A Divergence-Based Model of Synchrony and Distribution in Collaborative Systems

Paul Dourish

Technical Report EPC-1994-102

Copyright © Rank Xerox Ltd 1994.

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

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

A Divergence-Based Model of Synchrony and Distribution in Collaborative Systems

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

Abstract

Rather than embodying the behaviour of toolkit components directly in their implementation, metalevel techniques provide programmers with ways to override and revise them. The style of very flexible design is derived from specialisable generic models of application behaviour. This paper presents a generic model of distribution and synchrony in cooperative systems. Organised around the explicit management of divergence between parallel streams of activity, it links usage and implementation issues, as well as supporting an important class of applications which are not well supported by more traditional CSCW models. Examples are given showing this model as a basis for flexible design.

1 Introduction

CSCW toolkits, such as MMConf [Crowley *et al.*, 1990], Rendezvous [Patterson *et al.*, 1990] or GroupKit [Roseman and Greenberg, 1992] are a common basis for building cooperative applications. They support development by providing collections of ready-made groupware components and behaviours which can be combined and used in new applications. Using a toolkit lets the developer focus on the specific requirements of the application under development—argumentation support, map browsing, or whatever—without worrying about common CSCW features such as distributed interface management, which the toolkit will provide.

However, this reliance on "black box" components provided by the toolkit can become problematic. Toolkits embody models of system behaviour in their components, and these models are fixed and hidden from the application developer, locked behind "abstraction barriers". What can the programmer do when the standard components provided by the toolkit don't match an application's needs? For instance, how can a WYSIWIS toolkit support applications requiring different views for different participants?

In general, the application developer must hope that the toolkit's author has anticipated the requirements of applications constructed from the toolkit. If the toolkit author has not incorporated these requirements into the toolkit components, then the components cannot be adequately used. Any given toolkit is inherently limited in the range of applications or application domains which it can support.

I am concerned with the provision of *generic* and *revisable* models of behaviour within toolkits. Being generic, they describe a wider range of applications than the specific models of traditional toolkits. Being revisable, they can be adapted to new situations which would not be well supported by their default configurations. So by designing a toolkit around generic behaviours, we can make it much more flexible.

This focus of this paper is the development of a generic model of *distributed data management* and *interactional synchrony* in collaborative systems design. There are two primary uses of a model of this kind. First, it can be directly incorporated into a toolkit, providing a framework which can support a wide range of application behaviour. Second, as an abstract structure, it can be useful in analysing and characterising CSCW systems. This two-fold value is an important feature of the model; it means that it can be used to bridge between the issues of system use and system design.

The model tackles a number of problems with more traditional characterisations of the CSCW design space. In particular, it links the issues of distribution and synchrony, which are more usually treated independently; it provides a bridge between implementation strategies and their implications for user interaction; and it provides explicit support for a range of applications which are outside the scope of standard approaches, and which I class as *multi-synchronous*.

Before describing the structure and scope of this model, it's necessary to explain some of the context of this work. The following section discusses the way in which generic models can form the basis of flexible design when directly incorporated into open toolkits. I will then go on to discuss the traditional view of the CSCW design space and some problems with this approach which lead to a new model based on the management of *divergence*.

1.1 Background and Approach

My research is concerned with the provision of flexible, extensible and revisable system structure in a toolkit for CSCW systems. Using a technique called *metalevel programming* in combination with standard object-oriented approaches, I am building a toolkit based around generic descriptions of system behaviour. The system allows these generic models to be specialised and refined, to provide better support for particular applications [Kiczales *et al*., 1991]. The technical details of this approach are not appropriate for this discussion, and are presented elsewhere (see, for example, [Dourish, 1994]). However, a brief overview is useful in illustrating the importance of generic system models.

The metalevel approach augments the traditional elements of a program—those which describe a system's behaviour (the "base level")—with a specification of *how that behaviour is supported* (the "metalevel"). At its simplest, this can be thought of as a distinction between "what" and "how". In most systems this metalevel component is hidden in the implementation; but in *reflective* systems it is revealed to the programmer. In addition, the system maintains a *causal connection* between base level behaviour and the metalevel description [Smith. 1982; Maes, 1987]. The result is that system developer can use these facilities to describe how base level functionality should be mapped onto real behaviour. Through changes at the metalevel, the system's implementation can be tailored to for particular needs or circumstances. The causal connection guarantees that metalevel changes will be appropriately reflected in the system's behaviour.

For example, while the "base level" of a programming language comprises components such as variables, arrays, lists and control structures, the metalevel might describe how the data structure elements map onto memory and how control structures are implemented. A programmer who finds the abstractions offered by the programming language inappropriate for a particular application, then, can use the metalevel to adjust the implementation to suit the application's particular needs.

