



A Tale of Two Toolkits: Relating Infrastructure and Use in Flexible CSCW Toolkits

PAUL DOURISH and W. KEITH EDWARDS

Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
(E-mail: {dourish, kedwards}@parc.xerox.com)

(Received 5 February 1999)

Abstract. The design of software toolkits embodies a fundamental tension. On the one hand, it aims to reduce programmer effort by providing prefabricated, reusable software modules encapsulating common application behaviours. On the other, it seeks to support a range of applications, which necessitates avoiding an overly-restrictive commitment to particular styles of application behaviour.

We explore this tension in the domain of collaborative applications, which we believe are particularly subject to problems arising from this tension. Based on an analysis of the basic issues of flexibility in toolkit design, we explore opportunities for the design of toolkits which avoid application style commitments, with illustrations from two toolkits which we have developed. A comparative analysis of these two approaches provides insight into the underlying questions and suggests new design opportunities for toolkits that provide a framework for application enhancement and extension.

Key words: collaboration infrastructure, collaborative toolkits, reuse, specialisation, tailorability, toolkit design

1. Introduction

Flexibility and tailorability – the ability to accommodate individual differences in needs, use, style or task – are endemic problem in software systems. Most systems are created to be used by multiple people, in multiple organisations, on multiple computers, at multiple times, and generally in a wide range of settings that require different forms of support and engagement. Collaborative applications are, if anything, subject to even more stringent demands due to the extra requirements to accommodate individual differences within the same group, as well as the range of ways in which groups organise and engage in their activities. The problems of tailorability have been studied in the context of use, in both single-user and group settings, in investigations such as those of Trigg et al. (1987), MacLean et al. (1990), Nardi and Miller (1991) and Trigg and Bodker (1994).

We are concerned with flexibility in a different domain, the domain of software toolkit design. However, we are motivated by *just the same* concerns that motivated the studies cited above. In other words, even though we are going to be talking about software toolkits and discussing questions of application *implementation*,

our concern is with *use*. In particular, we are concerned with how toolkits can be designed to accommodate the wide range of potential applications and situations in use. This is a more complex problem than simply the design of flexible applications, because of the nature of toolkits and toolkit design.

Software toolkits ease the implementation of software systems by providing reusable components and behaviours designed to be applicable in a range of circumstances. Like building houses from prefabricated parts, this makes the application developer's task simpler (and, with luck, faster) since the work now focuses on assembling and configuring components rather than on creating them from scratch. As well as reducing effort and speeding application development, this reuse of software components also offers a common conceptual framework for application development, which can aid both application designers and users.

The components themselves arise out of common patterns of software structure that occur across a range of applications in a particular domain. So for example, a toolkit for constructing user interfaces provides interface "widgets" such as scrollbars and buttons which occur in different interfaces, and from which interfaces can be constructed, as well as commonly-occurring behaviours such as mouse-tracking, keyboard event processing and so on. A toolkit for image processing applications might provide common objects such as pixel arrays and color maps and common behaviours such as filtering, scaling and rotating image objects. Applications are constructed using these components as building blocks.

The designer of a software toolkit, then, is faced with a complex task. The toolkit designer doesn't write applications, but writes software to be *used* in applications; the details of those applications – what they will do, how they will do it, and who will use them – are not available at the time when the toolkit is being written. There are multiple levels of developers and users. The toolkit designer is developing software that implements a toolkit. The "users" of the toolkit are themselves programmers, who then generate applications which *themselves* have users. In order to provide a valuable and widely-applicable set of software components, the software toolkit designer must anticipate the needs of the application developers by anticipating the sorts of applications which will be created using the toolkit, and so in turn must anticipate the behaviour and requirements of the users of those applications. The toolkit developer's conception of the likely applications will influence the design of the software components provided in the toolkit.

It is from this two-level problem that the fundamental tension of toolkit design arises. On the one hand, the toolkit designer must offer pre-packaged software components, which come with pre-packaged behaviour and hence pre-packaged expectations of usage patterns. At the same time, the toolkit designer's goal is to produce a set of components that can be used in as many different applications as possible, thus making the application developers's job easier. More specialised toolkit components make the application development task easier, but limit the range of applications which can be developed. The most fundamental issue in toolkit design is to resolve this dilemma.

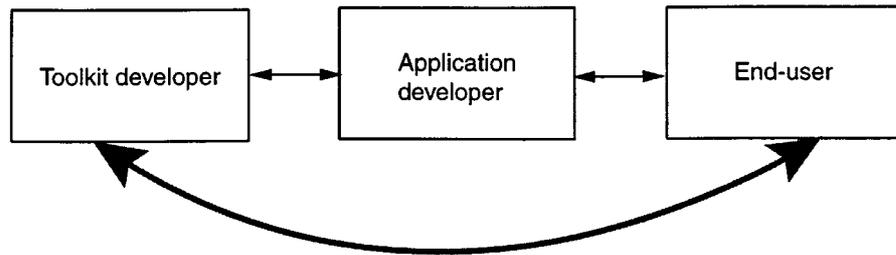


Figure 1. For the toolkit developer, anticipating the needs of application developers means anticipating the requirements of as-yet-unformated applications and the needs of as-yet-unknown users.

Typically, the two levels of the design problem – the design of the toolkit for the purposes of the application developer, and the design of the applications for the purposes of the user – are addressed independently. Although the needs and requirements of specific applications and their users clearly play a major role in the design of a toolkit, the decisions about toolkit design and application design are usually made in isolation. In this paper, we aim to develop an approach which brings them together, and which looks at software toolkit design in terms of the constraints that toolkits introduce for the users of applications which might be generated from that toolkit. Our concern is with an approach to toolkit design that preserves the benefits of reuse but supports application-specific programming and adaptation of the toolkit itself.

