a single session*. When various group members distributed over a wide area leave and join a session, the effective topology of the network changes, and this may require adaptation in the system's support for ongoing collaborative activity.

3.1.1 Tackling Flexibility

The systems outlined in Chapter 2 tackle these problems in various ways. Some, such as Suite, provide a range of options within the toolkit so that applications can vary in their selection of support strategies. Others, such as SEPIA, provide various run-time "modes" which embody different policies to match user needs. However, these various toolkits still suffer from serious restrictions, and are applicable only to a range of particular applications. The structures which the toolkits embody are not flexible enough to support the full range of collaborative applications which might be needed, nor the range of ways in which a particular application might be used. Any toolkit design embodies prior commitments to styles of application design, and hence to styles of working activity. The focus here, then, is on ways in which this comes about, and hence what can be done about it.

From the perspective of static flexibility, the typical approach is to limit the applicability of the toolkit to some particular range of applications or a particular style of interaction. From these sorts of commitments, we encounter the traditional distinctions between application and toolkit types: synchronous versus asynchronous, collaboration-aware versus collaboration-transparent, tightly-coupled versus loosely-coupled, and so on. However, as observational studies have highlighted, these are distinctions of use, rather than distinctions of design; and so the toolkit is not the appropriate place to make such decisions.

From the perspective of dynamic flexibility, they tend to provide little support for changes in group membership or working style since the structures and abstractions which they embody are fixed at within the toolkit or application. Where variability is available, it typically takes the form of interface tailoring, or tailoring of other "surface" features of the application (Bentley and Dourish, 1995).

The issues which implementational flexibility addresses are also, typically, hidden from the view of the applications developer using these toolkits. Abstractions of shared objects, workspaces, telepointers, etc., are too high-level to include or influence questions of, say, network topology.

3.1.2 Flexibility and Abstraction

Toolkits offer a range of abstractions which can be arranged as needed for particular applications or situations. The abstractions might include *components* (structural units which can be incorporated into programs or interfaces), *mechanisms* (procedures managing activities over these components) and *composition strategies* (ways of combining components and mechanisms to form larger units). Tailorability facilities might allow us to *parameterise* these abstractions, in order to capture some range of potential needs, by selecting from a number of pre-defined settings.

However, parameterisation is only half the problem. The abstractions themselves remain fixed, impenetrable and static, in contrast with the flexible and fluid nature of collaborative working activity. It is this mismatch between fixedness and fluidity which results in the problems of flexibility and appropriateness which have been noted; and this problem, the way that computational renditions of conceptual abstractions become static and lifeless, has been observed by a number of researchers studying CSCW (e.g. Robinson, 1993; Suchman, 1994; Button and Dourish, 1996).

Abstraction and formalisation are at the very heart of computer science; we can't build systems without abstractions with which to build them. Our strategy here is not to throw away abstraction, but to re-examine it and reconsider how we wish to use it in system design.

3.2 Abstraction

Abstraction is one of the fundamental tools of computer science and system design. It is the means by which we can break down large problems into small ones and, conversely, can combine small solutions to create large systems. It allows us to isolate one part of a system from another and consider the two separately. It is the key to analysis, modularity and reuse; and it is also, potentially, the source of a range of problems throughout systems design practice.

The traditional form of abstraction in systems design relies on three principal components—*black boxes*, *clients*, and the *abstraction barrier*, as illustrated in figure 3.1. The black box implements some abstraction, which is offered to clients at an abstraction barrier. The abstraction barrier is a point of separation between the client and the implementation; the concepts, terms and structures in which the abstraction is phrased at the abstraction barrier are the only ones which clients can use to manipulate and control the abstraction. In programming languages or systems, the abstraction barrier is often presented as an "interface". However, I use the term "abstraction" here because the same abstraction principles apply to other situations where the "interface" is less clearly articulated (such as when manipulating graphical objects in an interactive user interface, or performing memory references in a virtual memory system). Similarly, the term "barrier" refers to the way in which this serves to hide aspects of the implementation from the client. Behind an abstraction barrier, the internal details of an implementation are not revealed.

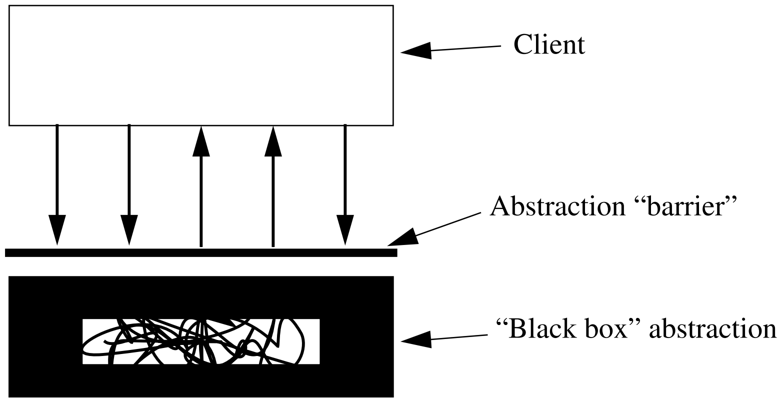


FIGURE 3.1: A traditional black box abstration locks implementation details away behind an abstraction barrier.

There are two important features of abstraction being employed here. First, *separation* divorces the use of the abstraction from the details of its implementation, allowing a client to use an abstraction without understanding all the details which lie behind it. Second, *generalisation* divorces the abstraction itself from any particular implementation, so that the implementation may be changed without changing the abstraction (and hence without forcing changes in clients which code to the abstraction). By using separation and generalisation in this way, systems can be modularised and their components reused. This model of abstraction runs through system design. Even the most basic elements of software systems, such as programming languages and instruction sets, are built upon it.

This notion of abstraction, as used in software development, is derived from the mathematical use of abstraction. However, software entities differ from mathematical entities. In software, the abstractions we offer are not truly "abstract". Instead, they are the visible aspects of underlying implementations, and, clearly, the implementation constrains the way in which the abstraction behaves. While any (correct) implementation of the abstraction will agree with the abstraction's specification, and hence operate in the "same" way, different implementation strategies will result in different performance characteristics, memory usage patterns, and so on. I will use the term "behaviour" to refer to the *manifestation* of these properties—that is, not just the semantics of the implementation, but also the details of its acceptable patterns of operation and performance.

Lists and arrays, for instance, are different implementations of a Collection abstraction. Although they might share an interface, they exhibit different performance characteristics (different behaviour). These particular examples happen to be so endemic to the problems we solve that we think of them as different abstractions; but the differences in their behaviour are *not expressed* in the abstraction. This variability is something we often depend upon in implementation; for instance, caching in memory systems and memoisation in programming language implementation are both techniques which *change* performance characteristics while maintaining the *original* abstraction.

This same variability, though, can also introduce significant problems. To illustrate these problems, consider a situation opposite to that described above—rather than one client and multiple possible implementations, consider a single implementation and multiple clients. This is a common arrangement—an operating system supporting a text editor, an email reader and a database, or a window system supporting a word processor, a spreadsheet and a game. The clients all make use of the same implementation, accessed through the same abstraction, in service of whatever functionality they themselves provide to their own clients. However, the clients have different needs and requirements. Imagine the problems which would arise if there were only a single implementation of a Collection class. Some clients would need array-style behaviour; some would need list-style behaviour; some might want hash-table behaviour. In fact, the more clients there are, the more likely it is that there are going to be conflicts in their requirements for the behaviour of the implementation. However, as observed above, the abstraction does not express the difference in behaviour. In fact, those aspects of the implementation which would cause a programmer to choose one over another are systematically hidden by the abstraction barrier.

In this case, it's not the particular abstraction itself which is at fault. The simple specification of the abstraction (the Collection abstraction, defined in terms of addition or removal of elements, lookup and searching) can be used effectively by all the clients. The problem is, first, in the fact that the "abstraction" isn't abstract at all, but is the interface to an implementation; and second, in the way in which a single implementation must serve multiple purposes. But this isn't some unusual special case; it's simply everyday reuse.

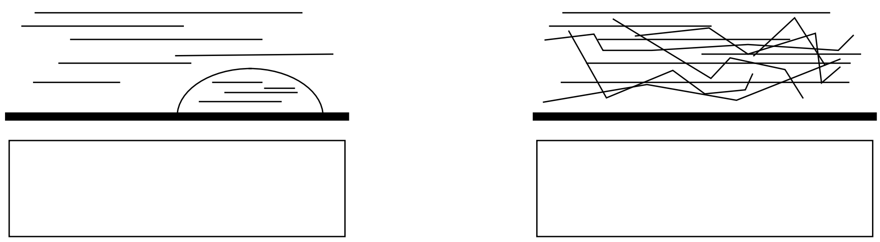
3.2.1 Mapping Dilemmas

The root of these problems can be explained in terms of mapping decisions, mapping conflicts and mapping dilemmas (Kiczales and Paepcke, forthcoming)[1]. A mapping *decision* occurs at any point where the implementor of an abstraction has to choose between a number of possible strategies for implementing some internal mechanism (that is, for mapping some aspect of the high-level abstraction onto the lower-level structures to which the implementation has access). A mapping *conflict* occurs when some the implementor makes the decision one way, but the needs of a client would be better met if the decision had been made another way. A mapping *dilemma* occurs when two clients of the same implementation require different mapping choices; whatever choice is made, a mapping conflict results.

Mapping decisions are orthogonal to the abstraction. They arise not from the structure of the abstraction itself, but from the way in which it is implemented. Thus, since mapping decisions are not part of the abstraction, they are not visible through the abstraction barrier. While it's clear that the incidence of mapping conflicts can be exacerbated by poor mapping decisions, it's important to recognise that the problems of mapping dilemmas in general are not the result of particular implementations or abstractions, but are *inherent in the model of abstrac-*

1. Although retaining Kiczales and Paepcke's terminology, I have redefined the terms somewhat. My breakdown is true to the spirit of their's, but not the letter. My terminology refers to points in the design process, while Kiczales and Paepcke's refers more to the outcome.

tion itself. As such, software developers encounter them every day, and employ a number of strategies to deal with them.



(a) (b)

FIGURE 3.2: Two common solution strategies—"hematomas" and "coding between the lines".

3.2.1.1 Hematomas

Two common strategies are identified by Kiczales (1992) as "code hematomas" and "coding between the lines". The hematoma strategy involves the duplication of some piece of abstraction functionality *above* the abstraction barrier, embodying a different set of mapping decisions (see figure 3.2a).

Consider the example (used by Kiczales, derived from Rao (1991)) of a spreadsheet application written over a window system. The window system offers a set of abstractions such as window, font, menu, and so on. At the level of offered abstractions, the cells of the spreadsheet (rectangular areas of screen space which can display text and respond to mouse and keyboard input) seem very like the windows of the window system (rectangular areas of screen space which can display text and respond to mouse and keyboard input). So, since the abstractions match, why not create one million windows to act as the cells for a 1000 by 1000 cell spreadsheet?

However, in conventional window systems, an attempt to use windows as cells are unlikely to succeed. The reasons for this sort of failure—either an explicit breakdown or a performance failure—are implementational, arising from such issues as a limit on the number of windows, structure traversal, memory usage and so forth. The implementational details (how mapping decisions concerning windows have been resolved) have a considerable effect on the ways in which the "window" abstraction can be used. They are not part of the abstraction itself, though, and so the abstraction itself offers no information about why this use of it fails (and similarly, it offers no prior warning). As system designers, familiar with the ways in which window systems work, we may recognise the resultant behaviours and the mapping dilemmas which cause them; but this is information extrinsic to the abstraction itself. We may intuit it, but we cannot perceive it by looking at the offered abstraction.

The hematoma strategy involves reimplementing some amount of the window system functionality above the window system abstraction, and coding the application to *that* version of the abstraction. In this case, the application developer might write a library of routines to manipulate spreadsheet cells within a single window, making different mapping decisions to exploit known properties of the application (e.g. that cells are of a uniform fixed size, that they are tiled, etc.). This library is the "hematoma". It is quite likely that the hematoma will reproduce a lot of code and functionality which exists in the implementation below; there may only be one or two mapping decisions to be resolved in a new way to suit this application. Typically, though, considerably more code (or functionality) will have to be reproduced than changed.

3.2.1.2 Coding Between the Lines

The second strategy, "coding between the lines", is another common solution to abstraction problems. It involves writing the client code in particular ways so as to avoid expected problems with the implementation underlying the abstraction. This strategy relies on the client programmer having some understanding of the mapping decisions and mapping choices made by the implementor of the abstraction. Such an understanding might come from direct experience, or simply from known common practice; but, again, the information is not offered in the abstraction itself. At the same time, since the code is being fashioned to fit a set of unexpressed lower-level concerns, the resultant structures are often obscure, convoluted and non-intuitive in terms of the desired behaviour at the module's level of abstraction.

A frequently-occurring, familiar example of this strategy involves the artful use of virtual memory. The abstraction offered by a canonical virtual memory implementation is simple enough, and very familiar to programmers. However, the breakdown of this abstraction and the resultant performance problems are equally familiar. It is seen in a range of cases, such as database or graphics systems, which exhibit poor locality of reference. In these cases, the common solution strategy is to find patterns of memory allocation and traversal which improve locality and hence perform well over typical (and, again, well-understood) virtual memory implementations. Unfortunately one consequence of this common strategy is to make the application code more complex to understand. There are now two elements mixed in together (the application code, and the stylised strategy to make it work efficiently). This "obscuring" problem is inherent in the "coding between the lines" strategy since there is no clean separation the two aspects of the implementation. As well as making the code more difficult to understand and maintain, it also makes it considerably more difficult to port, since the two aspects of the system's design—how the system itself operates, and how it is mapped onto underlying structures—are mixed up together.

3.2.2 Gaining Control over Abstractions

For their part, the implementors of the abstractions can also use a number of strategies to lessen these problems for their users (the implementors of the clients). As systems have become larger and more complex, and as hardware has improved and exposed more performance problems in software, strategies for overcoming these abstraction problems have become more common. One typical solution is to offer a number of different implementations to choose from (compiler optimisation strategies often operate this way). Another is to provide switches which allow application to select particular strategies. For instance, the UNIX system call *vadvise* allows application programmers to specify the style of memory access particular memory regions will experience, so that an effective paging strategy can be employed.

Recently, more radical solutions have been adopted, in various areas of system design. For example, a number of areas of operating systems design have been a focus of attention. The Mach operating system provides facilities for virtual memory behaviour to be controlled directly by application programs—the "external pager" (Rashid et al., 1987). Scheduler activations (Anderson et al., 1992) allow application control over thread facilities, addressing the design trade-offs involved in locating thread information and control in user space or kernel space. Cao et al. (1994) have explored application control over filesystem cache policies. More generally, flexible object-oriented operating systems such as Spring (Hamilton and Kougiouris, 1993) have allowed applications (or user-space code) a great deal of control over the implementation details of "lower-level" operating system abstractions (Khalidi and Nelson, 1993; Nelson et al., 1993).

Similar forces have resulted in the provision of greater client-specific flexibility in network communication protocols; the *x*-kernel supports composable micro-protocols which applications combine to produce lean protocol stacks incorporating just the particular features they need at a given time (O'Malley and Peterson, 1992; Bhatti and Schlichting, 1995). Similarly, responding to the call of Clark and Tennenhouse (1992), other groups have investigated ways in which application needs can be used to control protocol implementation (e.g. Braun and Diot, 1995).

3.3 Open Implementation

The sudden appearance of all these techniques in different areas suggests some fundamental, common problem. *Open Implementation* is a relatively new approach to tackling the issues which they raise, looking directly at the way they originate in the differences between mathematical and software abstractions. Originally developed in the area of programming language design and implementation, it has come to be applied to a number of other areas of system design, and provides a route to solving the problems of expression and flexibility raised in Chapter 2. Essentially, Open Implementation is an approach to computational architecture which "opens up" abstractions and provides clients with principled access to examine and control aspects of the implementation behind the abstraction barrier.

Probably the most important foundational principle behind the Open Implementation approach is *computational reflection* (Smith, 1982). The reflection principle states that a system can embody a *causally-connected representation* of its own behaviour, amenable to examination and change from within the system itself. The causal connection sets up a two-way relationship between the representation and the behaviour it describes; the representation is continually maintained in correspondence with the system's behaviour, and the behaviour itself is controlled through manipulation of the representation. So, a reflective system can examine the model (*introspection*) to reason about its own behaviour; and it can make changes to the model (*intercession*) to effect changes in its behaviour. The causally-connected self-representation creates a link between two "levels" of processing—the "base" level, which is the traditional domain of computation for any given system, and the "meta" level where the domain of computation is the system itself. Reflective models are clearly a route towards Open Implementations; however, reflection was originally developed outside this framework.

3.3.1 3-Lisp and the Infinite Tower

The first system to be designed along these lines was 3-Lisp (Smith, 1984), a semantically rationalised dialect of Lisp. The reflective link in 3-Lisp was achieved by defining, as part of the language itself, the means for programs to gain access to the data structures of their interpreters. Since the 3-Lisp interpreter is meta-circularly defined in terms of 3-Lisp, the 3-Lisp interpreter is, itself, a 3-Lisp program, and so must have access to an interpreter of its own, and so on. This sets up an "infinite tower of reflective processors", each interpreting the program at the level below (generally another 3-Lisp interpreter, except at the base of the tower) and, hence, each potentially accessible through the reflective facilities in the program at the level below it.

The infinite recursion of this conceptual model was discharged in the implementation of the *level-shifting interpreter* (des Rivières and Smith, 1984) which extends traditional interpretation mechanisms with the ability to create the data structures of interpreting programs lazily, if needed. Base-level (traditional) processing requires just standard interpretation structures, allowing a 3-Lisp program for some application domain to compute over representations of base-level concepts, so that, for example, a statistics program might compute over representations of data points, distributions and standard deviations, while a graphics program might compute over graphical points, polygons and filling patterns. Meta-level processing, or computation over representations of the program itself and its execution, might involve questions like "where was this procedure invoked?", or "how is this variable bound?", and employ's 3-Lisp reflective mechanisms to provide access to the lazily-created representation of interpreter structures.

While 3-Lisp provided an early demonstration of the concepts behind reflection, and spawned a number of other systems which embodied many of the same principles (e.g. 3-KRS (Maes, 1987), a knowledge representation language, and Brown (Friedman and Wand, 1984), a reflective extension of Scheme), reflection remained largely an area of fairly abstract research investigation. It was not until they were applied in the design of the Common Lisp Object System (CLOS) that the ideas behind it began to find their way into wider programming practice.

3.3.2 CLOS and the Metaobject Protocol

The Common Lisp Object System (CLOS) was an extension to the original Common Lisp design (Steele, 1984) and is a part of the later ANSI Common Lisp standard, defining an object model which extends Lisp's traditional functional style. At the base level, it is a powerful object programming language, based on generic functions, multi-methods and powerful method combination facilities[2]. In addition, CLOS is a reflective language; it provides mechanisms by which programmers can reach into the implementation, examining internal structures and

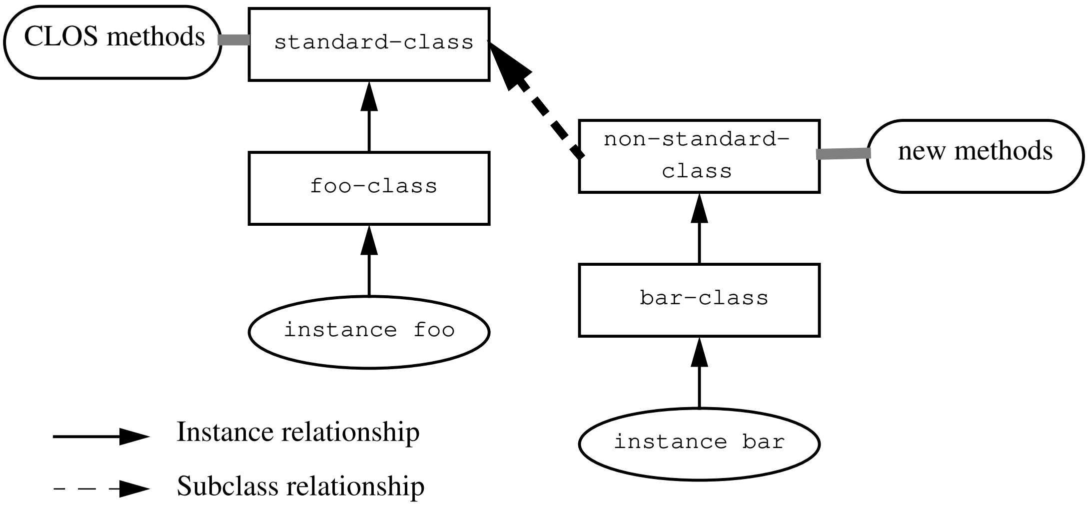


FIGURE 3.3: Instance `bar` behaves differently from instance `foo` because aspects of the class behaviour have been changed by the programmer through the metaclass of `bar-class`.

making changes to the object system's behaviour. By reaching into the language's implementation, programmers can tailor its elements to particular application domains or hardware platforms, or create language-level extensions.

The means by which this reflective access is made available differs from the "tower of reflective processors" used in 3-Lisp. In CLOS, the reflective mechanism is the *metaobject protocol* (MOP) (Kiczales et al., 1991). Essentially, the MOP is an explicit self-representation cast in object-oriented terms. The MOP defines the structures of the object system implementation as *metaobjects*, which represent the objects out of which CLOS itself is built—representations of classes, objects, generic functions, methods and so on. Generic function calling sequences, defined for these metaobjects, capture the behaviours in which the object system engages in the process of its execution. So, internal CLOS actions—such as allocating space for an object's slots, computing the class precedence list, or looking up slot values—are defined in terms of method invocation over metaobjects. The metaobject representation is accessible from within CLOS programs, just as the representations of application program structures would be. So, CLOS has an internal representation defined in CLOS, and accessible throughout the system in terms of CLOS structures and functionality.

Just as the meta-level representation in CLOS is an object-oriented representation, so the means of effecting modifications proceeds through object-oriented mechanisms such as subclassing and specialisation. Metaclasses—classes whose instances are themselves classes—are classes, and so are subject to the same form of manipulations as normal classes. A programmer might make a subclass of the standard metaclass, `standard-class`; call it `non-standard-class` (figure 3.3). By default, `non-standard-class` inherits all the behaviours of its parent, `standard-class`, and so instances of `non-standard-class` will behave as normal classes. However, the programmer can now define new methods which apply to instances of `non-standard-class` (i.e. they

2. Generic functions (Bobrow et al., 1986) are a means to combine the functional style of Lisp with the message-passing style of object-oriented programming. Roughly, generic functions play the same role as messages in traditional OO languages, naming behaviours for invocation on particular objects; however, they exist as functional objects, and, syntactically, are "applied" to objects rather than "sent" to them. Multi-methods are methods which dispatch on more than one object; any or all of the arguments to a generic function can be used to find a method to apply, rather than a single, distinguished object (as in, say, Smalltalk). Finally, method combination allows a number of different methods to be associated with a generic function invocation, and defines the ways in which methods don't simply override each other, but may, in particular circumstances, be combined and dispatched in sequence in response to a single generic function call.

are "specialised on" `non-standard-class`). Just as methods defined on a subclass of "rectangle" can produce rectangles which respond to the same protocol but behave differently, methods defined on a subclass of `standard-class` can produce classes which respond to the same protocol but behave differently. The methods defined on `standard-class` are those which implement the behaviour of the object system. So, if new methods are defined to override them, then the programmer can *change the way in the object system works*, for classes which are instances of `non-standard-class`. When the internal generic functions are called, CLOS will execute the programmer's new methods (rather than the predefined code) for any relevant metaobjects. Not to do so would violate the definition of CLOS as a standard CLOS program. In this example, when classes are created as instances of `non-standard-class`, rather than `standard-class` (that is, `non-standard-class` is their metaclass), they will behave with the programmer-supplied functionality for those methods which have been changed, while, for other methods, they will inherit the default behaviour associated with their metaclass's superclass, `standard-class`.

The metaobject protocol in CLOS, then, illustrates the integration of reflection and object-oriented programming. It was also the first reflective architecture to be widely developed and distributed. A number of implementations exist as parts of commercial products. It has been used not only to achieve compatibility with older object-oriented extensions to Lisp, and for improved performance on both stock hardware and special-purpose Lisp machines (two original motivations for its development), but also to provide semantic extensions to the language, such as in PCLOS, a persistent implementation of CLOS (Paepcke, 1988), and in ALV, which incorporates a constraint mechanism for CLOS object slots (Hill, 1993).

3.3.3 The Metaobject Protocol in General

More recently, the metaobject protocol approach has been exploited to incorporate a notion of language-backed deep customisation into systems in other domains. Examples include Silica (Rao, 1991; 1993), a reflective window system, and Anibus (Rodriguez, 1992), which uses reflection to tailor parallelisation strategies in parallel programming. Rao refers to this sort of reflection as *implementational reflection*—the support for programs to "become involved in" the implementation of the services on which they depend. Implementational reflection does not employ the meta-circularity[3] of previous reflective systems such as 3-Lisp or CLOS, but retains the essential element of reflection—explicit self-representation. It's through this representation that the internals of system behaviour can be revealed in a principled way, making them open to examination and manipulation. In turn, this mechanism provides for a "downward" information flow across abstraction barriers, so that application needs and requirements can be expressed to the lower levels of the system, and exploited for customised configuration and behaviour.

This Open Implementation approach clearly shares many of the same goals as more traditional separation of mechanism and policy (as discussed when COLA was outlined in Chapter 2). However, there are some critical differences. The split between mechanism and policy normally operates by, first, *decomposing* mechanisms (to remove policy decisions encoded within them), and, second, migrating policy *above* the abstraction barrier. Applications are responsible not only for defining policy, but also for implementing it; there is no policy below the abstraction barrier. In the metaobject protocol approach, however, "policy" resides in the code below the abstraction barrier, but under the control of the application.

There are three consequences. First, the base-level abstraction remains the level at which client and implementation interact. This abstraction is, typically, closer to the application's level than the raw implementation, and so critical facets of abstraction (such as information hiding, modularity and decomposition) are retained. MOP-based systems offer default behaviours, after all; the goal is to retain and enhance the abstraction, not to do away with it. Second, implementation tailoring (modification through the metalevel interface) takes effect at the

3. Meta-circularity is a form of implementational recursion. It refers to the (conceptual, at least) implementation of a programming language in itself. Lisp is traditionally implemented by a meta-circular interpreter (Steele and Sussman, 1978), which is the starting point for 3-Lisp-in-3-Lisp and CLOS-in-CLOS.

implementation level, rather than the client level, optionally making modifications available to other clients. In other words, the separation of base and meta levels is akin to a separation of concerns in the implementation design—the two concerns being the *what* and *how* of the implementation—while separation of mechanism and policy simply removes the policy elements, pushing them up to become the concern of each different applications. Third, by keeping policy within the implementation (but allowing control from outside), MOP systems can provide simpler modification interfaces. The self-representation which MOP systems present is the *inherent* (not absolute) structure of the system. So, this approach allows the implementation to retain a certain amount of control over elements which should not be revealed, and turns control over policy into a negotiation between the two system components. This element of partial revelation and partial control is critical in systems where control should not be ceded absolutely to individual clients (such as cases where the abstraction manages resources for the use of multiple client—such as physical memory control in a virtual memory system).

3.4 Reflection, Open Implementation and Abstraction

The metaobject protocol is a technique for implementing computational reflection, which in turn can be used to achieve Open Implementations (Kiczales, 1996)—systems which extend their traditional (base-level) abstraction for accessing functionality with a new (meta-level) abstraction for controlling aspects of the implementation. One important reason to do this is precisely the sorts of problems which were outlined earlier in this chapter, and which echo the problems of CSCW toolkits raised in the previous chapter—that is, to provide flexibility in system infrastructure so that a wide range of application functionality can be supported. A CSCW toolkit based on the Open Implementation approach, then, offers a route towards much more radical flexibility than is achieved by traditional techniques.

To date, only a handful of Open Implementations (largely based on metaobject protocols) have been developed, and there has been considerable variety in their domains and goals. As a consequence, it is difficult to make generalisations about the process of metalevel design. Based on some particular cases and insights, though, we can at least list some desiderata (which is how we will set out on the design of Prospero in the next chapter). First, a diversion into reflective self-representations is useful to set some context.

When thinking about self-representations in reflective systems, it's important to bear in mind that they are just that—*representations*. The causal connection, and in particular the computational effectiveness which it supports, can lead to a confusion between the representation and the mechanism which is represented. Similarly, the metaphoric relationship drawn between reflective systems and mechanical ones—a story in which mechanism is "exposed to view", and in which users can "reach in" to effect changes—can also contribute to this confusion. When thinking of the design of a MOP-based system, there are at least two important aspects of the representation *qua representation* to be borne in mind; *maintenance* and *partiality*.