Reflection has been primarily developed as a technique for flexible programming language design; however, it is applicable to other types of toolkits and frameworks. In each case, the approach "opens up" an implementation and allows restrictive design decisions to be revised.

The structure of the metalevel—that is, the components, behaviours and the ways in which they can be manipulated—reflects the range of possible system behaviours at the base level. So, the process of designing a reflective toolkit is rooted in the search for generic models of systems behaviour. The generic model at the metalevel provides the framework within which specific designs can be supported. Within this framework, an implementation can be thought of as a description of the particular infrastructure support needs of an application. The model provides the terms in which this description is given. Thus, *descriptive power* is an important characteristic of models, giving a measure of the applicability of the framework to a range of application domains.

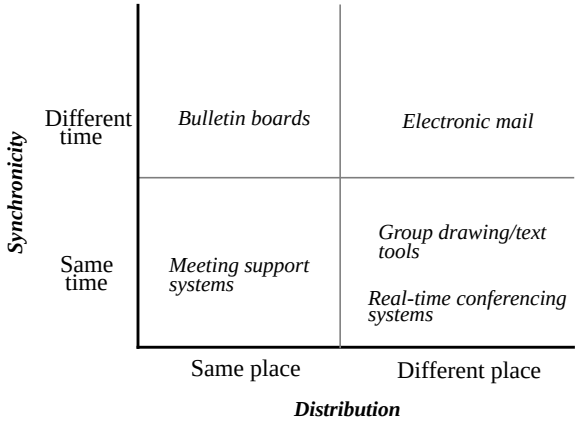

FIGURE 1: The traditional space/time characterisation of CSCW systems.

Reflective techniques of the sort outlined here provide a way for generic models to be directly incorporated into systems, providing a basis for extremely flexible application design. The model of distribution and synchrony presented in this paper is one step towards a reflective toolkit for collaborative applications. Elements of this model are described in sections 3–5; first, we will consider problems with other generic characterisations of the design space for CSCW.

2 The CSCW Design Space

Despite its relatively short history, there have already been a number of attempts to characterise the scope and structure of CSCW. Perhaps the earliest of these models is the space/time model shown in figure 1, which remains in widespread use (see, for example, [Johansen, 1988]; [Ellis *et al*., 1991]; [Rodden, 1991]; [Olson *et al*., 1993]). It characterises cooperative work along two orthogonal dimensions; work occurring at the same or different times (synchronicity), and in the same or different places (distribution).

For the purposes of this discussion, I will take two different perspectives on this diagram. From the *usage* perspective, we can think of the space dimension as concerning the location of collaborating individuals, and the time dimension as describing the temporal distribution of their actions. From the *design* perspective, we can think instead about the location and synchronicity of the computer-based representations of the artefacts which support their work, and of the interactions between them. So from the usage perspective, this model can be employed to describe working situations, in the search for appropriate support mechanisms;

while from the design perspective, it is used either as a way of delineating the scope of a system, or even as the basis of a system architecture.

2.1 Flaws

While the space/time characterisation remains a frequent basis for descriptions of the field as a whole, a variety of criticisms have been leveled at it. It has been found increasingly inadequate as a descriptive tool, and even less satisfactory as a basis for design. For the purposes of this paper, I want to consider three problems in particular: first, the issue of *synchronicity of interaction*; second, the *orthogonality* of the space/time dimensions; and third, the lack of support for *multi-synchronous* applications.

2.1.1 Synchronicity

The division of the synchronicity dimension—concerning the temporal relationships between the actions of collaborators—into "same time" and "different time" is very misleading.

First, this is not a simple two-way distinction; there is a spectrum of interactional styles which cannot be broken down into these two classes, however broadly they are defined. Whether we see interactions occurring at the "same" or "different" times is entirely dependent on how we analyse them. We will get different answers when we use different "grain sizes" or "time-bases" of the analysis. So, users may be engaged in a collaborative session at the same time, but performing individual tasks at different times; or tasks performed at the same time may be seen as being made up of differently-timed turns.

Second, from the design perspective, it can be extremely problematic to think of the synchronicity of interaction as a feature of the design. Synchronicity emerges as an *aspect of use*, conditioned by a variety of external factors. For instance, electronic mail is normally considered to be a paradigmatic example of "asynchronous" interaction; but it can easily be seen as enabling *synchronous* interaction in the kind of rapid-fire, one-line exchanges which often occur between experienced email users. Similarly, a group using a "synchronous" shared drawing program might establish a private "baton-passing" protocol, which essentially regulates fine-grained asynchronous behaviour. In each case, the synchronicity of the group's interactions depends on their use of a tool, not its design. Indeed, as a consequence, the synchronicity of interaction is dynamic, changing in response to the activities of individuals and of the group as a whole.