We look at this problem particularly in the domain of collaborative systems. Although the issues we raise for toolkit design are general ones, we believe that the demands of collaborative applications demonstrate them particularly well, not least because of the range of ways in which different groups can engage in superficially similar tasks.

We begin in Sections 2 and 3 by looking at the problem of flexibility in two domains. First we explore flexibility in the domain of collaborative infrastructures, and look at the approaches taken by previous work to support a range of applications with a limited set of toolkit components. Second, we examine flexibility in the context of use – that is, flexibility required by the social and interaction dynamics of collaboration. Our thesis is that there is a fundamental tension between these two forms of flexibility, made manifest in the ways in which software infrastructures embody commitments to particular working styles – commitments that may interfere with the requirements of both application developers and end-users. Section Four introduces a set of issues that highlight this tension.

From our discussion of the problem of flexibility we will then consider the ways in which software toolkit developers have addressed these problems, and consider toolkit flexibility in terms of the notions of genericity, extensibility, and specificity. We will then outline our approaches, and give illustrations from two systems we have developed – *Intermezzo* (Edwards, 1996a) and *Prospero* (Dourish, 1996a).

The details of these toolkits have been presented elsewhere, and we will not dwell on them here. Rather, our intention is to step back and consider the relationship between the techniques that they employ, and to try to learn from their orientation to the problems of flexibility.

The design of these toolkits is based on a reconsideration of the boundary between the application and toolkit. Like conventional toolkits, they offer application programmers common structures and run-time behaviours; but unlike conventional toolkits, they provide application programmers with the means to incorporate their own functionality to specialise those structures and behaviours to the needs of specific applications. Although *Intermezzo* and *Prospero* take different approaches, the common elements in their design suggest new opportunities for a model of collaborative toolkit design which helps bridge the gap between infrastructure and use by creating toolkits that can incorporate application-specific detail.

2. Flexibility in CSCW infrastructure implementation

The need to support a variety of application behaviours, and to support them across a wide range of potential implementation substrates (e.g. different network topologies and characteristics) has led collaboration toolkit developers to confront a bewildering array of potential areas of flexibility and control. The basic entities out of which a collaborative application might be constructed using a particular toolkit – abstractions such as “shared object” or “workspace” – embody a range of implementation decisions and options. We refer to this level of system provision as “infrastructure”, since it provides a common level of technological support on top of which applications will be implemented. We consider a number of elements of this infrastructure here.

2.1. ASPECTS OF INFRASTRUCTURE FLEXIBILITY

Data distribution concerns the ways in which the computational representations of data available to participants (the text in a collaborative writing system, the records in a multi-user database or the marks on a shared whiteboard) will be located, found, accessed and moved in a collaborative system. For example, a single copy of any data item may be located on a central server, or multiple copies may be accessible at each node in the network. Data objects may be stored at fixed locations through the course of a session or collaboration, or may move around according to the pattern of access which emerges. Access to this distributed data must be regulated so as to maintain consistency in each user’s view. Such *consistency control* may be achieved through regulating access to the shared data store, or by requiring clients to obtain locks on data items, or by outlawing sequences of action which might result in inconsistency. *Access* to the data store may be governed by permission settings, by roles, or by opportunity. *Sessions* may arise

simply through the individual actions of collaborators, or may be set up explicitly, and might be managed on an invitation basis, or regulated by an individual.

Dourish (1995a) outlines three aspects of flexibility in the design of CSCW systems. *Static* flexibility refers to the extent to which the system can support a variety of collaborative needs (such as “synchronous” or “asynchronous” work.¹) *Dynamic* flexibility reflects a system’s ability to respond to the changing circumstances of a collaborative session (such as changes in group membership). *Implementation* flexibility refers to the ability of a system to operate over a range of implementation substrates (such as different forms of network or network topology, or different approaches to data store management). To one extent or another, each of these is a concern for the development of any toolkit, just as a failure of any, in a toolkit or an application, can itself move from being an infrastructure problem to one of use.

2.2. EXISTING APPROACHES

To illustrate the ways in which toolkit developers attempt to offer flexibility to support different applications, we will briefly discuss programmer control in three sample systems – Suite, Oval and COLA.

Suite is multi-user application toolkit which focuses on editor-based interfaces implemented over a shared data model (Dewan, 1990; Dewan and Choudhary, 1992). Programmer control over aspects of the infrastructure is provided through the coupling mechanism between *shared active variables* and *interaction variables* (Dewan and Choudhary, 1995). Shared active variables are the abstract variables of the shared workspace, available (in one form or another) to the different participants in a collaborative session, while interaction variables are the local proxies of the shared active variables in any one user’s interface. In other words, if three people are editing a form simultaneously, there is one “shared active variable” representing the contents of a particular field, and three “interaction variables”, one in each person’s interface. User interaction happens “at” the interaction variables, and coordination between users happens “at” the shared active variables. The coupling between these variable types, then, is essentially where the multi-user interaction resides. Interaction variables are organised into *coupling sets*, which are in turn described by *coupling attributes* (controlling the degree of sharing of data values, presentation, etc.) and *transmission attributes* (controlling the circumstances under which one user’s changes will be reflected in the interfaces of others). By allowing programmers to select different attribute parameters for different sets of variables, Suite provides the opportunity to manage coupling fluidly for different application needs.

