# The Parting of the Ways: Divergence, Data Management and Collaborative Work

Paul Dourish
Rank Xerox Research Centre, Cambridge Laboratory (EuroPARC) and Dept of Computer Science, University College, London (dourish@europarc.xerox.com).

**Abstract**: Systems coordinating distributed collaborative work must manage user data distributed over a network. The strong consistency algorithms which designers have typically borrowed from the distributed systems community are often unsuited to the particular needs of CSCW. Here, I outline an alternative approach based on divergence and synchronisation between parallel streams of activity. From a CSCW perspective, this strategy offers three primary advantages. First, it is scalable, allowing smooth transitions from highly interactive collaboration to more extended, "asynchronous" styles of work. Second, it supports "multi-synchronous" work, in which parties work independently in parallel. Third, it directly supports observed patterns of opportunistic activities in collaborative working.

## Introduction: Distributed Data Management

Collaborative applications coordinate activities which may be distributed in time and/or space. Distribution in time means that activities may take place at different times, but are coordinated to achieve a unified effect (such as the production of a document). Distribution in space means that activities may take place on different computers, perhaps linked by a data network. Collaborative applications, then, are heir to a set of design problems which have arisen in the development of distributed computing systems (or just "distributed systems"), concerning the distributed management of data.

This paper considers strategies which can be employed to meet the conflicting demands placed on collaborative applications, in presenting users with a single, uniform data "space". We are primarily concerned here with "user data"; that is, largely the computational representations of artefacts which are manipulated directly by the system's users. So, in a collaborative writing system, user data would include the computational representation of the document, or of users' activities over that document.

There are a number of criteria must be met when this data is distributed across a network: *availability*—users should be able to gain access to data when they need it; *transparency*—users should not have to worry about patterns of data distribution, or the details of the distribution management; *consistency*—users should see identical (or, at least, consistent) views of shared data, even though they may be working at different places or different times; and *responsiveness*—data management should not interfere with the interactive response of the system.

However, these criteria place conflicting demands on an implementation. For example, *availability* can be enhanced by maintaining multiple copies of the data on different network nodes, increasing the probability that some copy will be available to a user when it is required. Unfortunately, this approach—data *replication*—conflicts with *consistency*, since two users can make incompatible changes to two copies of the same piece of data. Various strategies can be used to avoid or resolve these conflicts, but these, in turn, endanger *transparent* operation, by introducing more ways in which users can be exposed to the consequences and details of distributed data management. Different systems have different requirements, and differently prioritise these criteria.

These issues are endemic to distributed computer systems, and much research in the distributed systems community has focussed on their implications in areas such as distributed databases, distributed transaction processing and distributed file systems. This paper concerns the relationship between distributed data management and applications supporting specifically collaborative work. I will argue that traditional mechanisms which we might adopt from distributed systems are frequently ill-suited to the needs of collaborative systems. I will outline an alternative approach, which has been developed in Prospero, a toolkit for building collaborative applications, and show how it addresses problems in effectively supporting collaborative systems in an open and flexible way. Such flexibility is crucial to the design of a toolkit (rather than individual applications). First, though, I will outline some current approaches to the problems of distributed data management, and in doing so, set out some issues, parameters and terminology[1].

## Distributed Data and Collaborative Work

A range of approaches have been employed to manage distributed data problems in collaborative technologies. In this section, I will focus on the design decisions and trade-offs which system designers make, particularly in CSCW systems, before going on to outline some of the inherent problems and an alternative strategy.

### Distribution

One set of decisions concern the mechanisms which determine where a particular data structure will reside in the system at any given time—data *distribution*. The

[1] At this point, I beg the indulgence of more technical readers. In a spirit of fairness, I'll beg the indulgence of non-technical readers later on.

distinction between *centralised* and *replicated* approaches, has long been a concern for CSCW developers (Ahuja et al, 1990; Lauwers, 1990; Greenberg et al, 1992). Centralisation concentrates all the data at one point in the system; clients communicate with this central point to retrieve or update information, and so "consistency" is a trivial issue since there is only one copy of a data item at any time. Replication, however, allows multiple copies of data structures—possibly as many copies are there are participants. Replication improves availability, but complicates consistency management.