These examples argue that synchronicity is an emergent property of an interaction and is subjectively assessed from the point of view of a given participant. From this perspective, it seems to be a poor basis for a characterisation of the design space.

2.1.2 Orthogonality

In characterising applications around the issues of space and time, the model presents distribution and synchrony as independent, orthogonal dimensions.

However, from the design perspective, we cannot think of distribution and synchronicity as independent. The range of interaction styles which an application supports (from fine-grained "synchronous" working to regimented baton-passing to "asynchronous" draft-passing) depends on the way in which distributed data is managed and coordinated. For example, a centralised data model can introduce serialisation and latency effects which affect the access patterns of the group; and various consistency strategies for replicated data will also impose restrictions on interactional synchrony in order to operate. The choice of distribution strategy will constrain synchronicity at the higher level. Distributed data management is the "space" dimension from the design perspective.

So synchronicity and distribution are not independent; design decisions may affect either or both aspects of the system. They are not separate and distinct bases for a characterisation of cooperative support systems.

2.1.3 Multi-Synchronous Applications

The space/time model omits a large class of important applications—those which I term *multi-synchronous*.

The traditional model separates *synchronous* applications, such as real-time collaborative text editors, from *asynchronous* applications, such as source code control systems. In [Dourish and Bellotti, 1992], the concept of *semi-synchronous* systems was suggested as a bridge between these two by compiling elements of each. However, all three approaches are based on the notion of a *single thread of control* which is either collaboratively produced in real-time by the actions of a number of users, or passed back and forth between users as needed.

Multi-synchronous applications involve simultaneous but disconnected working; they manage more than one thread of control. This situation might arise, for instance, when collaborators each take copies of some shared artefact on separate laptop computers and process these copies individually before coming back together again. At a finer grain, these independent but parallel interactions can also result from network failures and partitioning.

In situations like these, different users' activities may proceed at the same time, but generally these activities are performed in the absence of any connection with other users. Information exchange only takes place when the users regroup. The standard model leaves no space for applications like these. Both the use of the system and the shared updates may be either synchronous or asynchronous (or both!). This

style of working is not captured by the space/time model, which is based on a single thread of control.

2.2 Reviewing the Space/Time Model

A generic characterisation of the design space for collaborative systems is a useful tool for analysis, comparison and design. However, it's clear that the space/time model is problematic in a number of respects. These problems, and especially the failure to handle multi-synchronous applications, suggest that general structure based around synchronicity and distribution is not an appropriate basis for the generic descriptions in a metalevel toolkit.

It would seem, then, that we must look for a new model to act as the basis of a toolkit of this sort. We can identify a number of desirable properties for such a model. Given the way in which activities and implementation are linked, we would like such a model to be able to describe both user-level and system-level activities. It should describe a wide range of applications with a small number of concepts. In particular, as the basis for a toolkit, it should be based around to specialisation as a means to move from the general to the particular.

3 Creating a New Design Framework

In trying to find a new basis for a generic descriptive model of the CSCW design space, we must start from some basic questions about the activities which CSCW systems are trying to support. What determines the various styles of interaction in which collaborators engage? What kind of resources do collaborative users employ in managing their work? And how do such resources interact with the system components, behaviours and mechanisms from which collaborative software systems are constructed?

The model presented in this paper has arisen from such considerations, particularly with respect to the generic description of potential strategies for the technical support of cooperative work. In what follows, the model is introduced iteratively. In this section, the issues of the *synchronisation* of parallel *streams of activity* are introduced. These lead us to the *management of divergence*, and then the provision of *consistency guarantees* within such a system.

3.1 Synchronisation

Since we found the twin dimensions of distribution and synchronicity inadequate as concepts on which to base a generic framework, we need to find a different basis for design. In this, I follow Dix and Beale [1992] in focussing on *synchronisation*.

While synchronicity, as discussed above, talks about the temporal distribution of individual actions, synchronisation concerns the extent to which individuals are able to see and make use of each other's activity. Whether individuals are engaged in activities simultaneously or at different times, it is only when they *synchronise* that they can exchange information and establish a common ground or understanding. It is synchronisation, rather than synchronicity, which regulates the progress of cooperative activity.

Previous work, studying use of the ShrEdit shared text editor [Dourish and Bellotti, *op. cit.*], has shown how awareness and representations of the activities of other participants are important resources for individuals trying to coordinate their own activity. In these cases, this kind of awareness allows participants to assess the progress of the group as a whole, as well as to organise their own work in relation to others' activities. In other words, the exchange of awareness information, whether explicit or implicit, is a form of synchronisation which allows groups to coordinate their actions. Elsewhere, Dix [1992] has used the notion of *pace of interaction* as a basis for analysing the technological mediation of communications, rather than using the bandwidth of the channel, and elsewhere has used synchronisation as the basis of an extension to version control for distributed workers [Dix and Miles, 1992]. This notion of pace is organised around the rates of synchronisation needed to manage tasks at different levels.