Oval is a “radically tailorable” toolkit for collaborative applications which is derived from a considerable body of research into the use of “semi-structured” information systems (Malone et al., 1995). Oval is named for its four basic components – *Objects*, *Views*, *Agents* and *Links* – out of which applications

are constructed. Objects are semi-structured records; Views are visualisations of objects and object collections; Agents are production rules which can be triggered by conditions such as Object field values; and Links are connection between objects, to form networks, hierarchies, etc. Oval's radical tailorability is essentially a form of end-user programming, allowing the construction of applications in terms of these basic components, and an environment which allows this to be done visually and incrementally. Users essentially combine and configure the pre-packaged components (such as particular Views) to create new applications. The "radical" aspect of this approach is that it applies techniques from traditional tailorable systems and uses them to create, essentially, a specialised visual programming language for collaborative application production.

COLA (Cooperative Objects in Lightweight Transactions) is a CSCW support platform based on a distributed system perspective (Trevor et al., 1995). COLA employs two primary mechanisms for achieving flexibility. The first is the use of "object adaptors" (Trevor et al., 1994), which allow objects to present different views and functionality, and to be handled differently from different perspectives within the system. The second, and more wide-ranging, is that COLA rigorously enforces a distinction between *mechanism* and *policy*. This approach is based on a strong separation between, on the one hand, the fundamental components of a system and their inherent behaviours (the mechanisms of the system) and, on the other, the way they are combined in order to create higher-level interactive behaviours (driven by policy). One common example in user interface development is a scroll-bar; the system can provide mechanisms (such as mouse-sensitive regions and a coupling between mouse movement and pane movement), without embodying a commitment to interface policy (such as whether the scroll-bar appears to the left or the right of the contents, which mouse buttons operate it, and where the arrow buttons might appear). Similarly, in COLA, policy decisions (concerning how users will work together) are left to the application, and the toolkit simply provides fundamental mechanisms necessary for the creation of a range of policy-driven instances.

In these and other systems, then, the concerns of flexibility in infrastructure have been a significant element of the development of collaborative toolkits, and these systems illustrate a range of solutions – parameterisation, customisation and the mechanism/policy separation – which have been employed to address them.

3. Flexibility in collaborative application use

The previous section examined the problems inherent in addressing flexibility from the standpoint of infrastructure developers; this section explores flexibility in the context of application use. Studies of cooperative working have repeatedly emphasised the importance of fluidity and flexibility of coordination and activity.

Dourish and Bellotti (1992) report on a study of the use of the ShrEdit shared text editor (McGuffin and Olson, 1992) in cooperative design tasks by small groups

in an experimental setting. In comparison with a number of other cooperative writing tools of the same period, ShrEdit is highly unstructured. The group can share any number of textual documents, in which each author has an individual insertion point, and in which any number of authors can be working concurrently. ShrEdit employs an implicit locking mechanism, but locks at the level of characters, so that authors only come into conflict when two of them attempt to place their edit point at exactly the same character in the document. Changes made by any author are reflected almost immediately in the windows of the other authors (presuming that their view is scrolled to include the areas where the others are working), so that they have real-time access to each other's work.

Dourish and Bellotti's observations repeatedly emphasise the fluid and self-organising nature of the cooperative group activity engaged in by a number of sets of authors. Far from finding the lack of structured support for the cooperative writing process problematic, the authors would negotiate and manage the structure of their collaborative process in a natural and straightforward way. Groups varied in how they did this; some took separate responsibility for different parts of a single, long document, some worked largely in their own documents and then integrated their work towards the end of the experimental period, and others would essentially engage in free-for-all activity over the whole document. Perhaps more importantly, no group worked solely in any one way; rather, the "shared feedback" that the synchronous shared workspace provided to the group as a whole acted as a resource both for the individuals and for the groups as a whole to manage the group process on a moment-by-moment basis. This shared context allowed them to respond to the immediate circumstances of their work, build on what they could see each other doing, and negotiate, both explicitly and implicitly, the informal division of labour by which their activity was organised.

Beck and Bellotti (1993) report on other experiences of co-authoring, in quite different settings; the primary collaboration described was the naturally-occurring collaborative writing of an academic conference paper by two authors separated by six thousand miles and eight time zones. (Elsewhere, Beck (1994) has reported on other studies of collaborative authoring that provide further supporting evidence for the observations made in this study.) A significant finding in this work was that, while the authors would establish a division of labour and a plan for the coordination of their joint work over the document, this pre-arranged plan would be regularly *and unproblematically* ignored in the actual performance of the work. That is, while the authors might divide up responsibility for two sections of document, they would each, in fact, make contributions to the other's section, whether to correct typos or to add substantive material relevant to their own sections, as relevant. In fact, Beck and Bellotti observe, the success of the collaborative process may in many cases depend on just this sort of "opportunistic" action that successfully advances the group's work while failing to uphold the separation of individual activities.

As we can see, the collaborative setting requires a fluid and flexible style of interaction to be effective. This requirement of flexibility in use is, however, often at odds with the choices made by toolkit developers in attempting to provide infrastructure flexibility, and discussed previously. In the next section, we explore the tensions between application infrastructure and application use.

4. The interaction of use and infrastructure

By definition, collaboration involves the activities of multiple users, the presence of whom affects the nature of the systems we build. On the systems side, our applications and toolkits must be cognizant of multiple input streams, must address consistency control, and may potentially have to deal with problems of distribution.