Maintenance refers to the way in which the representation is actively maintained by the reflective system. Elements of the representation are created as needed, and/or maintained in correspondence with elements of the system itself, rather than being continually present. The lazily-created reflective interpreter layers of the 3-Lisp implementation illustrate this. While the 3-Lisp model guarantees that the representation is always available when requested, it may not actually exist *until* its requested. At the point where it is created, the elements of the representation (or rather, of any instance of the representation) are a rationalisation of the system's state according to an idealised model. So, when designing the model, and considering the terms in which the meta-level interface is cast, it's important to remember the distinction between "exposed structure" and the *actual* implementation mechanisms; a distinction that the system maintains actively.

A second critical property, which follows from the maintenance of the representation, is its inherent *partiality*. The purpose of the representation is not to provide an absolute, decontextualised or impartial description of the system's activity. Rather, the representation describes some aspects of the system's behaviour for the purposes of some domain of expected behaviour. It reveals certain aspects of behaviour, and hides others; similarly, it supports certain forms of tailoring and modification, but not others. The representation is a *designed* artifact;

and, in line with perceived needs and expectations, the designer sets the bounds on the flexibility which it embodies.

3.4.1 CLOS and Telos

The representation, then, is guided more by expectations of use than it is by the structure of the implementation. This is particularly well illustrated by the comparison between two MOPs for ostensibly similar domains—CLOS and Telos. As discussed above, CLOS is the object system for Common Lisp; Telos, similarly, is the object system for EuLisp (Padget et al., 1993). Like CLOS, Telos incorporates a metaobject protocol by which programmers can make changes to the internal behaviour of the object system. The design histories and overall goals of EuLisp and Common Lisp (and hence, Telos and CLOS) are quite different—in particular, EuLisp and Telos were designed together, whereas CLOS was developed after the original Common Lisp design was complete—and to an extent the differences in the MOPs reflect this. However, a number of important distinctions originate in the different set of design criteria at work, and the designers' expectations about desired usage patterns (Bretthauer et al., 1993).

In some areas, the differences highlight the Telos developers' concerns with run-time efficiency, and so the Telos design moves as much computation as possible to load-time or define-time. For example, to introduce changes to the way in which slots are read, a CLOS MOP programmer would define new methods on the generic function `slot-value-using-class`, running specialised code when the slot is read. Using the Telos MOP, by contrast, this would be achieved by changing the class definition code to install a different slot-reading function; and so the burden of computation (and optimisation) is moved from run-time to define-time[4]. At the same time, Telos can be seen as making more aggressive use of the MOP in the language definition itself; Telos has a layered structure with different levels of functionality, and uses the MOP to achieve semantic extensions as successive layers emerge. For instance, Telos supports only single inheritance at level-1, but multiple inheritance at level-2; this extension is achieved through meta-programming.

3.4.2 Compile-Time and Run-time MOPs

The Telos developers are by no means the only group to have investigated the relationships between MOP design, areas of MOP control, and performance. PCL, the public-domain implementation of CLOS and the CLOS MOP, used aggressive compilation and optimisation techniques to "win back" the performance which was lost through the run-time overhead implied by the metaobject protocol (Kiczales and Rodriguez, 1990). The techniques used in PCL preserve the illusion of late-binding and run-time use of the MOP structure during the object system's operation, while moving as much processing as possible to define-time.

More recently, various projects have investigated the use of compile-time MOPs. These are metaobject protocols which are invoked (and therefore incur processing overhead) only during the compilation phase of program development, and therefore will not interfere with run-time performance. Anibus (Rodriguez, 1992) controlled the distribution and communication patterns of programs for parallel machines, but the strategies which it manipulated were compiled into the parallel programs; there was no run-time interference by the metalevel facilities. While the first version of Open C++ (Chiba and Masuda, 1993) incurred a high overhead by using a metaobject protocol to control virtual function invocation at run-time, Open C++ version 2 (Chiba, 1995) is entirely a compile-time protocol. The modifications which programmers supply to change details of the language implementation are aimed, not at how the run-time system behaves, but about how the targeted behaviours are *compiled*. Since the metaobject protocol controls compilation strategies, the metaobject protocol introduces no inherent run-time overheads. The Intrigue project (Lamping et al., 1992) represents an attempt to use the same sorts of compile-time techniques in a MOP-controlled compiler for Scheme.

---

4. Of course, this is a conceptual distinction, rather than a necessary implementational one. Kiczales and Rodriguez (1990) describe the use of partial evaluation in PCL, a highly-optimised public-domain implementation of CLOS, which achieves this sort of cost movement, but does it in the implementation rather than in the language definition.

Compile-time techniques have also been used in non-language settings. Maeda's (1996) MOP-controlled filesystem provides application programmers with control over filesystem caching policies, but does not force the filesystem to continually indirect through a metaobject layer or metalevel at run-time. The direction of this work is to transform what Rao (1993) calls the "Don't use, don't lose" policy into a policy of "Use, don't lose", by designing the metaobject protocol explicitly to minimise or eliminate run-time activity. At the same time, this yields a second benefit, which is a simplification of the metaobject protocol technology. If the MOP is designed to minimise the "performance hit", then obviously the effort which must be expended to regain performance is correspondingly reduced, and so the metaobject system need not (necessarily) require the level of complexity that, say, the PCL CLOS implementation does.

3.4.3 Computational Reflection and Metaobject Protocols

This chapter has introduced an architectural approach (*Open Implementation*), a design principle (*computational reflection*) and a programming technique (*metaobject protocols*). Before proceeding, it is valuable to spend some time on the relationship between these and some consequences.

Open Implementation is a general approach to the design and organisation of software systems. The approach is based on the exploration of the role of abstraction presented in this chapter. Reflection is a general computational principle (about opportunities and consequences of computational self-reference). Open Implementations may be reflective, but that is not inherent.

The metaobject protocol is one technique for implementing reflective systems. To explore its consequences, it is interesting to compare the implementation of two reflective systems presented in this chapter, 3-Lisp and CLOS. CLOS uses a metaobject protocol, but 3-Lisp does not.

In 3-Lisp, the reflective representation is provided through access to the language processor. 3-Lisp provides mechanisms for moving evaluations between levels of the processor stack (the "infinite tower"), as well as describing the structure of the processor (environments, continuations, etc.) so that the relationship between the base level and meta level is revealed. One consequence of this "level-shifting" approach is that points where reflection is used, and the means by which it used, are very clear in the program. Reflection is *directly accessible*, and is not mediated by any other language feature.

CLOS's reflective link takes the form of a metaobject protocol—a protocol defined on metaobjects, which are themselves explicit representations of structures and mechanisms used at the base level. However, the MOP's encoding of the reflective link in terms of object-oriented programming begins to blur the boundaries of reflective manipulation. The object-oriented encoding means that reflection is now mediated by the mechanisms of object-oriented programming—subclassing, specialisation, method inheritance, etc. So, the MOP approach does not provide *reflective procedures* in the sense in which 3-Lisp did. Reflection is achieved through applying standard object-oriented techniques, but applying them to metalevel structures (the metaobjects and their methods). In other words, in the MOP approach, the reflection arises through the *objects* of actions, not through the *actions* themselves.

3.5 Summary

This chapter has introduced: first, a framework by which to understand the flexibility problems which were encountered in Chapter 2; and second, a route towards a solution.

The problems of flexibility in CSCW toolkits, as with the design of other toolkits, libraries, languages, lies in the range of purposes to which a client or application might put the structures and mechanisms which the toolkit provides. The effective matching of application needs to the toolkit's facilities implies either that the two be designed together (or with considerable knowledge of each other's behaviour and requirements), or that the client somehow be able to "see through" the abstraction barrier which separates them. The problems introduced by this opaque barrier become gradually worse as the number of potential clients increases (and with it, the range of potential client behaviours). So the traditional model of abstraction in system design, which is meant to support the reuse of implementations, is actually getting in the way of doing precisely that.

The Open Implementation (OI) approach provides a new way of thinking about the relationships between a client, the abstraction which the client is using, and the implementation which realises that abstraction. Drawing on computational reflection as a way of relating the abstraction and the implementation, Open Implementations provide clients not only with abstractions which they can use, but also with the means to examine and manipulate those abstractions. Using these facilities, clients can adapt aspects of the systems which support them, tailoring them and specialising them to their own particular needs.

This approach has been demonstrated in a number of systems, including programming languages, window systems and operating systems. The problems which OI addresses are precisely those which were encountered in the discussion of CSCW toolkit flexibility in the previous chapter, and so it is an excellent candidate for potential solutions. In the next chapter, I will start off by outlining aspects of the OI design approach, guidelines and desiderata, and then move on to discuss how I have applied OI to the problems of the CSCW toolkit, Prospero.

Chapter 4: Applying Reflection in a CSCW Toolkit

4.1 Introduction

Chapter 2 examined a number of CSCW toolkits and their support for flexibility. It showed how the decisions which had been made in the design of the toolkits' facilities limited their applicability, by committing to particular approaches to collaboration support. In Chapter 3, this problem was discussed with reference to very general notions of abstraction in software design, and the Open Implementation approach was introduced as a technique to address these problems. Open Implementations reveal aspects of system structure and behaviour, providing applications (clients of the abstractions) with principled means for examining and manipulating the internal operation of the abstractions. As a result, clients of the abstractions can become involved in how the infrastructure supports their operation, and so can tailor the implementation of system abstractions to their own particular needs.

This chapter and those which follow introduce the design and implementation of Prospero, a CSCW toolkit. Prospero uses Open Implementation and computational reflection to address the problems identified in the design of CSCW toolkits. This chapter is concerned with general design issues, and the scope of the toolkit; Chapters 5 and 6 will introduce the particular techniques developed in Prospero, and show how the apply to the problems of CSCW flexibility.

4.2 Designing Open Implementations

How would the design of an open CSCW toolkit proceed? To date, there are few examples of complete Open Implementation designs, although there are examples of OI-style concepts in otherwise traditional systems (such as the referenced work with composable microprotocols, or external paging facilities). On the basis of these experiences, what has emerged is not so much a process for doing Open Implementation design, but more a set of principles which should guide specific attempts at creating Open Implementations. Emerging work on Open Implementation Analysis and Design (OIA/D) (Kiczales et al., 1995) represents an early attempt to draw out these principles.

I will describe a number of these principles separately. However, it should be noted (and will become clear) that they are strongly related to each other.

4.2.1 Scope Control

One critical property is *scope control*, the ability to restrict attention (and changes) to the particular set of objects which should be affected. The ability to maintain and manipulate different scopes not only sets up protection boundaries, but also allows for different behaviours to be mixed together in a single system.

In CLOS, for instance, this is achieved through the class/metaclass mechanism. Since class behaviours are encapsulated by metaclasses, new behaviours are only introduced into those classes which specify a new, modified class as their metaclass. Introducing a change to slot access or method despatch will not affect every class in the system. The metaclass mechanism bounds the effect of the change, restricting its scope. At the same time, this also allows multiple behaviours to coexist. While a change to the slot access mechanism can be introduced for a new metaclass, the default behaviour exists alongside it, associated with the original metaclass. Indeed, any number of new behaviours might be introduced, and the scope control introduced by the metaclass namespace allows them to sit alongside each other without interference. A similar approach is used in Silica, through the use of specific "contracts" between types of windows and their subwindows.

The ability to name and distinguish between sets of alternative behaviours is not the only important factor in maintaining scope control. It is also critical that the groupings and categories to which these can be applied are at appropriate levels of granularity. For example, in CLOS, it would be unwieldy to have to discuss metaclass-level behaviours individually for each instance of a class, or to have to talk about all classes at once. CLOS associates these behaviours with classes, which are a convenient unit of scope for the flexibility which CLOS provides. So the provision of scope control means not only setting up the boundaries between different scopes, but also providing an appropriate level of granularity for bounding the effects of changes introduced at the metalevel.

4.2.2 Conceptual Separation

A second key property is the separation of conceptual concerns expressed by the metalevel interface. Again, this is essentially a scoping issue, but of a different sort—scope control addresses *which objects* will be affected by a particular change, while conceptual separation is concerned with the *extent of the behaviours* which are affected.

The meta-level interface expresses a range of different behaviours and aspects of the system's internals; the principle of conceptual separation states that the separation between different aspects of the system's internal behaviour should be expressed in a similar separation between those aspects of the interface used to control them. So, it should be possible to introduce a change to one aspect of the system's behaviour, relatively independently of the other aspects which the metalevel interface may control; and similarly, it should be possible to do this using only specific aspects of the metalevel interface relevant for that concern, without having to bring in (or even, perhaps, understand) the other areas. Simple changes should be simple to introduce.

The separation of base and meta levels can be thought of in terms of a traditional "separation of concerns"; conceptual separation, here, deals with the further separation of concerns *at the metalevel.* Perhaps even more than the other principles, this highlights the fact that the metalevel interface is designed to support a particular range of customisation behaviours, based on the designer's expectations or understandings of the system's particular domains of use. For example, the CLOS metaobject protocol opens up certain areas of processing (such as slot access, method invocation and slot initialisation—while leaving other areas unexplored (such as name binding and argument processing). This is done on the basis of the perceived needs of CLOS users and implementors. The separation of concerns in the metalevel interface reflects assumptions and expectations about the different behaviours likely to be tailored independently.

4.2.3 Incrementality

Incrementality deals with the ways in which changes introduced into the system relate to, and build upon, existing or default behaviours.

The provision of a meta-level interface, and thereby a means to change the system and adapt it to particular needs and circumstances, does not relieve the system designer of the burden of designing a good system. Open Implementations are intended to be usable designs with the metalevel interface as an added facility. They should have good default behaviour.

The default behaviour serves two ends. First, it provides the standard functionality of the toolkit. It should be usable in a normal range of circumstances, without any appeal to the metalevel interface. Second, when the metalevel interface is used to introduce changes, the default behaviour should be a basis for reuse. Incrementality concerns this second use of default behaviours. This principle states that it should be possible to introduce new behaviours by incrementally changing old ones, specifying just what's new and different relative to the original behaviour. The programmer should not have to recraft the behaviour from scratch, but rather use the default behaviour as a baseline. So, the default behaviour is provided not only as a usable system in its own right, but also as the basis for redefinition and reconstruction.

4.2.4 Robustness

Open Implementation allows applications to introduce modifications to the infrastructures which support them. In the presence of these sorts of changes, how can robustness be maintained? Although this issue has been recognised as critical from the outset, relatively little work has addressed it directly until recently.

As a first step, the use of *scope control* can be seen, to an extent, as a damage limitation exercise to preserve robustness, by providing the mechanisms to limit changes to specifically those objects in the system to which they need to apply. So, if a CLOS programmer designs a new slot lookup mechanism which is slow or flawed, then those changes are restricted to the programmer's new classes and so the behaviour of the whole system is not endangered. Similarly, in metacircular systems, this scope control approach prevents a programmer's (potentially incorrect or damaging) changes from affecting the internal behaviour of the system. At least, it prevents *accidental* incursions. In many systems, the programmer could deliberately specify internal system components as the objects of change.

Another approach to robustness is the use of *declarative* interfaces to the metalevel, rather than *procedural* ones. With a procedural interface, the metalevel programmer provides new code which will be executed as part of the operation of the underlying implementation. With a declarative interface, on the other hand, the metalevel programmer describes properties of desired state. In this case, the system retains control over its own mechanisms, incorporating the declared properties and balancing them against other needs within the system. A declarative model provides fewer opportunities for a poorly-considered metalevel change to undermine the system's behaviour or performance.

Some aspects of metalevel design lend themselves more naturally to a procedural approach, and some to a declarative. Both may be present within a single system. For example, recent OI work related to operating systems, and most particularly Maeda's OI filesystem (Maeda, 1996), has distinguished between areas of *abstraction mapping* and *resource allocation*. Abstraction mapping deals largely with the way in which a single process (or a single client) sees and manipulates underlying abstractions, while resource allocation involves a balance between the simultaneous requirements of multiple clients. Maeda's metalevel interface uses more procedural interfaces for control of abstractions, and more declarative interfaces for resource allocation concerns, so that a single client will not be able to adversely affect the allocation of resources for others.

4.3 CSCW: The Locus of Flexibility

A second question to be addressed is, what is the locus of flexibility? What sort of flexibility should be provided, and to whom? We can revisit the distinctions outlined in Chapter 2, looking at them from the OI perspective:

1. *Programmer/Toolkit Flexibility.* The traditional point of application of reflective/OI concepts is at the toolkit level, allowing the programmer to adapt elements of the toolkit to the needs of particular situations, extending the range of applications which can be built with the toolkit.

2. *Programmer/Application Flexibility.* A second possible point of application would allow the programmer, in an application, to appeal to the self-representation to support flexible or adaptive behaviour. The control which the programmer would exert could be used to adapt the application dynamically to particular circumstances of use (such as current patterns of interaction or network topology).
3. *User/Application Flexibility.* Finally, means of introspection and intercession could be offered at run-time to the users of the system. The user interface could present them, singly or collectively, with the means to tailor the internal mechanisms of the system to their own preferences, process or procedures.

These approaches are different but not unrelated, and a case made for any one of them is, essentially, a case for all. They can also be seen as a progression, each building upon the flexibility mechanisms provided in the previous approach.

Prospero's design adopts the first approach, using reflection and OI techniques to open up the toolkit to programmer specialisation. This is the approach most directly related to previous work in Open Implementations (such as in CLOS, Silica or Anibus). Aspects of the second approach are being explored in attempts to incorporate into OI practice lessons and techniques from work on adaptive computation (Lopes, 1996). Similarly, the third issue is being addressed separately in the development of a notion of interface *accounts*—the use of computationally reflective models as "stories" which systems tell about their own structure and behaviour, as a resource for improvised action (Dourish, 1995).

4.4 CSCW Toolkits: Areas of Concern

When existing CSCW toolkits were presented in Chapter 2, attention focussed on ways in which specific systems had made provision for flexibility and diversity in creating applications. The discussion above, though, illustrates that a hodge-podge of the various facilities they provided would be of little help in an Open Implementation. Instead, this section lays out the various areas of concern and functionality which those toolkits addressed, as a starting point in the design of Prospero. In particular, I will detail five particular areas of CSCW toolkit concern which emerge from current practice.

4.4.1 Data Distribution

Data distribution concerns the ways in which user and system data are distributed across a network. Issues which arise include:

1. *Replication*. Are multiple copies of data items available at once at different points in the network? If so, how many? How are decisions to replicate data made? How are multiple copies of data items held consistent?
2. *Migration*. Are data units held at a single location for their lifetime, or for the lifetime of a session? If data units move from place to place, what determines when they should move, and to where?
3. *Location*. How can objects be located? What determines the mapping between an object presented at the user interface, an internal representation, and the current location of that object?

Like the other concerns to be presented in this section, these data distribution issues are normally addressed and encapsulated at the lower levels of the system. The details of object replication, for instance, would normally be seen as lying below a "shared object" layer (which itself is encapsulated below the level of applications themselves). However, these are precisely the areas to which Prospero is addressed; areas of "low-level" concern which, as shown in previous chapters, have a significant impact on patterns of collaboration and interaction.

4.4.2 Concurrency and Exclusion Control

CSCW systems coordinate the activity of multiple users, across time and space. At different times, different numbers of users, corrected to varying degrees over a network, may be simultaneously working in the same workspace. How is the concurrent activity of these users managed? How does the system ensure that simultaneous action doesn't introduce inconsistency into the network? How is inconsistency avoided or conflict managed?

4.4.3 Representations of Activity

Both experimental studies (e.g. Dourish and Bellotti, 1992) and observational studies (e.g. Heath and Luff, 1992) have illustrated the ways in which individual activities in collaborative work are organised around perceptions and understandings of the ongoing work of others. As a result, an important area of design is the provision for representations, explicit or implicit, of the activities of other users. These may be integrated into the shared workspace (e.g. Hill et al., 1992), or provided externally; be at different levels of specificity; focus on current or previous activity; focus on task-intrinsic or extrinsic features; focus on the character or context of activity (Dourish and Bellotti, 1992); encode or mediate information (Bentley and Dourish, 1995); etc.

4.4.4 User Interface Linkage

In CSCW systems, it is not only the application, but also the interface, which is distributed. Different CSCW systems provide different levels of interface linkage. Two possible examples of interface linkage patterns are illustrated in figure 4.1. The strongest form of linkage is the screen-based sharing of systems, in which partici-

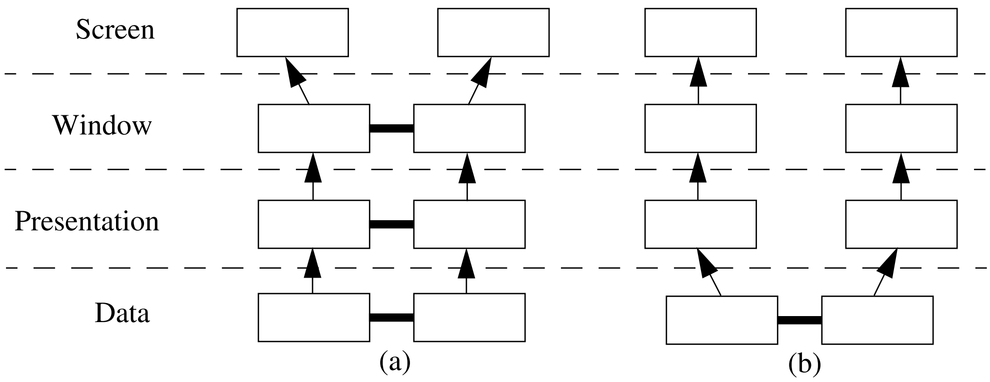


FIGURE 4.1: Distinguishing styles of interface linkage in terms of the level of user and group control.

pants' screens mirror each other (shown in 4.1(a)). A more flexible form of sharing replicates only window contents; this is supported by Shared-X, for example. More flexible yet are systems which share only application data, and may allow users to have different views of the data concurrently (shown in 4.1(b)).

Again, forms of interface linkage will clearly affect the styles and patterns of group interaction. (For example, see Stefik et al. (1987b) and Tatar et al. (1991) for accounts of the relationship between different interface linkage strategies in Colab and user experiences.) This, then, is another area where flexibility in a CSCW toolkit is important.

4.4.5 Conference and Session Management

Finally, areas of concern for a CSCW toolkit are the management of conferences and sessions. Existing systems vary in their definition of these terms. For the purposes of this discussion, I will regard a conference as a longer-lived unit than a session. Conferences are persistent even when there are no active users. Users may come and go, joining and leaving conferences as they please. A session is a period of activity within a conference. A session begins when the first active user joins the conference, and ends when the last active user leaves.

Conference management, then, is largely concerned with issues of naming and identification; the association of data and users together to form a single, identifiable conference entity. Session management, on the other hand, is primarily concerned with the recording, preservation and presentation of information about user activities. That is, the conference is the means by which a user might identify a collaborative activity in which they want to engage; the session is the means by which they catch up with ongoing activity and their own activity becomes associated with the data (and hence shared with other users).

4.5 Design Concerns in Prospero

The previous section highlighted some design themes. Each reflects a different set of concerns, but clearly all are relevant to overall issues in the interaction between system structure and user behaviour, and each is an area where the provision of flexibility in a toolkit is important.

The two themes which will be explored in particular in this dissertation are *data distribution* and *concurrency and exclusion management* (which, for convenience, I will jointly describe as *distributed data management*). There are two reasons for this narrowing of focus. The first is purely practical; any effort must start somewhere. The second reason is that these are the areas of greatest leverage. Tackling the problems of distributed data management provides a base for handling issues in activity representation and user interface linkage. Distributed data management subsumes both.

4.5.1 Distributed Data Management subsumes Activity Representation

In supporting flexibility in the representation of user and group activities—flexibility intended to support the creation of varied applications and styles of use—the primary problem is in the separation and identification of related activities. As will be the discussed in Chapter 5, CSCW systems traditionally take a fairly simple view of user action, which supports the illusion of a *single stream of activity*, partitioned in time or space between a number of users. This simplifies distributed data processing tremendously since, even when data objects are replicated, there is only ever one single stream of activity over the specific data representations. However, Chapter 5 will introduce an alternative mechanism which directly supports multiple, parallel streams of activity. Not only is this a better match to the actual patterns of user activity, but it is also a necessary step toward the level of flexibility and scalability in data management which is required.

At the same time, however, it creates a new abstraction which gives the programmer direct access to encapsulations of user activity. This separation between the activities of different users or groups in turn provides for distinctions between styles of activity representation within the user interface. As such, then, the data distribution and control model which will be introduced is also, critically, the basis for flexibility in representations of activity.

4.5.2 Distributed Data Management subsumes User Interface Linkage

The structure of figure 4.1 shows that the variety of interface linkage in CSCW systems can be analysed in terms of the level at which control passes from a single user to the group. In figure 4.1, there are four levels (screen, window, presentation and data), although more can be incorporated. In this model, where a single user has control over window placement, size, view and representation of the shared workspace, the user interfaces are "loosely coupled". In "tightly coupled" mode, on the other hand, more facets of the presentation (in particular, traditionally, the representation used and the view of the shared workspace presented in a window) will be under group control rather than user control. The meaning of "control", in this case, concerns the way in which changes are communicated to the group. Some aspect of user interface behaviour which is under "user control" can be modified by the user without affecting the equivalent aspects of other users' interfaces; whereas, if it is under "group control", there may be a protocol restricting access or changes, and any change introduced by a single user will be communicated to other members of the group (by being reflected in changes in their user interfaces). In other words, decisions about user interface linkage are concerned with which aspects of the user interface components are shared by the group, and which are privately controlled by individuals.

This characterisation of interface linkage in terms of information sharing leads to an obvious simplification. By its very nature, a CSCW system already provides a mechanism to control data sharing—the standard data distribution and control mechanism which is used for application data. This same mechanism can also be used to share, distribute and manage access to *user interface* data. When this happens, user interface linkage can be managed by placing aspects of the user interface controls into the shared workspace; and control over interface linkage is achieved through control over which aspects of the user interface components are shared in this way. So, within a toolkit or application, we can achieve considerable control over the extent of user interface linkage available to us by giving the programmer or user control over the way in which user interface components can be selectively moved in to, and out of, the shared workspace.

One way to conceptualise this is to think of a CSCW system which provides users with a view over some shared data space (such as in a collaborative drawing tool, for instance). Conceptually, the user interface is a window

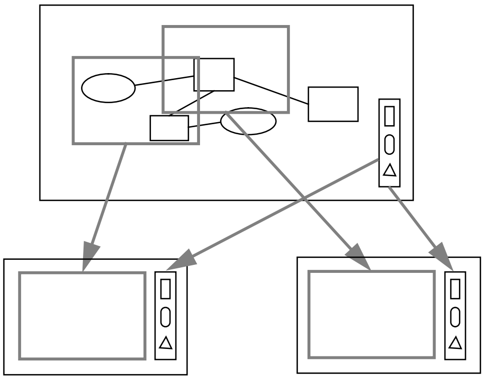


FIGURE 4.2: Using the shared workspace for user interface linkage. The two interfaces shown on the bottom have different views into the shared object space, but they share a tool palette.

onto a shared data space lying behind. The proposal, then, is that user interface components in the user's window can be "pushed through" to the shared data space behind, taking on shared semantics (as shown in figure 4.2). So, a scroll bar which is at the user window level provides control over the user's own view, and changes to that scroll bar do not affect the other user's view; the scroll bar, and its control data, is local and private. However, when the scroll bar is "pushed through" to the shared workspace, it becomes a shared scroll bar, controllable by either user (under whatever access control mechanism or floor control policy applies to the shared workspace), and with shared effect.

4.6 Communicating Application Semantics

Clearly, the key design criterion here is flexibility. It is, after all, in pursuit of a new, deep form of toolkit tailorability that Prospero adopts the reflective approach. This flexibility is intended to address issues of use as well as issues of design; and hence much of the control over the parameters of system flexibility comes from "above". Variability in purposes and usage patterns drives flexibility in application areas and in user and group interaction, which in turn drives flexibility in data representations and distribution control, etc.

Traditional models of abstraction in toolkit building work in the opposite direction. Designing a language, toolkit or other infrastructural system inevitably involves making decisions, which then become articulated and embodied in the system's implementation. These decisions—whatever their details might be—constrain the actions and behaviour of applications which are built on top of the infrastructure. So, instead of requirements and needs flowing down from the top—from use, to application, to infrastructure—we actually observe constraints flowing upwards from the bottom—from the infrastructure, to applications, to forms of use.