We can use this concept of synchronisation to re-examine the two perspectives discussed earlier—the usage perspective and the more systems-oriented design perspective. From the usage perspective, synchronisation subsumes the notion of synchronicity; the traditionally *synchronous* systems are those which are highly synchronised, while *asynchronous* systems have a much lower rate of synchronisation. Within these broad classes, we can talk about fine-grained differences; the kinds of synchronisation differences which affect the shared feedback and synergy effects discussed by Dourish and Bellotti. From the design perspective, we can think of synchronisation as a means to describe the coordination of distributed data and the sharing of state information between components of CSCW systems. In other words, we can use this general concept of synchronisation as a way of unifying the space and time dimensions in the traditional model, both from usage and design perspectives.

3.2 Streams of Activity

In order to take this further as a basis for generic description and design, the next question to be asked is: what is being synchronised?

From the design perspective, it might be tempting to suggest that it is the *data and representations* held by each user's system which are synchronised. Indeed, since many CSCW systems *do* manage multiple copies of user data, and these copies must be maintained consistently and coherently, there is certainly a synchronisation issue here. However, the strategies by which this coherence is maintained are very much a

systems-level issue. Questions of replication and consistency are not directly of interest to a group of users (except when they break down). By the same token, if we were to focus on this level of description of synchronisation, then it would be very difficult to correlate those descriptions with the kinds of working styles which we want to support.

In contrast, then, we might attempt to talk in terms of the synchronisation of *activities*. This is clearly a level of description which is more meaningful to users. However, in addition to the problem of meaningfully defining what we mean by "activities", descriptions at this level are an inadequate basis for describing different implementation strategies and policies. It is, perhaps, *too* high-level.

Instead, we choose an intermediate level, and talk in terms of the synchronisation of *streams of activity*. In this context, we define a stream of activity as a sequence of user actions upon shareable data. Note both that while these actions are at the user level, they are not necessarily actions which the user explicitly initiated; and that "shareable" data is not necessarily shared.

Figure 2 shows the general picture. As time progresses, two users generate a sequence of actions on some shared representation of the artefacts supporting their work. At particular intervals, their streams are synchronised. This synchronisation may involve an exchange of information, a coordination of their work, *etc*. Intentionally, the model in itself says nothing about either the mechanics or the frequency of synchronisation. Indeed, the intervals between points of synchronisation could range from fractions of seconds to months. Synchronisation need not be jointly initiated, but can be opportunistically performed by either party. And synchronisation need not result in a unification of the states on either side; data might well remain inconsistent, although now information about the inconsistency has passed between the streams.

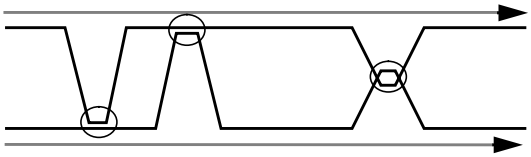

FIGURE 2: Parallel streams of activity are synchronised at intervals as work progresses.

We can interpret streams of activity widely and synchronisation can be as simple as glancing at a colleague's screen to get an idea of what they're doing (essentially, synchronising between their on-screen data and my idea of what's there) to the exchange of information between our applications so that we each have full information about the state of each other's interface.

So this level of description can be used to mediate between the usage and design perspectives. From the usage perspective, a single stream encapsulates the actions of one user; the synchronisation of these streams opens up issues of the interplay of independent actions and the coordination of users' activities. From the design perspective, the streams model provides us with a way of linking cooperative work concerns into areas such as parallel processing, transaction management and distributed data access—the components from which an implementation may be constructed.

3.3 Direct Synchronisation

In contrast with a number of other approaches, this view of "streams of activity" deals with explicit synchronisation *directly between the separate streams* as the mechanism for managing consistency. In many, if not most, other systems, the focus is quite different. Instead of synchronising independent streams, a variety of devices are used to reduce activities to a single stream. For instance, in "synchronous" systems, this may be achieved through the automatic serialisation which arises from centralising the data; or locks may be used to allow merely a single stream of activity at a time over a data item. The data-passing (or "draft-passing") mechanism employed by many "asynchronous" systems can be seen as the ultimate lock; access is denied because the data simply isn't available.

A number of CSCW application problems can arise from the attempt to reduce activities to a single stream. For instance, overly restrictive floor control policies which interrupt the free flow of work, or rigid activity structures, can result from the use of a single-stream model in situations involving multiple simultaneous activities. By using a model of direct synchronisation and multiple streams, the generic model should be able to support these types of applications more naturally.