The presence of multiple users also has implications for the social aspects of the systems we create. A multiuser application becomes a medium for interpersonal exchange, with all of the potential concerns (as well as benefits) which that entails (Bentley and Dourish, 1995). Examples of the more socially-oriented aspects of collaborative systems that must be addressed include privacy of participants, awareness, and support for the rapidly shifting roles of the participants in the collaboration.

So, these concerns, the “social” and the “technical”, are deeply intertwined. However, despite this, they are typically *explored* in isolation; not just in different papers, but in different rooms at conferences. Toolkit designers focus primarily on the range of applications they wish to support in motivating and evaluating the flexibility of their toolkits. Those engaging in user studies or studies of collaborative work settings typically do this with reference to the facilities provided by specific applications or the requirements for new ones. In other words, the focus of the toolkit designer suggests that applications can be thought of independently of the situations in which they will be used; while those studying collaborative activities suggest, inversely, that applications can be studied independently of the toolkits and infrastructure facilities which give rise to them. We believe that neither position is workable.

Our concern, then, is not with the implications for applications of flexible patterns of group work, nor with the implications for application development of flexible infrastructure. Rather, we are primarily concerned with the direct interaction between use and infrastructure, and its consequences.

For example, the decision of a particular toolkit to provide locking, or optimistic serialization, or some other concurrency control mechanism affects the style of use of the applications built with that toolkit.² For instance, one way to ensure consistency is to require that any process (and so, any user) must hold a lock on a shared object before that object can be modified. However, a heavyweight locking mechanism like this can interfere with the ways in which a group will organise their work. For example, if each region (or each pixel) of a shared whiteboard is thought of as a shared object, then a heavyweight locking strategy would prevent

two users from drawing lines that cross (since they would both have to modify the same region at once, at the point of intersection). Clearly, this is an inappropriate degree of structure for a casual, lightweight interaction, but since it is imposed by the toolkit (which handles shared objects and consistency management), the application developer has little control.³ Greenberg and Marwood (1994) have examined a range of such concurrency control techniques and their influence on interaction.

Consider as a second example a toolkit that uses the notion of roles to establish access control for data objects. In a shared text editor application, these roles might include editor, author, and commentator. Such roles codify a set of social practices that exist at a certain moment in time, but are not easily adaptable to new situations and cannot accommodate the moment-to-moment shifting interactions which characterize so much of interpersonal communication. Previous work by Dewan et al. (1994) and Neuwirth et al. (1990) has noted such problems.

Both of these examples illustrate how choices in infrastructure design can influence the user interface and multiuser “social interface” of our systems.

The point of these examples is not to say that certain styles of locking or access control are necessarily bad; rather, they show that any choice of implementation strategy can potentially influence application functionality and hence usage. Any toolkit designer endeavors to accommodate a wide range of applications. But designers of toolkits for collaborative systems must take special care they not only allow but *support* the construction of applications that are responsive to the fluid styles of interaction required by collaboration.

5. Aspects of flexibility

In order to consider the relationship between the functionality and variability offered in a toolkit and the requirements of both application developers and end-users, we need some ways to think about toolkit flexibility. We will consider flexibility here in terms of three concepts – *generic*, *extensible* and *specialisable* systems.

Generic systems exploit the fundamental aspect of toolkit design that we introduced at the start of the paper. Toolkit components are designed to be applicable in a range of circumstances; that is, they are generic. Designs emphasise common functionality, independent of circumstances. The extent to which components are generic varies from case to case. For instance, consider a user interface component that provides a scroll bar. One way to do this (the “scroller” approach) is to provide a component that can be attached to a window, and that can be used to move the viewport so that the window pans over content in a larger workspace. Another way to do it (the “slider” approach) is to provide a component that users can move around and that controls the value of an associated variable. Clearly, the slider approach can be used to create a scrollbar (by moving the window contents when the variable changes), although this involves more work by the application

developer than the pre-packaged scrollbar approach; so the slider approach is more generic. In the “generic” mode, then, the variability in the system component is how it is connected to the rest of the system; generic components are built to be “plugged in” to systems in a variety of contexts. The component itself doesn’t change, but is general enough to apply widely.

The idea of extensible systems refers to the opportunities the toolkit offers programmers to extend its functionality by incorporating new objects and behaviours as if they were predefined ones. Extensible toolkits are ones whose functionality can be extended beyond their original boundaries. For example, Suite provides not only particular styles of interface coupling, but also the means to define new ones that can be used in just the same ways as the originals. So, in extensible systems, the variability lies in the way that new behaviours can be made available alongside existing ones, as new parts of the design.

Specialisation refers to the ability to *adjust* (rather than extend) toolkit structures to meet the demands of specific applications and specific application requirements. We draw a distinction, then, between the creation of new objects and behaviours (extensibility) and the modification of existing ones (specialisation). This distinction is a fine one, but crucial; there are many cases in which extensibility fails because of the internal relationships between components. For example, consider a case where we wish to modify a collaborative toolkit so make it suitable for use in presentations, and in particular where we want to give it a shared cursor that is considerably larger than the default. In an extensible toolkit, we could create a new shared cursor object and make it larger and more prominent, but it would only be available in applications that used our new type of cursor. Existing applications, or existing components that used cursors, would be unaffected and would still be using the original cursor. In other words, extensibility creates new functionality, whereas specialisation augments or refines existing behaviour.