This is exactly the problem—application requirements pushing down against implementation constraints pushing up—which the Open Implementation approach addresses. Essentially, the characteristic of Open Implementation which will be employed in the next few chapters is its provision for exploiting higher-level semantics in lower-level system components. The information which is passed down is not only a set of requirements but also an account of how these requirements are realised. By appeal to reflective models, Open Implementation allows clients to specify not only *what* is to be done, but also aspects of *how* it is to be done, so that the specific details of the client's behaviour can be used to specialise the implementation.

4.7 An Overview of Prospero

So far, I have dealt largely with general issues motivating the application of the Open Implementation approach to CSCW, and some design issues which arise as a result. The next two chapters will be concerned with specific techniques developed in Prospero, a prototype OI CSCW toolkit. While the details of Prospero will be unfolded in those discussions of particular techniques and areas of concern, it is appropriate to preface those discussions with a high level description of Prospero and some background on what role it plays, how it appears to programmers, and the general concerns of the role which computational reflection plays in its design and use.

4.7.1 Base Level Programming in Prospero

Prospero is designed as a toolkit for building CSCW applications. Its users are programmers, who are in turn designing applications to support cooperative work. The abstractions which Prospero offers, then, are ones which can be used to support cooperative work. The basic abstractions which it presents are divergence and synchronisation of *streams* of *actions* (to be discussed in Chapter 5) and consistency management through the exchange of *promises* and *guarantees* (to be discussed in Chapter 6). Programmers use streams, actions, promises and guarantees to describe and implement their CSCW applications. These abstractions and the functions for their manipulation (creating them, and using them to support cooperation) constitute the base level interface.

4.7.2 Metalevel Programming in Prospero

Prospero is implemented in CLOS (CMU Common Lisp 17c, with an optimised CLOS implementation based on PCL of September 1992). As already discussed, CLOS is a reflective language; however, reflection in Prospero is independent of reflection in CLOS. Prospero, and the abstractions it employs to support cooperative applications and reflective processing over those abstractions, could also be implemented in a non-reflective language. The style of Prospero, however, lends it more to implementation in a symbolic, dynamic language, so it would be easier (or more natural) to reimplement in Smalltalk than in C++. The general principles which will be discussed in the following chapters, however, are independent of the implementation language, and in particular, are independent of the language's reflective facilities.

Just as reflection in CLOS is used to give the programmer control over aspects of its implementation, reflection is used in Prospero to give the application programmer control over the strategies which Prospero uses to implement the abstractions offered at the base level interface. The metalevel interface comprises those mechanisms which provide metalevel control. This is done through a metaobject protocol. In other words, Prospero does not provide a set of functions which change specific aspects of its implementation, but rather it makes internal components available as metaobjects, and documents the protocol by which they are related. Abstractions such as "stream" and "promise" are *used* at the base level; but they are *themselves* the direct objects of attention at the metalevel, through which issues such as synchronisation and editing can be reached (as shown in figure 4.3).

The metaobject protocol approach to reflective programming, as presented in CLOS, was discussed earlier (section 3.4.3). Just as in CLOS, Prospero's metaobject protocol approach to reflection uses subclassing, inheritance and specialisation as mechanisms to effect metalevel control. As discussed earlier, subclassing, inheritance and specialisation are not, themselves, reflective techniques. Reflection arises in the way in which this control is related to the base level; in particular, in the way that the focus of attention at the metalevel is the explicit representations of the abstractions used at the base level. The same relationship characterises the reflective link in 3-Lisp, even though 3-Lisp does not model it in terms of object-oriented programming mechanisms.

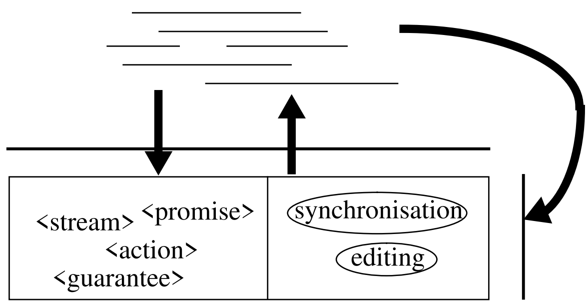


FIGURE 4.3: Base and meta-level in Prospero. Prospero offers abstractions and mechanisms for building collaborative applications, as well as meta-level control over the ways in which those abstractions and behaviours are realised.

For the purposes of this dissertation, Prospero is designed as a demonstration of the value of Open Implementation applied to CSCW. In Chapter 7, examples will be presented of applications built using Prospero, to show how Prospero's reflection can be used by programmers to tailor the toolkit to specific application needs. Data distribution and consistency management are the particular areas which Prospero opens up to programmer control. Clearly, a number of other areas (and, as previously demonstrated, the relationships between them) are also critical to the design of effective, flexible CSCW systems, but are beyond the scope of this dissertation. Although this chapter has already discussed some of these areas in general, it is valuable to discuss some of them here in the context of Prospero in particular, to explore their consequences and help frame the discussion in the chapters which follow.

4.7.2.1 User Interface

Prospero does not provide functions for user interface construction or management. Instead, it relies upon external facilities provided in whatever environment is available.

Clearly, user interface issues are critical to the development of successful CSCW applications. More significantly, here, this dissertation pursues an argument on the relationship between user interface and "low level" systems issues in CSCW design. From this perspective, then, it might seem unusual that Prospero does not include such facilities.

There are two significant points here. The first is that the argument I have pursued on the relationship between user interface and infrastructure design focuses particularly on the architectural issues in infrastructure, arguing that they should be opened up to make them accessible from higher levels, so that precisely this issue about the relationship of interface and infrastructure can be addressed. It is this "opening up" which Prospero serves to demonstrate. The second, and consequent, point is that this mechanism is then generally independent of the mechanism by which buttons, scroll bars and graphical objects are managed at the interface.

In the applications which will be shown, the user interfaces have been constructed and managed using the Garnet user interface software (Myers et al., 1990). Garnet's user interface model is presented in terms of a prototype-based object system with dynamic constraints, and specialisable interactors to encapsulate interactive behaviours independent of the graphical structures with which they are associated. As it turns out, this model fits easily and naturally with the mechanisms used in Prospero. Indeed, the interactors mechanism would be an obvious starting-point for extensions of Prospero's approach into the user interface domain.

Since Prospero does not manage the user interface itself, it also has no way to handle the management of user events and their relationship to system events. In the longer examples in Chapter 7, ways of managing this relationship will be shown.

4.7.2.2 Session Management

Prospero provides no direct support for session management. This includes a number of areas of functionality, including finding and naming sessions, and catching-up with existing sessions.

Within the Prospero model, session discovery and session-based resource discovery are, largely, temporally isolated, occurring in the initialisation phase. This provides an opportunity to exploit an external session management facility. As an ongoing activity, on the other hand, sessions provide a context for the naming of resources (by delimiting a scope). An object-naming mechanism is used in the examples in Chapter 7 to avoid potential name conflicts, but a fuller implementation would locate these naming issues along with session management.

4.7.2.3 Network Interface

Prospero does not provide access to the underlying network interface. The implementation uses CMU Common Lisp's "Wire" package for Lisp-based remote execution, but this is neither visible from the base level nor available for manipulation at the metalevel.

There are a number of reasons for this. The first, and most mundane, is that there is no commonly-accepted standard for interprocess communication in Common Lisp, and so access to this level of communication would render the toolkit hopelessly implementation-specific. The second reason, also primarily a practical concern, is that interprocess communication is itself a black box offered within the language (and often, as in this case, implemented as through a foreign-function interface to a library written in another programming language). As a consequence, it offers no opportunity for reflective introspection and intercession.

In contrast, the third reason is higher-level design issue. The goals of Open Implementation are to provide principled access to the underlying implementation, and to provide this in such a way that it can be effectively used to control those strategies relevant for higher-level use of the abstraction. The implication, as considered in the earlier sections of this chapter, is that the reflective representation through which this can be achieved, is, itself, a representation. It is crafted with particular purposes in mind and, like any other representation, it hides some aspects and accentuates others. In other words, this representation abstracts over the specifics of any given implementation, so as to be portable across implementations, and to focus on particular issues from the perspective of the base level interface.

Network communication and data communication are managed in Prospero through the use of streams and actions, to be introduced in Chapter 5. These are realised as metaobjects, so that they are, in turn, the terms in which modifications to the implementation strategy are given. The application programmer can use Prospero to gain control over these strategy decisions without "falling down" to an implementation layer below; and so the data management facilities in Prospero do not deal in the specifics of hosts, network addresses, sockets and server processes.

4.7.2.4 Conceptual Separation and Areas of Concern

On one level, the inclusion and exclusion of different areas of concern is problematic in an Open Implementation, since the OI approach talks directly about the potential interactions between issues on either side of the abstraction barrier (between the clients and the implementation). However, the goal of *principled* access to implementation strategies and decisions to which OI is directed *requires* the exercise of just this sort of partiality. Two points are relevant here.

First, the goal of opening up an abstraction is to make it amenable to particular forms of manipulation. The reflective representations which might be offered are, quite specifically, *representations* of the system's implementation (rather than, at the other extreme, simply the source code of the particular implementation being used). This is done for two reasons: first, to provide conceptual leverage in the meta-code, so that strategies can be described at a relatively high level, close to the model used in the base code, rather than in the implementation itself; and second, to make the representation independent of the specific implementation (as in the multiple implementations of the CLOS MOP, offering the same metaobject protocol over different underlying implementations).

Second, conceptual separation is an important feature in the design of an Open Implementation. The open facilities should be structured so as to allow a programmer to focus on particular areas of implementation strategy (particular mapping decisions, or sets of related decisions). If the metalevel is to be usable, it must provide this sort of structure. The corollary of this principle of conceptual separation is that we should be able to talk, relatively independently, about the different areas of concern which we wish to "open up" within an Open Implementation.

As a result, then, we can use specific areas—in this case, distributed data management and consistency management—to illustrate the application of the Open Implementation approach to toolkit concerns in CSCW, and to demonstrate the gains which result, without having to open up absolutely everything. In the course of Chapter 5 and 6, as the specific techniques in Prospero are detailed, they will be focussed on specific areas of concern, and specific problems which emerge in current approaches to their implementation in toolkits.

4.8 Summary

Drawing on the problems with toolkit flexibility in Chapter 2, and the Open Implementation approach introduced in Chapter 3, this chapter has set out the issues addressed in the design of Prospero. Drawing on a set of design desiderata for metalevel systems, it introduced five areas of specific concern in the design of flexible support for CSCW applications. These are areas of traditionally low-level design concerns, and which are typically locked behind the abstraction barriers of the toolkit. However, earlier chapters have shown that the decisions made at these levels can have a significant impact on the forms of collaboration and interaction which applications can support. So it is at this level that a toolkit can employ information from above—semantic features of the application and the domain—to resolve implementation decisions in the light of specific application needs.

There are two areas which Prospero deals with in particular—the management of distributed data, and control over consistency. These two areas, and the techniques which Prospero introduces to provide programmer control over them, will be the subjects of the next two chapters.

This approach of exploiting application semantics within the infrastructure is one which will be applied first to data distribution (replication and access), and then to consistency management.

Chapter 5: Divergence, Data Management and Collaborative Work

5.1 Introduction: Distributed Data Management

Collaborative applications coordinate activities which may be distributed in time and/or space. Distributed in time, activities may take place at different moments but are coordinated to achieve a unified effect (such as the production of a document). Distributed in space, activities may take place on different computers perhaps linked by a data network, over notionally shared artifacts and representations. So collaborative applications are heir to a set of design problems which have arisen in the development of distributed systems concerning distributed data management.

In this chapter, I consider strategies to meet the conflicting demands placed on collaborative applications in presenting users with a single, uniform data "space". I present a new model of distributed data management specifically for collaborative work, and show how it is developed and used in Prospero. I am primarily (but not exclusively) concerned here with "user data"; that is, largely the computational representations of artifacts which are manipulated directly by the system's users. So, in a collaborative writing system, user data would include the representation of the document, or of users' activities over that document, rather than data structures controlling the system's own behaviour. This focus is reflected when establishing requirements and criteria for data management strategies, although (as highlighted in Chapter 4) the distinction between user and system data will emerge as less than clean-cut; for instance, representations of user activity straddle this distinction.

I will begin by examining aspects of the problem—the criteria by which strategies are evaluated, and the ways in which CSCW systems have addressed these problems, particularly drawing from the distributed systems tradition. As has already been outlined, traditional distributed systems algorithms do not map easily onto the needs of collaborative systems, because of their implications for interaction; and in a toolkit, we need to be able to support a wide range of interactive strategies for different applications. After setting out the scope of the discussion, I will characterise the standard approaches in terms of *inconsistency avoidance*, rather than *consistency management*. I will then outline an alternative model, based on cycles of *divergence* and *synchronisation*, which satisfies the flexibility requirements discussed in earlier chapters. After introducing this divergence-based model, and discussing its the advantages in collaborative settings, I will present a series of brief examples to illustrate how it can be used.

5.2 Criteria

Distributed data management systems attempt to meet a range of criteria to greater or lesser degrees. Different applications or domains will emphasise different criteria, but some of the central ones are:

1. *Availability.* Users should be able to gain access to data when they need it, wherever it might be in the network;

2. *Transparency.* Users should not have to worry about the details of distribution or the implementation in order to use shared data;

3. *Consistency.* Data items may be accessed from different points in the network. Users should see identical (or, at least, consistent) views of shared data, even though they may be working at different places or different times;

4. *Responsiveness*. Data management should not interfere with the system's interactive response.

Unfortunately (if predictably), these place conflicting demands on an implementation. For example, availability can be enhanced by maintaining multiple copies of the data on different network nodes. However, this "replication" approach conflicts with consistency, since it potentially introduces the opportunity for two users to make simultaneous incompatible changes to two copies of the same piece of data. Various strategies can be used to avoid or resolve these conflicts, but these, in turn, endanger transparent operation, by introducing more ways in which users can be exposed to the consequences and details of data distribution. Mechanisms can be introduced to maintain coordination between the replicas behind the scenes, but the processing involved to ensure consistency may interfere with responsiveness. Different systems have different requirements, and must give these criteria differing priorities.

These problems are endemic to a wide range of traditional distributed computing systems, such as distributed databases and distributed file systems, and solutions have been developed in these domains. Similarly, problems of distributed data management have been addressed by the CSCW toolkits discussed in previous chapters. The critical distinction between (most) CSCW applications and (most) distributed systems is the role of user and group interaction over distributed data. First, distributed data is often directly represented and manipulated in a user interface, which significantly increases the response time requirements of the system. Second, the shared data space is a critical channel of communication between users (Dourish and Bellotti, 1992) and so the choice of data management strategy may affect the patterns of group interaction. For instance, a system which ensured that consistency was maintained *at the point of manipulation* would not be suitable in systems which use a shared space to present a continual, ongoing awareness of group work to all users. So before distributed systems approaches can be applied to CSCW, or mined for possible solutions, we must first consider the particular relationship between distributed data and collaborative work.

5.3 Distributed Data and Collaborative Work

Our starting point is an examination of the design issues behind existing strategies (such as those illustrated in Chapter 2).

5.3.1 Distribution

The first set of design decisions concern the mechanisms which determine where a particular data structure will reside in the system at any given time—data *distribution*. We can distinguish two orthogonal choices here.

The distinction between *centralised* and *replicated* approaches has long been a concern for CSCW developers (Ahuja et al., 1990; Lauwers et al., 1990; Greenberg et al., 1992). Centralisation concentrates data at one point in the system; clients communicate with this central point to retrieve or update information. Centralisation favours simplicity and consistency (which is trivial since there is only one copy of a data item at any time), potentially at the cost of availability and responsiveness in large networks or groups. Replication allows multiple copies of data structures to coexist, which improves availability, but complicates consistency management.

The distinction between centralised and replicated architectures speaks only to the number of copies of any given data item, but not to their location. Most systems are *static*; the "location" of any given piece of data is fixed through the course of system execution, or a session. However, this is generally not inherent to the approaches. In a *dynamic* system, data items may migrate from location to location. Distributed document management systems, and some workflow systems, use this approach, mimicking the physical movement of documents and responsibility from one user to another. Some other distributed systems (such as Obliq (Cardelli, 1995)) employ a similar approach in which data migrates to the site of computation.

5.3.2 Management

The critical question for any distribution strategy, however, is how consistency can be maintained in the face of the simultaneous activity of multiple users

Often this is a "non-problem". Many applications do not require absolute consistency in user data. For instance, in a "shared whiteboard", absolute consistency is rarely a concern. Conflicts over a single-plane bitmap are minimal and non-intrusive. In these circumstances, the system would be unlikely to attempt to maintain data integrity rigorously. Indeed, the overhead of many consistency management strategies would interfere with the responsive performance and free-form interactive style that a shared whiteboard requires. In more structured applications, however, consistency can be vital. The inconsistency acceptable on a shared whiteboard is unacceptable in a spreadsheet or source control system.

Inconsistency generally arises through misorderings in applying changes to user data at different sites. User actions arise independently at different points in the network, and are then propagated to other users. This distributed activity introduces timing problems; events may arrive at different nodes in different, unpredictable sequences. To maintain consistency, the system must ensure that each client sees the result of these changes applied *in a consistent order*.

In a centralised system, since everyone sees the single, unique copy of any data item, they see the same changes arise; there is a single, network-wide consistent ordering of events. Only one event can be processed at a time, so changes which arrive at the "same" time will be processed separately, in some specific (if arbitrary) order. So, centralised data storage inherently introduces a *serialisation* of change events, which, while potentially unpredictable, maintains consistency.

Replicated systems can also achieve consistency through global serialisation. The simplest approach, used in one form or another in many CSCW systems, is *data locking*. In this approach, clients must request and hold locks on data before they can make modifications. Since only one client at a time can hold the lock on an specific item, simultaneous changes are prevented and consistency is maintained. Locks operate at different granularities, from the whole document down to individual objects or insertion points, and the request/grant cycle for locks may be explicitly handled by users or implicitly managed on their behalf by the system. Whatever the details of their implementation, their role remains the same—to avoid inconsistency by preventing simultaneous action on data items.

The use of floor control is a particular form of this approach. Floor control regulates the activity of multiple users over a shared space, setting up mechanisms by which one at a time can "gain the floor". Most floor control policies can be seen as managing locks on the entire workspace, restricting activity to one individual at a time. This is *input multiplexing*—reducing multiple input channels (one or more per individual) to a single channel (the input channel to the workspace). Mechanisms such as "baton-passing" and "round-robin" divide access between participants so that only one has control at any point. The participant who has the floor essentially holds a lock on the entire workspace. Since no-one else can contribute until the "lock" is released, consistency is maintained.

5.4 The Emergence of Inconsistency

The variety of data management strategies is testament to the fact that no single approach is applicable in all cases. In part, this is simply due to the considerable variation in the needs of CSCW systems. In addition, as discussed previously, it is because the choice of management strategies has strong implications for the interface and for the nature of collaborative interaction in a CSCW system. Data distribution examples abound. Centralisation strategies introduce delays into the interaction cycle; locking strategies make parts of the workspace unavailable to participants; and replication strategies can result in inconsistency between user displays.

In other words, 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 the designer must be constantly aware of the way in which application distribution and interface distribution are mutually influential. Decisions about data distribution must be made in the context of application needs, since application details affect the relative importance of timely response, activity feedback, flexible interaction and strong consistency.

These issues are particularly important when building a CSCW toolkit such as Prospero, which will be used to create a wide range of applications. The toolkit designer is even more distant from end-users than is the developer of individual applications; and so, as argued in Chapters 1 and 2, it becomes critical to understand the implications of distributed data strategies for particular usage situations. Here, we need to find a general characterisation of distributed data management in CSCW.

5.4.1 Streams of Activity and Inconsistency Avoidance

The distributed systems approach to distributed data has typically been organised around the data objects themselves. Indeed, we saw in Chapter 2 that most CSCW systems also manage shared data in terms of the fan-out between a single abstract "shared" object and the multiple representations of it which float around the network. Data items might be grouped together at particular locations, respond to messages, coordinate with each other, or be maintained in "virtual synchrony" (Birman and Joseph, 1987). However, if we want to try to understand data distribution for specifically *collaborative* work, we need to be able to see it in terms of user activity, and so draw out the relationship between models of activity and models of distribution. Shared objects are not very enlightening; we need to choose another unit of analysis. What's more, as highlighted in the discussion of scope control in Chapter 4, it must be one which gives us flexible, controllable access to the ongoing actions of users at the interface and components of the system, if it is to be useful not only as for analysis but also for toolkit manipulation.

Coming from the user side, rather than the system side, I will present this discussion of data management in terms of *streams of activity*. These streams will be the unit of analysis here. A stream of activity is a set of sequentially-related operations, coherent over time. Normally, the producer of a stream of activity is a single user sitting at a workstation; however, other agents may be responsible for producing streams (either as proxies for users, or perhaps under programmatic control, as in a session recorder/player). Although most streams are associated with a single source, a single user may produce multiple streams concurrently (perhaps acting in two roles, or in two workspaces, concurrently). When we come to look at examples, later in this chapter, and subsequently in Chapter 7, we will see in more detail how streams are used as an abstraction for sequences of action. For the moment, though, a stream of activity can be thought of as capturing the actions of a single user in a single session.

What does a streams-oriented view tell us about the approaches to distributed data management discussed so far? The most interesting answer is that streams are nowhere to be seen, either from the user or the data perspective. In fact, the action of most systems seems to be to *eliminate* the notion of separate streams. To achieve a unified view of the data, they produce a unified view of activity over it.

In other words, most approaches to data management in CSCW deal with *inconsistency avoidance* rather than *consistency management*[1]. Rather than working to achieve data consistency, they erect barriers (such as asynchronous access, floor control and locks) to prevent inconsistency arising in the first place. This approach comes from the distributed systems tradition; the system manages the action of the separate components to avoid inconsistency. However, applying this strategy to collaborative work is problematic. Here, the distributed entities are users, not programs; and users are generally less prepared to accept the imposition of global mechanisms to constrain their activity!

1. One notable exception, *operational transformation*, will be discussed shortly.

Since inconsistency arises through the simultaneous execution of conflicting operations, the simplest approach to avoiding inconsistency is to avoid simultaneous action over individual data items. This approach attempts to define a *single, global stream of activity* over the data space. Asynchronous access achieves this by sharing one stream between multiple participants, one at a time. Floor control policies and locking mechanisms do likewise, at a finer granularity. Mapping collaborative activities onto a single stream of activity establishes an inherent serialisation and avoids inconsistency.

This inherent serialisation is not appropriate for Prospero, since the goal of supporting a wide range of applications implies that the interactive limitations which that approach implies will be too restrictive. Instead, Prospero embodies an alternative approach which abandons this attempt to construct a single stream of activity out of multi-user activity. Instead, it begins with a picture of *multiple, simultaneous streams of activity*, and then looks to *manage divergence* between these streams. Divergence occurs when two streams have different views of the data state; it is the *emergence of inconsistency*. This could arise through simultaneous execution of conflicting operations at different points in the network, or through a lag in propagating compatible operations from one site to another. The divergence model works to maintain data integrity by coordinating multiple streams of activity and resolving conflicts, rather than by attempting to eliminate the multiple streams and prevent conflict.

The general divergence view does not imply any particular number of parallel streams of activity. So it encompasses the traditional views outlined earlier; they correspond to the special case of just one stream. Divergence between multiple streams of activity is the *more general case* which subsumes attempts to maintain a single thread of control. This generality is critical to the design of a toolkit.

5.5 Divergence

So the first move is to regard collaborative activity as the progress of *multiple, simultaneous* streams of activity. We no longer have the fiction of a single stream of activity to work with. However, data distribution is *all about* maintaining the fiction of a shared data store, even though each individual data item might be replicated and distributed. Multiple, simultaneous streams of activity can lead to inconsistency in the actual data representations being manipulated. So, as outlined above, the second step is to view inconsistency as *divergence* between these streams' views of data (or, at a more abstract level, the divergence of the system as a whole from a unified view of the data).

Hence, we can see distributed data management in terms of the *re-synchronisation* of divergent streams of activity. As collaboration progresses, the streams continually split and merge, diverge and synchronise. At synchronisation, they re-establish a common view of the data; further activity will cause them to diverge again, necessitating further synchronisation later. This pattern is characteristic of the divergence model, and the focus of the rest of this chapter and the next is how the pattern can be analysed, represented, manipulated and exploited.

Although the toolkits which were presented in Chapter 2 largely adopted the single-stream view common to many CSCW systems and toolkits, there are some examples of approaches, within CSCW and in other related fields, more reminiscent of the divergence model. Before going on to discuss divergence and CSCW in more detail, it's worth briefly reviewing some of the other approaches related to divergence, and discussing the extent to which they capitalise on these ideas.

5.5.1 Divergence and Versioning

The view of collaborative data manipulation as continual divergence and synchronisation is quite similar to the principles of versioning systems. Versioning is a technique which maintains a historical record of the instances in time, or *versions,* of objects or files. Versioning can be applied at different granularities—workspaces, files, objects, etc.

In CSCW, version systems have been used to track the different stages of evolution of objects in a shared workspace. They typically allow multiple versions of an object to exist at once, and in some, multiple versions can be simultaneously active. For example, GMD's CoVer (Haake and Haake, 1993) uses a version system to manage the cooperative work. CoVer is a version server which has been used to support the SEPIA collaborative hypertext system (Streitz et al., 1992), by providing explicit versioning support in addition to straightforward object store functionality.

There are two principal differences between the divergence approach and the versioning approach. The first is that versioning systems like CoVer in CSCW tend to emphasise the creation and management of parallel versions, rather than the subsequent integration of different versions (divergent streams). Merging (synchronisation, in the divergence model) is typically something which happens outside the versioning framework itself. Munson and Dewan (1994) provide a framework organised around version merging, but, again, they primarily emphasise versioning and merging within a context of "asynchronous" work. This reflects the second difference, which is that most collaborative versioning systems tend to regard versions as fairly heavyweight objects. Versioning might be applied to a document, or to document sections, and perhaps, occasionally, even paragraphs; but they would tend not to use versions to represent the divergence resulting from two users inserting different characters in a synchronous collaborative text editor. This heavyweight data-driven approach means that versioning systems are rarely as scalable as the divergence model must be (discussed in more detail below).

5.5.2 Divergence and Replicated Databases

Replicated database research has also addressed questions of divergence. In a replicated database, multiple copies of all or part of the database are maintained in parallel, to increase availability. Database queries might be satisfied by activity over these various copies of the data, and a range of mechanisms might be used to propagate updated information between the distributed components of the data store. The relationship between Prospero's mechanisms and those of replicated database research will be discussed in more detail when the consistency mechanisms are presented in Chapter 6, but a brief outline of the divergence issues is relevant here.

In database work, consistency is normally maintained by supporting a transaction model, which decomposes database activity into a sequence of transactions. Transactions group related operations for atomic execution; since transactions execution is all-or-nothing, consistency can be maintained. In replicated databases, research focuses on the detection of transaction conflicts and on finding an execution order which avoids potential conflicts. Various approaches can be used to sustain the transaction model under replication. For instance, distributed conflict detection can be used to generate the consistent serialisation globally, rather than individually at each replication point; or rollback techniques can be used with an optimistic concurrency model, so that conflicting transactions can be undone and re-executed later.

These techniques place the detection, avoidance and management of conflicts *within* the database itself; unlike the divergence proposal, the application is typically *not* involved in the conflict management process. This is generally true when collaborative applications are based on database technology. However, there are times when this model must break down. In Lotus Notes, for example, users interact directly with document databases replicated amongst different sites but largely disconnected from each other, and so conflicts can occur during periods of simultaneous work (as here). However, in these cases, Notes merely flags the conflict and carries on, rather than providing any means for conflict resolution. Replicated databases deal with some problems which divergence raises; however, they generally do not directly exploit divergence to support multi-user activity.

5.5.3 Divergence and Operational Transformation

An alternative technique which has been employed effectively in a number of collaborative systems is operational transformation (Ellis and Gibbs, 1989; Karsenty and Beaudouin-Lafon, 1992; Nichols et al., 1995[2]). This

2. Not all of these authors use the term "operational transformation" for their approach, but it will serve here as a generic term.

approach perhaps comes closest to the divergence mechanism presented here, in that it explicitly manages multiple streams, and attempts to resolve conflicting activities rather than prevent them.