Orthogonal to this is the issue of data location. While most systems are static—that is, the "location" of any given piece of data is fixed during execution—this is not an inherent property. In a dynamic system, objects may *migrate* from location to location. Distributed document management systems, or Workflow systems, might use this approach, moving the active data set from user to user. This technique may be used along with either centralised or replicated approaches.

## Management

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

In many cases, this is a "non-problem". Many applications simply do not require absolute consistency in their user data. For instance, in a "shared whiteboard", absolute consistency is generally not a concern and the system would be unlikely to attempt to rigorously maintain data integrity. However, in more structured applications, consistency can become a key requirement. The kinds of inconsistency acceptable on a shared whiteboard are unacceptable in a spreadsheet.

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

In a centralised system, this is not a problem. Since everyone sees the single copy of any data item, they see the same changes arise; there is a single, network-wide consistent ordering of events. Only one event can be processed at a time, so changes which arrive at the "same" time will still be processed seperately, in some specific (if arbitrary) order. Thus, a centralised approach to data storage inherently introduces a *serialisation* of change events, which, while potentially unpredictable, maintains consistency.

Replicated systems can also achieve consistency through global serialisation. The simplest approach is *data locking*. System components declare their intent to perform modifications on a data item by requesting a "lock" on the data item. Once the change has been made, the lock is released and is available to other clients. Since only one client at a time can hold the lock on an specific item, simultaneous changes are prohibited and consistency is maintained. Locks can be defined at different granularities, from the whole document down to the level of individual objects or insertion points, but their role remains the same—to avoid inconsistency by preventing simultaneous action on data items.

Many common floor control policies can be regarded as locks on the entire workspace, restricting activity to one individual at a time. This is *input multiplexing*—the reduction of multiple input channels (one or more per individual collaborator) to a single channel (the input channel to the workspace). Mechanisms such as baton-passing and round-robin divide access between the participants so that only one has control at any point. Essentially, that participant holds a lock on the entire workspace; no other participant can contribute until she loses control (relinquishes her “lock”), and so consistency is maintained.

## Managing Divergence

The variety of data management strategies is testament to the fact that no single approach is applicable in all cases. In part, this is simply due to the considerable variation in the needs of CSCW systems. In addition, it is because the choice of management strategies has strong implications for the interface and for the nature of collaborative interaction in a CSCW system (e.g. Greenberg and Marwood (1994)). Collaborative systems differ crucially from other distributed systems in that not only the application, but also the interface, is distributed. The trade-offs between availability, transparency, consistency and responsiveness must be made with this in mind, and so design must be constantly mindful of the way in which application distribution and interface distribution are mutually influential.

These issues become particularly problematic when trying to design a CSCW toolkit, which will be used to create and support a wide range of applications. The toolkit designer is even further separated from end-users than is the developer of individual applications; and so it becomes even more difficult to understand the implications of distributed data strategies for particular usage situations. Here, we need to develop a general characterisation of distributed data management in CSCW.

### Inconsistency Avoidance and Streams of Activity

We begin with a simple but crucial observation; that most approaches to data management in CSCW deal with *inconsistency avoidance* rather than *consistency management*. Rather than working to achieve data consistency, they erect barriers which prevent inconsistency arising in the first place. This is a distributed systems approach; the system manages the action of the separate components to avoid inconsistency arising. However, applying this same strategy to collaborative systems is problematic. Our distributed entities are users, not programs; and they're less prepared to accept the imposition of global mechanisms to constrain their activity!

Since inconsistency arises through the simultaneous execution of conflicting operations, the simplest approach to avoiding inconsistency is to avoid simultaneous action over individual data items. In other words, this approach attempts to define *single, global stream of activity* over the data space. Various common elements of CSCW systems embody this model of a single stream of activity. Asynchronous access to the workspace—using the distribution of work in time as a means to control access—does just this, by sharing the single stream between multiple participants, one at a time. Floor control policies do likewise, as do conventional locking mechanisms, from the point of view of individual data items; locks ensure that each item is subject to a single thread of control, currently available to whoever holds the lock.