4 Divergence

The previous section has outlined a CSCW model based on multiple, parallel streams of activity, each of which acts upon potentially replicated representations of artefacts, and with variable rates of synchronisation between the streams. A natural consequence is that the replicated copies of a single representation could possibly differ, either through unsynchronised actions in different streams or through conflicting actions. The emergence of such differences is *divergence*, and it is central to this type of model. Indeed, we could say that it is only through the appearance of divergence, either in the data or the interface, that synchronisation becomes necessary. We can, then, reduce the issue of synchronisation, both in terms of distribution and synchrony of interactions, to the *management of divergence*.

The structure of the resulting model is shown in its most general terms in figure 3. A *base object* is a system representation of some shareable artefact supporting a group's work, such as a paragraph in a collaborative text editor. In a collaborative system, this object is implemented as a *replication group*—a set of (one or more) synchronised copies which may be distributed between different users' interfaces across a network. At some point, called the *divergence point*, an unsynchronised change takes place in one or more elements of the replication group through the actions of some user. Since the change is unsynchronised, this results in differing copies of the same base object—there is a divergence in the copies of the object. The replication group is *partitioned* into a number of versions, each of which is a replication group of its own, containing one version of the object. The set of replication groups is known as a *version group*; they are differing copies derived from the same base object. At some later point, the *synchronisation point*, the elements of the version group are brought back together to form a single replication group.

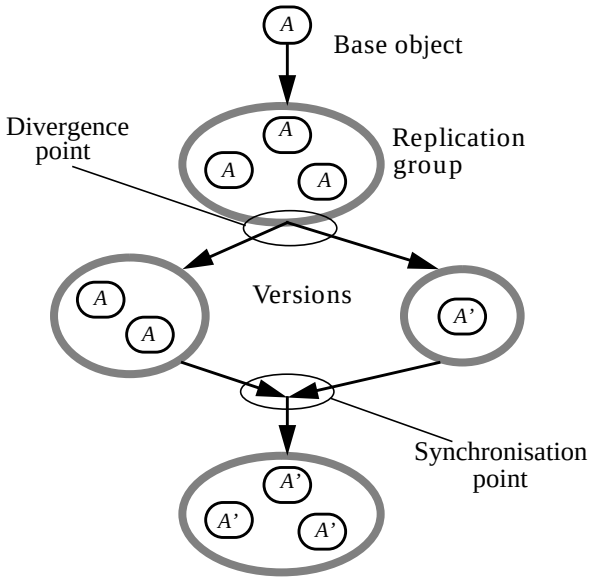

FIGURE 3: The replication group representing an object is partitioned when divergence occurs.

This sequence is intentionally very general; particular cases and applications are specialisations of the general model. For instance, in a real-time multi-user graphics editor, divergence might occur when one user moves an object in the workspace, and synchronisation is achieved by broadcasting this change to the other users before any more edit actions are performed. In a multi-synchronous text editor, however, there might be many changes which occur between divergence and synchronisation, occurring as each user pursues individual work.

There are four particular elements which this general model leaves explicitly undefined:

1. The granularity of base objects in relation to the artefacts of work. In a text-editor, the base objects might be documents, sections, paragraphs or characters; or they could be arbitrary units which the users manipulate (*e.g.* selections);
2. The criteria for divergence. Applications will vary in the amount of change which they can accommodate before divergence is said to have occurred;
3. Related to the last are the criteria by which consistency can be re-established, which affects the kinds of synchronisation that can be performed;
4. The mechanism of synchronisation—that is, the means by which members of a version group will be coalesced into a single replication group.

The point of leaving these undefined is that they are points in which specific behaviours can be used to create different applications. In the generic design, these are the primary features which can be specialised and extended, to give a variety of behaviours within the same model.

4.1 Divergence Management Strategies

One significant reason for adopting this model is that it is highly scalable. In tightly synchronised systems—those in which either the criteria for divergence are fairly loose, or the time intervals between divergence and synchronisation are very short—there is minimal divergence. This is clearly the case in centralised systems, which have a single effective stream of activity, but also applies to a number of other systems exhibiting very high levels of synchronisation. In less synchronised systems, we can use the model to define *operational transformation* mechanisms, such as those of Ellis and Gibbs [1989] or Beaudouin-Lafon and Karsenty [1992]. At the other extreme, when systems are very loosely synchronised, this model covers the use of explicit version management, such as that explored in versioned SEPIA [Haake and Haake, 1993].

To match this variability in scale, there is a wide range of strategies which can be used to manage divergence, as appropriate for particular applications. Note again that the model itself does not define or impose any strategy; but it allows such strategies to be employed. The strategy used to manage divergence and synchronisation defines the way the system appears to users and the types of application which can be supported.

