Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit

Paul Dourish

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

Abstract

CSCW toolkits are designed to ease development of CSCW applications. They provide common, reusable components for cooperative system design, allowing application programmers to concentrate on the details of their particular applications. The underlying assumption is that toolkit components can be designed and implemented independently of the details of particular applications. However, there is good evidence to suggest that this is not true.

This paper presents a new technique which allows programmers to express application requirements, so that toolkit structures can be adapted to different circumstances. Prospero is a toolkit which uses this technique to meet different application needs flexibly.

Keywords: application control, CSCW toolkits, Prospero, consistency management, consistency guarantees.

1 Introduction

CSCW toolkits (such as Rendezvous [Hill et al., 1994], GroupKit [Roseman and Greenberg, 1996] or Suite [Dewan and Choudhary, 1992]) are systems which make it easier for programmers to develop CSCW applications. They provide generic, reusable components and behaviours which application programmers can incorporate into their systems. Components might include telepointers, shared data objects, or mechanisms to join and leave conferences. Using these components to capture common system elements, programmers can concentrate on the particular details of their own, specific applications.

There is a critical assumption which underpins this sort of reuse. It is that the components provided by the toolkit can be designed independently of particular applications, and can be reused wholesale to meet the different needs of different applications. However, there is evidence that this assumption is problematic. This evidence suggests that the *usage patterns* of CSCW applications depend, in a detailed way, on the specifics of the component design. In other words, the details of toolkit components are as much of a factor in supporting collaborative behaviour as the details of the application. The independence of toolkit from application is undermined.

This relationship, between details of system design and details of use, suggests that we should take a new approach to toolkit design, and this paper will discuss a new technique developed in Prospero, a prototype CSCW toolkits. First, though, we will consider two examples of studies which highlight the problem.

1.1 The Relationship between Design and Use

Dourish and Bellotti [1992] describe experimental studies of a collaborative text editor, supporting groups of three authors in brainstorming and design tasks. The editor, ShrEdit [McGuffin and Olson, 1992], gives each author a separate edit point in a synchronously-shared text workspace. Each can move around the document and work independently, although mechanisms are provided for synchronising views, tracking others and so forth. An implicit region-locking mechanism helps maintain consistency and avoid conflicts. Working in a shared data space, authors can see the effects of each other's work as it is performed.

Dourish and Bellotti's analysis highlights the collaborators' continual use of the visualisations of each other's work to maintain an ongoing awareness of the work of the group. This allows them to continually relate their own work to that of others and of the group as a whole, and so achieve their informal "division of labour". Unlike systems using explicit roles or information exchange to manage group behaviour and provide a sense of ongoing activity, ShrEdit's shared feedback approach is more open and flexible and leads to much more fluid transitions between forms of working, by providing a continual sense of both the character and content of other people's work. Similar mechanisms have been shown to operate in real-world collaborations in physical settings [e.g. Heath and Luff, 1992].

Greenberg and Marwood [1994] discuss a range of distributed systems techniques used in CSCW systems to manage collaborative access to distributed data. CSCW systems must maintain representations of users' work which may need be to be visible or accessible to different users at the same time, and techniques for managing this are frequently embedded in CSCW toolkits. Greenberg and Marwood show that common mechanisms, such as locking and serialisation, introduce temporal dependencies which restrict the patterns of collaboration in which groups can engage. For instance, the overhead of "locking" data representations can interfere with free-wheeling interaction (such as brainstorming); and the use of "rollback" techniques can actually cause data to change under the users' feet.

The fundamental point which Greenberg and Marwood point out is that the “distributed system” elements of collaborative applications, which are often embodied in toolkit components, cannot be considered independently of the interactional ones. They are mutually influential.

1.2 Toolkit Structures and Application Needs

The studies outlined above illustrate that the activities and interactions of collaborating individuals are organised not only around the work that they’re doing, but also around the details of the tools they have for doing that work—the collaborative technologies which support them.

This observation undermines the “independence assumption” at the heart of toolkit design. The independence assumption is that the collaborative components which are embodied in a toolkit are independent of the applications in which they will be used. Toolkit designers are concerned with the reusability of their components in a wide range of applications and circumstances. If they provide shared data objects, then they want application programmers to be able to use those in any situation where shared data objects are needed. Unfortunately, we’ve seen that the details of *how* objects are shared has an impact on the kinds of sharing and collaborative behaviour that takes place. Toolkit designers have to provide sharing mechanisms to make their shared data objects work; but these mechanisms cannot be completely independent of the application’s requirements concerning patterns of interaction.