Generic, extensible and specialisable techniques are by no means mutually exclusive routes to flexibility in toolkit design. Instead, they are *aspects* of flexibility, reflecting particular styles and approaches. We introduce the terms as a form of characterisation, not as a taxonomy. Toolkits, or even particular techniques, are not uniformly of one sort or another, but tend to combine aspects of each. For instance, take the example of the larger shared cursor again. The original cursor object may have some parameters that could be adjusted to control its appearance. Perhaps it has a “big” flag that doubles its size. Perhaps it offers four different size settings, or perhaps ten. Perhaps it has an internal control for adjusting its scale to any size. Perhaps it offers similar controls over its color, or its transparency. At what point does this sort of parametric control cease to be a form of genericity and begin to be a form of specialisation? There is no clear dividing line; we use the terms instead to reflect aspects of how the toolkit’s flexibility is offered, rather than as absolute categorisations.

The generic, extensible and specialisable approaches give us a framework to consider the issues of flexibility in toolkit design. Generic and extensible toolkits

provide their users (the developers of applications) with the means to create and incorporate new behaviours. In contrast with fixed techniques, they give the application developer much more control, and begin to blur the distinction between “toolkit” and “application” (or between “toolkit programming” and “application programming”). Specialisation blurs that boundary further, by providing a means to incorporate understandings about the application requirements and behaviour into the infrastructure the toolkit provides. In the next section, we discuss two toolkits we have developed and show their approaches to specialisation.

6. New approaches to CSCW toolkit flexibility

We will illustrate these issues and potential solutions with reference to two particular toolkits that we have designed and implemented: *Intermezzo* (Edwards, 1996a) and *Prospero* (Dourish, 1996a). *Intermezzo* is designed to support the *coordination* aspects of collaboration: the tasks associated with rendezvous of participants, awareness, and policy. A principal element in the toolkit’s design is that it reifies the setting in which the collaboration occurs by gathering and exposing information about user activity to applications. Further, applications can modify the behavior of the toolkit based the situational context of the collaboration represented by this activity information. *Prospero* (Dourish, 1996a) deals largely with the areas of distributed data management and consistency control. It exploits an architectural approach called “Open Implementation” (Kiczales, 1992, 1996) in which the abstractions and mechanisms offered by a system (such as a toolkit) can not only be used by its clients (applications), but can also be examined and manipulated, and hence specialized to the needs of particular situations.

While *Intermezzo* and *Prospero* were developed independently, and take different approaches to the problem of toolkit flexibility, they have a number of interesting features in common. First, they are motivated by many of the same concerns with the interaction of toolkit structure and application use; and second, they share a common technical concern with flexibility through specialisation rather than through genericity. This section presents the approaches taken by both systems to provide flexibility to application writers.

Intermezzo and *Prospero* have both been described elsewhere. Our goal here is not to provide another presentation of their structure and use. Instead, we use the two systems to illustrate the specialisation approach to infrastructure flexibility and customisation, as a stepping stone towards a fuller discussion of the approach to infrastructure development based on the relationship between infrastructure and use.

6.1. EXAMPLE: FLEXIBILITY THROUGH AWARENESS IN INTERMEZZO

Intermezzo addresses flexibility by providing an infrastructure by which some of the technical aspects of collaborative applications can be mediated by input about

the social setting in which the collaboration occurs. For example, a collaborative writing application may need to accommodate different styles of session management or access control during the course of a collaboration based on the shifting goals and needs of the participants. To support this form of dynamic flexibility, *Intermezzo* brings information about the participants of the collaboration, their activities, and their environment into the realm of the toolkit. This information is made available to applications (modulo privacy restrictions) in a machine-parsable format. Further, the toolkit itself uses this information about situational context to regulate its internal operations.

Intermezzo relies on information about the state of the world that is represented as a database of objects describing user activity. Representations of activity are hierarchical and allow application-specific “views” of the world state at any number of semantic levels. Activity information has a structured format and supports links between related objects. In the *Intermezzo* model, applications themselves are responsible for maintaining the global view of the world, and the toolkit provides support (much of it automatic) for publishing and updating this information. For example, when an *Intermezzo*-based collaborative writing tool is started, it will publish information describing itself (the tool), its users, and the documents being edited. The format of the published data allows it to be searched, updated, and viewed easily by other applications.

The principal way in which *Intermezzo* provides flexibility to applications is by allowing them to adapt not just their own behaviour, but also the behavior of the toolkit in reaction to the changing dynamics of the world in which they are run. Applications change toolkit behaviour in two ways. The first is by downloading code into the runtime system that runs in response to changes in world state. This downloaded code can be tied to any change in the environment, and can directly affect application or toolkit state.

Downloaded code, which is written in an extended version of the Python language (Van Rossum, 1995), can be directly executed by the runtime system at the time it is transmitted; can be set to “fire” when a particular change on a specific object occurs in the activity database; or can fire whenever the database state achieves a certain “pattern,” as described via a pattern matching language. By associating portions of application code with objects “in the world,” application behavior can be directed and influenced by changes in situational context.

The second way applications can change the behavior of the toolkit is through the use of situationally-based access control (Edwards, 1996b). *Intermezzo* uses strong, cryptographically-secure access control throughout. The objects to which access is controlled are not just the objects created by the application to maintain domain-specific application state, they also include objects used internally by the toolkit. By constraining access to toolkit functions and data, applications can effect global changes in the behaviour of the toolkit itself. For example, application code can provide situational control over inter-application behaviour such as session management through this technique (Edwards, 1994).

The actual access control settings of various objects – both in the application and in the toolkit – are under the control of a language-based policy system that uses information about the situational context as an input. Applications use the access control system by writing “policies” – which are essentially access control templates – and binding them to “roles.” Unlike traditional roles, however, the membership of an Intermezzo role is *described* by a predicate function, rather than *defined* by a membership list. This technique allows applications to be written in terms of general descriptions of user behavior and context, rather than more limiting specific definitions. For example, an application could respond to general description of “the set of people currently in my lab,” rather than the more static and restrictive *a priori* enumeration of the names of specific people often found in my lab.