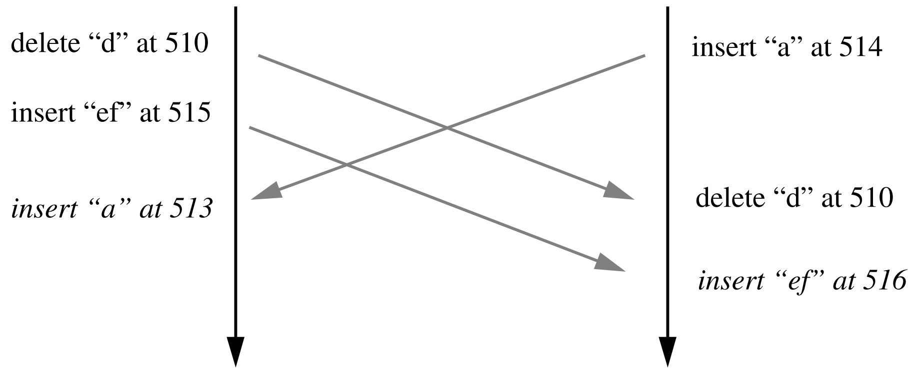


FIGURE 5.1: In the operational transformation approach, records of commands are transformed before execution to reflect differences between the execution contexts at each host.

Operational transformation employs a model of multiple streams, and uses a transformation matrix to process records of remote operations before applying them locally. This "transformation" uses information about the different contexts in which the operations arose, and the history of operations executed at each site. The transformation eliminates the effect of operations which "pass on the network", so that they execute in different orders at each site. The result of a transformation is an operation which can be applied locally to achieve the same effect as the operation had when it was originally executed at the remote site in a different context. The transformation re-contextualises the operation. For example, if two users both delete the same character on a line, and send their operations to each other, then the transformation matrix would replace each incoming "delete" operation with a null operation, since the character has already been deleted. The transformation matrix comprises rules for transforming any operation which might have "passed" any other on the network, and so be executed out of order.

Clearly, this approach is much closer to the divergence model advocated here, but there are two principal differences. First, just as versioning approaches have typically emphasised asynchronous activity, operational transformation has typically emphasised synchronous; as will be illustrated, Prospero's use of divergence attempts to be more general, incorporating other models of collaboration. 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 which is potentially more widely applicable.

5.6 Capitalising on Divergence

Much of what's critical about the divergence view is what it *doesn't* say, because those areas of openness are the keys to the specialisable nature of the model. So far, nothing has been said about the defined units of activity, or what constitutes a "stream"; nothing has been said about the granularity of "divergence" *per se* and how it is recognised; and nothing has been said about the timescale on which divergence and resynchronisation takes place. In fact, this openness is critical to the particular advantages of divergence for CSCW, and are features exploited for flexibility in Prospero.

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-synchro-*

nous—which are supported poorly or not-at-all by existing approaches. Third, it directly supports common patterns of working activity based on observational studies which are at odds with the models embodied in most systems today. This section will consider each of these in more detail.

5.6.1 Scalability

Scalability refers to graceful operation across some dimension of system design. In particular, the scalable dimension here is the "pace of interaction" (Dix, 1992); or, more technically, its relationship to the period of synchronisation.

The period of synchronisation determines 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 so 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 which communicates the effects of each user's actions to all participants "as they happen". The synchronous element arises from the short delay between divergence (an action taking place) and synchronisation (the action being propagated to other participants). This is one end of the "pace of interaction" dimension.

At the other end of the spectrum, synchronisation can take place much less frequently in comparison to the actions of the users. In this case, there is considerably more divergence, arising from different sorts of activities which 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. An 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. Scalability across this dimension allows Prospero to generalise across this distinction.

5.6.2 Multi-Synchronous Applications

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-written 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 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; not synchronous, nor asynchronous, but *multi-synchronous* work.

Multiple, parallel streams of activity is a natural way to support 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) to achieve a consistent state and progress group activity.

Here I am concerned with the *nature* of synchronisation, 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. The important factor here is the support for multi-synchronous working within this model of distributed data management.

5.6.3 Supporting Opportunistic Work

Divergence does not simply support a different working style; it's also a means to support *more naturally* the other styles to which CSCW has traditionally addressed itself. In studies of collaborative authoring, Beck and Bellotti (1993) highlighted the opportunistic way in which much activity was performed. In particular, they pointed to the ways in which opportunistic action on the parts of individual collaborators often went *against* predefined roles, responsibilities or plans. Individuals acted in response to specific circumstances; while the plans and strategies formed *one* guide to their actions, they were by no means the only factors at work, and in each of their case studies, they observed occasions on which agreements about who would do what and when were broken. Critically, these broken agreements are neither unusual nor problematic; opportunistic activity is simply part of the natural process of collaboration. Suchman (1987) has, of course, made similar telling observations about the status of plans as resources for action rather than as rigorous constraints upon it.

These observations suggest that CSCW systems should be wary of reifying plans and using pre-formed strategies to organise collaborative activity, since they are often opportunistically broken in the course of an activity. Turn-taking floor control policies, or partitioning a workspace into separate regions accessible to different individuals, are examples of technological approaches which structure user interaction around plans of this sort. Once again, this highlights the contrast between the particular needs of CSCW systems and traditional distributed systems, and shows that a distributed *interface* is an important consideration. To support the sort of opportunistic working described by Beck and Bellotti, then, CSCW technology must *relax* rules about exclusion and partitioning— exactly the rules which have been employed to maintain the fiction of the single stream of activity.

So the same sorts of mechanisms which 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 effective*. 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.

5.7 Divergence in Prospero

We can now look at how divergence works in practice. The examples presented in this section simply illustrate the basic ideas; more detailed and complex examples will be examined in Chapter 7, once the consistency guarantees mechanism has been presented.

Prospero's structure and programming model was introduced in Chapter 4. The examples here illustrate how its data distribution mechanisms, modelled in terms of the divergence and synchronisation of streams of activity, are exploited in the development of CSCW applications. The examples also show how toolkit tailoring is achieved by specialising the programming model. Since Prospero is defined in terms of generic functions to relate the different behaviours and system components, new mechanisms introduced in specialised classes will affect its internal behaviour.

These examples will particularly show how divergence in Prospero supports a wide range of application strategies. They take the form of code fragments illustrating the framework's specialisation to the needs of particular applications. After presenting the examples themselves, I will step back to consider the structure of the framework.

Some points should be noted. First, the examples have been considerably simplified to illustrate the main points for this chapter. In particular: the interaction between divergence management and consistency guarantees has been omitted until consistency guarantees are presented in Chapter 6; some functions are included with the examples for explanatory purposes which are actually defined by Prospero; and some examples use library functions which wrap around the core Prospero functions and hide details not relevant to these examples (such as the Lisp RPC mechanism underneath the Prospero implementation).

Second, these examples operate on three levels at once, and it is critical to a conceptual understanding that these are kept separate. The first is that of the example applications used to illustrate the ideas; the code, the applications and the implementation strategies. The second is the use of programming structures to realise these applications: the way in which subclass and specialisation mechanisms are used, and the way in which Prospero's facilities are integrated into applications. The third level, which is the most important for my current purposes, is the use of divergence itself to provide a programming framework; the way in which the multiple streams model lends itself to a particular structure of application programming, and how the pattern of divergence and synchronisation is reflected in the application. Since the examples have been structured to highlight this third level, liberties have been taken with application requirements and efficient programming.

5.7.1 Example: Shdr

Shdr is a simple replicated shared whiteboard application. It was designed outside the divergence framework; it was originally written in 1990 to support experiments in remote collaboration over long distances. Actions from the user interface are performed on the user's own copy of the data, and are recorded in a local buffer of activity records. Periodically, buffer contents are sent to other participants using a simple, high-level protocol (that is, one defined in terms of domain—drawing— events, rather than lower-level user interface events like mouse-clicks and key-presses). The update frequency varies, but generally the history is transmitted multiple times per second.

We can reconstruct shdr's approach in the divergence framework (figure 5.2). Local actions create divergence from a shared view of the whiteboard until synchronisation, when history records are exchanged. Each user's actions are associated with a particular stream, where they are recorded until synchronisation.

```
(defmethod locally-perform-action :after ((action <edit-action>))
   (add-action-to-stream action *my-stream*))

(defmethod add-action-to-stream ((action <edit-action>) (stream <stream>))
   (push action (stream-actions stream))

(defmethod add-action-to-stream :after (action (stream <bounded-stream>))
   (if (full-p stream)
      (synchronise stream (stream-remote stream))))

(defmethod synchronise ((stream <bounded-stream>) (remote <remote-stream>))
   (dolist (action (reverse (stream-actions stream)))
      (propagate-action-to-stream action remote))
   (stream-reset stream))

(defmethod propagate-action-to-stream (action (stream <remote-stream>))
   (remote-call (stream-host streams) incorporate-action action))
```

FIGURE 5.2: Mapping shdr's strategy in the Prospero framework.

User actions are explicitly represented within a class hierarchy rooted in the abstract class `<action>`. Different actions are instances of its subclasses. Here, we use the subclass `<edit-action>` for actions which have an effect on the data store (such as making or erasing a mark, but not cursor movement).

Activity streams are also explicitly represented, under the abstract class `<stream>`. Two subclasses of `<stream>` are used here. The first, `<remote-stream>`, represents the streams of other users; the second,

```
(defmethod add-action-to-stream ((action <edit-action>) stream)
   (push action (stream-actions stream)))

(defmethod add-action-to-stream ((action <synchronise-action>) stream)
   (synchronise stream (stream-remote stream)))

(defmethod synchronise (stream (remote <remote-stream>))
   ;; as figure 5.2 ...
   ...)

(defmethod propagate-action-to-stream (action (stream <remote-stream>))
   ;; as figure 5.2 ...
   ...)
```

FIGURE 5.3: Check-in/check-out strategy with synchronisation events.

`<bounded-stream>`, is a particular kind of local stream with specialised behaviours, particular to the way that shdr manages user data. A `<bounded-stream>` accumulates local actions and periodically flushes them to other participants.

We define shdr's strategy in Prospero by writing specific methods on a generic function framework which in turn describes the general model that Prospero embodies. These are the hooks onto which specialised behaviour can be hung. For instance, the generic function[3] `locally-perform-action`, which Prospero uses to operate on the local copy of user data, is a place to "attach" the association of user actions with a specific stream. This is defined for `<edit-action>` operations, rather than all `<action>` operations, since only the actions which cause a change in the data store contribute to divergence. Next, the test for whether a bounded stream is "full" and needs to be synchronised is made after any new action record is stored there, and so the after-method we define for `add-action-to-stream` specialises on `<bounded-stream>` rather than `<stream>`, so it applies only to bounded streams.

5.7.2 Example: Source Code Control

The second example is a traditional source code control system in a collaborative programming environment. It uses a check-in/check-out model for software components or modules, in which modules must be explicitly "checked out" of the repository before they can be modified. Checking a module out prevents other users from modifying it until it is "checked in" again; so this is a locking mechanism at the granularity of code modules.

Code for this example is shown in figure 5.3. After the first example, most of the structure for this is already provided. We already have a means to accumulate and distribute sets of changes which arise in one place or another, which can be reused here (and, in fact, is defined in Prospero, and so would not need to be included in the previous example). In this example, however, there is a one major difference from the first, which is that synchronisation is an explicit, user-initiated event. "Check-in" and "check-out" are actions which mark points when views of the data store should be synchronised.

Just as the first example distinguished between actions in the interface which did or did not have an effect upon the shared data, this example distinguishes between actions which do or do not force synchronisation. These user-initiated synchronisation. actions are members of a new action class, `<synchronise-action>`. In normal editing, the system accumulates the action records, as before; but for synchronisation actions, the synchronisation function is invoked. The use of explicit command objects in a class hierarchy allows these two

3. The keyword ":after" designates that the method defined for locally-perform-action in this example is an "after-method", a CLOS feature which will be explained in more detail in Chapter 7. Essentially, "after-methods" run after the primary (standard) methods for this generic function have been run.

styles of synchronisation, both explicit and implicit, to be handled within the same object-oriented framework. This mechanism—class-based encoding—will be discussed in more detail in Chapter 6.

5.7.3 Example: Multi-synchronous Editing

The final example here considers the implications of multi-synchronous working—the case not handled by traditional CSCW toolkits and systems.

At this stage, multi-synchronous activity is no different at the point of divergence (although Chapter 6 will deal with ways in which a fuller implementation would use consistency guarantees to limit or constrain divergence). Once again, we can accumulate actions until some synchronisation action occurs, either automatically or by user request. This, however, is the point at which a more complex strategy is required. In the first example, we could simply ignore data consistency problems; and in the second, asynchronous access ensured that such problems never arose. In this example, we have to be aware of the possibility of mutually inconsistent changes and act accordingly. So the focus of attention in this case is on the synchronisation procedures.

The code in figure 5.4 illustrates two points. The first is that synchronisation now requires explicit processing (i.e. it is not simply the transmission of information); and the second is that it is now the *mutual* achievement of both parties (i.e. its no longer sufficient for the originating side to send the information and move on, but rather both sides must be involved).

The approach is very simple. For the first time, the synchronisation procedure makes use of the return value of `propagate-action-to-stream`, which carries back information from the remote side. In particular, it returns the “intermediate results” of synchronisation; that is, a form of the data which has been modified to reflect the resolution of conflicts. This must now be reintegrated into the local stream’s view, so the integration procedure is something in which both sides are involved.

We also see the way in which `incorporate-action` processes records of activities originating in some other stream. In this case, we use the simplest strategy. If the remote action is an edit action, and if it is compatible with local changes, then it is applied. If not, then we simply combine the two pieces of text as a unit to be processed by the users later. (This technique—aggregation—hinges upon a distinction between “syntactic” and “semantic” consistency to be detailed in Chapter 6.) Since the open strategy used in Prospero allows specialised definition of functions such as `compatible-p` and `locally-perform-action`, then we can be quite loose in what is accepted, and what constitutes compatibility; it may be possible to make recourse to more information than is recorded here.

5.7.4 Specialisation in Prospero

As well as illustrating the details of the divergence mechanism, these examples also show the pattern of specialisation and modification in Prospero; in other words, they illustrate Prospero as an Open Implementation. The

```
(defmethod synchronise (stream (remote <remote-stream>))
   (dolist (action (reverse (stream-actions stream)))
      (integrate (propagate-action-to-stream action remote)))
   (stream-reset stream))

(defmethod propagate-action-to-stream (action (stream <remote-stream>))
   (remote-call (stream-host stream) incorporate-action action))

(defmethod incorporate-action (action <edit-action>)
   (if (compatible-p action) (locally-perform-action action)
      (aggregate action)))
```

FIGURE 5.4: Supporting multi-synchronous activity.

features of MOP-based programming were discussed in Chapters 3 and 4. Now that some Prospero examples have been presented, a number of specific aspects are worthy of note.

First, Prospero provides default behaviours which embody mechanisms for collaborative data management. This is what toolkits do and so, in this respect, Prospero is not particularly different from other toolkits (although the detail of Prospero's management strategies differs from those of other toolkits). Second, and critically, Prospero structures these mechanisms in an object-oriented framework and reveals elements of this framework to applications as a means to introspection and intercession. Prospero, then, provides two, orthogonal interfaces to the functionality of its collaboration support mechanisms. The first, *base-level* interface provides facilities which clients use to *create* collaborative applications. The second, *meta-level* interface allows internal functionality to be *specialised* to the needs of particular applications. Design decisions are not hidden behind traditional abstraction barriers but are open to manipulation, so the toolkit can support a wider range of application requirements than would otherwise be possible.

5.8 Summary

Managing the consistency of distributed data is a critical issue for many collaborative systems. However, the interactive nature of CSCW systems means that many techniques which might be adopted from other areas of distributed systems engineering are not appropriate. Even when they can be used, their implications often limit them to a restricted set of applications and so they are not suitable for use in a toolkit to support a wide range of applications.

Distributed data management in CSCW needs to reflect the practicalities of collaborative work and group interaction. Unfortunately, most traditional mechanisms are oriented around preventing conflicts of action from occurring, which in turn forces the introduction of constraints upon individual action in a collaborative setting.

Prospero embodies an alternative approach. Rather than creating the illusion of a single stream of activity, it is based on divergence and synchronisation between multiple, parallel streams. This approach focuses on the resolution of conflicts, rather than on their prevention. The divergence and synchronisation strategy is particularly suited to CSCW applications, and, as a *specialisable* model, it can be used as flexible basis for development. In an Open Implementation framework, it allows applications to become involved in the data distribution aspects of toolkit behaviour, so that Prospero's procedures can be tailored to the particular needs of the application, and reflect the interactional patterns of the domain.

So far, however, only half of the picture has been presented. This chapter has focussed on the divergence model and shown how divergence and synchronisation can be used to manage distributed data in collaborative work. The other side of the coin is the means by which Prospero can constrain divergence to enhance collaboration. In just the same way as with divergence and synchronisation, this will begin with the semantics of applications and look for ways in which they can be usefully encoded and exploited at the toolkit level, using representations called consistency guarantees.

Chapter 6: Consistency Management and Consistency Guarantees

6.1 Introduction

Chapter 5 introduced and demonstrated the first half of Prospero's integrated model of distributed data management and consistency control. Divergence builds explicitly on a model of parallel, ongoing streams of activity, corresponding to the simultaneous actions of individual group members. Synchronisation between divergent views of shared data occurs periodically, at intervals from milliseconds to days, corresponding to the pace of group interaction. The Open Implementation framework which Prospero provides not only allows application developers to use divergence and synchronisation to manage distributed data in applications, but also enables them to specify new data distribution strategies in terms of the divergence/synchronisation model, incorporating these specialisations into the toolkit.

As was explained, the divergence model attempts to synchronise data, rather than to unify the actions of multiple participants into a single stream of activity. However, the process of synchronisation can be both simplified and enhanced in two ways: first, by allowing application control over the types of consistency to be achieved, and second, by providing some way of constraining or characterising divergence.

This chapter introduces the mechanisms Prospero provides to support consistency management. In section 6.2, I will outline the basic problem and two basic mechanisms. The first—*variable consistency*—is a useful technique which can be used in applications so that user activity can continue in the face of "hard" synchronisation problems. The second—*consistency guarantees*—generalises the traditional notion of lock-based exclusion and, like the divergence/synchronisation model itself, sets up the framework in which application developers can tailor and extend the toolkit. Section 6.3 compares consistency guarantees to related approaches, particularly in distributed database research. In section 6.4, I will show how the consistency mechanisms are embodied in Prospero, before going on to look at some examples of their use in section 6.5.

6.2 Constraining Divergence—Two Techniques

The model of continual divergence and synchronisation, introduced in Chapter 5, can be used to capture and describe the behaviour of a wide range of CSCW systems. More importantly, as the examples illustrated, it also provides the basis of an *implementational* description of a variety of data management mechanisms, both familiar and novel. Conceptually, the basic notions of divergence and synchronisation map onto implementational structures very well and provide a framework in which data replication and distribution models can be encoded; and this implementational use of the general model is the basic means to exploit the metalevel approach in Prospero.

However, before we can apply the divergence model practically, there is a problem to be addressed. The divergence model *per se* makes no commitment to the nature or extent of the divergence. Arbitrary actions can be performed over the data during the divergence phase, which can last arbitrarily long before synchronisation.

However, the longer two streams of activity remain active but unsynchronised, the greater is their potential divergence and, in turn, the more complex it becomes to resolve conflicts at synchronisation-time. Indeed, this model cannot guarantee that the system will *ever* be able to resolve two arbitrary streams into a single, coherent view of the data store. Essentially, unconstrained divergence leads to arbitrarily complex synchronisation; and that can be a practical inconvenience, to say the least.

However, there are clearly approaches which can be taken to address this problem. After all, if the divergence model can be used to characterise the data management strategies of other systems, then it should also be able to model the techniques which those other systems use to avoid these problems. Prospero extends the basic divergence model introduced in Chapter 5 with two techniques for constraining and managing divergence between streams—variable consistency and consistency guarantees.

6.2.1 Variable Consistency

The first approach, which was foreshadowed in the third example at the end of the previous chapter, is quite straightforward. Just as, earlier, we used divergence and synchronisation to express application-specific strategies for data management, so we can also exploit application-specific models of consistency. The application can become involved in not only the definition of consistency which the toolkit uses, but also in a process of selecting between degrees of consistency, and strategies for managing them. Like data distribution and replication, data consistency is an issue for the application, not for the toolkit.

In particular, Prospero distinguishes between two forms of consistency, labelled *syntactic* and *semantic*. By "semantic" consistency, I mean that the data store contains no inconsistencies from the perspective of the application domain. The data is fit for its intended purpose; textual data can be read and understood, spreadsheet data can be computed, etc. This is the conventional, intuitive form of consistency in most collaborative and distributed systems. When traditional CSCW systems deal with consistency, they deal with semantic consistency.

Appeal to "syntactic" consistency, on the other hand, allows for semantic inconsistencies, but ensures that the data store is *structurally* sound, so that some kind of activity can continue. Achieving syntactic consistency implies only that the two streams now share a common view of the data. It does not guarantee, however, that content is now in its final form. Typically, achieving syntactic consistency means doing the minimum work necessary to achieve a common view of the data space.

6.2.1.1 An Example

Consider a multi-user text editor, supporting a collaboration between two authors, A and B. At some point, during a period of divergence, each author has introduced some changes to the same paragraph. When the streams corresponding to each author are synchronised, the collaborative system must find some ways to make their views of the data store consistent.

In some cases, the changes which the authors have introduced may be quite minor. Perhaps the changes do not overlap, or perhaps they are easily composable. Many small edits—correcting typos, or making minor adjustments—are of this form. In these cases, the system can straight-forwardly combine the changes which A and B have introduced, and present each of them with a new version of the paragraph which incorporates their joint work. However, there is no guarantee that this can be achieved. In the face of more extensive changes, there may be no way to merge the edit actions. In the worst case, perhaps, both A and B have completely rewritten the paragraph. The two new pieces of text bear no strong relationship to each other, and yet the collaborative system must somehow "synchronise" A and B's views of the data store.

One simple approach to making the state (semantically) consistent would be to resolve the conflict by simply selecting one paragraph or other for the final text, and making it available to each author. The selection could be done on any grounds; most likely, the more-recently authored paragraph would "win" and replace the other. So, if the system were to discard the earlier of the two paragraphs, and incorporate the newer text into the shared data, then it would be preserving semantic consistency. (Clearly, consistency does not imply "correctness".)

However, this "lossy" approach is not necessarily the best suited to the needs of collaborating authors, even though the synchronisation procedure is straight-forward.

An alternative mechanism would be to retain *both* the paragraphs within a structure which flags this as a conflict which the system cannot resolve—essentially preserving the text for the authors to sort out later. This approach preserves syntactic (structural) consistency, enabling further work by the collaborators while not losing any data. As it happens, it also mirrors the behaviour of some collaborative writers using single-user tools (Beck and Bellotti, 1993). By only preserving syntactic consistency in some cases, rather than semantic consistency, a divergence-based system can achieve synchronisation more often, and continue operation in the face of potential problems. Consistency from the users' perspective is often not the same as consistency from that of the system.

6.2.2 Consistency Guarantees

The second technique, which will be the focus of the rest of this chapter, is to enhance consistency management by allowing the toolkit to exploit application semantics. This requires some explanation, since Prospero is designed independently of specific applications; how can application semantics be used in the toolkit mechanisms?

The solution to this conundrum lies in the *way* in which the toolkit is general. Although the notion of divergence and synchronisation, as a basis for managing a collaborative data store, is a general one, it is not *realised* as a generality; instead, it emerges as it is applied in *specific* applications. It is general (rather than particular), but concrete (rather than abstract); and the generality is made particular when it is used to implement a particular application, using the general structures in a particular way. In turn, these particular solutions do more than simply particularise the mechanisms of the toolkit, but also draw in the details of the application domain.

In other words, in any given case the application developer can employ specialist knowledge of the application semantics to describe locally-effective techniques for synchronisation. The application's specific patterns of collaboration over structured information offer the opportunity to define more effective synchronisation strategies; a programmer is better able to describe synchronisation of spreadsheet data than synchronisation of "data". Chapter 4 showed that this is a feature of the Open Implementation. The abstractions provided in the toolkit are specifically designed to be specialised in a range of circumstances, drawing on more detailed information available in particular settings.

In the examples in the previous chapter, we saw how the synchronisation process could be defined specifically for different applications, and in particular, the third example shows the use of aggregation (to achieve syntactic consistency) in an application situation which made it acceptable.

However, while using application-specific synchronisation might *postpone* some of the problems of unbounded divergence, the problems remain with us. So far, we have improved the synchronisation process, but still not done anything to constrain divergence. We need to go further; and so Prospero introduces the notion of *consistency guarantees* as a control for the divergence process. However, before outlining Prospero's approach in more detail, I will begin with the mechanism they generalise—locks.

6.2.2.1 Constraining Divergence with Locks

As discussed in Chapter 5, the most obvious traditional mechanism for constraining divergence (or, more accurately, for avoiding it altogether) is *locking*. Locking is widely used in current CSCW systems. Implicitly or explicitly, a user obtains a "lock" for some or all of the data store. Since update access is restricted to clients holding a current lock, the availability of locks controls the emergence of divergence; and since, in typical configurations, only one client can hold a lock on a given piece of data at any time, divergence is avoided. This sort of locking behaviour can also be exhibited by systems in which locks don't appear explicitly in the interface; floor-control algorithms and other forms of asynchronous access are also particular cases of the general locking approach.

A wide range of locking strategies exist, varying in how locks are requested, obtained, granted and relinquished, what kinds of operations require locks, and the granularity of data units controlled by a single lock. However, the basic pattern remains the same, and so do the basic problems of locking for CSCW applications. Locking is a *pessimistic* concurrency strategy; on the assumption that any conflict could be damaging, it prevents conflict arising in the first place. Locking restricts activity on the data store, and hence restricts the activity of users, due to the interaction of application, infrastructure and interface discussed by Greenberg and Marwood (1992).

The pessimistic strategy of traditional locking is quite appropriate in many applications, to avoid the danger of conflict and potential inconsistency. For applications in which data integrity is critical, and intra-group interactivity low—such as collaborative software development—locking strategies (such as the check-out model) can be valuable, appropriate and effective. In other applications, though, strict locking mechanisms can interfere with group interaction. Some systems, such as the ShrEdit shared text editor (McGuffin and Olson, 1992) use *implicit* locks, which are silently obtained and released in the course of editing activity, to reduce the level of interference and overhead. However, the locking strategy is still visible to the group through the effect it has on the interface, even in cases where working activity would not result in conflict or inconsistency (Dourish and Bellotti, 1992). In the case of even less structured, free-form data collaboration such as a shared whiteboard, the interactionaloverheadeven the interactional overhead of implicit locking becomes unwieldy.

This potential for inflexibility makes pure locking an inappropriate model in Prospero, since the commitment to pessimistic concurrency control undermines the different requirements of different applications. The toolkit must embody more flexible mechanisms which can be adapted or appropriated for a range of application needs and interactional styles. Clearly, something more flexible than locking—even when supported by a range of strategies—is needed.

6.2.2.2 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. Locking is essentially a means by which a client[1] receives some guarantee of future consistency in exchange for a description of the client's future activity; the client commits to restrict edit activity to the locked area, and the server commits to achieving consistency when the edit is complete, by locking out other users. So this generalisation is the first principle of consistency guarantees: regard locks as *guarantees of achievable consistency*, given in exchange for *promises about future activity*.

Immediately, this view has a number of interesting implications.

First, there's clearly a wide range of such guarantees which could be made. When we think in terms of guarantees of consistency, then we can consider distinguishing between different *degrees* of consistency, and the fact that a guarantee may only hold for limited consistency (in the worst case, perhaps, just syntactic consistency). Determining the achievable level of consistency is the responsibility of the server, based on currently-issued promises and the information about future activity which the client provides. The nature of these client "promises" will be discussed in more detail shortly; for now, though, it is enough to say that they are characterisations of expected behaviour, such as whether the client will simply read data, write new data but not delete anything current, delete or modify existing data, and so forth. Similarly, they might be more or less restricted to particular areas of the workspace (that is, they have variable *extent*).

Second, the promises could vary in specificity and detail, just as the guarantees can vary. The level of specificity of a client promise, or the region over which it extends, might vary from application to application, from client to client, or from moment to moment, depending on the immediate circumstances of the collaboration. By defin-

1. Although I use the terms *client* and *server*, these mechanisms also apply to peer-to-peer structures. In fact, Prospero uses a peer-to-peer model.

ing the model in terms of promises and guarantees, rather than simply in terms of locks, we gain the ability to exploit this openness within the toolkit and in particular applications.

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 becomes obvious that this is a *negotiation*; a client may make increasingly restrictive promises in exchange for increasingly strong guarantees of consistency.

The use of this sort of mechanism allows better interleaving of activity. From the server side, having more details of future activity means that better decisions can be made about which actions can be simultaneously performed by multiple users. From the client side, the ability to accept weaker guarantees than locks may allow activity to proceed where otherwise it would be blocked. This flexible interleaving retains the important *predictive* element of locking—that is, the client still makes “up-front” promises of future activity which give the server a better picture of the extent of future divergence and so enable more informed decision-making.