For example, an implementor might choose to adopt some *automatic strategy* for the management of divergence. The operational transformation algorithms cited above are examples of such an approach. Alternatively, in sufficiently constrained domains, it might be possible to use domain

information to accomplish automatic merging of version group elements.

In other cases, it might be more appropriate to employ *user-driven strategies* rather than automatic ones. The simplest strategy in this case would simply be to flag the divergence, bringing it to the attention of the group (or appropriate members of the group, perhaps just a single moderator) for some action to be taken. An application might assist this by not only detecting the divergence but also providing some analysis, such as a "visual diff". So as not to prevent further work when conflicts are not immediately resolved, another option would be to support "version aggregation", where two concurrent versions of some object are combined as an aggregate object and manipulated as a single object except where internal details are needed. Indeed, different strategies could co-exist within an application, so that different strategies can be used for different objects, users or activities.

The divergence model, then, allows for the generalised synchronisation of potentially simultaneous streams of activity over a shared workspace. Instead of avoiding conflict—and hence limiting applicability to particular situations where this is possible—the model handles it through the explicit management of divergence, using various strategies to resolve or manage conflicts and maintain consistency in the workspace. In a toolkit, the metalevel approach provides a means for creating and using these different strategies within a consistent framework.

5 Locking and Consistency Guarantees

Although a number of possible strategies have been outlined as potential divergence management mechanisms which could be invoked at the synchronisation point, it is clear that the kinds of strategies which can be adopted depend very much on the sort of divergence which has taken place. If an application allows arbitrary divergence before synchronisation, then it may be arbitrarily complex to resynchronise the streams.

This arbitrariness is highly problematic for design. How can it be constrained? One approach might be to try a series of different synchronisation strategies at synchronisation time, continuing until one is found which can deal with the extent of divergence which has occurred. However, this is also extremely unsatisfactory, since different strategies achieve different levels of synchronisation. The choice of divergence management strategy has a strong influence on the kinds of interactions in which the group can engage. In this "serial" strategy, users would not know which particular synchronisation mechanism would be used until the synchronisation occurred; and so, only then would they discover how successfully their activities could be synchronised. Using different strategies in this way would result in a highly unpredictable interface and pose severe problems for collaborating groups.

Clearly, then, in most cases we cannot allow arbitrary divergence to occur. This suggests that some mechanism must be provided by which, at the divergence point, guarantees can be made about the kind of synchronisation which will be possible, based on information about the sort of divergence which is likely.

The ultimate guarantee, used in many existing systems, is a "strong" lock. When a participant holds a lock, it guarantees that only that participant can make changes. This implies that later synchronisation is simple, being merely a case of replacing the original object with the new, updated copy. Locking prevents divergence, and so synchronisation is trivial. However, this strong guarantee of future synchronisation behaviour is bought at the expense of support for simultaneous working by multiple participants. A lock held by one stream blocks progress in other streams over the locked objects. This is especially problematic in the case of disconnected or multi-synchronous working, as outlined earlier.

Since support for this sort of working, amongst others, is an explicit goal of the divergence-based model, a more flexible approach is needed. Through the same process of generalisation that we used before, we can provide a more abstract notion of "consistency guarantees". Consistency guarantees are similar to locks, in that they are delivered by the system as "promises" that some form of consistency can be maintained within the shared data space in the presence of changes made by individual users. They fulfill two functions:

1. for the system, they provide advance information about likely changes. This information can be used to gauge the likely extent of divergence, either to select an appropriate strategy to be used for synchronisation, or to collect information which will be useful at the synchronisation point;
2. for the user, they provide a guarantee that changes to artefact representations can be consistently integrated back into the shared workspace for future progress.

These guarantees are more general than simple locks, though, and can be used to make a variety of "promises" about consistency. A strong lock is certainly one sort of guarantee. However, through specialisation, we can create guarantees with different properties, by varying the basic definition along different dimensions. For instance, guarantees can be variable in terms of the sorts of operations they will support during the divergence, *e.g.* guarantees that consistency can be maintained if only the internal details of an object may change, but not its structure. They may vary in the type of synchronisation or merging which they will support, *e.g.* one type of guarantee if automatic strategies are required, and another if manual user-driven strategies are acceptable. A third form of variability might be in terms of the form of consistency which can be maintained, *e.g.* if cer-

tain types of inconsistency are acceptable in particular domains or circumstances.

So consistency guarantees are the final element of this model, balancing divergence and synchronisation by providing a means for the system to anticipate the requirements of synchronisation. Unlike standard locking mechanisms, these are guarantees about synchronisation, not about divergence. Divergence will still occur. In particular circumstances, an application might even allow a user to make more changes than are supported by the guarantee it holds, presumably with suitable warnings; and in this case re-synchronisation may simply not be possible. This kind of flexibility derives from focussing on the re-establishment, rather than the maintenance, of consistency.