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

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

The purpose of exploring this divergence-based view is in pursuit of a generic, specialisable model of distributed data management. By *generic*, I mean that this model describes, in general terms, a range of distribution strategies which can be or have been adopted in a variety of systems. By *specialisable*, I mean that any particular example can be operationally described as a refinement of the general model. The model, then, is not simply a tool for the analytic description of CSCW architectures and implementations; it can also be used to generate new ones, as the basis of an implementation. It has been developed as part of Prospero, a toolkit for CSCW application design which employs explicit specialisable models as a basis for highly flexible, open-ended design (Dourish, 1995a); and the framework which it provides is the basis for creating data management strategies in CSCW applications.

## Divergence

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

### Divergence and Versioning

This view of continual divergence and synchronisation is similar to that of versioning systems, which maintain a historical record of the versions of some object which have existed over time. They typically allow multiple versions of an object to exist at once, and in some, multiple versions can be simultaneously active. GMD's CoVer (Haake and Haake, 1993) uses a version system to manage the cooperative work. CoVer, however, emphasises the creation and management of parallel versions rather than the subsequent integration of different versions (divergent streams). Munson and Dewan (1994) go further in providing a framework explicitly organised around version merging, but, like Haake and Haake, they primarily emphasise versioning and merging within a context of "asynchronous" work, rather than as a more general approach to distributed data management. I want to consider the wider use of divergence as a general strategy (discussed in more detail below).

## Divergence and Operational Transformation

An alternative technique which has been employed effectively in a number of collaborative systems is operational transformation (Ellis and Gibbs, 1989; Beaudoiun-Lafon and Karsenty, 1992). Operational transformation employs a model of multiple streams, and uses a transformation matrix to *transform* records of remote operations before applying them locally, using information about the different contexts in which the operations arose. Clearly, this approach is much closer to the divergence model advocated here,but there are two principal differences. First, just as versioning approaches have typically emphasised *asynchronous* activity, operational transformation has typically emphasised *synchronous*; as will be discussed, Prospero's model attempts to be more general. Second, operational transformation relies upon the transformation matrix to resolve conflicts (easier in the tightly-coupled, synchronous domain); whereas Prospero employs a more general notion of sychronisation which potentially offers a much wider scale of applicability.

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

## Divergence and Replicated Databases

One area of research in which divergence has been considered is replicated database management. In a replicated database, multiple copies of all or part of the database are maintained in parallel, in order to increase availability. The relationship between this proposal and replicated databases is discussed in detail elsewhere (Dourish, 1995b), but an outline is appropriate here.

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

Both of these techniques, and in general the use of the transaction model, place the detection, avoidance and management of conflicts *within* the database itself; unlike this proposal, the application itself is typically not involved in the conflict management process. This is generally true when database technology is used as infrastructure for collaborative applications. However, there are times when this model must break down. In Lotus Notes, for example, users interact directly with document databases which are replicated amongst different sites but largely disconnected from each other, and so conflicts can occur during periods of simultaneous work (as here). However, in these cases, Notes merely flags the conflict (i.e. it maintains what I will call "syntactic consistency") and carries on, rather than providing any means for conflict resolution.

Replicated databases deal with some problems which divergence raises; however, they generally do not directly exploit divergence to support multi-user activity.

## Capitalising on Divergence

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

### Scalability

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

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

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

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

## Multi-Synchronous Applications

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

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

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

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

## Supporting Opportunistic Work

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

In studies of collaborative authoring, Beck and Bellotti (1993) highlighted the opportunistic way in which much activity was performed. In particular, they pointed to the ways in which opportunistic action on the parts of individual collaborators often went *against* pre-defined roles, responsibilities or plans. Individuals acted in response to specific circumstances; while the plans and strategies formed *one* guide to their actions, they were by no means the only factors at work, and in each of their case studies, they observed occasions on which agreements about who would do what and when were broken. Critically, these broken agreements are neither unusual nor problematic; this opportunistic activity is part of the natural process of collaboration. (Suchman (1987) has, of course, made similar telling observations about the status of plans as resources for action rather than as rigorous constraints upon it.)