6.2.2.3 Breaking a Promise

This promise/guarantee generalisation still suffers one of the major problems with the locking approach applied to CSCW. Since it states that divergence is preceded by a description of expected activities, then the possibility of opportunistic activity is still restricted. This was raised earlier as a criticism of traditional locking mechanisms, which interfere with the way in which collaborative work proceeds naturalistically. Obviously a redesign should address this problem.

The second principle of the consistency guarantees approach is introduced to deal with this: *a client can break a promise, in which case the server is no longer held to its guarantee*. So the characterisation of future activity which a client makes—its promise—may not be binding; when the time comes, the client (or the user) may actually do something else. However, in this case, the server can no longer be held to the guarantee it made of the level of consistency which can be achieved. The guarantee is only honoured when the promise is kept.

With this second principle in place, the consistency guarantee mechanism provides more direct support for opportunistic working styles, as well as allowing for the multi-synchronous applications introduced in Chapter 5. Just as in naturalistic work, stepping outside previously-agreed lines is not prevented; but the mechanism provides stronger guarantees when used cooperatively by both client and server. Of course, the user need not (often, *should* not) be exposed to this complexity and unpredictability. In Prospero, these facilities are provided so that they can be appropriately deployed (or not) by an application developer. The developer might choose *not* to exploit the second principle *in a given application*, where application requirements or usage patterns would make it inappropriate. Examples might include cases where the resulting conflicts may be too difficult to synchronise later, or where loss of integrity in the data-store would be unacceptable. 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.

6.3 Consistency and Concurrency in Database Research

The variable consistency mechanism outlined in section 6.2.1 used knowledge of application semantics to specialise and improve the synchronisation process. Essentially, the consistency guarantee approach outlined in section 6.2.2 uses knowledge of application semantics—and the semantics of particular operations—to increase the *opportunities* for concurrency and parallel activity. Perhaps unsurprisingly, similar approaches have been explored in database design, since database management systems also involve multi-user activity over shared and perhaps replicated data. Barghouti and Kaiser (1991) provide a comprehensive survey of advanced concurrency control techniques. However, since databases tend to hide the activities of multiple parties from each other (preserving the illusion of sole access to a system), the primary (although not exclusive) focus of the database community has been on using concurrency to improve performance rather than to open up data models for collaboration. Two aspects of database research are particularly related to the consistency guarantees approach: *semantics-based concurrency* and *application-specific conflict resolution*.

6.3.1 Semantics-Based Concurrency

Database systems use a transaction model to partition the instruction stream. Transactions provide serialisation and atomicity. However, transactions might be executed in parallel or interleaved, without interfering with these properties, if the system can detect that there is no conflict between them. The interaction-time and response characteristics of database systems are frequently such that the calculation of appropriate serialisation orders for transaction streams has no significant impact on interactive performance. However, shared data stores supporting interactive collaborative systems 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 must 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 or interleaved 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 which issues a read request but doesn't use that result as part of a later computation (or does, but is robust to particular changes). It could, quite safely, be executed in parallel with another which writes that same data. Although no actual conflict would occur, conventional transaction systems would signal a *read/write* conflict, and so 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, applied before commit-time to validate transaction schedules. His approach uses predefined sets of conflicting operations, derived from the data type specifications. Farrag and Oszu (1989) exploit operation semantics by introducing a breakpoint mechanism into transactions, producing transaction schedules in which semantically-safe transaction interleavings are allowed.

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

6.3.2 Application-Specific Conflict Resolution

A second approach from database research which is relevant to the consistency guarantees mechanism is the use of application-specific conflict resolution. The Bayou system, under development at Xerox PARC, is a replicated database system for mobile computers, which are frequently active but disconnected from their peers. It provides a mechanism by which client applications can become involved in the resolution of database update conflict which can occur with replicated, partially-disconnected databases (Demers et al., 1994).

Bayou write operations can include *mergeprocs*—segments of code which are interpreted within the database system and provide application-specific management of conflicts. For example, in a meeting scheduling application, a write (carrying a record of a scheduled meeting) might be accompanied with code which 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) which 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 the way in which infrastructure support is configured to their particular needs.

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

6.4 Encoding Promises and Guarantees

The use of activity descriptions and consistency guarantees, as outlined above, provides a framework in which the semantics of operations and applications can be used to improve concurrency management for collaborative work. Before we can go on to look at some examples of these techniques in use, however, the issue of representation must be addressed. How can Prospero represent and encode the semantic properties on which consistency guarantees are based?

6.4.1 Semantics-Free Semantics

The primary role of the semantic descriptions which are the basis of this mechanism is to provide a *point of 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—the system's ability to identify and subsequently recognise semantic properties—rather than on a detailed, structured semantic account of user-level operations. So while the properties on which we would like to base our descriptions are *semantic* properties, the descriptions themselves do not have to *have* semantics. Application programmers need a way to refer to semantic properties, but not a language of semantics. It is sufficient to be able to distinguish and recognise semantic property `foo`, without having to give an account of what `foo` *means*.

This simplifies the problem immensely, by turning it from a *description* problem into a *naming* problem. Since the particular semantic properties which are useful in managing concurrency are entirely application-specific, they are named—for the purpose of coordination—by the application developer. What's required of Prospero, then, is the means to name them, to associate them with particular operations, and subsequently to recognise them in the process of managing promises and synchronising streams.

6.4.2 Class-based Encoding

Prospero accomplishes this through *class-based encoding*. The semantic properties for an application are named as classes in an object-oriented framework. Particular operations are represented explicitly as command objects (Berlage, 1994); that is, invocations of any operation are represented explicitly as objects within the system. Each instance of a command object represents a particular invocation, along with any relevant parameters and contextual information. Command objects multiply inherit from the classes which represent their semantic properties; the objects themselves represent the commands, while the class hierarchy encode the semantic properties of the commands' actions.

The use of explicit command objects is, in itself, a useful mechanism for representing sequences of actions and arriving at appropriate mechanisms for resolving conflicts which might arise; but encoding semantic properties in the inheritance structure of the command objects yields two particular benefits for the problems which Prospero seeks to address. First, the mechanism is inherently extensible. The application developer can create new semantic properties from existing ones within the same mechanism that 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. In the application, behaviours related to different semantic properties (or combinations of them) are written separately as methods specialised on the relevant classes, rather than in a complex, monolithic synchronisation handler. This allows the programmer to rely on the object system's dynamic dispatch mechanism to match semantic properties (classes) to associated behaviours (methods) for particular command objects. In turn, this encourages a modular separation of code segments based on the semantic properties themselves, rather than on the procedural resolution of those properties.

6.5 Using Consistency Guarantees

To provide a more detailed illustration of the use of consistency guarantees in collaborative applications, this section presents two more extended examples, along with the framework Lisp code which implements them. As with the examples in the last chapter, these are small fragments intended to illustrate the general principles; for explanatory purposes it is sometimes necessary to include more code, and sometimes less, than is actually needed with Prospero. Similarly, the focus here is simply on those aspects of Prospero concerned with consistency guarantees; the examples illustrate the use and manipulation of the "promise" and "guarantee" abstractions. Chapter 7 will present more detailed examples of using the Prospero implementation as a whole. Primarily, the examples here show how application-specific semantic properties can be used within a toolkit framework to manage concurrency. Clearly, semantically-informed concurrency control could be used in hand-coded applications, on a case-by-case basis; the issue here is how these application-specific features can be exploited within a generalised toolkit.

6.5.1 A Shared Bibliographical Database

A simple example of an application whose collaborative performance can be enhanced by exploiting semantic information is a shared database for bibliographical information. The key property which we want to exploit in this example is that updates to the database are normally non-destructive. Updates will typically add new information, rather than removing or changing information already present. Simultaneous appends are much less likely to cause conflicts that simultaneous revisions, but would be prevented by standard "strong" locks.

```
(let ((guarantee (request (remote-stream) <read> <safe-write>)))
  ;; ... editing actions ...
  (synchronise-with-guarantee (my-stream) (remote-stream) guarantee))

(defmethod grant-guarantee (stream (operation <safe-write>))
  ;; ... always ok ...
  (let ((guarantee (construct-guarantee <auto-consistent> stream)))
    (push guarantee (stream-guarantees (local-stream))))

(defmethod grant-guarantee (stream (action <write>))
  ;; ... sometimes restricted ....
  (if (compatible-action <auto-consistent> <write>)
      (construct-guarantee <refused-guarantee>)
    (push (construct-guarantee <auto-consistent> stream)
          (stream-guarantees (local-stream))))

(defmethod grant-guarantee (stream object (action <read>))
  ;; ... always ok ...
```

FIGURE 6.1: Methods defining access to the shared bibliographical database..

We can take advantage of this feature by introducing a class of actions which correspond to non-destructive writes. A standard access mode for the collaborative database during disconnected operation, then, would be the combination of read and non-destructive write, and could be encoded in Prospero as shown in figure 6.1.

So, in the initial code fragment, the editing actions are bracketed by a request/synchronisation pair. The generic function `request` requests[2] a guarantee for the local data stream, specifying that the expected behaviours will be of types `<read>` and `<safe-write>` (non-destructive writes). The guarantee that it receives is subsequently used as part of the synchronisation process.

As far as making the guarantee is concerned, the application adopts the policy that, like read capabilities, safe-write capabilities can be granted to multiple clients at a time.

The generic function `grant-guarantee` deals with the server side of the transaction. The programmer has defined methods to handle the specific cases here where application semantics are to be used. So, the server grants guarantees for read operations and for safe-write operations (although they receive different levels of consistency, which are also class-encoded). However, for general write operations, a guarantee is only issued when no other guarantee has been granted to another writing client. Guarantees are recorded so that they may be used as the basis of later decision-making, as well as for synchronisation purposes later.

If this application were coded using only strong locks, then any update activity would lock out other users. By using consistency guarantees, the programmer can specialise the locking mechanism to accommodate the particular semantics of this application, and so arrange to increase concurrent cooperative work.

6.5.2 Collaborative Text Editing

The previous example showed the selective granting of consistency guarantees based on characterisations of expected behaviour—the semantics of activity during the period of divergence. This second example illustrates the use of semantic properties in synchronisation. Consider a collaborative text editing system in which multiple authors work on a single document, obtaining guarantees at the level of paragraphs or sections. As in the previous example, the guarantees obtained before divergence are passed along at synchronisation-time. At this point, the guarantee must be examined to verify that only promised actions were performed.

```
(defmethod synchronise-with-guarantee (stream action-list guarantee)
  (let ((promise (guarantee-promise (find-guarantee guarantee))))
    (if (valid? action-list (promise-properties promise))
        (simple-synchronise stream action-list)
      (salvage-synchronise stream action-list))))

(defmethod simple-synchronise ((stream <stream>) action-list)
  (dolist (action action-list)
    (synchronise-action stream action *stream*)))

(defmethod salvage-synchronise-action ((stream <stream>) (action <action>))
  (if (action-conflict? action (stream-history *stream* :relative-to stream))
      ;; definite conflict
      (syntactic-locally-perform-action action)
    (if (guarantee-conflict? action *guarantee-table*)
        ;; potential conflict
```

FIGURE 6.2: Methods for synchronisation of the collaborative writing example.

2. In this example, error conditions—and in particular, the refusal of a guarantee—have been omitted for clarity.

So there are two cases (distinguished in figure 6.2 by the predicate `valid?`). In the first case, the actions performed by the client are those which were given in the promise. The promise has been upheld. In this case, synchronisation should be straightforward since the server was in a position to know what actions were expected beforehand. At this point, then, the type of the stream can be used to determine the appropriate synchronisation method (as in the examples presented in Chapter 5).

In the second case, however, the system detects that the actions performed do not match those listed in the promise. The client has broken its promise. There are various ways in which this situation could have arisen; and, critically, since a number of them are important features of naturalistic work practice, we would like to provide as much support for them as possible. The generic function `salvage-synchronise` is called to provide fall-back synchronisation. In this case, salvaging involves stepping through the actions attempting to apply them one-by-one. By comparing the classes of the operations (that is, their semantic characterisation) with the activities of other streams, their compatibility can be determined. Actions compatible with activities performed (and guarantees granted) since the divergence point can be applied directly; other actions must be processed specially.

Here there are three different means of applying potentially conflicting actions locally. In the case of no conflicts we can use `locally-perform-action` which incorporates the remote actions into the local data store. However, there are two cases of potential conflict. The first is where the remote operation conflicts with an action arising in another stream. In this case, the application reverts to syntactic consistency by calling `syntactic-locally-perform-action`, which applies the action preserving syntactic, rather than semantic, consistency. In the second case, the remote action conflicts with a guarantee which has since been made to some other stream. In this case, there are clearly various things that could be done; the application developer here chooses to apply the operation tentatively, although it may be necessary, later, to undo this and move to syntactic consistency instead. Note that this decision—to maintain consistency at the expense of actions under broken promises—is a decision which the application developer, rather than the toolkit developer, can make in particular circumstances. The default structures of the toolkit may provide frameworks around such decisions, but they can be revised to suit particular application needs.

6.6 Summary

Chapter 5 presented a model of distributed data management which arose, in part, from the requirement to address the interactional component which distinguishes CSCW applications from other distributed systems. However, control over the data distribution is only one half of the puzzle. The same distinction (the distribution of the interface as well as data and application and its implications) must be also taken into account when considering concurrency control in collaborative systems. Traditional algorithms typically maintain consistency by restricting concurrency. However, just as with data distribution, this approach is unsatisfactory in general, as it often interferes with the flexible management of group activity.

The semantics of specific applications can be exploited to increase concurrency while maintaining adequate consistency in a collaborative data store. By looking in detail at the semantic properties of particular actions in a CSCW system, we can find operations which can be performed in parallel without leading to inconsistency. Prospero exploits this by allowing application developers to encode and use specific semantic properties of the application domain and the system's operations to enhance support for concurrency.

This chapter has introduced the notion of *consistency guarantees* as a parallel technique to increase the effectiveness of the explicit semantics approach. Essentially, consistency guarantees generalise locks, regarding them as guarantees of some level of achievable consistency. This more flexible interpretation allows applications to balance freedom of action against eventual consistency as appropriate to the particular circumstances of use. In addition, by allowing clients to break their promises of future activity (and hence not holding the server to its guarantee of later consistency), and by falling back to a model of syntactic consistency when necessary, we can support opportunistic work without completely abandoning the synchronisation of parallel activities.

Prospero's goals are two-fold: first, to be able to create applications which more naturally support the flexibility of everyday activity; and second, to achieve a wider scope than traditional toolkits by finding ways to shed commitments to particular styles of application design and interaction support. Both in the divergence and synchronisation strategy presented in the previous chapter, and in the consistency strategies presented in this chapter, a common approach has helped to address each of these goals. Rather than trying to make the system sufficiently general that it is independent of the semantics of particular applications—and thereby unable to capitalise upon them—Prospero uses the Open Implementation approach to allow the specifics of application semantics to be integrated with toolkit facilities, so that those areas of concern (such as data distribution and concurrency control) typically associated with toolkit functionality can not only exploit application specifics, but can be tailored to application needs.

Chapter 7:
Using Prospero: Application Examples

7.1 Introduction

The last two chapters have introduced the main abstractions and mechanisms which Prospero provides for creating collaborative applications, as well as the means by which these can be specialised to meet the needs of specific situations. Small examples were used to illustrate how these were embodied in the toolkit and used in practice. The purpose of this chapter is to pull together the various ideas encountered earlier by presenting longer, more detailed examples which also serve to illustrate the range of behaviours which the toolkit can support.

7.2 Application Structure

While applications are free to select and use toolkit features in whichever ways are appropriate, there is a general schema which characterises most applications. The scheme, illustrated in figure 7.1, has two sections—initialisation and general running.

The initialisation phase sets up and initialises the various structures which are used while the application is running. There are three primary areas of responsibility. The first is the description of the semantic properties which will be employed by the consistency guarantee mechanism described in Chapter 6. This will establish a set of properties by which the actions of various streams can be classified and described, as well as the range of levels of consistency which can be achieved for various sets of potential operations. The second is the stream or streams which will be associated with a given application instance. While there will commonly be a single stream per running process, corresponding to the actions of the user which it represents within the system, any given process may actually be responsible for the management of multiple streams (e.g. if it is a user-less "server" process, if the user is engaged in multiple tasks, or if a user's activity is separated into multiple streams, perhaps for data and interface). New streams may, of course, be established while the application is running; but since the stream is our primary unit of dissection and analysis, the establishment of a stream is always seen as preceding any further operation. Finally, the collaborative session must be established. As discussed in Chapter 4, session management is not one of the areas of toolkit functionality which this work has addressed. It is handled through other means (in general, probably through coordination with another server) and so we will not deal with this in any particular detail; however, before the application becomes operational, it must locate the other streams with which it is associated, and with which it will synchronise.

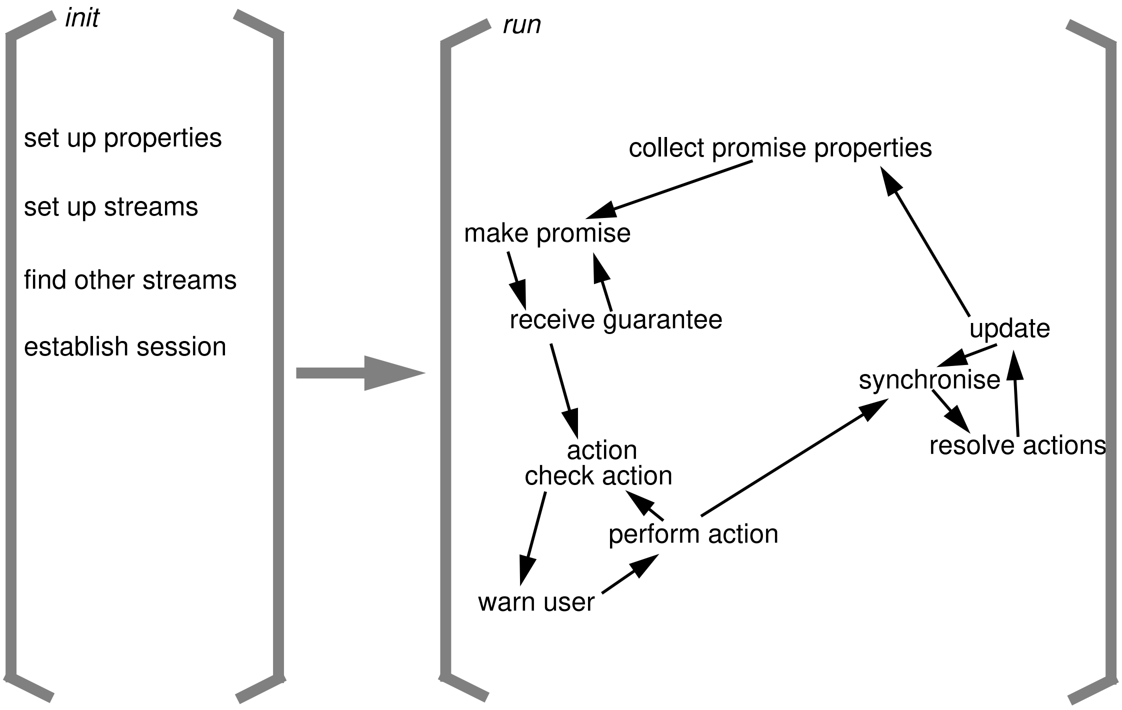


FIGURE 7.1: The schmatic structure of applications written in Prospero.

Once these initialisation tasks have been completed, the application is ready to begin general operation. This is structured as a set of nested cycles. In general, it proceeds as follows. First, for a single cycle, a set of properties describing likely future operations must be collected. This may be done in a number of ways—from explicit user information, on the basis of available operations at this moment, heuristically from recent activities, etc. Once these properties have been collected, the application enters the promise cycle. The properties are collected together to form a promise, which is then submitted to the "guarantee authority"—usually a peer stream. As described in Chapter 6, this authority returns a guarantee of the level of consistency achievable at a future synchronisation point on the basis of the actions described in the promise. There may be a cycle at this point, as the application revises its guarantee in order to receive the promise of a level of consistency which it finds acceptable—that is, the application and the guarantee authority may negotiate a level of activity and consistency suitable for the purposes of either.

When the guarantee is established, the application enters the main action cycle, which is the divergence phase from the stream-oriented view presented in Chapter 5. Like the promise phase, this is a loop. First, the user performs some action at the interface. This action is (optionally) compared against the promise properties collected earlier to determine whether or not it is in keeping with the promise on the strength of which the guarantee was obtained. The application may chose to warn the user about conflicts here, or (again, if the programmer wishes) may refuse operations which do not conform to the promise. If the action is acceptable (either within or without the parameters of the promise), it is performed[1] locally, and the application loops to accept more user input.

At some point—triggered, as we saw in Chapter 5, either by internal action or an external event—the divergence phase ends and synchronisation takes place. The various actions which the user has performed over shared data during the divergence phase are resolved against other streams, which may also have accumulated their own actions during this time. The system works to achieve a level of consistency or stability. Action streams are resolved against each other, and the local state is updated to reflect the newly established shared consistent state.

1. Actually, it may simply be noted, rather than performed; but for the moment, we will regard these as equivalent from the perspective of the local stream.

7.2.1 Specialisation through Subclassing

As demonstrated in previous chapters (and to be shown further here), Prospero allows applications to become involved in the implementation of the infrastructure which supports them, so that toolkit structures can be adapted to the needs of particular applications and situations. This is done by subclassing and specialising revealed toolkit features. Subclassing allows the application programmer to create new, more specific forms of toolkit structures; and since these generic structures are related to each other through generic function invocations, new methods can be defined on these to introduce new behaviours to the system.

CLOS's extensive method combination facilities make this approach particularly attractive. In most object-oriented languages, new methods defined for some class override the equivalent methods defined for classes further up the class hierarchy. However, CLOS also allows more-specific methods to invoke the less specific methods themselves. In addition, methods can be specified to run before, or after, the primary methods for a class; before- and after-methods are also subject to inheritance, according to different rules than primary methods (so that all the before- and after- methods for an object will be run, in a specified order, while only one primary method will typically be called). CLOS's method combination facilities allow application programmers to introduce new pieces of toolkit functionality very simply within the generic function framework specified by Prospero.

7.2.2 Configuring or Extending the Base Level

Before going on to look at Prospero's generic function framework, there is one particular feature of the use of the base/meta distinction in Prospero which is worth exploring. This aspect of Prospero's design represents a departure from earlier OI designs.

In Open Implementations, the separation of base and meta interfaces is normally organised around the distinction between *what* the client requires of the abstraction, and *how* the abstraction should go about providing (aspects of) that functionality. One way of thinking about this is that the base level sets the terms of the abstraction, while the meta level *configures* that abstraction appropriately for the needs of the client. The meta interface will typically deal in terms of different sorts of objects—those used (on some level) to realise the abstraction. In this way, Open Implementations "open up" the abstractions through the meta interface.

Prospero uses Open Implementation to the same end—that is, to allow applications to specialise toolkit facilities to their own needs. However, the base level object in Prospero—streams, promises, actions, etc.—are very abstract. They are quite distant from the implementation level and much closer to the application level. This in turn affects the way in which the metalevel works. Activity at the metalevel in Prospero largely specialises the base level with semantic features of the application domain (such as the properties of actions, or stream types); and once this has been done, those semantic features become available for base-level programming. In other words, we can think of this not so much as *configuring* the base level, but more as *extending* it. So it is not simply that the metalevel specialises the structures of the base level, and the implementation which lies behind it; but it specialises the *base level itself* to the needs of the application.

```
<stream>
  <local-stream>
  <remote-stream>

  (stream-name <stream>)
  (stream-host <stream>)

  (stream-actions <local-stream>) ==> <list> of <action>
  (stream-peers <local-stream>) ==> <list> of <remote-stream>
  (stream-promise <local-stream>) ==> <promise>
  (stream-guarantee <local-stream>) ==> <guarantee>

  (synchronise <local-stream> <remote-stream>)
  (synchronise-with-guarantee <local-stream> <remote-stream>
    <guarantee>)

<promise>
  <null-promise>
  <lock-promise>

  (make-promise %extent <action> <action> ...) ==> <promise>
  (respond-to-promise <promise>)
    (respond-to-promise-from-stream <promise> <remote-stream>)
    (acceptable-promise <promise> <local-stream>) ==> <guarantee>

  (promise-properties <promise>) ==> <list> of <action>

<guarantee>
  <null-guarantee>
  <full-guarantee>

  (guarantee-authority <guarantee>) ==> <remote-stream>
  (get-guarantee <remote-stream> <promise>) ==> <guarantee>
  (redeem-guarantee <remote-stream> <guarantee>)

<action>

  (add-action-to-stream <action> <stream>)
  (propagate-action-to-stream <action> <remote-stream>)

  (locally-perform-action <action>)
    (simple-synchronise <stream> %action-list)
    (tentative-locally-apply-action <action>)
    (syntactic-locally-apply-action <action>)
    (salvage-synchronise <stream> (<action>...))
    (salvage-synchronise-action <stream> <action>)
```

FIGURE 7.2: Prospero class structure and generic function framework.

.

7.3 The Generic Function Framework

The programmer interface to using and manipulating Prospero structures is defined by the generic function framework. In CLOS, generic functions play the role that messages take in Smalltalk, but within a functional context. Calling a generic function invokes CLOS's method combination mechanism to find appropriate methods which implement the generic function for the objects given as parameters to the generic function. Prospero's behaviour is defined not only in terms of the generic functions to which internal objects respond, but also the generic functions which will be used in their execution. These give programmers who wish to modify internal aspects of Prospero a finer grain of control.

Figure 7.2 details the class structure and generic function framework supplied by Prospero. The class structure details the classes and subclasses of objects over which Prospero acts. As in previous chapters, the convention is adopted that classes are lexically distinguished by angle brackets, and subclasses are indicated by indentation. The basic classes are `<stream>`, `<promise>`, `<guarantee>` and `<action>`. Along with the classes, figure 7.2 describes generic functions associated with them and, where relevant, other generic functions which are called in the course of executing them (again, detailed by indentation).

The class `<stream>` has two subclasses, `<local-stream>` and `<remote-stream>`, which represent streams local to or remote to the local host. General functions—for naming and locating streams—are defined on the stream `<class>`, while more detailed functions for examining and manipulating streams are defined on `<local-stream>`, since a host cannot manipulate streams on remote hosts. As implied by the stream model, user actions are associated with streams, not with hosts.

In addition to the basic class `<promise>`, Prospero also provides a subclass, `<lock-promise>`. This refers to the general form of promise associated with traditional locking mechanisms (that is, complete control over some region of the shared workspace).

Two subclasses of `<guarantee>` are also provided. The first, `<null-guarantee>`, is an explicit statement that no guarantee is made. The second, `<full-guarantee>`, is a total guarantee, such as might be given in a successful response to a `<lock-promise>`.

The most complex class structure is associated with the basic class `<action>`; this is also the area in which the most information must be supplied by an application programmer. It is through the refinement of `<action>` that the detailing of application semantics (necessary for the guarantee mechanism) is performed, as outlined previously in Chapter 6. Promises are constructed from actions, or (more commonly) are predefined.

Most of the generic functions and library functions outlined here have been seen, in one form or another, in the examples laid out in the previous chapters. The rest of this chapter will present and discuss two longer examples which show how they are used together, in full applications.

7.4 Sample Applications

This section presents two sample collaborative applications created using Prospero. These examples demonstrate the various facilities and principles which have been explored in the previous chapters, putting them together in larger, more detailed examples and demonstrating both the range of the toolkit and the styles of programming which it supports. The first example is a simple graphical editor; the second is a bug-tracking database.

The code for the examples is presented in Appendix A. The applications were written with CMU Common Lisp and the PCL implementation of CLOS, and use the Garnet toolkit to provide the user interface components. Important features to note when considering these examples include the relationship of application to collaboration code, the form of the link between the applications and Prospero and the use of previously-detailed mechanisms such as subclassing and refinement to express application needs.

7.4.1 Eureka, a Collaborative Polyline Editor

Section 1 of Appendix A contains a code listing of a synchronously-shared polyline[2] editor called Eureka (because "it's very simple"). In this section, I will discuss the code, show how it uses the Prospero mechanisms discussed previously, and highlight particular pieces.

The code for Eureka is divided between two files. The first, `eureka.lisp`, is presented in section 1.1 of Appendix A, and contains most of the Prospero-related functionality. The second, `ui.lisp`, is presented in section 1.2. It is largely concerned with managing the user interface, although it also has some important "glue" functions which relate interface action to Prospero functionality, as will be discussed.

Eureka uses roughly the same sort of mechanism as was shown in the "shdr" example in Chapter 5. However, somewhat less of the details of stream management are provided here, because they are actually internal parts of Prospero's functionality, which were included with the example code in Chapter 5 for completeness.