By mediating the systems-oriented aspects of access control, session management, and awareness with input about the context in which a collaboration is occurring, Intermezzo provides applications with *dynamic* flexibility – the ability to match and adapt to the changes in a group’s interactions over time. Intermezzo enables applications to more easily support groups that move fluidly between various styles of work.

6.2. EXAMPLE: FLEXIBLE CONSISTENCY CONTROL IN PROSPERO

One of Prospero’s major areas of concern is consistency control for collaborative applications. Consistency control is the mechanism by which a collaborative system ensures that the potentially simultaneous actions of multiple users over a shared data space do not result in inconsistent views of the data space. For instance, consider a medical system supporting shared access to patient records. To ensure speedy response, the system might replicate the database, placing copies of the patient records at different points in the network. If two users were to simultaneously update the same record, and their changes were to be made to their own copies of the records before being sent across the network, then an inconsistency would have arisen, since their respective copies of “the same” record would display different information.

Since the shared spaces of collaborative systems can be modeled as databases, most CSCW systems have looked to the models provided in distributed database design for consistency mechanisms. The most common mechanisms in database design are forms of “locking,” in which a “lock” for data to be modified is obtained before the changes can be made. The locking mechanism forces clients to declare their intent to update records before the updates are actually made, and so provides an opportunity to avoid inconsistency by having the system refuse to grant locks if the actions might conflict with others. If multiple clients simultaneously request “read locks” (locks obtained by clients which intend to read data), then they can all be granted, because there is no opportunity for inconsistency to arise, since read operations will not change the data. However, if one client has already been

granted a “write lock” for a piece of data, then no other client will be granted a second “write lock” until the first client has finished, because two simultaneous writes might result in inconsistency – a write/write conflict. In many situations, an outstanding write lock might also prevent any further read locks being granted, because the reading client might receive data which has become out of date.

The database model, with its read and write semantics, is very generic; it supports a wide range of potential collaborative activities. However, drawing on the interaction of use and infrastructure outlined above, we can identify a number of problems. Locking out operations because of potential conflicts can interfere with the smooth progress of collaboration; and the read and write semantics are such that many activities look like conflicts even if, in fact, they will not lead to inconsistency. For example, consider two users working on a bibliographical database, which records citation details for publications. Adding two records at once will not lead to conflicts; either they’re the same record (in which case, they match to the same new record to be added, which should be an acceptable operation), or they’re different records (in which case, they should both be added). However, to the database substrate, these will look like two write operations, and so a conflict will be flagged. In other words, the configuration of the infrastructure, a database storage layer with a conflict avoidance mechanism based on read and write semantics, will interfere with the execution of collaborative work, forcing users to interleave their activities.

Prospero’s approach is to allow consistency control to be specified in terms of the domain semantics of particular collaborative applications (Dourish, 1996b). The paradox, of course, is that this makes consistency management a toolkit concern, but yet the toolkit, as a general facility, must be free of exactly the type of specific application features from which we want to construct this mechanism. The toolkit itself, then, does not operate in terms of pre-defined application semantics (although it provides a reusable and extensible core set of potential properties applicable in some range of situations). Instead, it provides the framework *within which* they can be defined. So, the developer of the bibliographical database example would describe the semantic properties of application operations (such as the non-destructive writes implied by adding new entries to the database), and then the toolkit can manage consistency in these terms. The toolkit is specialised to the needs of the particular implementation.

Prospero uses a pairwise comparison of operation properties, much like the read/write comparison model, to decide when sets of operations might conflict. However, using the metalevel control (that is, the mechanism for programmers to modify or augment internal toolkit facilities) allows the programmer to gain control over this process and cast it in terms of the application specifics. Since the comparison is done in terms of operations which are meaningful at the application level, rather than simply those meaning at the infrastructure level, the toolkit can support a range of behaviours specific to each situation, which would be disallowed by a traditional approach implemented over a standard database model. In this

way, a link is achieved between the configuration of the infrastructure and the particularities of each particular application.

7. The design of flexible CSCW toolkits

Intermezzo and Prospero illustrate two different approaches to achieving flexibility in toolkit design. However, they share a common set of concerns, and in particular, a common reaction to the traditional mechanisms of toolkit flexibility.

The standard approach to flexibility in toolkit design (in any domain, not just CSCW) is through *generic design*. This approach involves the design of *generic* toolkit facilities, applicable to the widest range of applications. For instance, a standard database model, with access control and consistency management based on “read” and “write” access, is highly generic, and can therefore serve as an infrastructure for a very wide range of CSCW applications. Access to the shared data store can be implemented in terms of read and write operations, in whatever way the application might then use those reads and writes to support the specific application needs. As a generic mechanism, then, the traditional approach takes the read/write database model as a useful basis for the design of a toolkit which can be used to build CSCW applications of all sorts.

Intermezzo and Prospero take almost the *opposite* approach to the provision of flexibility in CSCW toolkits. Their primary concern is with the design of effective, functional CSCW applications, and so they are concerned with the highly *specific* behaviours observed in collaborative settings. As we have explored in this paper, these behaviours depend in crucial ways on the variety of infrastructural configurations within which cooperation takes place.