The implication here is clear. We must be wary of introducing technology which inappropriately reifies plans and use pre-formed strategies to organise collaborative activity since observational studies show that they are opportunistically broken in the course of an activity. Turn-taking floor control policies, or partitioning a workspace into separate regions accessible to different individuals, are examples of technological approaches which structure user interaction around plans of this sort. Once again, this contrasts the particular needs of CSCW systems with traditional distributed systems, and shows that a distributed *interface* is an important consideration. To support the sort of opportunistic working described by Beck and Bellotti, then, our technology must relax rules about exclusion and partitioning; exactly the rules which have been employed to maintain the fiction of the single stream of activity.

So the same sorts of mechanisms which were described earlier as supporting multi-synchronous collaboration have, in fact, a wider range of applicability; they support a more naturalistic means of *making asynchronous collaboration work*. Divergence is a direct consequence of these ways of working; and so a model of distributed data management based on a pattern of repeated divergence and sychronisation fits well with support for a wide range of working styles.

## Constraining Divergence: Consistency Guarantees

The previous sections have outlined a model of distributed data management based on a continual cycle of divergence and synchronisation, and shown how this approach fits naturally with the needs of CSCW systems.

However, there's a problem; and it's one which must be addressed if we hope to use divergence as a strategy for *building* CSCW systems rather than simply talking about them. At any given point, how can we maintain reasonable expectation that synchronisation will be possible? If two streams diverge arbitrarily, how can we be sure that a consistent view can be constructed later?

### Syntactic and Semantic Consistency

The answer to this question has two components. The first lies in the very general nature of "synchronisation". The notion of synchronisation is in not meant to imply that consistency can be achieved automatically. Certainly, it *may* be possible in many cases—particularly where divergence is slight, or user activity over the data is highly structured—to resolve divergence by purely automatic mechanisms; but this automation is not central to the model. In other cases, conflict resulolution may require human intervention. However, crucially, we can make a distinction between *semantic* and *syntactic* consistency. By “semantic” consistency, I mean that the data is internally “consistent” and “appropriate for its intended use”. By “syntactic” consistency, I mean merely that two streams see the same view of the data, even if that view doesn’t necessarily make sense in context.

Consider an example in collaborative writing again. Some changes—simple changes in formatting, text insertion, spelling correction and so forth—can be automatically integrated and so synchronisation is largely automatic (indeed, unless this were true, it would be impossible to build shared editing systems at all). Others, however, are conflicts which require human intervention. For instance, if two authors have separately reworked the same paragraph in such a way that the new paragraphs can not be integrated textually, then clearly the authors should be responsible for deciding which paragraph text should be used, and how the conflict can be resolved. So human intervention is required to achieve semantic consistency; but a different form of consistency—syntactic—can be achieved without human intervention. The system can apply the same approach which collaborative authors might well employ when out-of-touch with each other; preserving *both* texts, along with some marker that “this choice remains to be resolved”. This approach is *aggregation*—the combination of unresolvable data elements to form a single larger unit. Aggregation achieves syntactic consistency, which retains the property we require at the system level—that the two streams share a view of the user data. It is sufficient for the two individuals involved to be able to carry on with their work for the moment, although they will have to come back and sort out the problem later, together.

So, by maintaining semantic consistency when possible, resorting to syntactic consistency when necessary, and potentially using weak techniques such as aggregation, we can achieve a *working* level of consistency under a variety of circumstances. However, we can do more to help ensure that this works smoothly.

## Consistency Guarantees

The second aspect of our solution is technological. We can augment the divergence and synchronisation framework with something to manage expected divergence—consistency guarantees.

Clearly, we can be more confident about achieving consistency if we have some idea of what type of divergence is likely to occur. The longer the periods of divergence, the less sure we can be about this, and hence about achieving consistency. If we knew in advance what sort of actions were likely to occur on a stream before the next point of synchronisation, we could make some kind of guarantee of the degree of consistency which can be achieved.

In Prospero, consistency guarantees explicitly represent these interactions. Before divergence, one stream can “describe” the likely actions which will occur during the period of divergence. For instance, if a user has opened a document for reading only, then it’s likely that no changes will be made. Alternatively, it may be possible to say that the expected changes are all structural, rather than affecting the content, or that the user will only add information but not delete any. In exchange for this, the client can receive a statement of the level of synchronisation which can likely be achieved at the next synchronisation point—a consistency guarantee. Again, these are explicit computational artefacts in Prospero. Essentially, the guarantee says, "if only actions of those sorts occur, given other declarations of expected activities in other streams, this level of consistency should be achievable when synchronisation occurs."