Each site maintains a local stream for the processing of locally-originated commands. This is a `<bounded-stream>`. Prospero provides bounded streams as a subclass of `<local-stream>`; they are local streams which have a fixed upper bound to the number of commands which can be executed before synchronisation will occur. So, a Eureka client will gather and record user action until the bound is reached, at which point synchronisation will take place.

The early parts of the file `eureka.lisp` correspond to the initialisation phase of the general application scheme outlined in section 7.2 (this chapter). The first particularly interesting component here is the creation of command object classes for various sorts of operation, in lines 14–24. In this segment, three classes are created which correspond to three different forms of action over shared objects (creating them, editing/deleting them, and moving them). These are all subclasses of the Prospero-supplied base class `<action>`, and each defines slots which can be used to record the details of specific executions of the commands. A set of functions (defined at lines 33–42) are set up as wrappers to create command objects with appropriate values, in response to the execution of commands in the user interface.

The code segment beginning at line 49 sets up the streams; the global `*local-stream*` is an instance of `<bounded-stream>`, as discussed above, and it is associated with instances of `<remote-stream>` which refer to peer clients on other hosts. Since this example uses the pre-defined behaviour of `<bounded-stream>`, this code is straightforward.

The code at the end of the file, however, at lines 78–84, is more interesting. For the classes of objects corresponding to user interface commands—the command object classes defined earlier—Prospero needs to know how to execute them (since local execution is one of the defined behaviours associated with adding an object to a bounded stream). The framework defines that this will be done through the generic function `locally-perform-action`, and so lines 78–84 set up methods for this generic function, specialised on the various command object classes defined earlier in the file. Garnet uses a "retained" object model, which maintains representations of all interface objects using KR (its object system). As a result, the interface representation *is* the shared workspace, and so the only behaviour we need to associate with the execution of these commands is the interface behaviour. So the methods defined on `locally-perform-action` for each class of command object call back to user interface glue functions which take command objects and perform the user interface actions. Clearly, of course, these user interface actions could be performed directly by `locally-perform-action`; but the separation introduced here helps maintain the clarity of the code by decoupling collaboration functionality from interface functionality.

The second file, `ui.lisp`, deals mainly with maintaining and controlling the user interface. As a result, the bulk of that file deals with the mechanisms and structures of the Garnet interface toolkit, rather than those of Pros-

---

2. A polyline is a connected sequence of line segments, essentially forming an unclosed polygon.

pero. Garnet details do not concern us here, although the interaction mechanism are relevant. Garnet uses the *interactors* model, in which interactive behaviours are reified as objects within the toolkit which are then declaratively associated with a combination of windows (or other areas of applicability) and conditions (under which they will become operative). So the calls to `create-instance` on lines 25, 32 and 39 of `ui.lisp` define interactors associated with the Eureka window.[3] Associated with each of these interactors is a callback function which is called when the interactor operates. The callback functions themselves are defined at the end of the file (line 66 onward), where there is more of interest from the perspective of this chapter.

The function `add-polyline` defined at line 66 is the callback function for the polyline creation interactor. It is called when the various points have been defined for a polyline, the interactor has completed, and the Garnet polyline object itself is to be created. In a traditional Garnet application, this function could then call `create-instance` to create a polyline object and associate it with the window. However, in this application, the callback function is a hook into the Prospero mechanism. Recall that the local performance of any operation will be associated, by Prospero, with action on the local stream. However, the role of `add-polyline`, in this case, is to recognise that an interaction has taken place and, rather than executing it, to create a corresponding command object which will be associated with the local stream. So `add-polyline` creates an object of the class `<object-create-object>`, and calls `add-action-to-stream` to associate it with the local stream.

At this point, internal Prospero mechanisms take over, processing the command object and so forth. At some point, however, Prospero will have to cause the local execution of the command object, through the generic function `locally-perform-action`. As we saw above, this was defined, for the various command objects process by this application, to call a glue function which would execute the associated user interface command. These functions are also defined in this final section of `ui.lisp`. The function at line 72, `ui-add-polyline-from-create-obj`, takes an instance of `<object-create-object>`, extracts the relevant information from it, and create a new Garnet polyline. So these two functions (`add-polyline` and `ui-add-polyline-from-create-obj`), taken together, form the glue between the user interface for a standard single-user application, and a Prospero-based multi-user application, by managing the communication between the two components. Similarly, the code from line 81 onwards defines appropriate pairs of glue functions for other actions, in precisely the same way.

There are two particular things to be observed in this example. The first is the declarative style by which collaboration behaviours are associated with interface behaviours. Individual and collaborative activity is managed in terms of the streams model, and inter-process communication is dealt with purely in terms of the synchronisation of streams. In this case, because we have been able to rely upon the default behaviour of `<bounded-stream>`, the application has not had to get involved in the synchronisation process at all. Changes to the synchronisation process—for instance, to the period of synchronisation, or to used timed, rather than bounded, streams—is accessible to the application (being directly associated, through generic dispatch, with the objects which the application is manipulating) but is encapsulated when not needed. The second feature is the localisation of Prospero-related functionality in a few areas. The impact on the core user interface code—which would have to be written, in pretty much the same way, for a single-user application—is minimal.

7.4.2 Bugspray, a Bug-Tracking Database

Section 2 of Appendix A presents the code for a second example application, a bug-tracking database called Bugspray. This example illustrates very different patterns of collaborative activity (and infrastructure support) than those demonstrated by Eureka, but supported within the same framework.

The scenario of use is a typical two-stage customer support operation. On the front line are telephone operators who receive calls from customers with product queries and problems. Specific problems are logged in the database, with information about the platform, product, etc. The database also holds records of known bugs, fixes

3. Garnet uses a private prototype-based object system, called KR; create-instance is KR's equivalent of CLOS's make-instance.

and workarounds, as well as a mechanism for sets of comments to be attached to problems, as a means of discussion. The second stage of database processing is the "stitching together" of these various records; associating problems with bugs, bugs with fixes and so on. This may be done by the telephone operators, but also by offline customer support staff, product staff, and so on. So there are three different sorts of processing going on over this database—the entry of new records, modification of existing records, and manipulations of the overall structure (by linking records together).

As with Eureka, the code of Bugspray is divided across a number of files, concerned with different aspects of the implementation. These files are presented as separate subsections of section 2 in Appendix A. The first, `bugs.lisp`, contains the core database functionality. `Ui.lisp` deals with maintaining the user interface and linking user interface actions into application functionality. `Common.lisp` contains elements common to both the server side and the client side; client and server specific functionality is in `client.lisp` and `server.lisp`. `Guarantees.lisp` provides the code which handles consistency guarantees. Finally, `flatten.lisp`, `dbio.lisp` and `misc.lisp` contain other supporting functions; they are included for completeness, but will not be of concern here.

Bugspray uses a client/server model in which the primary copy of the database resides on the server (although all or part of the database is cached at the client). The database consists of a set of records, instances of the classes defined in `bugs.lisp`. There are five sorts of records—problems (that is, problem reports), comments, bugs, fixes and workarounds. (For our purposes here, there is no difference between fixes and workarounds, and in fact they're related in the class hierarchy through their parent class, `<solution>`.) The record class definitions in `bugs.lisp` are normal CLOS definitions. Object slots hold either text strings (for platform, symptoms, comments, etc.) or "links"—pointers to other records, such as from a problem report to a comment upon it. In a single-user system, these might more normally be stored as direct pointers, but here, a linked record is referred to using the unique record identifier generated for it at definition time. Using these symbolic identifiers means that the application does not need to map between host-specific and host-independent representations when communicating with remote systems.

The file `ui.lisp` sets up a Garnet-based user interface, which creates a control window and dialogue boxes for the entry and manipulation of records. As with Eureka, the details of the user interface are not relevant here, being more concerned with the mechanism of Garnet than anything to do with Prospero. The interface allows records to be entered and linked together, as well as being used to control synchronisation. User interface components manipulate the database by calling the various methods on the record classes which were defined along with the record classes in `bugs.lisp`.

The interface allows users to switch between different access modes to the database—entry and linkage. The access mode determines the type of promise which is made. The function `ui-set-mode` at line 107 is called when a mode switch is made. The processing of promises will be explained subsequently.

Most of the Prospero-specific processing begins in `common.lisp`, setting up elements common to both client and server sides. In particular, lines 14–37 set up the action properties which are used by this application. These classes allow actions which operate over the content of the database (record data fields) and its structure (record link fields) to be distinguished. The immediate subclasses of `<action>` defined at lines 14–21 (`<append-action>`, `<structure-change-action>`, `<content-change-action>`, and `<no-change-action>`) are not application actions, but properties which those actions can have; application actions are then defined in terms of these. The existence of `<set-field-action>` separate from `<change-field-action>` denotes a special case, where the field that is set belongs to a newly-created, unsynchronised record. Since there is no change to synchronised content when this happens, this action inherits from `<no-change-action>`, not from `<content-change-action>`. The class `<change-field-action>` is reserved for changes over synchronised data.

`Common.lisp` then provides functions for setting up the streams. Whereas Eureka used bounded streams, Bugspray uses `<explicit-synch-stream>`, another predefined class in Prospero, which uses explicit synchronisation to control data sharing as in the check in/out example in Chapter 5.

Similarly to the actions, `guarantees.lisp` defines the promises and guarantees which might be made according to these actions. Promises are defined at lines 22–29; two classes of promise, `<structure-promise>` and `<content-promise>`, correspond to intents to make structural changes or content changes. A third promise, `<structure-change-promise>`, includes the possibility of making both. The guarantees which Bugspray uses are the two basic guarantees which Prospero offers as part of its default behaviour—`<full-guarantee>` and `<null-guarantee>`.

As well as defining the promises themselves, `guarantees.lisp` also provides code which determines compatibility between promises, on the basis of which guarantees are issued. This, again, is an example of a point where a programmer provides details of application semantics for use by the internal mechanisms of the toolkit. In this case, the application defines methods on the generic function `compatible-promises` which reflect their potential interactions. The interactions here are quite simple. Structural promises and content promises are compatible with each other, but not with themselves. This reflects the fact that multiple users might simultaneously update records (the entry/edit phase) and link them together (the comment/integrate phase), but that two users linking records or editing records at the same time may introduce incompatible changes. Three methods are defined for the generic function `compatible-promise`. The first (line 37) marks promises as compatible, by default, while the second two (lines 40 and 44) mark the exceptions. Note that, since `<structure-content-promise>` inherits from both `<structure-promise>` and `<content-promise>`, these methods will find it incompatible with any other promise. This exploits the declarative class-based encoding mechanism described in Chapter 6. The return value from compatible-promise can be a guarantee to be returned, but in this case we simply use true and false (t and nil), which are interpreted as `<full-promise>` and `<null-promise>`.

Next, `client.lisp` deals with the integration of Prospero mechanisms and client database manipulations. This is managed differently in Bugspray than it was in Eureka. Bugspray defines CLOS "after-methods" on the record manipulation generic functions to perform the Prospero behaviours. CLOS's method combination facility ensures that these methods are executed after the execution of any relevant primary method when a generic function is invoked. For instance, the method definition for `new-object` at line 34 of `client.lisp` is an after-method (designated by the keyword `:after`), so that calls to `new-record` will first cause the "primary" method at line 58 of `bugs.lisp` to be executed (adding the record to the local record table), and then call this after-method. `Client.lisp` uses these after-methods to create command objects which represent actions over Bugspray record objects, and then add them to the local stream. Other after methods (at lines 17 and 24 of `client.lisp`) are defined for other actions over records.

The function `request-guarantee` at line 73 makes a promise, based on the current mode; it is called by `ui-set-mode` in `ui.lisp` (line 107) whenever a mode change is made. The two modes—entry and linkage—determine what sort of promise is to be made. User interface functions (`ui-request-guarantee` at line 149 of `ui.lisp`) inform the user if the server refuses the desired guarantee due to concurrent activity by other users.

Finally, `server.lisp` deals with server-side functionality. Essentially, this simply deals with the server-side execution of client-initiated commands, by defining a number of methods on the generic function `locally-perform-action`, corresponding to the various types of actions which might arise.

The remaining files are included to complete the application, but are not of immediate interest or relevance here. `Flatten.lisp` contains functions which map from CLOS objects onto printable forms which can be sent over the WIRE remote function package in CMU Common Lisp, in which these applications were written. `Dbio.lisp` deals with loading and saving databases and caches to disk; and `misc.lisp` provides some ancillary support functions.

7.4.3 Bugspray in Use

Since Bugspray is a considerably larger application than the others presented in the past few chapters, this section will consider how it operates by stepping through how it works.

Most of the processing takes place in the client. The user sets an access mode to the database. On the basis of this, the client makes a promise to the server, characterising upcoming activity for that mode. This characterisation is compared against the promises which are currently offered to other clients. Compatibility is resolved using application-specific information supplied, in this case, by specialised methods for the generic function `compatible-promises`. In this example, this is the most significant specialisation of Prospero functionality necessary to support the application, and is only a few lines long.

A guarantee is returned, which is checked; the user is informed if the desired guarantee is not granted. The user can then enter, edit and link together records, from the graphical user interface. Action over the client-side records causes actions to be generated and added to the local stream. This is done through after-methods on the generic functions for creating records and manipulating fields. So, as user action continues, records of that activity are recorded and stored.

When the user selects "Synchronise" from the command menu, the local stream is synchronised with the server's stream. Prospero processes the local records and sends them to the server. At the server end, they are then applied to the database. The synchronisation framework is set out by Prospero; application details are attached through methods on generic functions, such as `locally-apply-action`. This involves more slight specialisation—to produce forms of the action instances which are acceptable to the transport system (in `flatten.lisp`, lines 5–12).

7.5 Flexibility

The previous sections have introduced and described two longer examples of the use of Prospero in building collaborative applications, illustrating the ways in which application code and collaboration code are related, and the ways in which applications can specialise Prospero structures to their own needs. The particular end to which Prospero has been developed is greater flexibility. So, having seen the two examples separately, it is worth stepping back to discuss the ways in which Prospero addresses problems raised earlier.

First, I will briefly to compare the two applications in terms of their different requirements and mechanisms. Next, I will discuss how the flexibility in Prospero which these applications exploit compares to the facilities in other toolkits discussed earlier. Finally, I will discuss how the use of reflection and OI techniques makes this possible.

7.5.1 Comparing Eureka and Bugspray

The two applications, Eureka and Bugspray, clearly differ considerably in their domains, which is what we require of a toolkit. However, and more importantly, they also differ considerably in their structures and styles. Consider various differences:

1. Eureka uses a peer-to-peer data management policy; Bugspray adopts a client/server approach.
2. Eureka uses a completely replicated representation of user data; Bugspray uses a centralised data store (with partial replication in each client cache).
3. Eureka supports highly synchronous interaction; Bugspray is more variable in its access patterns, from semi-synchronous to largely asynchronous.
4. Eureka uses bounded synchronisation, leading to frequent, regular and transparent data sharing; Bugspray uses an explicit synchronisation model, allowing users to update their changes as needed.
5. Eureka allows each user free access to the workspace; Bugspray uses an application-specific consistency management policy.

It is certainly not the case that the approach adopted by one example is right, and that adopted by the other is wrong. Nor is it the case that one approach subsumes the other, or even that one particular option for each design decision is more correct or more general. Rather, these and other design decisions reflect the way in which infrastructure requirements and patterns of collaborative activity interact; the decisions can only be resolved in the context of particular applications or scenarios. What's more, looking at new applications will not simply require different sets of options for these various decisions, but instead will introduce entirely new candidate solutions for these decisions, as well as opening up new areas for design decision-making. In other words, supporting these applications means supporting the different strategies which they might use; mapping the infrastructure supplied by the toolkit onto the needs of the application, rather than the other way around.

7.5.2 Flexibility In Prospero and Other Toolkits

In Chapter 2, six existing CSCW toolkits were described, with particular focus on their support for programming flexibility. Having now seen the core elements of Prospero's design, and examples of applications developed to demonstrate its flexibility, it seems appropriate to return to those systems and contrast the flexibility in Prospero with that offered in the other systems. Could they be used to build these two applications, and if not, why not?

There are two sets of reasons why it would be difficult or impossible to generate these two applications using the other toolkits. One set is fairly simple; the second is more significant.

7.5.2.1 Surface Flexibility

The first set of reasons arise from the inability of some toolkits to provide specific high-level features of the application designs. For instance, Oval and Suite do not provide support for the forms of interface variability; Suite operates in terms of textual interaction, while Oval provides graphical interaction only through pre-defined views which do not include arbitrary graphical objects. While they could both support the Bugspray application, which uses a simple editor-like visualisation of structured data, neither has support for the forms of graphical interaction in Eureka. This is a more significant issue in the case of Oval, since it is intended to be used for end-user variability, without further programming; since Suite is organised as a library for use within other programs, it may be possible to build graphical interfaces, although no such applications have been described in the Suite literature.

Similarly, Oval and MEAD do not provide control over the patterns of data sharing and consistency which are illustrated by the use of free-for-all access in Eureka and application-specific consistency control in Bugspray. This level of control is simply outside their design requirements. Similarly, while Suite provides flexible data management and consistency controls, its control is provided in terms of rules based on predefined conditions and facilities. It is not cast in terms of application-specific needs, as was exploited in the case of Bugspray's use of consistency guarantees.

7.5.2.2 Architectural Flexibility

The second set of reasons, however, are more relevant to the basic design of the toolkits, and to the use of the OI approach in Prospero.

Some of the toolkits described in Chapter 2 have no support for the forms of architectural variability seen in the examples presented here. Eureka is fully-replicated, highly synchronous, with automatic synchronisation. Bugspray is centralised with distributed caching, semi-synchronous to asynchronous, with explicit synchronisation. However, amongst the toolkits, Oval, GroupKit and Rendezvous are fully synchronous[4], and Oval and Rendezvous share an emphasis on centralised architectures. Suite and COLA also emphasise synchronous operation, and although they provide opportunities for more flexible control, these do not extend to the more asynchronous use of Bugspray. MEAD, similarly, assumes synchronous working (although, again, this level of architectural flexibility is simply not an issue in its design, since it focuses more on visualisation and user interface coupling).

4. That is, they provide only for real-time connections between processes. Clearly, asynchronous models of work can be modelled through synchronous interactions with a permanently-running server.

Critically, where mechanisms exist for deeper flexibility in the six toolkits of Chapter 2, their use of traditional abstraction techniques requires that the programmer "drops down" to the implementation level in order to gain control. For instance, COLA's separation of mechanism and policy means that the entire data management and consistency control mechanisms must be implemented within the application, requiring that it deal with a new level of abstraction. The two levels are inextricably mixed. GroupKit's "open protocol" approach constrains this slightly by dealing in terms of a specific protocol for managing, say, data consistency, but still requires a complete implementation; there is no provision for the incremental definition of new mechanisms and the optional reuse of existing facilities, since open protocols are *completely* open. In other words, while the OI approach is designed to *allow* programmers to *become involved in* aspects of the infrastructure which supports their applications, these other approaches *require* programmers to *take responsibility* for them.

7.5.3 OI and Reflection in Prospero

The value of Prospero, then, is not in the particular *set of options* it offers to application programmers, and the different included mechanisms from which they can pick and chose. Rather, it lies in the provision of a framework *within which* new behaviours and structures can be defined. Each of the applications has taken elements from the Prospero toolkit and tailored them to its specific needs, whether that be to the local versus the remote effects of executing actions, or the means by which promises can be compared. These specialisations were performed simply and concisely, and fit naturally into the general structure for collaborative action which Prospero sets up and implements. Furthermore, the code within Prospero which implements various specialisations employed by these two examples is similarly straightforward. The difference between `<bounded-stream>` and `<explicit-synch-stream>`, for example, requires three lines of Lisp code; and the addition of application-specific consistency management in Bugspray was only a few lines of code.

The use of OI techniques, and the metaobject protocol in particular, is critical to the way in which this flexibility is achieved.

First, it provides the structures for programmers to gain control over aspects of the implementation. This means not only the opportunity to create new structures and mechanisms which are usable within the toolkit, but also modifications which are seamlessly integrated into the toolkit's internal mechanisms (such as changes to stream synchronisation control, which then take immediate effect within the running system).

Second, it provides the means to do this more extensively than a parameterised approach will allow. That is, extensions are made not only through the structural aspects of the object-oriented encoding (the MOP as a reflective representation), but also through the use of metacode, rather than simply "meta-switches". The difference between the MOP approach and pure parameterisation is best seen in comparisons with Oval.

Third, the metaobject protocol retains the use of high-level specifications which "dropping down" to the implementation level would preclude. The components which metacode addresses are just those which base-level code uses. Programmers express metacode in terms of the requirements on these reified metaobject classes, while other, implementation-specific details which lie underneath remain hidden. The same metalevel interface can be maintained across implementations of Prospero, since the metalevel interface is written in terms of the *revealed structure* of the system, rather than the details of its implementation.

This, in turn, encourages metaprogramming in terms of the specifics of the application, rather than the specifics of the toolkit implementation. So, for instance, the use of consistency guarantees represents the expression of application-specific requirements, rather than the re-implementation of consistency management in the toolkit (as would be required by, say, GroupKit).

Each of these elements—application-specific control over aspects of the underlying system's behaviour, through programmatic access to a revealed model of its inherent structure—derives directly from computational reflection (self-referential models) and the metaobject protocol (object-oriented meta-structure) as elements of the Open Implementation approach.

7.6 Summary

Building on the concepts and mechanisms introduced primarily in Chapters 5 and 6, this chapter has presented two longer worked examples of applications built using the prototype Prospero implementation. These two examples vary significantly in their scope, structure and style of interaction; the first is a synchronous, replicated shared graphical editor, and the second is a more asynchronous, centralised collaborative database application. Although the styles, and hence internal organisation, of these two applications differ considerably, they are both supported within the framework which Prospero defines and implements. Application programmers can use specialisation and refinement to tailor the basic Prospero mechanisms and structures (such as <stream> and <promise>), as well as pre-defined and default derivatives of these (such as <bounded-stream> and <lock-promise>) to match the needs of their particular applications or domains, while maintaining the overall structure of their code and effecting a simple separation between application code, toolkit use and toolkit specialisation.

Chapter 8: Summary and Conclusions

We have now seen the various elements of this work—motivations, analysis, development and use. In this chapter, I will briefly summarise the preceding material, reiterate some of the principal points to show how this work has addressed them, and then point the way towards opportunities for further investigation.

8.1 Recapitulation

I began (in Chapters 1–3) by describing a problem—that existing CSCW toolkits are too inflexible to support the range of behaviours needed by collaborative tools. This problem has two aspects: one social, and one technical. The social aspect, as illustrated by a number of experimental and naturalistic studies, is that collaborative technologies are typically ill-suited to the flexible and open ways in which collaborative activity proceeds. The technological aspect is that traditional approaches to the design of toolkits require toolkit developers to make implementation decisions which subsequently restrict the ways in which those toolkits can be used and, hence, the range and form of the applications which can be built using them.

These two aspects are related. In Chapter 3, I drew on recent work on Open Implementation to analyse these problems in terms of the use of *abstraction*, in both toolkits and applications. This analysis suggests a particular form of solution—the use of Open Implementation techniques to open up a CSCW toolkit, resulting in a system in which the components and mechanisms that the toolkit offers can be manipulated, controlled and specialised by application developers to match the needs of particular applications and usage situations.

The main body of the dissertation (chapters 4–7) has outlined such a solution. Chapter 4 described the basic form and design principles behind Prospero, a CSCW application toolkit designed using Open Implementation techniques. Chapters 5 and 6 introduced two novel techniques which the toolkit embodies. Each of these techniques is focussed on providing ways to map implementations structures onto application needs, rather than the other way around.

The first technique is the *divergence/synchronisation approach*, a descriptive and implementational framework for managing collaborative data spaces. Divergence and synchronisation cast the problem of distributed data control in terms of managing and resolving inconsistency, rather than attempting to avoid it. This means that the application can get involved in replication control. This framework is specifically designed for the ways in which potentially replicated data is processed in *collaborative* rather than simply distributed environments, and, as such, gives much more direct support for collaborative activity. Specifically, it provides support for scalable synchrony (by opening up "synchronicity"), opportunistic work (by allow application-controlled inconsistency) and multi-synchronous action (by turning "floor control" into an application issue).

The second technique, *consistency guarantees*, supports the management of consistency over such a collaborative data store. Consistency guarantees not only provide direct support for collaborative activity (unlike, for example, the locks of traditional distributed or multi-user databases), but are also a means for application programmers to express the semantics of application operations. Application semantics provide a much richer basis for decisions about concurrency than would be available if all actions had simply to be mapped to the most general read/write-semantics model. As a result, collaborative applications developed using Prospero have increased potential for concurrency and opportunism as appropriate for the particular application (rather than pre-conceived notions of allowable concurrency embedded within the toolkit's design).

These techniques have been demonstrated and implemented in the prototype Prospero toolkit. To supplement the smaller examples which Chapters 5 and 6 used to illustrate technical points, Chapter 7 presented two full examples—Eureka, a shared graphical editor, and Bugspray, a multi-user bug-tracking database. Individually, these examples both illustrate how Prospero's structures and principles are used in designing collaborative applications, and how the relationship between toolkit facilities and application programming is managed. More importantly, when taken together, these two examples illustrate the flexibility which Prospero embodies. They demonstrate how a single toolkit can embody radically different models of collaboration, and how applications can revise and adapt toolkit mechanisms for their own needs.

8.2 Claims and Goals

A number of claims and goals were set out in the introduction as the principal elements of this dissertation.

1. *That the evidence of empirical and naturalistic studies of cooperative work demonstrates that usage issues and system issues are fundamentally linked.*

   While there are many examinations of the use of interactive technology (supporting both individual users and collaborating groups) which demonstrate the ways in which the technology systematically undermines patterns of everyday action, these critiques have typically focussed on design features of the particular technologies in use. Here, I have drawn upon a range of experimental and naturalistic studies, emphasising the ways in which they show activity to be organised around the fine detail of interactive technologies, emerging not only from the interface but as reflections of the implementation. On this basis, I have argued for a much stronger relationship between the organisation of computational mechanism and the organisation of activity around it. In other words, the relationship between system design and system use is more than interface-deep. The design of more effective interactive and collaborative technologies, then, must begin with an understanding of these sorts of interactions and their implications.

2. *That the reevaluation of abstraction in software engineering, set out by research in Open Implementation, applies to these issues as encountered in CSCW.*

   In Chapter 3, I introduced the principles behind recent work on Open Implementation (and their conceptual foundations in the work of computational reflection). A fundamental precept of this work is that abstraction in software engineering differs fundamentally from abstraction in mathematics. It argues that computational abstractions are not wholly abstract, but rather are the "visible components" of underlying, concrete implementations—implementations which carry with them constraints and implications for their use.

   I have demonstrated that this analysis can be carried over to the sorts of abstractions typically provided in collaborative systems and toolkits for the design of collaborative systems. Abstractions such as "shared object", "workspace" or "telepointer" are only useful in such toolkits as the interface to implementations; but those implementations carry with them implications for the nature of collaborative activity which can be carried out over those abstractions. Further, the abstract interfaces themselves provide no expression of these constraints. It is issues of just this sort which lead to the problems described above—breakdowns arising from the separation of user activity and system behaviour (abstract manipulation and actual implementation).

3. *That OI/reflective principles can be used to design a novel CSCW toolkit.*

   The obvious result of an OI-style analysis of the flexibility problem in CSCW is an OI-style solution to that problem. I have presented Prospero, a prototype CSCW toolkit. Prospero builds on the OI analysis by opening up toolkit abstractions—structures and mechanisms—to examination and manipulation by application clients. The clients can "become involved" in the implementation of the infrastructure which supports them. The concepts which Prospero embodies (principally, the divergence/synchronisation framework, and the use of consistency guarantees) are designed not just to support cooperative work, but as a basis for programmatic extensions and specialisation of Prospero's internal mechanisms.

4. *That a toolkit built along these lines yields significant improvements in design (and hence, usage) flexibility.*

   Two examples presented in Chapter 7 to help demonstrate the flexibility achieved through Prospero's use of Open Implementation techniques. These applications vary across a number of dimensions. They differ not only in their application domains—the traditional (and motivational) area of flexibility in toolkit design—but also in the nature and structure of their implementation. These implementational variances—synchronous versus asynchronous, peer-to-peer versus client-server, centralised versus replicated—are ones which cross-cut the barriers which traditional toolkit designs erect. Chapter 7 showed how the other toolkits introduced in Chapter 2 would either fail altogether to support these two different applications, or would require the programmer to "step down" into the code of the implementation (if this were available) and provide implementation-specific extensions and modifications. Prospero's model allows customisation at a high-level through the metalevel interface.