6 Applying the Generic Model

We have now assembled all the components of a generic model of the distribution and synchrony elements of collaborative systems. The relationships between the areas of the model are shown in figure 4. Divergence is used to mediate between the issues of synchrony and distribution, and divergence is managed by building on the use of consistency guarantees as a means of achieving synchronisation. This model encompasses a number of areas of activity in CSCW design. When we apply the notion of divergence to issues of distribution, we address the question of replication of data and the management of consistency; when we apply it to synchrony, we address issues of the interaction of separate user activities .

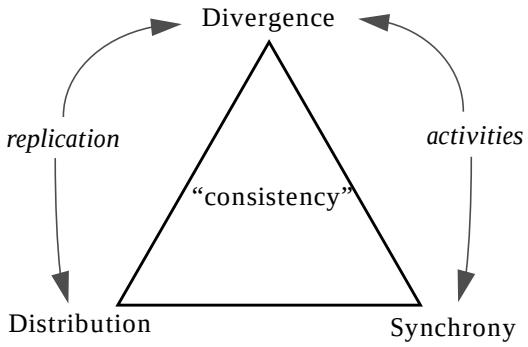

FIGURE 4: Divergence provides a link between issues of distribution and synchrony, organised around a generalised view of consistency.

There are two ways in which we can use this model—as a basis for analysis and as a basis for design.

It supports analysis through the modelling of applications in terms of streams, replication groups, divergence and synchronisation and consistency guarantees. We have already seen that the basic model of synchronised streams of activity can be used to model both fine-grained real-time cooperative working and much looser asynchronous or multi-synchronous working. Similarly, the previous section has outlined how consistency guarantees can be used to model a variety of coordination styles for group working.

An important role for the model, however, is as the basis for a toolkit. The toolkit is based on the reflective principles outlined in section 1.1, and embodies the generic model using object-oriented programming techniques. The result is that standard components (such as **`stream`**) or standard operations (such as **`synchronise`**) can be subclassed and augmented or incrementally redefined. This means that the system developer has the opportunity to tailor the support provided in the toolkit to the requirements of an application.

The generality implicit in the model, then, is the basis of flexibility in the toolkit; this is the flexibility required to support a wide range of applications. By employing different implementation strategies or configurations at the metalevel, we can use the toolkit structures in very different ways. Consider three examples.

Example 1: In an object management system, divergence and synchronisation take place only on explicit user actions on individual objects, called **`check-out`** and **`check-in`**. At check-in time, a modified object can either be added to a set of checked-in versions, or an optional user-assisted merging facility can be called to unify the copies. This mechanism can clearly be supported through use of the basic mechanisms provided by the divergence model. The result is a basic multiple version system, as might be used to support cooperative software development.

Example 2: In a real-time shared text editor, each character entry represents a divergence (in this case, between the stream of the user pressing the key, and the other users). Between each keystroke and the resultant change to the text, the application captures its current state, recording the latest edit changes received from other users. This is a consistency guarantee, since this record can be used, along with the character entry command, to synchronise concurrent edit streams when the command is sent to the workstations of other users. In this case, we have simulated the mechanisms behind the dOPT algorithm [Ellis and Gibbs, 1989] used for fine-grained coordination in the GROVE editor.

Example 3: So-called "epidemic" algorithms are a means by which distributed, replicated databases can be maintained (see, *e.g.* [Demers *et al.*, 1987]). Each node in a network maintains part or all of a copy of the database, and will receive transactions (additions, deletions or changes to records). Periodically, nodes exchange information about recently observed events. Nodes pass on information received from other nodes, perhaps at random (a technique colourfully called "rumour-mongering") and so changes spread throughout the system, maintaining the consistency of the "global" database. Again, this can be implemented using a suitably specialised version of the generic model,

capturing the sequence of transactions at a node as a stream. Divergence occurs as individual transactions are submitted to each node, either directly by a (human) database maintainer, or indirectly through inter-node information exchange; synchronisation occurs as conflicts between previously-held information and new transactions are resolved (typically through simple time-stamping).

The purpose of these examples is to give a flavour of the way in which the model provides a basis for flexibility. Specialisation and customisation of toolkit components responsible for divergence detection, replication group partitioning, synchronisation and so on allow the general framework to be applied to a wide range of application scenarios.

The three examples differ greatly in terms of the synchronicity of interaction they exhibit, the nature of the task and of the consistency which is maintained, and the kinds of tasks for which they might be useful. However, they are captured by a single framework. In itself, this is useful; but as the basis of a toolkit which embodies the same flexibility and variability as the model, they point towards a much more open notion of design and a much wider scope of applicability than has generally been possible for traditional toolkits.