Consistency guarantees are a more general mechanism than traditional locks, although they share certain properties. Consistency guarantees are used to manage simultaneous action (rather than to avoid it, which is the role of locks); and as a result, they embody more limited guarantees of later consistency (while locks guarantee absolute consistency). However, they share the principle of providing information about activities in advance, in exchange for guarantees of later consistency. We must be careful to avoid the problems of locking described above, such as poor support for opportunistic working. So Prospero introduces the following principle: the client can break its "promise" about expected behaviour, in which case the system will no longer be held to its guarantee. If the client, or the user, performs actions which were not part of its declaration, then it may only be possible to achieve some weaker form of consistency (e.g. through aggregation).

We use consistency guarantees, then, as a way to manage expectations, but not to enforce activity. Space is too limited here to go into the full details of this approach and the way in which it is embodied in Prospero; and in later sections, I will pass over the relationship between divergence, synchronisation and consistency guarantees. A fuller discussion is presented elsewhere (Dourish, 1995b).

## Divergence in Prospero

Now that the major components of the divergence approach, and its benefits, have been outlined, we can look at how it operates in practice.

Prospero is a CSCW toolkit written in Common Lisp which has been designed to provide application developers with a great deal of flexibility in tailoring the toolkit's components and strategies to the needs of specific applications or usage situations. It employs computational reflection (Smith, 1984) and open implementation (Kiczales, 1992) to open up the implementation and allow application developers—the toolkit's users—principled access to internal aspects of the toolkit. This approach exploits specialisable generic models of the sort outlined here.

Prospero's data distribution strategies are based on divergence. The divergence/synchronisation patterns form a generalised framework within which particular mechanisms are implemented. This is encoded in an object-oriented class hierarchy; new strategies are developed by specialising these descriptions.

Here, I will present examples to illustrate the use of the divergence mechanism in Prospero and show how divergence supports a wide range of application strategies. The examples take the form of code fragments[2] illustrating the framework's

[2] At this point, and as promised, I beg the indulgence of *non*-technical readers. However, the structure of the code fragments is more important than their detail.

specialisation to the needs of particular applications. After presenting these examples, I'll step back to consider the structure of the framework itself.

Before going into more detail, though, there are a number of points which should be made. First, the examples here have been considerably simplified to illustrate the main points in the space available. In particular, as suggested above, the interaction between divergence management and consistency guarantees has been omitted. Second, these examples operate on three levels at once, and it's critical to a conceptual understanding of the approach that these are kept separate. The first level is that of the example applications used to illustrate the ideas; the second level is the use of programming structures to realise these applications, and the third level—the most important for this exposition—is the use of the divergence model in providing a programming framework. The examples have been structured to highlight this third level; the result is that certain liberties have been taken with application requirements and efficient programming.

## Example: Shdr

Shdr is a simple multi-user shared whiteboard application with a replicated architecture. Originally designed outside the divergence framework, its approach lends itself well to that model. Individual user actions are performed on the user's own copy of the data (the record of whiteboard marks), and are recorded in a buffer of activity records. Periodically, the accumulated history is sent to other participants using a simple protocol defined at the level of semantically-meaningful events (that is, in terms of drawing actions, rather than in terms of mouse movements or other input operations). The frequency of updates is variable, but generally the event history will be transmitted multiple times per second.

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

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

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

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

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

Figure 1: Mapping shdr's strategy into the Prospero framework.

We can reconstruct the approach used by shdr within the divergence framework, as in figure 1. Local actions create divergence from a shared view of the whiteboard until synchronisation occurs when the history records are exchanged. Each user's actions are associated with a stream of their own, and are accumulated in the stream data structure until synchronisation occurs.

In Prospero, user actions are explicitly represented within a class hierarchy rooted in the abstract class `<action>`. Different types of action are instances of subclasses of this class. In this example, we use the subclass `<edit-action>` to represent those actions which have an effect on the data store. So, actions which result in drawing activities (such as making or erasing a mark) would be classes as instances of `<edit-action>`, whereas those which have no effect on the data itself, such as cursor movement, are not.