Prospero is a prototype CSCW toolkit which addresses these sorts of problems. Most toolkit designs try to exploit the general applicability of components to a range of situations, leaving application programmers to map application needs onto the general facilities which the toolkit provides. Prospero’s approach is different. Using a recent software abstraction technique called Open Implementation [Kiczales, 1996], Prospero allows application programmers to tailor the toolkit, and match toolkit facilities onto the needs of the application, rather than the other way around.

This approach is used to give application programmers control over a number of different areas of toolkit functionality. This paper describes a mechanism called *consistency guarantees* which Prospero uses to give applications control over the consistency management mechanisms in the toolkit. Consistency mechanisms are the parts of the toolkit which ensure that data representations remain consistent even though users may attempt to make simultaneous, conflicting changes (such as when one user changes an object’s colour to blue, and another changes it to red). The goal is to allow the programmer to express aspects of the application domain, so that the toolkit can manage consistency in a way which is responsive to the needs of the particular applications.

The next section outlines some background, and briefly summarises how Prospero deals with the distributed data over which consistency mechanisms operate. The section which follows introduces promises and guarantees, the fundamental mechanisms in the new consistency technique. I will then discuss Prospero’s mechanisms in comparison with some other techniques, before providing examples of how programmers can use these techniques to make the toolkit responsive to application requirements.

2 Divergence in Prospero

Although this paper concentrates on consistency management, rather than data distribution, a brief sketch of the data management mechanism is necessary here to set the scene[1].

Prospero’s data distribution strategy is based on *divergence* [Dourish, 1995b]. Some approaches (such as centralisation or floor control) control data flow and management by maintaining a model of one-at-a-time action; that is, only one person can operate over the data at once, and so only one copy of any data item is “active” at a time. In Prospero’s model, on the other hand, multiple users can act over data items at once. These separate actions can cause different users to have different views of the data; this is *divergence*. The complementary operation is the *synchronisation* of these divergent views to re-establish a common view of the data. So data management takes the form of the continual divergence and synchronisation of views of the data. Particular threads of activity which diverge from each other are called *streams*.

Dourish [1995b] discusses three primary benefits of this strategy in a CSCW toolkit.

1. It can be applied across a range of synchronisation frequencies. Frequent synchronisation results in “synchronous”-style interaction, where group members can observe each other’s work as it progresses; infrequent synchronisation is more akin to traditional “asynchronous” application styles.
2. Since it incorporates a notion of resolvable, simultaneous work, it provides support for parallel, disconnected activity—“multi-synchronous” work. Parallel simultaneous activity is a common style of working, but traditionally CSCW systems have provided poor support for it.
3. For the same reason, it also supports the sorts of opportunistic activity in collaborative work which are revealed by observational studies (e.g. [Beck and Bellotti, 1993]), in which pre-arranged plans of activity and divisions of labour—where they even exist—are subject to local reorganisation and rearrangement.

With that background, we can now focus on consistency management in particular.

3 Constraining Divergence

In database terms, Prospero’s divergence/synchronisation strategy is an *optimistic* one. It presumes that simultaneous actions will probably not result in conflict, but that if conflict *does* occur, things can be sorted out later. Locking, on the other hand, is a *pessimistic* strategy; it presumes that simul-

1. The reader is referred to [Dourish, 1995a] and [Dourish, 1995b] for more information and background on the techniques used.

taneous operations are likely to lead to conflict, and so should be prevented.

Pessimistic strategies guarantee the maintenance of consistency, since they prevent the simultaneous action which would lead to inconsistency on the first place. On the other hand, optimistic strategies support more open styles of working. Prospero uses an optimistic strategy because the freedom and flexibility it provides is better suited to the needs of collaborative work. The price of this freedom is that the toolkit must provide explicit means to maintain consistency.

The problem is that the divergence model *per se* makes no commitment to the nature or extent of the divergence. The longer two streams of activity remain active but unsynchronised, the greater their potential divergence, and so the more complex it becomes to resolve conflicts at synchronisation-time. Indeed, there's nothing to say that the system will *ever* be able to resolve two arbitrary streams into a single, coherent view of the data store. Essentially, unconstrained divergence leads to arbitrarily complex synchronisation; and that can be a practical inconvenience, to say the least.

3.1 Variable Consistency

The first approach used in Prospero is to distinguish between *syntactic* and *semantic* consistency.

By "semantic" consistency, I mean that the data store contains no inconsistencies from the perspective of the application domain. The data is fit for its intended purpose. This is the conventional, intuitive form of consistency in collaborative and distributed systems. Appeal to "syntactic" consistency, on the other hand, allows for semantic inconsistencies, but ensures that the data store is structurally sound, so that some kind of activity can continue.