   Critically, Prospero does not provide a *parameterised* implementation, in which users can select one of a set of pre-defined behaviours for each component of an application. Rather, it provides a framework within which new behaviours and mechanisms can be crafted through the programmatic extension and specialisation of revealed aspects of the toolkit's internals. The view that Prospero provides onto aspects of its internal structure, and the opportunities that it offers for applications to tailor and specialise this structure according to their particular needs, are the essence of the Open Implementations approach; and also the means by which Prospero offers considerably greater flexibility and control than traditional toolkits.

8.3 Future Work and Opportunities

The analysis and design presented in this dissertation have been pursued as an experiment to demonstrate the validity of the techniques and their applicability to CSCW as a domain where concerns of openness, flexibility and revisability are paramount. Building further on this, then, there are a number of areas for further research investigation.

1. *Development of Prospero*. The most obvious area for future work is in further extensions to the Prospero toolkit itself.

   A number of areas for possible future developed were discussed in Chapter 4, as areas which the prototype toolkit does not address. For instance, extension to the user interface is one obvious area, which would provide a better integration of user interface actions (particularly "intermediate" actions, such as dragging and selection) with the actions which Prospero sees, as well as opening up more opportunities for the integration of user interface components with the shared data space (discussed in section 4.5.2). Similarly, many interesting aspects of application-specific control have been left unaddressed by Prospero in the areas of session management, object naming/identity and object namespace scope control. Areas such as these, omitted in the initial implementation so as to concentrate on core elements which demonstrate the application of OI principles, are candidates for the same sorts of development as distributed data management and consistency control have received here.

   Another important area is in direct support for dynamic flexibility through dynamic reflection, described in Chapter 3. The accessible meta-level model is the basis for this, by providing mechanisms through which applications can change the model in order to change their behaviour dynamically. In the prototype Prospero implementation described here, this can be achieved through the exploitation of Lisp's inherent dynamism. Using CLOS's `change-class` facility, for example, a programmer could switch between different models of synchronisation at run-time by, for example, changing a stream from a bounded-stream to an explicit-synch-stream. However, Prospero itself does not provide support for this, even though it is clearly a highly desirable facility even when implemented in a more static programming language.

2. *Open Implementations*. A number of areas open for further work focus on the development of Open Implementation techniques and, in particular, their application to CSCW.

   First, the design of Open Implementations is at an early stage of development, and general techniques building on the generalised experiences of OI developers are only slowly being developed (e.g. recent work on OIA/D (Kiczales et al., 1995)). Each new OI experience, and each application to a new domain, brings refinements and insights into the model. As described in Chapter 7, one interesting aspect of the Open Implementation in Prospero is the way in which applications *enrich*, rather than *configure*, the base language. The ways in which this happens, its consequences, and its applicability to new domains, all remain avenues for fruitful investigation in the development of the OI technique.

   Second, as the focus of OI techniques has broadened from its original grounding in programming language semantics and applications, researchers from other areas have begun to adopt aspects of the OI approach and apply them to their own work. This has included a number of investigations in distributed systems and distributed operating systems of the value of reflective and metalevel techniques (e.g. Chiba and Masuda, 1993; Okamura and Ishikawa, 1994; Stroud and Wu, 1995). These investigations aim principally at dynamic control and configuration of distributed systems and augmentation of programming languages in support of distributed programming, so typically at a lower level than the work presented here; their focus is infrastructure (that is, "below" the application). However, they reflect a concern with the mutual implication of programming, distribution and concurrency issues, and as such, share some of the motivations which have driven this work. They point towards an opportunity to use reflective techniques to integrate system and application issues by using metalevel information to coordinate the needs of both. The subsequent issues of layered metaobject protocols is one which, similarly, requires investigation, and which (despite the occasional use of CLOS's reflective facilities in support of my own) has been outside the scope of this research.

3. *Technical and Social Issues in Systems Design*. A fundamental motivation for much of this work has been the relationship between technical and social concerns in the development of interactive technologies. From its inception, CSCW has had a strong concern with the social aspects of group work, as well as with technological opportunities. However, in the work presented here, I have been particularly concerned with the ways in which these issues are mutually influential and can be integrated in the search for some common solution.

   The use of reflective techniques towards this end is particularly interesting because of the way that they cut across the boundaries which normally separate social concerns (and sociologists) from technical concerns (and technologists). In other recent work, I have been investigating the use of explicit, computationally reflective representations in traditional single-user interactive systems, orienting towards them as "accounts" which systems offer of their own action (Dourish, 1995; Dourish et al., 1996). This use of accounts, and the notion of accountability on which it draws (based on the role of accountability in Garfinkel's ethnomethodology (1967)), has provided the underpinnings for a new form of relationship between social science and system design (Button and Dourish, 1996).

   An important area of investigation which this opens up, then, is the use of these sorts of techniques—techniques like OI, which break down the traditional barriers of computational systems—as a means to take sociological investigations of technology use and make their insights "real" in design; to embed them not simply in the methods by which we design computer systems, but in the computer systems themselves.

8.4 Concluding Remarks

In this research, I have taken a set of insights into the nature of CSCW, derived from experimental and naturalistic studies of collaborative work, and, from these, developed and demonstrated a new approach to the design of CSCW technologies. In doing so, I have presented not only these ideas themselves, but also some mechanisms for CSCW implementation. In addition, I have looped back on myself, and encountered and developed a set of understandings about the nature of the relationship between technological design and social analysis which are now spurs to further work.

It is not uncommon, perhaps, to come to this point in writing a dissertation—the final paragraph—only to discover that it is not the end at all, but the beginning. The work reported here has demonstrated the applicability and utility of a set of principles: that usage implications drive down deep into the heart of computational design, that abstraction “barriers” function better as semi-permeable membranes, and that the representations at the heart of computer systems design are, themselves, up for negotiation and reworking. These principles, however, map out a considerably more vast terrain than can be covered here. I look forward to exploring it further, and to meeting others there.

Epilogue

And now my charms are all o'erthrown
And what strength I have's mine own
Which is most faint: now t'is true
I must here be released by you.

But release me from my bands
With the help of your good hands
Gentle breath of yours my sails
Must fill, or else my project fails,
Which was to please. Now I want
Spirits to enforce, art to enchant
And my ending is despair,
Unless I be relieved by prayer
Which pierces so that it assaults
Mercy itself and frees all faults
As from your crimes would pardon'd be
Let your indulgence set me free.

—Prospero's closing speech from "The Tempest"

Collected References

Ahuja, S., Ensor, J., and Lucco, S. (1990). "A Comparison of Application Sharing Mechanisms In Real-time Desktop Conferencing Systems", in *Proc. ACM Conference on Office Information Systems COIS '90* (Boston, Mass.), pp. 238–248.

Anderson, T., Bershad, B., Lazowska, E., and Levy, H. (1992)."Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM Transactions on Computer Systems, 10(1), pp. 53–79.

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

Beck, E. and Bellotti, V. (1993). "Informed Opportunism as Strategy: Supporting Coordination in Distributed Collaborative Writing", in *Proc. Third European Conference on Computer-Supported Cooperative Work ECSCW'93* (Milano, Italy), pp. 233–248.

Bentley, R. (1994). *"Supporting Multi-User Interface Development for Cooperative Systems"*, PhD Thesis, Department of Computing, Lancaster University, England.

Bentley, R., Rodden, T., Sawyer, P., and Sommerville, I. (1992). "An Architecture for Tailoring Cooperative Multi-User Displays", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92* (Toronto, Canada), pp. 187–194.

Bentley, R. and Dourish, P. (1995). "Medium vs. Mechanism: Supporting collaboration through customisation", in *Proc. European Conference on Computer-Supported Cooperative Work ECSCW'95* (Stockholm, Sweden), pp. 133–148.

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

Bhatti, N. and Schlichting, R. (1995). "A System for Constructing Configurable High-Level Protocols", in *Proc. ACM SIGCOMM'95* (Boston, Mass.).

Birman, K. and Joseph, T. (1987). "Exploiting Virtual Synchrony in Distributed Systems", in *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pp. 123–138.

Bly, S. and Minneman, S. (1990)."Commune: A Shared Drawing Surface", in *Proc. ACM Conference on Office Information Systems COIS '90* (Boston, Mass.), pp. 184–192.

Bobrow, D., Kahn, K., Kiczales, G., Masinter, M., Stefik, M., and Zbydel. F. (1986). "CommonLoops: Merging Lisp and Object-Oriented Programming", in *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA '86* (Portland, Oregon).

Bowers, J., Button, G. and Sharrock, W. (1995). "Workflow from Within and Without", in *Proc. European Conference on Computer-Supported Cooperative Work ECSCW'95* (Stockholm, Sweden), pp. 51–66.

Braun, T. and Diot, C. (1995). "Protocol Implementation Using Integrated Layer Processing", in *Proc. ACM SIGCOMM'95* (Boston, Mass.).

Bretthauer, H., Davis, H., Kopp, J., and Playford, K. (1993). "Balancing the EuLisp Metaobject Protocol", Lisp and Symbolic Computation, 6(1/2), pp. 119–138.

Brink, T. and Gomez. L. (1992). "A Collaborative Medium for the Support of Conversational Props", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'92* (Toronto, Canada), pp. 171–178.

Brink, T. and Hill, R. (1993). "Building Shared Graphical Editors Using the Abstraction-Link-View Architecture", in *Proc. Third European Conference on Computer-Supported Cooperative Work ECSCW'93* (Milano, Italy), pp. 311–324.

Button, G. and Dourish, P. (1996). "Technomethodology: Paradoxes and Possibilities", in *Proc. ACM Conference on Human Factors in Computing Systems CHI'96* (Vancouver, Canada), pp. 19–26.

Cao, P., Felten, Ed., and Li, K. (1994). "Implementation and Performance of Application-Controlled File Caching", in *Proc. of First Symposium on Operating Systems Design and Implementation*, pp. 165–178.

Cardelli, L. (1995). "A Language with Distributed Scope", in *Proc. 22nd ACM Symposium on the Principles of Programming languages*, pp. 286–297.

Chiba, S. (1995). "A Metaobject Protocol for C++", in *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA'95* (Austin, Texas), pp. 285–299.

Chiba, S. and Masuda, T. (1993). *"Designing with an Extensible Distributed Language with a Meta-Level Architecture"*, in Proc. European Conference on Object-Oriented Programming ECOOP93 (Kaiserlautern, Germany).

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

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* (Los Angeles, California), pp. 329–342.

Cypher, A. and Smith, D.C. (1995). *"KidSim: End User Programming of Simulations"*, in *Proc. ACM Conference on Human Factors in Computing Systems CHI'95* (Denver, Colorado), pp. 27–36.

De Michaelis, G. and Grasso, A. (1994). "Situating Conversations with the Language/Action Perspective: The Milan Conversation Model", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'94* (Chapel Hill, North Carolina), pp. 89–100.

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. First IEEE Workshop on Mobile Computing Systems and Applications* (Santa Cruz, California).

Dewan, P. (1990). "A Tour of the Suite User Interface Software", in *Proc. ACM Conference on User Interface Software and Technology UIST'90* (Snowbird, Utah).

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

Dewan, P. and Choudhary, R. (1995). "Coupling the User Interfaces of a Multiuser Program", ACM Transactions on Computer-Human Interaction, 2(1), pp. 1–39.

Dix, A. (1992). "Pace and Interaction", in *People and Computers VII: Proc. of HCI'92*, York, UK, 1992.

Dourish, P. (1995). "Accounting for System Behaviour: Representation, Reflection and Resourceful Action", in *Proc. Third Decennial Conference on Computers in Context CIC'95* (Aarhus, Denmark), pp. 147–156.

Dourish, P. and Bellotti, V. (1992). "Awareness and Coordination in Shared Workspaces", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '92* (Toronto, Canada), pp. 107–114.

Dourish, P., Adler, A., and Smith, B.C. (1996). "Organising User Interfaces around Reflective Accounts", in *Proc. Reflection'96* (San Francisco, California).

Eisenberg, M. (1995). "Programmable Applications: Interpreter meets Interface", SIGCHI Bulletin, 27(2), pp. 68–93.

Ellis, C. and Gibbs, S. (1989). "Concurrency Control in a Groupware System", in *Proc. ACM Conference on Manamagement of Data SIGMOD'89*, (Seattle, Washington).

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

Fischer, G. and Girgensohn, A. (1990). "End-User Modifiability in Design Environments", in *Proc. ACM Conference on Human Factors in Computer Systems CHI'90* (Seattle, Washington).

Fish, R., Kraut, R., Leland, M., and Cohen, M. (1988). "Quilt—A Collaborative Tool for Cooperative Writing", in *Proc. ACM Conference on Office Information Systems COIS '88* (Palo Alto, California).

Floyd, S., Jacobson, V., McCanne, S., Lui, C-G., and Zhang, L. (1995). "A Reliable Multicast Framework for Light-weight Sessions and Application Layer Framing", in *Proc. ACM SIGCOMM'95* (Boston, Mass.).

Friedman, D. and Wand, M. (1984). "Reification: Reflection Without Metaphysics", in *Proc. ACM Conference on Lisp and Functional Programming* (Austin, Texas).

Garfinkel, H. (1967). *"Studies in Ethnomethodology"*, Prentice-Hall, New York.

Garfinkel, D., Gust, P., Lemon, M., and Lowder, S. (1989). *"The SharedX Multi-User Interface User's Guide, Version 2.0"*, Software Technology Lab Report STL-TM-89-07, Hewlett-Packard Laboratories (Palo Alto, California).

Greenberg, S. (1991). "Personalisable Groupware: Accommodating Individual Roles and Group Differences", in *Proc. European Conference on Computer-Supported Collaborative Work ECSCW91*, (Amsterdam, Netherland), pp. 17–32.

Greenberg, S., Roseman, M., Webster, D., and Bohnet, R. (1992). "Human and Technical Factors of Distributed Group Drawing Tools", Interacting with Computers 4(3), pp. 364–392.

Greenberg, S. and Marwood, D. (1994). "Real-Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'94* (Chapel Hill, North Carolina), pp. 207–218.

Greif, I. and Sarin, S. (1986). "Data Sharing in Group Work", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '86* (Austin, Texas).

Grinter, R. (1995). "Using a Configuration Management Tool to Coordinate Software Development", in Proc. ACM Conference on Organisational Computing Systems COOCS'95 (Milpetas, California).

Gust, P. (1988). "Shared X: X in a Distributed Group Work Environment", unpublished paper presented at Second Annual X Technical Conference.

Haake, A. and Haake, J. (1993). “Take CoVer: Exploiting Version Management in Collaborative Systems”, in *Proc. InterCHI’93* (Amsterdam, Netherlands), pp. 406–413.

Haake, J. and Wilson, B. (1992). “Supporting Collaborative Writing of Hyperdocuments”, in *Proc. ACM Conference Computer-Supported Cooperative Work CSCW ‘92* (Toronto, Canada), pp. 138–146.

Hamilton, G. and Kougiouris, P. (1993). *“The Spring Nucleus: A Microkernel for Objects”*, Sun Microsystems Laboratories Technical Report TR-93-14 (Mountain View, California).

Harper, R. and Hughes, J. (1993). “What A F-ing System! Send ‘em all to the same place and then expect us to step ‘em hitting: Making technology work in air traffic control”, in Button (ed), *“Technology in Working Order”*, pp. 127–144, Routledge, London.

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* (Boston, Mass), pp. 187–199.

Heath, C. and Luff, P. (1992). “Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Line Control Rooms”, Computer Supported Cooperative Work, 1(1–2), pp. 69–95.

Heath, C., Jirotka, M., Luff, P. and Hindmarsh, J. (1995). “Unpacking Collaboration: the Interactional Organisation of Trading in a City Dealing Room”, Computer-Supported Cooperative Work, 3(2), pp. 147–165.

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

Hill, R. (1992). “The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications”, in *Proc. ACM Conference on Human Factors in Computing Systems CHI ‘92* (Monterey, California), pp. 335–342.

Hill, R. (1993). “The Rendezvous Constraints Maintainance System”, in *Proc. ACM Symposium on User Interface Software and Technology UIST’93* (Atlanta, Georgia).

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

Hill, W., Hollan, J., Wroblewski, D., and McCandless, T. (1992). “Edit Wear and Read Wear”, in *Proc. ACM Conference on Human Factors in Computing Systems CHI’92* (Monterey, California), pp. 3–10.

Hughes, J., Randall, D. and Shapiro, D. (1993). “From Ethnographic Record to System Design: Some Experiences from the Field”, Computer Supported Cooperative Work, 1(3), pp. 123–141.

Jeffay, K., Lin, J., Menges, J., Smith, F., and Smith, J. (1992). “The Architecture of the Artifact-Based Collaboration System”, in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW’92* (Toronto, Canada), pp. 195–202.

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.

Karsenty, A. and Beaudouin-Lafon, M. (1993). “An Algorithm for Distributed Groupware Applications”, in *Proc. 13th International Conference on Distributed Computing Systems (ICDCS)*, pp. 195–202.

Khalidi, Y. and Nelson, M. (1993). *“The Spring Virual Memory System”*, Technical Report SMLI TR-93-009, Sun Microsystems Laboratories (Mountain View, California).

Kiczales, G. (1992). *“Towards a New Model of Abstaction in Software Engineering”*, in Proc. IMSA’92 Workshop on Reflection and Metalevel Architectures (Tokyo, Japan), pp. 1–11.

Kiczales, G. (1996). “Beyond the Black Box: Open Implementation”, IEEE Software, January, pp. 8–11.

Kiczales, G. and Rodriguez, L. (1990). “Efficient Method Dispatch in PCL”, in *Proc. ACM Conference on Lisp and Functional Programming* (Nice, France), pp. 99–105.

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

Kiczales, G., DeLine, R., Lea, A., and Maeda, C. (1995). “Open Implementations Analysis and Design” (Tutorial Notes)*, Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA ‘95* (Austin, Texas).

Kiczales, G. and Paepcke, A. (forthcoming). *“Open Implementations and Metaobject Protocols”*, MIT Press, Cambridge, Mass.

Lai, K.-Y. and Malone, T. (1988). “Object Lens: A Spreadsheet for Cooperative Work”, in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW’88* (Portland, Oregon), pp. 115–124.

Lamping, J., Kiczales, G., Rodriguez, L. and Ruf, E. (1992). “An Architecture for an Open Compiler”, in *Proc. IMSA’92 Workshop on Reflection and Metalevel Architectures* (Tokyo, Japan), pp. 95–106.

Lauwers, C., Joeseph, T., Lantz, K., and Romanow, A. (1990). “Replicated Architectures for Shared Window Systems: A Critique”, in *Proc. ACM Conference on Office Information Systems COIS ‘90* (Cambridge, Mass.), pp. 249–260.

Lopes, C. (1996). “Adaptive Parameter Passing”, in Proc. International Symposium on Object Technologies for Advanced Software ISOTAS’96 (Japan).

Mackay, W. (1989). “How Do Experienced Information Lens Users Use Rules?”, in *Proc. ACM Conference on Human Factors in Computing Systems CHI’89* (Austin, Texas).

Mackay, W. (1990). “Patterns of Sharing Customisable Software”, in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW ’90* (Los Angeles, California), pp. 209–222.

Mackay, W. (1991). “Triggers and Barriers to Customising Software”, in *Proc. ACM Conference on Human Factors in Computer Systems CHI ‘91* (New Orleans, Louisiana), pp. 153–160.

MacLean, A., Carter, K., Moran, T., and Lövstrand, L. (1990). “User-Tailorable Systems: Pressing the Issues with Buttons”, in *Proc. ACM Conference on Human Factors in Computing Systems CHI ‘90* (Seattle, Washington).

McGuffin, L. and Olson, G. (1992). *“ShrEdit: A Shared Electronic Workspace”*, CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory, University of Michigan.

Maeda, C. (1996). “A Metaobject Protocol for Accessing File Systems”, in *Proc. International Symposium on Object Technologies for Advanced Software ISOTAS’96* (Japan).

Malone, T., Grant, K., Lai, K.-Y., Rao, R., and Rosenblitt, D. (1987). “Semi-Structured Messages Are Surprisingly Useful for Computer-Supported Coordination”, ACM Transactions on Office Information Systems, 5(2), pp. 115–131.

Malone, T., Lai, K.-Y., and Fry, C. (1995). “Experiments with Oval: A Radically Tailorable Tool for Cooperative Work”, ACM Transactions on Information Systems, 13(2), pp. 175–205.

Maes, P. (1987). *“Computational Reflection”*, Technical Report 87.2, Artificial Intelligence Laboratory, Vrije Universiteit, Brussels.

Miller, D., Smith, J., and Muller, M. (1992). “TelePICTIVE: Computer-Supported Collaborative GUI Design for Designers with Diverse Experience”, in *Proc. ACM Symposium on User Interface Software and Technology UIST’92* (Monterey, California), pp. 151–160.

Munson, J. and Dewan, P. (1994). "A Flexible Object Merging Framework", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'94* (Chapel Hill, North Carolina), pp. 231–242.

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

Myers, B., Guise, D., Dannenberg, R., Vander Zanden, B., Kosbie, D., Pervin, E, Mickish, A. and Marchal, P. (1990). "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". IEEE Computer, 23(11), pp. 71–85.

Nardi, B. (1993). *"A Small Matter of Programming: Perspectives on End-User Computing"*, MIT Press, Cambridge, Mass.

Nardi, B. and Miller, J. (1991). "Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development", in Greenberg (ed.), *"Computer-Supported Cooperative Work and Groupware"*, Academic Press.

Nelson, M., Khalidi, Y., and Madany, P. (1992). *"The Spring File System"*, Technical Report SMLI TR-93-10, Sun Microsystems Laboratories (Mountain View, California).

Neuwirth, C., Kaufer, D., Chandhok, R., and Morris, J. (1990). "Issues in the Design of Computer Support for Co-authoring and Commenting", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '90* (Los Angeles, California), pp. 183–196).

Neuwirth, C., Chandhok, R., Kaufer, D., Erion, P., Morris, J., and Miller, D. (1992). "Flexible Diff-ing in a Collaborative Writing System", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '92* (Toronto, Canada), pp. 147–154.

Neuwirth, C., Kaufer, D., Chandok, R., and Morris, J. (1994). "Computer Support for Distributed Collaborative Writing: Defining Parameters of Interaction", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'94* (Chapel Hill, North Carolina), pp. 145–152.

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* (Pittsburgh, Pennsylvania).

O'Malley, S. and Peterson, L. (1992). "A Dynamic Network Architecture", ACM Transactions on Computer Systems, 10(2), pp. 110–143.

Okamura, H. and Ishikawa, Y. (1994). "Object Location Control Using Meta-level Programming", in *Proc. European Conference on Object-Oriented Programming ECOOP'94* (Bologna, Italy), pp. 299–319.

Ousterhout, J. (1994). *"Tcl and the Tk Toolkit"*, Addison-Wesley, Reading, Mass.

Padget, J., Nuyens, G., and Bretthauer, H. (1993). "An Overview of EuLisp", Lisp and Symbolic Computation, 6(1/2), pp. 9–98.

Paepcke, A. (1988). "PCLOS: A Flexible Implementation of CLOS Persistence", in *Proc. European Conference on Object-Oriented Programming ECOOP'88.*

Patterson, J. (1991). "Comparing the Programming Demands of Single-User and Multi-User Applications", in *Proc. ACM Conference on User Interface Software and Technology UIST '91* (Hilton Head, South Carolina).

Patterson, J., Hill, R., Rohall, S., and Meeks, S. (1990). "Rendezvous: An Architecture for Synchronous Multi-User Applications", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'90* (Los Angeles, California), pp. 317–328.

Rao, R. (1991). “Implementational Reflection in Silica”, in *Proc. European Conference on Object-Oriented Programming ECOOP 91* (Geneva, Switzerland).

Rao, R. (1993). “The Silica Window System: The Metalevel Approach Applied More Widely”, in Paepcke (ed), *“Object-Oriented Programming: The CLOS Perspective”*, MIT Press, Cambridge, Mass.

Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. (1987). “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures”, in *Proc. ACM Conference on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, California).

Rein, G. and Ellis, C. (1991). “rIBIS: A Real-Time Group Hypertext System”, Intl. Journal of Man-Machine Studies, 34, pp. 349–367.

des Rivières, J. and Smith, B.C. (1984). “The Implementation of Procedurally Reflective Languages”, Xerox PARC Technical Report ISL-4 (Palo Alto, California).

Robinson, M. (1993). “Designing for Unanticipated Use...”, in *Proc. European Conference on Computer-Supported Cooperative Work ECSCW’93* (Milano, Italy), pp. 187–202.

Rodriguez, L. (1992). “A Study on the Viability of a Production-Quality Metaobject Protocol-Based Statically Parallelizing Compiler”, in *Proc. IMSA’92 Workshop on Reflection and Metalevel Architectures (Tokyo, Japan)*, pp. 107–112.

Roseman, M. and Greenberg, S. (1992). “GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications”, in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW’92* (Toronto, Canada), pp. 43–50.

Roseman, M. and Greenberg, S. (1993). “Building Flexible Groupware Through Open Protocols”, in *Proc. ACM Conference on Organisational Computing Systems COOCS’93* (Milpetas, California), pp. 279–288.

Roseman, M. and Greenberg, S. (1996). “Building Real-Time Groupware with GroupKit, a Groupware Toolkit”, ACM Transactions on Computer-Human Interaction, 3(1), pp. 66–106.

Smith, B.C. (1982). *“Reflection and Semantics in a Procedural Language”*, MIT Laboratory for Computer Science Report MIT-TR-272 (Cambridge, Mass.).

Smith, B.C. (1984). “Reflection and Semantics in Lisp”, in *Proc. ACM Symposium on Principles of Programming Languages* (Salt Lake City, Utah) pp. 23–35.

Steele, G. and Sussman., G. (1978). *“The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One and Two)”*, AI Memo No. 453, MIT Artificial Intelligence Laboratory (Cambridge, Mass.).

Steele, G. (1984). *“Common Lisp: The Language*” (first edition), Digital Press, 1984.

Stefik, M., Foster, G., Bobrow, D., Kahn, K., Lanning, S., and Suchman, L. (1987a). “Beyond the Chalkboard: Computer Support for Collaboration and Problem-Solving in Meetings”, Communications of the ACM, 30(1).

Stefik, M., Bobrow, D., Foster, G., Lanning, S., and Tatar, D. (1987b). “WYSIWIS Revisited: Early Experiences with Multiuser Interfaces”, ACM Transactions on Office Information Systems, 5, pp. 147–186.

Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schütt, W., and Thüring, M. (1992). “SEPIA: A Cooperative Hypermedia Authoring Environment”, in *Proc. ACM European Conference on Hypertext ECHT ‘92* (Milano, Italy).

Streitz, N., Gissler, J., Haake, J., and Hol, J. (1994). “DOLPIN: Integrated Meeting Support Across Local and Remote Desktop Environments and Liveboards”, in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW’94* (Chapel Hill, North Carolina), pp. 345–358.

Stroud, R. and Wu, Z. (1995). "Using Metaobject Protocols to Implement Atomic Data Types", in *Proc. European Conference on Object-Oriented Programming ECOOP'95* (Aarhus, Denmark).

Suchman, L. (1983). "Office Procedures as Practical Action: Models of work and system design", ACM Transactions on Office Information Systems, 1, pp. 320–328.

Suchman, L. (1987). *"Plans and Situated Actions: The problem of human-machine communication"*, Cambridge University Press, Cambridge, UK, 1987.

Suchman, L. (1993). "Technologies of Accountability: Of Lizards and Aeroplanes", in Button (ed.), *"Technology in Working Order: Studies of Work, Interaction and Technology"*, pp. 113–126, Routledge, London.

Suchman, L. (1994). "Do Categories have Politics? The language/action perspective reconsidered", Computer-Supported Cooperative Work, 2(3), pp. 177–190.

Tatar, D., Foster, G., and Bobrow, D. (1991). "Designing for Conversation: Lessons from Cognoter", Intl. Journal of Man-Machine Studies, 34, pp 185–209.

Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M. and Welch, B. (1994). "Session Guarantees for Weakly Consistent Replicated Data", in *Proc. International Conference on Parallel and Distributed Information Systems* (Austin, Texas).

Trevor, J., Rodden, T., and Mariani, J. (1994). "The Use of Adapters to Support Cooperative Sharing", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'94* (Chapel Hill, North Carolina), pp. 219–230.

Trevor, J., Rodden, T., and Blair, G. (1995). "COLA: A Lightweight Platform for CSCW", Computer Supported Cooperative Work, 3, pp. 197–224.

Trigg, R., Moran, T., and Halasz, F. (1987). "Adaptability and Tailorability in NoteCards", in Bullinger and Shackel (Eds.) *Proc. INTERACT '87*, North-Holland, 1987.