The implication, then, is that applications require widely different infrastructure provisions, rather than a single highly generic mechanism onto which a variety of application requirements can somehow be “mapped”. The question of flexibility in Intermezzo and Prospero, then, is how the toolkit can be specialised and adapted to the particular needs of a given collaborative situation, rather than how that particular situation can be described in terms of whatever toolkit components happen to be lying around. The latter would argue for an arbitrarily large “vocabulary” of toolkit building blocks. Instead, we argue for the ability to specialise the toolkit in a “deep” way to the needs of its applications. The generic design approach attempts to remove from the infrastructure any dependency on particular contexts of use, but it provides no means for context to be re-established. Specialisation gives us a way to incorporate context again, and this is what Intermezzo and Prospero do. In doing this, though, they focus on different concerns.

Intermezzo primarily addresses *dynamic flexibility*: the ability of applications to track the changing circumstances in which they are run and adapt their behavior accordingly. The infrastructure mediates its own operation through application-supplied code that takes as input a repository of information about the application’s context. Architecturally, Intermezzo takes a “programming language” approach,

providing specialised languages in which the needs of specific applications can be described.

Prospero primarily addresses *implementation flexibility*: the ability of applications to “open up” the internal constructs of the toolkit and adapt it to their particular requirements. Applications can modify and specialize toolkit-internal constructs, which the toolkit will then use to provide application support. At the same time, it provides novel mechanisms such as the divergence/synchronisation approach to distributed data management (Dourish, 1995b) which generalise across the traditional boundaries of collaborative systems, as a means to address (by reformulating) issues of static flexibility. Architecturally, Prospero is based on the “Open Implementation” approach which makes aspects of the toolkit, traditionally only available for “use” by applications, amenable to examination, modification and control. (It is perhaps interesting that the Open Implementation approach is derived from work in the theory and design of programming languages. Perhaps this work lends further support to the theory that Computer Science is a bell-shaped curve around programming language design and implementation.)

The different architectures, however, belie a shared underlying technical goal: to let application code “push semantics” into the toolkit to accomplish particular goals. This goal is motivated by both technical and social or observational needs. Technically, providing such a mechanism is a solution to the perennial problem of flexibility in toolkit design. Socially, semantic features (the “meaning” and behavioural consequences of toolkit configurations) are seen to be crucial in the emergence of forms of group behaviour, and so are ineliminably the concern of application developers and users, not of toolkit designers, who, by definition, are separated from the situations in which the toolkit mechanisms will be realised in particular applications and put to use in particular circumstances.

8. Conclusions

The days when CSCW applications were built by hand, from scratch, are long behind us. Not only is such an approach increasingly technologically impractical (as systems get larger and more complex, and as users demand more and more from the applications they use), but it is also unviable commercially. Like applications in any other domains, CSCW applications depend on toolkits to provide them with standard, reusable approaches to application design and mechanisms to be deployed. The critical concern for the designer of a CSCW toolkit is the range of behaviours (and hence of applications) which can be supported.

The design of toolkits, and the flexibility implied by the range of mechanisms they provide and the range of ways in which they can be combined, has typically been approached purely as a technical problem. However, the radical degrees of flexibility implied and required by collaborative work, and observable in studies of groups working together, undermines this position. Observational studies have pointed to the range of ways in which collaborators organise (and reorganise) their

work, and these studies illustrate how group behaviour is critically dependent not simply on the “high-level” facilities which collaborative applications offer, but also on the “low-level” issues of infrastructure configuration which lies beneath these applications.

These observations suggest that the problem of toolkit flexibility is not one of providing generic mechanisms, but rather is one of gaining control over the relationship between the application and the infrastructure. Semantic issues cannot be locked inside a toolkit, inaccessibly to the application designer and the user; rather, the semantics required of the toolkit come from applications and circumstances of use. In other words, the focus for design should be the three-way relationship amongst toolkits, applications and use, rather than the traditional pair of two-way relationships (toolkit/application and application/use). In this way, the problem of tailorability – incorporating local modifications and specialisations to adapt a tool to the needs of a particular set of users – is a problem not only for interface design, but for the deeper areas of system development.

We have outlined and motivated these problems by appeal to a number of studies, and shown how two toolkits, *Intermezzo* and *Prospero*, tackle these problems. *Intermezzo* and *Prospero* were designed independently, employ different computational architectures and address different domains of concern to the designer of a CSCW toolkit. However, they share a concern with the relationship between application and infrastructure, and with making this accessible so that application developers can free themselves from the rigid models which are embodied (and often hidden) in CSCW toolkits. *Intermezzo* and *Prospero* both emphasise flexibility through application specialisation, rather than flexibility through abstraction and genericity.

The general concern which motivates these two systems, and the mechanisms which they embody, point the way towards a new model of toolkit design which is grounded not simply in the technical concerns of generic infrastructure mechanisms, but rather which is based in understandings of how collaboration works.

Acknowledgments

Prospero was designed and implemented while Paul was employed at the Rank Xerox Research Centre (formerly EuroPARC) and studying at University College, London; *Intermezzo* was designed and implemented while Keith was studying in the College of Computing at Georgia Institute of Technology. We would like to thank the voices that spoke to us and told us what to say. Some of these belonged to Beth Mynatt, John Stasko, Dik Bentley, Jon Crowcroft, Beki Grinter, and Prasun Dewan.

Notes

1. We do not believe these terms are unproblematic – hence the “scare quotes” – but use them to refer to a set of relatively well-understood application usage situations recognised in the literature.
2. We will discuss approaches to consistency control in more detail later.
3. The most likely strategy in this case would be to write the whiteboard application in such a way that each user has a private cache which is then updated in the background, so that users don't have to deal with locks. This is a more complex structure that the application requires, but is necessitated by the underlying structure of the toolkit. This is what Kiczales (1992) called “coding between the lines.”