As an example, consider a multi-user text editor which attempts to resolve a conflict—the same paragraph has been completely rewritten, separately, by two different authors. If the system were to throw away (say) the earlier of the two paragraphs, then it would be preserving semantic consistency (although the reader should note that consistency does not imply "correctness"). However, this "lossy" approach is not necessarily the best suited to the needs of collaborating authors, even though the synchronisation procedure is straight-forward. An alternative mechanism would be to retain *both* the paragraphs within a structure which flags this as a conflict which the system cannot resolve—essentially preserving the text for the authors to sort out later. This approach preserves syntactic (structural) consistency. By only preserving syntactic consistency in some cases, rather than semantic consistency, a divergence-based system can achieve synchronisation more often, and continue operation in the face of potential problems. Consistency from the users' perspective is often not the same as consistency from the system's.

3.2 Using Application Semantics

The key observation which lies behind the variable consistency approach above is that the toolkit components, themselves, are not the final arbiters of "consistency". Instead, the toolkit can focus on making the data consistent *for the purposes at hand*. In other words, it is taking advantage of details about the application domain and the circumstances in which the toolkit is being used.

However, while using application-specific synchronisation might *postpone* some of the problems of unbounded divergence, the basic problem of unbounded inconsistency remains with us. The same basic technique—taking advantage of application semantics—can be applied to this problem. Prospero introduces the notion of application-specific consistency guarantees to control for the divergence process using details of particular circumstances. The roots of this mechanism lie in the strategies of existing systems.

3.3 Constraining Divergence with Locks

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

As outlined earlier, Greenberg and Marwood [1994] discuss some issues surrounding concurrency control in CSCW systems. Most strategies for managing distributed data have arisen in arenas such as distributed databases, distributed file systems, etc. Greenberg and Marwood point to a range of ways in which these approaches have interactional implications. Collaborative systems differ from many traditional distributed systems in that *in CSCW, not only the application, but also the interface, is distributed.* The choice of concurrency management strategy can have a significant impact on the styles of interaction which an application can support. One obvious example is the way in which the temporal properties of concurrency control strategies, such as relative execution times of actions over shared data, can interfere with interactional requirements in the interface. Similarly, approaches which apply a post-hoc serialisation on user actions may introduce unexpected interface behaviours (such as undo-ing actions).

Locking is a very general approach. A wide range of locking strategies have been used in CSCW systems, varying in how the locks are requested, obtained, granted and relinquished, what kinds of operations require locks, and the granularity of data units controlled by a single lock. However, the basic pattern (lock-act-release) remains the same, and so do the basic problems of locking for CSCW applications. Locking is a pessimistic concurrency strategy; on the assumption that any conflict could be damaging, it prevents conflict arising in the first place. Locking restricts activity on the data store, and hence restricts the activity of users.

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

Prospero is a toolkit for creating collaborative applications, and so it must embody more flexible mechanisms which can be adapted or appropriated for a range of application needs and interactional styles. Clearly, something more flexible than locking—even when supported by a range of strategies—is needed.

3.4 Promises and Guarantees

In an attempt to find a more flexible approach than the strict locking mechanism, and one more attuned to the needs of a CSCW toolkit, our starting point is with a generalisation of the traditional locking process. Locking is essentially a means by which a client[2] receives some guarantee of future consistency ("no other user can make changes, so consistency will be maintained") in exchange for a prediction of the client's future activity ("changes will only be applied to the locked region"). So we state this as the first principle: *locks are guarantees of achievable consistency*.

Immediately, this view has a number of interesting implications. First, there's clearly a wide range of such guarantees which could be made. Normally, locks are all-or-nothing guarantees. When we think in terms of guarantees of consistency, then we can consider distinguishing between different *degrees* of consistency, and the fact that a guarantee may only hold for limited consistency (in the worst case, perhaps, just syntactic consistency). Determining the achievable level of consistency is the responsibility of the server, based on currently-issued promises and the information about future activity which the client provides. The nature of these client "promises" will be discussed in more detail later; for now, though, it's enough to say that they are characterisations of expected behaviour, such as whether the client will simply read data, write new data but not delete anything current, delete or modify existing data, and so forth. So, second, these promises could vary in specificity and detail, just as the guarantees can vary. Third, and perhaps most importantly, when we think of this exchange as being less absolute than the strict locking exchange (an absolute guarantee for an absolute promise), then it becomes obvious that this is a negotiation; a client may make increasingly restrictive promises in exchange for increasingly strong guarantees of consistency. The promise/guarantee cycle is the basis of the consistency guarantees approach.