Winograd, T. (1986). "A Language/Action Perspective on the Design of Cooperative Work", in *Proc. ACM Conference on Computer-Supported Cooperative Work CSCW '86* (Austin, Texas).

Appendix A:
Code of Application Examples

This appendix presents the code of the example applications described and discussed in Chapter 7.

1 Eureka

1.1 eureka.lisp

```
(unless (find-package “EUREKA”)
   (make-package “EUREKA” :use ‘(“LISP” “KR” “PROSPERO” “PCL”)))
(in-package “EUREKA”)

;;;
;;; Eureka (because “it’s so simple”).
;;; eureka.lisp
;;;

;;;
;;; set up property definitions as subclasses of <action>.
;;;

(defclass <object-create-action> (<action>)
  ((name :initarg :name :accessor create-object-name)
   (points :initarg :points :accessor create-object-points)))

(defclass <object-edit-action> (<action>)
  ((action :initarg :action :accessor edit-action-action)
   (object-name :initarg :name :accessor edit-action-object-name)))

(defclass <object-move-action> (<action>)
  ((points :initarg :points)
   (object :initarg :object)))


;;;
;;; wrapper to create command objects based on UI information. these
;;; are called from the glue functions, and then the command objects
;;; are added to streams.
;;;

(defun object-create-object (name points)
  (make-instance ‘<object-create-action> :name name :points points))

(defun object-delete-object (obj-name)
```

```
  (make-instance ‘<object-edit-action> :action :delete
                 :name obj-name))

(defun object-move-object (obj)
  (make-instance ‘<object-move-action>
                 :points (gob-points obj) :object obj))


;;;
;;; set up local and remote streams.
;;;

(defvar *local-stream*)

(defun start-local-stream ()
  (setq *local-stream*
        (make-instance ‘<bounded-stream> :name “NAME FOO”
                                         :host “HOST FOO”)))

(defun connect-remote-stream (host local-stream)
  (let ((rs (make-instance ‘<remote-stream> :name “REMOTE FOO”
                                            :host host)))
    (push rs (stream-peers local-stream))))

(defun start-streams (remote-host)
  (connect-remote-stream remote-host (start-local-stream)))

;;; need to find other streams. get a clue from the init fn
;;; just one for the nonce
(defun setup-streams (host)
  (let ((rs (create-instance ‘<remote-stream> :name “FOO TEMP NAME”
                             :host host)))
    (push (stream-peers *local-stream* rs))))


;;;
;;; handle the remote execution of command objects. these functions
;;; basically just associate command object classes with the ui
;;; functions which perform them.
;;;

(defmethod locally-perform-action ((action <object-create-action>))
  (ui-add-polyline-from-create-obj action)
  (opal:update window))

(defmethod locally-perform-action ((action <object-edit-action>))
  (ui-delete-polyline-from-edit-obj action)
  (opal:update window))
```

1.2 ui.lisp

```
(unless (find-package “EUREKA”)
   (make-package “EUREKA” :use ‘(“LISP” “KR” “PROSPERO” “PCL”)))
(in-package “EUREKA”)

;;;
;;; Eureka (because “it’s so simple”).
;;; ui.lisp
;;;

```

```
(load (merge-pathnames “polyline-creator-loader”
                       user::garnet-gadgets-pathname) :verbose t)

;;;
;;; start-ui initialises the ui and sets up three interactors.
;;; poly-inter creates new polylines; quit-inter looks for
;;; keypresses to quit the ui; and delete-inter watches for object
;;; deletions.
(defun start-ui ()
  ;;(setq *lush-activity-record* nil)
  (create-instance ‘window inter:interactor-window
                   (:title “Eureka”))
  (create-instance ‘strokes-agg opal:aggregate)
  (s-value window :aggregate strokes-agg)

  (create-instance ‘poly-inter garnet-gadgets:polyline-creator
         (:start-event :leftdown)
         (:start-where `(:in ,window))
         (:input-filter nil)
         (:selection-function #’add-polyline)
         (:running-where t))

  (create-instance ‘quit-inter inter:text-interactor
         (:window window)
         (:continuous nil)
         (:start-where t)
         (:start-event ‘(:any-keyboard :except #\delete))
         (:final-function #’stop-ui))

  (create-instance ‘delete-inter inter:text-interactor
        (:window window)
        (:continuous nil)
        (:start-where `(:leaf-element-of ,strokes-agg))
        (:start-event #\delete)
        (:final-function #’delete-polyline))

  (opal:add-component strokes-agg poly-inter)

  (opal:update window))

(defun stop-ui (inter obj event string x y)
  (declare (ignore inter obj event string x y))
  (opal:destroy window))


;;;
;;; glue functions
;;;
;;; we need to relate the user interface to the prospero side. so,
;;; we produce glue functions for each of the interactors, which causes
;;; the appropriate prospero behaviour to happen, usually the creation
;;; of a new command object. we also have another function which will
;;; take the command object and produce the appropriate user interface
;;; behaviour. these are going to be linked by locally-perform-action.
;;;

(defun add-polyline (gadget points)
  (declare (ignore gadget))
  (let* ((new-name (gentemp “EUREKA”))
```

```
          (cmd-obj (object-create-object new-name points)))
     (add-action-to-stream cmd-obj *local-stream*)))

(defun ui-add-polyline-from-create-obj (create-obj)
  (let ((new-polyline (create-instance
                        (create-object-name create-obj)
                        opal:polyline
                        (:point-list
                         (copy-list (create-object-points create-obj))))))
(opal:add-component strokes-agg new-polyline)))


(defun delete-polyline (inter obj event string x y)
  (declare (ignore inter obj event string))
  (let ((line (opal:point-to-component strokes-agg x y)))
    (when line
          (let ((cmd-obj (object-delete-object (name-for-schema line))))
            (add-action-to-stream cmd-obj *local-stream*)))))

(defun ui-delete-polyline-from-edit-obj (edit-obj)
  (let ((polyline (symbol-value (intern (edit-action-object-name
                                          edit-obj)))))
    (opal:remove-component strokes-agg polyline)))

```

2 Bugspray

2.1 bugs.lisp

```
(unless (find-package “BUGS”)
  (make-package “BUGS” :use ‘(“LISP” “PCL” “KR” “PROSPERO”)))
(in-package “BUGS”)

;;;
;;; bugspray -- multi-user bug-tracking database
;;;

;;;
;;; class definitions for the various entry types.
;;;
(defclass <bugs-record> ()
  ((author :initform (getenv “USER”) :accessor record-author)
   (entry-slots :accessor entry-slots)
   (id :initform (generate-id) :initarg :id :accessor record-id)))

(defclass <bug> (<bugs-record>)
  ((symptoms :initform “” :initarg :symptoms :accessor bug-symptoms)
   (hypothesis :initform “” :initarg :hypothesis :accessor bug-hypothesis)
   (problems :initform nil :initarg :problems :accessor bug-problems)
   (entry-slots :initform ‘(symptoms hypothesis))))

(defclass <problem> (<bugs-record>)
  ((symptoms :initform “” :initarg :symptoms :accessor problem-symptoms)
   (customer :initform “” :initarg :customer :accessor problem-customer)
   (contact :initform “” :initarg :contact :accessor problem-contact)
   (platform :initform”” :initarg :platform :accessor problem-platform)
   (comments :initform nil :accessor problem-comments)
   (date :initform “” :initarg :date :accessor problem-date)
```

```
   (entry-slots :initform `(symptoms customer contact platform))))

(defclass <comment> (<bugs-record>)
  ((text :initform "" :initarg :text :accessor comment-text)
   (entry-slots :initform `(text))))

(defclass <solution> (<bugs-record>)
  ())

(defclass <fix> (<solution>)
  ((platform :initform "" :initarg :platform :accessor fix-platform)
   (reference :initform "" :initarg :reference :accessor fix-reference)
   (entry-slots :initform `(platform reference))))

(defclass <workaround> (<solution>)
  ((platform :initform "" :initarg :platform :accessor workaround-platform)
   (text :initform "" :initarg :text :accessor workaround-text)
   (entry-slots :initform `(platform text))))


;;;
;;; first, methods for creating and updating records.
;;;

(defmethod initialize-instance :after ((record <bugs-record>) &rest initargs)
  (declare (ignore initargs))
  (new-record record))

(defmethod new-record ((record <bugs-record>))
  (record-object record))

(defmethod update-record-slots ((record <bugs-record>) slot-value-list)
  (dolist (slot-value slot-value-list)
          (let ((slot (intern (car slot-value)))
                     (value (cadr slot-value)))
            (setf (slot-value record slot) value))))


;;;
;;; second, methods for linking various sorts of records.
;;;

(defmethod add-link ((problem <problem>) (comment <comment>))
  (setf (problem-comments problem) (append (problem-comments problem)
                                                                       (list (record-id
comment)))))

(defmethod add-link ((bug <bug>) (problem <problem>))
  (setf (bug-problems bug) (append (bug-problems bug)
                                                         (list (record-id problem)))))


;;;
;;; the mapping betweeen record objects and ids is made using a
;;; hash-table which operates either as the database or as a cache
;;; (depending).
;;;

(defvar *record-table* (make-hash-table))

```

```
(defun id-to-object (id)
  (gethash id *record-table*))

(defun record-object (object)
  (let ((id (record-id object)))
    (setf (gethash id *record-table*) object)))


```

2.2 ui.lisp

```
(unless (find-package “BUGS”)
  (make-package “BUGS” :use ‘(“LISP” “PCL” “KR” “PROSPERO”)))
(in-package “BUGS”)

(load (merge-pathnames “motif-prop-sheet-win-loader”
                              user::garnet-gadgets-pathname) :verbose t)
(load (merge-pathnames “motif-menubar-loader”
                              user::garnet-gadgets-pathname) :verbose t)
(load (merge-pathnames “motif-scrolling-menu-loader”
                              user::garnet-gadgets-pathname) :verbose t)
(load (merge-pathnames “motif-error-gadget-loader”
                              user::garnet-gadgets-pathname) :verbose t)

;;;
;;; ui.lisp
;;; user interface functions for bug tracking database
;;;

(defvar *selection* nil)
(defvar *access-mode* nil)

(defun start-controller ()
  (create-instance ‘window inter:interactor-window
    (:background-color opal:motif-gray)
    (:title “BugSpray controller”))
  (create-instance ‘agg opal:aggregate)
  (s-value window :aggregate agg)

  (create-instance ‘menubar gg:motif-menubar
    (:items
     ‘((“File” nil
          ((“Quit” stop-controller)))
       (“Mode” ui-set-mode
          ((“Entry”) (“Linkage”)))
       (“Object” ui-new-object
          ((“Problem”) (“Comment”) (“Bug”) (“Fix”) (“Workaround”)))
       (“Command” nil
          ((“Edit” ui-edit-selection) (“Delete” ui-delete-selection)
           (“Link” ui-link-selection) (“Synchronise” ui-synchronise))))))
  (opal:add-component agg menubar)

  (create-instance ‘scrolllist gg:motif-scrolling-menu
    (:top 30) (:width 150)
    (:title “Current objects:”)
    (:items nil)
    (:toggle-p nil)
    (:menu-selection-function #’ui-select-item)
    (:item-to-string-function #’object-to-string))
  (opal:add-component agg scrolllist)
```

```
  (opal:update window))

(defun init-controller ()
  (maphash #’(lambda (key value) (declare (ignore key))
                 (add-menu-item value)) *record-table*)
  (opal:update window))

(defun stop-controller (gadget menuitem submenuitem)
  (declare (ignore gadget menuitem submenuitem))
  (opal:destroy window))

;;; helper functions for the controller window, mainly for handling
;;; menu selections, etc.
;;;
(defun ui-select-item (gadget menuitem)
  (declare (ignore gadget))
  (setq *selection* (g-value menuitem :item)))

(defun add-menu-item (item)
  (let ((items (g-value scrolllist :items)))
    (unless (member item items)
      (s-value scrolllist :items (append items (list item))))))

(defun ui-edit-selection (gadget menuitem submenuitem)
  (declare (ignore gadget menuitem submenuitem))
  (when *selection*
          (dialogue-enter-record *selection*)))

(defun ui-delete-selection (gadget menu submenuitem)
  (declare (ignore gadget menu submenuitem))
  (when *selection*
          (let ((items (g-value scrolllist :items)))
            (s-value scrolllist :items (remove *selection* items)))))

(defun ui-link-selection (gadget menu submenuitem)
  (declare (ignore gadget menu submenuitem))
  (when *selection*
          (let ((s *selection*))
            (gg:display-query
             (create-instance nil gg:motif-query-gadget
                          (:string “Select end-point of link”)
                          (:button-names ‘(“OK” “Cancel”))
                          (:foreground-color opal:motif-gray)
                          (:modal-p nil)
                          (:selection-function #’(lambda (gadget item)
                                                                  (declare (ignore gadget))
                                                                  (ui-link-selected item s))))))))

(defun ui-link-selected (item oldselect)
  (when (and (string= “OK” item) (not (eq *selection* oldselect)))
    (add-link *selection* oldselect)))

(defun ui-new-object (gadget menuitem submenuitem)
  (declare (ignore gadget menuitem))
  (let ((obj (make-instance (item-to-classname submenuitem))))
    (dialogue-enter-record obj)))

(defun ui-set-mode (gadget menuitem submenuitem)
```

```
  (declare (ignore gadget menuitem))
  (setq *access-mode* (intern (string-upcase submenuitem)
                                          (find-package “KEYWORD”)))
  (if *current-guarantee*
      (redeem-guarantee *server-stream* *current-guarantee*))
  (setq *current-guarantee* (request-guarantee)))

(defun ui-synchronise (gadget menuitem submenuitem)
  (declare (ignore gadget menuitem submenuitem))
  (synchronise *local-stream* *server-stream*))

;;; functions for the motif dialogue box for entering any form
;;; of report object
;;;
(defmethod dialogue-enter-record ((record <bugs-record>))
  (let ((items (mapcar #’(lambda (x) (list (symbol-name x)
                                                                    (slot-value record
x)))
                              (entry-slots record))))
    (gg:pop-up-win-for-prop
     (create-instance nil gg:motif-prop-sheet-with-ok
                              (:items items)
                              (:ok-function #’(lambda (gadget)
                                                                 (dialogue-ok gadget
record)))
                              (:accept-function #’(lambda (gadget)
                                                                    (dialogue-ok gadget
record)))
                              (:cancel-function #’(lambda (gadget)
                                                                    (dialogue-cancel
gadget record))))
     200 150 (format nil “Entry form for ~A” (class-name (class-of record))))))


(defun dialogue-ok (gadget record)
  (update-record-slots record (g-value gadget :changed-values))
  (add-menu-item record))

(defun dialogue-cancel (gadget record)
  (declare (ignore gadget record))
  (format t “DIALOGUE CANCEL~%”))

;;;
;;; ui support for guarantees
;;;
(defun ui-request-guarantee ()
  (let ((guarantee (request-guarantee)))
    (if (eq (class-of guarantee) (find-class ‘<null-guarantee>))
          (gg:display-error-and-wait
           (create-instance nil gg:motif-error-gadget
                                      “Requested guarantee was NOT granted.”)))
    (setq *current-guarantee* guarantee)))

;;; misc. helper functions
;;;

(defun item-to-classname (item)
  ;; (intern (format nil “<~A>” (string-upcase item))))
  (intern (concatenate ‘string “<“ (string-upcase item) “>”)))

```

```
(defmethod object-to-string ((obj <bugs-record>))
  (format nil "~A: ~A" (class-name (class-of obj)) (record-id obj)))

(defmethod object-to-string ((foo t))
  (format nil "~A" foo))

```

2.3 common.lisp

```
(unless (find-package "BUGS")
  (make-package "BUGS" :use '("LISP" "PCL" "KR" "PROSPERO")))
(in-package "BUGS")

;;;
;;; common.lisp
;;; Components and definitions common to client and server sides.
;;;

;;; define semantic properties as subclasses of action, and then the
;;; various application operations in terms of those.
;;;

(defclass <append-action> (<action>)
  ())
(defclass <content-change-action> (<action>)
  ())
(defclass <structure-change-action> (<action>)
  ())
(defclass <no-change-action> (<action>)
  ())

(defclass <bugs-action> (<action>)
  ((id :initarg :id :accessor action-id)
   (fields :initform nil :initarg :fields :accessor action-fields)
   (values :initform nil :initarg :values :accessor action-values)))

(defclass <create-object-action> (<bugs-action> <append-action>)
  ((type :initarg :type :accessor create-action-type)))
(defclass <find-action> (<bugs-action> <no-change-action>)
  ())
(defclass <set-field-action> (<bugs-action> <no-change-action>)
  ())
(defclass <change-field-action> (<bugs-action> <content-change-action>)
  ())
(defclass <add-link-action> (<bugs-action> <structure-change-action>)
  ())

;;;
;;; check send action prevents action loops
;;; refuse attempts to send actions to the host which sent them.
;;;
(defmethod check-send-action ((action <bugs-action>) (stream <remote-stream>))
  (let ((host (stream-host stream))
        (source (action-source action)))
    (if (string-equal (subseq source (length host)) host)
        nil
      t)))


;;;
```

```
;;; stream stuff; initialisation and setup
;;;

(defvar *local-stream* nil)
(defvar *server-stream* nil)
(defvar *client-streams* nil)

(defun server-setup-streams (&rest client-hosts)
  (setq *local-stream* (make-instance ‘<explicit-synch-stream>
                                                       :name *ident*
                                                       :host *host*))
  (dolist (host client-hosts)
            (server-add-client host)))

(defun server-add-client (host)
  (push (make-instance ‘<remote-stream> :host host)
          (stream-peers *local-stream*)))

(defun client-setup-streams (host)
  (setq *local-stream* (make-instance ‘<explicit-synch-stream>
                                                       :name *ident*
                                                       :host *host*))
  (setq *server-stream* (make-instance ‘<remote-stream> :name “Bugspray server”
                                                        :host host))
  (push *server-stream* (stream-peers *local-stream*)))

(defmethod find-local-stream ()
  *local-stream*)
```

2.4 guarantees.lisp

```
(unless (find-package “BUGS”)
  (make-package “BUGS” :use ‘(“LISP” “PCL” “KR” “PROSPERO”)))
(in-package “BUGS”)

;;;
;;; guarantees.lisp
;;; code related to consistency guarantees for bugspray application.
;;; both client and server sides
;;;

;;;
;;; basic approach -- the application defines various sorts of guarantees
;;; which can be given, and then the comparative conditions under which
;;; they might be granted.
;;;
;;; a structure-promise implies that only structural changes will be
;;; made. a content-promise implies that only content-changes will be
;;; made. a structure-content-promise implies that structural and content
;;; changes will be made.
;;;

(defclass <bugs-promise> (<promise>)
  ())
(defclass <structure-promise> (<bugs-promise>)
  ())
(defclass <content-promise> (<bugs-promise>)
  ())
(defclass <structure-content-promise> (<structure-promise> <content-promise>)
  ())
```

```
;;;
;;; granting guarantees.
;;;
;;; in bugspray, we just want to look for structure/structure conflicts
;;; and content/content conflicts
;;;
(defmethod compatible-promises ((new <bugs-promise>) (old <bugs-promise>))
  t)

(defmethod compatible-promises ((new <structure-promise>)
                                                  (old <structure-promise>))
  nil)

(defmethod compatible-promises ((new <content-promise>)
                                                  (old <content-promise>))
  nil)


```

2.5 client.list

```
(unless (find-package “BUGS”)
  (make-package “BUGS” :use ‘(“LISP” “PCL” “KR” “PROSPERO”)))
(in-package “BUGS”)

;;;
;;; client.lisp
;;; Client-side database components for the bug-tracking database
;;; application.
;;;


;;;
;;; first, various functions for performing database actions. these
;;; then create and process action records. add these as after-methods.
;;;

(defmethod update-record-slots :after ((record <bugs-record>) slot-value-list)
  (let ((id (record-id record)))
    (dolist (slot-value slot-value-list)
       (let ((slot (intern (car slot-value)))
               (value (cadr slot-value)))
           (generate-update-record record id slot value)))))

(defmethod add-link :after ((from <bugs-record>) (to <bugs-record>))
  (let ((from-id (record-id from))
          (to-id (record-id to)))
    (when *local-stream*
            (add-action-to-stream
             (make-instance ‘<add-link-action> :source *ident* :id from-id
                                      :fields (list to-id) :values nil)
             *local-stream*))
    (format t “! link ~A ~A~%” from-id to-id)))

(defmethod new-record :after ((record <bugs-record>))
  (let ((id (record-id record))
          (type (class-name (class-of record))))
    (when *local-stream*
            (add-action-to-stream
```

```
              (make-instance ‘<create-object-action> :source *ident*
                                        :id id :type type)
              *local-stream*))
    (format t “-> <CREATE-OBJECT-ACTION> ~A ~A~%”
               (class-name (class-of record)) (record-id record))))

(defun generate-update-record (record id slot value)
  (let ((update-type (if (new-record-p record) ‘<set-field-action>
                             ‘<change-field-action>)))
    (when *local-stream*
             (add-action-to-stream
              (make-instance update-type :source *ident* :id id
                                        :fields (list slot) :values (list value))
              *local-stream*))
    (format t “-> ~A ~A ~A ~A~%” update-type id slot value)))

;;; new-record-p
;;; a record is “new” if it’s been created since the last point of
;;; synchronisation (and hence, changes to its fields cannot conflict
;;; with activities elsewhere).
;;;
(defun new-record-p (record)
  (let ((id (record-id record)))
    (pending-action-p *local-stream*
                             #’(lambda (action)
                                        (and (eq (class-of action)
                                                      (find-class ‘<create-object-action>))
                                           (eq (action-id action) id))))))

;;;
;;; promise and guarantee, based on the access mode
;;;
(defvar *current-guarantee* nil)

(defun request-guarantee ()
  (let ((promise (case *access-mode*
                              ((:entry) (make-instance ‘<content-promise>))
                              ((:linkage) (make-instance ‘<structure-promise>))
                              (t (make-instance ‘<structure-content-promise>)))))
    (get-guarantee *server-stream* promise)))

;;;
;;;
(defmethod add-action-to-stream ((action <bugs-action>) (stream <local-stream>))
  (push action (stream-actions stream)))


;;;;

(defun init-load-database ()
  (let ((host (stream-host *server-stream*)))
    (host-load-database host)))

```

2.6 server.lisp

```
(unless (find-package “BUGS”)
  (make-package “BUGS” :use ‘(“LISP” “PCL” “KR” “PROSPERO”)))
(in-package “BUGS”)
```

```
;;;
;;; server.lisp
;;; Server-side database components for the bug-tracking database
;;; application.
;;;

(defmethod locally-perform-action :before ((action <bugs-action>))
  (format t "received ~A ~A from ~A~%" (class-name (class-of action))
            (action-id action) (action-source action)))

(defmethod locally-perform-action ((action <create-object-action>))
  (let ((record (make-instance (create-action-type action)
                                          :id (action-id action))))
    (format t "Created ~A record with id ~A~%" (create-action-type action)
              (record-id record))
    (record-object record)))

(defmethod locally-perform-action ((action <set-field-action>))
  (format t "locally performing <set-field-action>~%")
  (let* ((record (id-to-object (action-id action)))
          (field (car (action-fields action)))
          (value (car (action-values action))))
    (format t "Setting ~A of ~A to ~A~%" field record value)
    (setf (slot-value record field) value)))

(defmethod locally-perform-action ((action <change-field-action>))
  (format t "locally performing <change-field-action>~%")
  (let* ((record (id-to-object (action-id action)))
          (field (car (action-fields action)))
          (value (car (action-values action))))
    (format t "Setting ~A of ~A to ~A~%" field record value)
    (setf (slot-value record field) value)))

(defmethod locally-perform-action ((action <add-link-action>))
  (let* ((from-id (action-id action))
          (to-id (car (action-fields action)))
          (from-obj (id-to-object from-id))
          (to-obj (id-to-object to-id)))
    (add-link from-obj to-obj)))

```

2.7 flatten.lisp

```
(unless (find-package "PROSPERO")
   (make-package "PROSPERO" :use '("LISP" "PCL")))
(in-package "PROSPERO")

(defmethod object-to-wireform ((obj bugs::<bugs-action>))
  (list (class-name (class-of obj)) (action-source obj)
         (bugs::action-id obj) (bugs::action-fields obj)
         (bugs::action-values obj)))

(defmethod object-to-wireform ((obj bugs::<create-object-action>))
  (list (class-name (class-of obj)) (action-source obj)
         (bugs::action-id obj) (bugs::create-action-type obj)))

(defun wireform-to-object (wireobj)
  (let ((obj (make-instance (intern (symbol-name (first wireobj)) "BUGS"))))
    (setf (action-source obj) (second wireobj))
```

```
    (setf (bugs::action-id obj) (third wireobj))
    (if (eq (class-of obj) (find-class `bugs::<create-object-action>))
          (setf (bugs::create-action-type obj) (fourth wireobj))
      (progn
          (setf (bugs::action-fields obj) (fourth wireobj))
          (setf (bugs::action-values obj) (fifth wireobj))))
    obj))


```

2.8 dbio.lisp

```
(unless (find-package “BUGS”)
  (make-package “BUGS” :use `(“LISP” “PCL” “KR” “PROSPERO”)))
(in-package “BUGS”)

;;;
;;; dbio.lisp
;;; Various things for saving, loading, uploading and downloading
;;; the record database.
;;;

;;;
;;; record-to-wireform provides a printable (transmissable) version of each
;;; record type
;;;
(defmethod record-to-wireform ((record <problem>))
  (list `<problem> `id (record-id record) `author (record-author record)
        `symptoms (problem-symptoms record)
        `customer (problem-customer record) `contact (problem-contact record)
        `platform (problem-platform record) `date (problem-date record)
        `comments (problem-comments record)))

(defmethod record-to-wireform ((record <workaround>))
  (list `<workaround> `id (record-id record) `author (record-author record)
        `text (workaround-text record)
        `platform (workaround-platform record)))

(defmethod record-to-wireform ((record <fix>))
  (list `<fix> `id (record-id record) `author (record-author record)
        `reference (fix-reference record)
        `platform (fix-platform record)))

(defmethod record-to-wireform ((record <comment>))
  (list `<comment> `id (record-id record) `author (record-author record)
        `text (comment-text record)))

(defmethod record-to-wireform ((record <bug>))
  (list `<bug> `id (record-id record) `author (record-author record)
        `symptoms (bug-symptoms record)
        `hypothesis (bug-hypothesis record)
        `problems (bug-problems record)))

;;;
;;; wireform-to-record does the inverse
;;;

(defun wireform-to-record (wireform)
  (let* ((type (first wireform))
           (id (third wireform))
```

```
           (initlist (cdddr wireform))
           (record (make-instance type)))
    (remhash (record-id record) *record-table*)
    (dolist (pair (pairify initlist))
              (setf (slot-value record (car pair)) (cdr pair)))
    (setf (record-id record) id)
    (record-object record)))

(defun database-to-wireform (database)
  (let ((dblist nil))
    (maphash #'(lambda (key value)
                        (declare (ignore key))
                 (push (record-to-wireform value) dblist)) database)
    dblist))

(defun dump-database ()
  (database-to-wireform *record-table*))

;;;
;;; file manipulation
;;;

(defun write-database (file database)
  (with-open-file (stream file :direction :output)
     (write (database-to-wireform database) :stream stream)))

(defun read-database (file)
  (with-open-file (stream file :direction :input)
     (read stream)))

(defun load-database (file)
  (let ((db (read-database file)))
    (dolist (record db)
      (wireform-to-record record))))

(defun new-database (file)
  (clrhash *record-table*)
  (load-database file))

;;;
;;; host-based database manipulation
;;;

(defun host-load-database (host)
  (let ((db (host-read-database host)))
    (dolist (record db)
      (wireform-to-record record))))

(defun host-read-database (host)
  (let* ((wire (prospero::host-wire host))
           (db (wire:remote-value wire (dump-database))))
    (dolist (record db)
      (wireform-to-record record))))

(defun host-new-database (file)
  (clrhash *record-table*)
  (host-load-database file))
```

2.9 misc.lisp

```
(unless (find-package “BUGS”)
  (make-package “BUGS” :use ‘(“LISP” “PCL” “KR” “PROSPERO”)))
(in-package “BUGS”)

(defvar *host* (unix:unix-gethostname))
(defvar *ident* (concatenate ‘string *host*
                                         (princ-to-string (unix:unix-getpid))))

(defun getenv (var)
  (cdr (assoc (intern var (find-package “KEYWORD”)) ext:*environment-list*)))

(defun generate-id ()
  (gentemp (concatenate ‘string “BUGS-” (string-upcase *host*))))

(defun pairify (list)
  (if (null list) nil
    (let ((a (car list))
           (b (cadr list)))
      (cons (cons a b) (pairify (cddr list))))))
```