References

- Beck, E. and V. Bellotti (1993): Informed Opportunism as Strategy. In *Proceedings of the Third European Conference on Computer-Supported Cooperative Work ECSCW '93, Milano, Italy*. Dordrecht: Kluwer.
- Beck, E. (1994): *Practices of Collaboration in Writing and their Support*. PhD thesis, University of Sussex (Technical Report CSRP 340, ISSN 1350 3162).
- Bentley, R. and P. Dourish (1995): Medium vs. Mechanism: Supporting Collaboration Through Customisation. In *Proceedings of the European Conference on Computer-Supported Cooperative Work ECSCW '95, Stockholm, Sweden*. Dordrecht: Kluwer.
- Crowley, T., P. Milazzo, E. Baker, H. Forsdick and R. Tomlinson (1990): MMConf: An Infrastructure for Building Shared Multimedia Applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '90, Los Angeles, California*. New York: ACM.
- Dewan, P. (1990): A Tour Through the Suite User Interface Software. In *Proceedings of the ACM Symposium on User Interface Software and Technology UIST '90, Snowbird, Utah*. New York: ACM.
- Dewan, P. and R. Choudhary (1992): A High-Level and Flexible Framework for Implementing Multi-User Interfaces. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '92, Toronto, Canada*. New York: ACM.
- Dewan, P., R. Choudhary and H. Shen (1994): An Editing-based Characterization of the Design Space of Collaboration Applications. *Journal of Organizational Computing*, vol. 4, no. 3, pp. 219–240.
- Dewan, P. and R. Choudhary (1995): Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Computer-Human Interaction*, vol. 2, no. 1, pp. 1–39.
- Dourish, P. and V. Bellotti (1992): Awareness and Coordination in Shared Workspaces. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '92, Toronto, Canada*. New York: ACM.
- Dourish, P. (1995a): Developing a Reflective Model of Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, vol. 2, no. 1, pp. 40–63.
- Dourish, P. (1995b): The Parting of the Ways: Divergence, Data Management and Collaborative Work, in *Proceedings of the European Conference on Computer-Supported Cooperative Work ECSCW '95, Stockholm, Sweden*. Dordrecht: Kluwer.
- Dourish, P. (1996a): *Open Implementation and Flexibility in a CSCW Toolkit*. PhD dissertation, Department of Computer Science, University College, London.
- Dourish, P. (1996b): Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '96, Boston, MA*. New York: ACM.

- Edwards, K. (1994): Session Management in Collaborative Applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '94 Chapel Hill, North Carolina*. New York: ACM.
- Edwards, K. (1996a): *Coordination Infrastructure in Collaborative Systems*. PhD dissertation, College of Computing, Georgia Institute of Technology, Atlanta, Georgia.
- Edwards, K. (1996b): Policy and Roles in Collaborative Applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '96, Boston, MA*. New York: ACM.
- Greenberg, S. and D. Marwood (1994): Real-Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work CSCW '94, Chapel Hill, North Carolina*. New York: ACM.
- Hill, R., T. Brinck, S. Rohall, J. Patterson and W. Wilner (1994): The Rendezvous Architecture and Language for Multi-User Applications. *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 2, pp. 81–125.
- Kiczales, G. (1992): Towards a New Model of Abstraction in the Engineering of Software. In *Proceedings of the IMSA Workshop on Reflection and Metalevel Architecture, Tokyo, Japan*.
- Kiczales, G. (1996): Beyond the Black Box: Open Implementation. *IEEE Software*, pp. 6–11, January.
- McGuffin, L. and G. Olson (1992): *ShrEdit: A Shared Electronic Workspace*. CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory, University of Michigan.
- MacLean, A., K. Carter, T. Moran and L. Lovstrand (1990): User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the ACM Conference on Human Factors in Computing Systems CHI '90, Seattle, Washington*. New York: ACM.
- Malone, T., K.-Y. Lai and C. Fry (1995): Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. *ACM Transactions on Computer-Human Interaction*, vol. 13, no. 2, pp. 175–205.
- Nardi, B. and J. Miller (1991): Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development. In Greenberg (ed.): *Computer-Supported Cooperative Work and Groupware*. Academic Press.
- Neuwirth, C.M., D.S. Kaufer, R. Chandhok and J. Morris (1990): Issues in the Design of Computer Support for Co-authoring and Commenting. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '90*. New York: ACM, pp. 183–195.
- Roseman, M. and S. Greenberg (1996): Building Real-Time Groupware with GroupKit, a Groupware Toolkit. *ACM Transactions on Computer-Human Interaction*, vol. 3, no. 1.
- Trevor, J., T. Rodden and J. Mariani (1994): The Use of Adapters to Support Cooperative Sharing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '94, Chapel Hill, North Carolina*. New York: ACM.
- Trevor, J., T. Rodden and G. Blair (1995): COLA: A Lightweight Platform for CSCW. *Computer Supported Cooperative Work*, vol. 3, pp. 197–224.
- Trigg, R., T. Moran and F. Halasz (1987): Adaptability and Tailorability in Notecards. In Bullinger and Shackel (eds.): *INTERACT '87*. North Holland.
- Trigg, R. and S. Bodker (1994): From Implementation to Design: Tailoring and the Emergence of Systematization in CSCW. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW '94, Chapel Hill, North Carolina*. New York: ACM, pp. 45–54.
- Van Rossum, G. (1995): *Python Reference Manual Release 1.3*. October 13 (available as <http://www.python.org/doc/ref/ref.html>).