Activity streams are also explicitly represented, under the abstract class `<stream>`. Two subclasses of `<stream>` are used here. The first, `<remote-stream>`, represents non-local streams (i.e. the streams of other users); the second, `<bounded-stream>`, is a particular kind of local stream with specialised behaviours. It captures those behaviours which are particular to the way in which we want to use streams in this example; the way that shdr manages use data. In particular, a `<bounded-stream>` is one which accumulates some number of local actions and periodically flushes them to other participants.

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

## Example: Source Code Control

As a second example, consider a traditional source code control system in a collaborative software engineering environment. Typically, these will use a check-in/check-out model for software components or modules, together with a dependency mechanism which records relationships between them.

With the previous example under our belts, most of the structure of this example is already provided for us. We already have a means to accumulate and distribute sets of changes which arise in one place or another, which can be reused here. The most important change we have to make, as illustrated in figure 2, concerns user-initated synchronisation. The code fragment uses a new class of action (called `<synchronise-action>`) to distinguish user operations which explicitly force

[3] I use CLOS terminology here for object-oriented concepts. In Smalltalk, the closest relative of a "generic function" is a "message"; in C++, a "virtual function".

synchronisation. For normal editing activities, the system simply accumulates the action records, as before. However, for synchronisation actions, the synchronisation function is invoked.

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

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

(defmethod synchronise (stream (remote <remote-stream>))
   ;; as figure 1 ...
   ...)