7 Summary and Ongoing Work

Starting with a number of problems which arise from the traditional space/time characterisation of CSCW applications, this paper has outlined a new model which unifies issues of distribution and synchrony at both user and system levels. The goal of this model is to provide a framework for flexible CSCW design, in the form of a reflective toolkit which uses object-oriented techniques, and in particular incremental specialisation, to allow the application developer to revise internal toolkit implementation decisions.

The basis of this model is the explicit management of divergence between potentially simultaneous streams of activity over a shared workspace. Streams of activity correspond to user and system actions at one user's node. Divergence occurs when actions across the shared workspace are unsynchronised, and hence two user's views (or copies) of the shared workspace differ. Cooperation and coordination is achieved through the periodic synchronisation of streams. In the model, a collaborative system regulates divergence and synchronisation through the use of general consistency guarantees.The mechanisms by which divergence is admitted, consistency guarantees made, and later synchronisation achieved, are points within this model where a different strategies can be adopted. This openness leads to a wide range of resultant tools applicable in different situations, and supporting different working styles.

The model provides a way of describing a wide range of CSCW systems, including systems which are not catered for by previous approaches. In particular, it looks on many existing systems as special cases of *multi-synchronous* working, which is poorly handled by more traditional approaches, especially in the disconnected case.

This model has been developed as part of the ongoing development of a flexible toolkit for building adaptive CSCW applications, and, at the time of writing, an implementation of this model is under way. In combination with generic models of other areas of CSCW system design, such as interface linkage, the divergence model should result in a toolkit which avoids the typical problems of premature commitment in toolkit implementation, leading to a much wider range of applicability than is usually available to application developers.

Acknowledgments

The model presented here has its beginnings in conversations with Alan Dix and John Lamping, to whom I am very grateful. Jon Crowcroft provided useful feedback on the ideas, and Victoria Bellotti, Dik Bentley, Steve Freeman and Tom Rodden made valuable comments on earlier drafts of this paper.

References

[Beaudouin-Lafon and Karsenty, 1992] Michel Beaudouin-Lafon and Alain Karsenty, "*Transparency and Awareness in a Real-Time Groupware System*", Proc. ACM Symposium on User Interface Software and Technology UIST'92, Monterey, California, November 1992.

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

[Demers *et al.*, 1987] Alan Demers, Mark Gealy, Dan Greene, Carl Hauser, Wes Irish, John Larson, Sue Manning, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry and Don Woods, *"Epedemic Algorithms for Replicated Database Maintenance"*, in Proc. ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987.

[Dix, 1992] Alan Dix, "*Pace and Interaction*", Proc. of HCI'92, York, UK, 1992.

[Dix and Beale, 1992] Alan Dix and Russell Beale, "*Information Needs of Distributed Workers*", University of York, 1992.

[Dix and Miles, 1992] Alan Dix and Victoria Miles, "*Version Control for Asynchronous Group Work*", Univerity of York, UK, 1992.

[Dourish, 1994] Paul Dourish, "*Developing a Reflective Model of Collaborative Systems*", EuroPARC Technical

Report EPC-94-101, Rank Xerox EuroPAC, Cambridge, UK, 1994.

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

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

[Ellis *et al*., 1991] Clarence Ellis, Simon Gibbs and Gail Rein, "*Groupware: Some Issues and Experiences*", Communications of the ACM 34(1), January 1991.

[Haake and Haake, 1993] Anja Haake and Jörg Haake, "Take CoVer: Exploiting Version Support in Cooperative Systems", Proceedings of InterCHI'93, Amsterdam, Netherlands, April 1993.

[Johansen, 1988] Robert Johansen, " *Groupware: Computer Support for Business Teams*", The Free Press, New York, 1988.

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

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

[Maes, 1987] Pattie Maes, *"Concepts and Experiments in Computational Reflection"*, Proc. ACM Conf. Object-Oriented Programming Systems, Languages and Applications OOPSLA'87, Orlanda, Florida, 1987.

[Olson *et al.*, 1993] Judith Olson, Stuart Card, Thomas Landauer, Gary Olson, Thomas Malone and John Leggett, "*Computer-Supported Cooperative Work: Research Issues for the 90's*", Behaviour and Information Technology, 12(2), pp. 115–129, 1993.

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

[Rodden, 1991] Tom Rodden, "*A Survey of CSCW Systems*", Interacting with Computers, 3(3), pp. 319–353, 1991.

[Roseman and Greenberg, 1992] Mark Roseman and Saul Greenberg, "*GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications*", Proceedings of ACM Conference on Computer-Supported Cooperative Work CSCW'92, Toronto, Canada, November 1992.

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