(defmethod propogate-action-to-stream (action (stream <remote-stream>))
   ;; as figure 1...
   ....)
```

Figure 2: Check-in/check-out strategy.

## Example: Multi-synchronous Editing

The first example modelled a fairly standard "synchronous" tool while the second used an application type normally considered asynchronous. As a final example, let's consider the implications of multi-synchronous working.

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

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

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

Figure 3: Supporting multi-synchronous activity.

With the exception of the possible use of consistency guarantees, which are omitted here due to space considerations, multi-synchronous activity is no different at the point of divergence Once again, we can accumulate actions until some synchronisation action occurs, either automatically or by user request. This, however, is the point at which a more complex strategy is required. In the first example, we could simply ignore data consistency problems, and in the second, asynchronous access ensured that such problems didn't arise. In this example, we have to be aware of the possibility of mutually inconsistent changes and act accordingly. So the focus of attention in this case is on the synchronisation procedures.

The code frament in figure 3 illustrates two points. The first is that synchronisation is now requires processing (i.e. it's not simply the transmission of information); and the second is that its now the mutual achievement of both parties (i.e. its no longer sufficient for the originating side to send the information and move on).

The approach is very simple. For the first time, the synchronisation procedure pays attention to the return value of `propagate-action-to-stream`, since this can return information from the remote side. Here, we work to the model that integration work will be done by the remote stream, which may pass back modified data to reflect the resolution of conflicts; and so it  must be reintegated into the local stream's view. We also see the way in which `incorporate-action` is processes records of activities originating in some other stream. In this case, we use the simplest strategy; if the remote action is an edit action, and if it is compatible with local changes, then it is applied, and if not, then syntactic consistency is achieved through aggregation. Since the open strategy used in Prospero allows specialised definition of functions such as `compatible-p` and `locally-perform`, then we can be quite loose in what is accepted, and work to achieve semantic consistency when possible.

### Specialisation in Prospero

From these examples, a pattern has built up of the way in which Prospero supports a diverse range of applications. First, Prospero provides a set of default behaviours which embody mechanisms for collaborative data management. This is the function of a toolkit, and so in this respect, Prospero is not particularly different from other toolkits. (Clearly, the detail of Prospero's management strategies differs from the particular strategies employed in specific other toolkits; but in the more general sense of the provision of management mechanisms, it accords to the standard model of toolkits.) Second, and critically, Prospero structures these mechanisms within an object-oriented framework and reveals to applications and application developers elements of this framework as a means to introspection and intercession. Prospero, then, provides two, orthogonal interfaces the functionality of its collaboration support mechanisms. The first, traditional or *base-level* interface provides facilities which clients can *use* to create collaborative applications. The second, *meta-level* interface provides a means by which internal functionality can be specialised to the needs of particular applications. Design decisions are not locked in and hidden behind traditional abstraction barriers but are revealed so that they can be manipulated, so the toolkit can be used to support a much wider range of application requirements than would otherwise be possible (Dourish, 1995a).

## Summary

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

This paper has outlined an alternative approach. Rather than attempting to maintain the illusion of a single stream of activity, it is based on divergence and synchronisation between multiple, parallel streams. This general approach is particularly suited to the requirements of CSCW applications, and, as a *specialisable* model, it can be used as flexible basis for development. Along with the consistency guarantee mechanism, divergence forms the basis of the distributed data management in Propsero, a reflective toolkit for the design of collaborative applications. Prospero is a vehicle for the exploration of issues of flexibility and openness in the design and use of collaborative applications; and the use of divergence is a critical component of its open approach to CSCW design.

## Acknowledgements

Alan Dix and John Lamping provided inspiration, while Dik Bentley, Jon Crowcroft and the conference reviewers improved the quality of exposition.

## References

Ahuja, S., Ensor, J. and Lucco, S. (1990): “A Comparison of Application Sharing Mechanisms in Real-time Desktop Conferencing”, in *Proc. ACM Conf. Office Information Systems COIS’90*, Boston, 1990.

Beaudouin-Lafon, M. and Karsenty, A. (1992): "Transparency and Awareness in Real-Time Groupware Systems", in *Proc. ACM Conf. User Interface Software and Technology UIST'92*, Monterey, Ca., November 1992.

Beck, E. and Bellotti, V. (1993): “Informed Opportunism as Strategy”, in *Proc. Third Eruopean Conference on Computer-Supported Cooperative Work ECSCW’93*, Milano, Italy, 1993.

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

Dourish, P. and Bellotti, V. (1992): “Awareness and Coordination in Shared Workspaces”, in *Proc. ACM Confe. Computer-Supported Cooperative Work CSCW’92*, Toronto, Canada, 1992.

Dourish, P. (1995a): “Developing a Reflective Model of Collaborative Systems,” *ACM Transactions on Computer-Human Interaction*, 1995 (in press).

Dourish, P. (1995b): *“Consistency Guarantees: Exploiting Operation Semantics for Consistency Management in Collaborative Systems”*, Rank Xerox EuroPARC Technical Report, Cambridge, UK, 1995.

Ellis, C. and Gibbs, S. (1989): “Concurrency Control in a Groupware System”, in *Proc. ACM Conf.Manamagement of Data SIGMOD’89*, Seattle, Washington, 1989.

Greenberg, S., Roseman, R., Webster, D. and Bohnet, R. (1992): “Human and Technical Factors in Distributed Group Drawing Tools”, *Interacting with Computers*, 4(3), pp. 364-392, 1992.

Greenberg, S. and Marwood, D. (1994): “Real-time Grouopware as a Distributed System: Concurrency Control and its Effect on the Interface”, in *Proc. ACM Conf Computer Supported Coooperative Work CSCW’94*, Chapel Hill, North Carolina, 1994.

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

Kiczales, G. (1992): “Towards a New Model of Abstraction in the Engineering of Software”, in *Proc. Workshop on Reflection and Meta-level Architectures IMSA’92*, Tokyo, Japan, 1992.

Lauwers, C., Joseph, T., Lantz, K. and Romanow, A. (1990): “Replicated Architectures for Shared Window Systems: A Critique”, in *Proc. ACM Conf. Office Information Systems COIS’90*, Boston, Massachussetts, 1990.

Munson, J. and Dewan, P. (1994): “A Flexible Object Merging Framework”, in *Proc. ACM Conf. Computer-Supported Cooperative Work CSCW’94*, Chapel Hill, North Carolina, 1994.

Smith, B. (1984): “Reflection and Semantics in LISP”, in *Proc. ACM Symposium on Principles of Programming Languages POPL*, Salt Lake City, Utah, 1984.

Suchman, L. (1987): *“Plans and Situated Actions”*, Cambridge University Press, Cambridge, UK, 1987.