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

However, this generalisation still suffers one of the major problems with the locking approach applied to CSCW. Since divergence is still preceded by a description of expected activities, the possibility of opportunistic activity is still restricted. This was raised earlier as a criticism of traditional locking mechanisms, which interfere with the way in which collaborative work proceeds naturalistically. Obviously we would like to address this problem in our redesign. So we introduce the second principle: *a client can break a promise, in which case the server is no longer held to its guarantee*. So the characterisation of future activity which a client makes—its promise—may not be binding; when the time comes, the client (or the user) may actually do something else. However, in this case, the server can no longer be held to the guarantee it made of the level of consistency which can be achieved.

With this second principle in place, the consistency guarantee mechanism provides more direct support for opportunistic working styles. Just as in naturalistic work, stepping outside previously-agreed lines is not impossible; but the mechanism provides stronger guarantees when used cooperatively by both client and server. Of course, the user need not (often, should not) be exposed to this complexity and unpredictability. In a toolkit, these facilities are provided so that they can be appropriately deployed (or not) by an application developer. A developer might choose *not* to exploit the second principle *in a given application*, where application requirements or usage patterns would make it inappropriate. These might include cases where the resulting conflicts may be too difficult to synchronise later, or where loss of integrity in the data-store would be unacceptable. In other cases, an application developer might want to warn the user when such a situation was likely to occur, so that an informed decision could be made as appropriate to the particular circumstances. The framework supports these behaviours, but doesn't require them.

So, adding consistency guarantees to Prospero provides a way to overcome the problem of unbounded divergence; they act as a curb to Prospero's optimism. They provide some of the predictable consistency which pessmistic strategies support, but in a way which is sensitive to patterns of

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

collaborative work (rather than simply distributed systems). The examples in section six will show how these potential benefits are realised in actual applications.

4 Related Approaches in Database Research

The variable consistency approach outlined in section 3.1 used knowledge of application semantics to specialise and improve the synchronisation process. Essentially, the consistency guarantee mechanism introduced in section 3.4 uses knowledge of application semantics—and the semantics of particular operations—to increase the *opportunities* for concurrency and parallel activity.

Perhaps unsurprisingly, similar approaches have been explored in database design, since database management systems also involve multi-user activity over shared and perhaps replicated data. Barghouti and Kaiser [1991] provide a comprehensive survey of advanced concurrency control techniques. However, since databases tend to hide the activities of multiple parties from each other (preserving the illusion of sole access to a system), the primary (although not exclusive) focus of the database community has been on using concurrency to improve performance rather than to open up data models for collaboration. Two aspects of database research are particularly related to the consistency guarantees approach: semantics-based concurrency and application-specific conflict resolution.

4.1 Semantics-Based Concurrency

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

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

However, this is a very expensive way to maintain the transaction model, since the analysis of conflict is very coarse-grained. In the absence of transaction conflicts, the system can guarantee that the transactions can safely be executed in parallel. On the other hand, the presence of a conflict does not imply that inconsistency *will* result. For example, consider a transaction which issues a read request but doesn't use that result as part of a later computation (or does, but is robust to particular changes). It could, quite safely, be executed in parallel with another which writes that same data. However, that would signal a *read/write* conflict and the potential concurrency would be lost. More generally (and more practically), transaction concurrency (and hence throughput) could be improved with more detailed access to transaction semantics, or to application semantics.

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

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

4.2 Application-Specific Conflict Resolution

A second approach from database research which is relevant to the consistency guarantees mechanism is the use of application-specific conflict resolution. The Bayou system, under development at Xerox PARC, is a replicated database system for mobile computers, which are frequently active but disconnected from their peers. In most systems, disconnection is an unusual state, and the systems can normally be assumed to be connected to each other; but in mobile applications, disconnection is the rule, rather than the exception.

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

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

5 Encoding Promises and Guarantees

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

5.1 The Programming Interface

Prospero is written in Common Lisp. Applications built with Prospero are Lisp programs; Prospero is available as a library of routines which programmers can incorporate into their code.

Drawing on the Open Implementation structure, Prospero offers two interfaces to application programmers. The first—called the *base* interface—is a traditional library interface. It consists of a set of classes representing the basic structures of the toolkit, such as streams and guarantees, and provides methods on those classes which encode toolkit functionality. Application programmers make instances of these classes, and call the Prospero functions to manipulate them (e.g. to add an action to a stream, to synchronise two streams, or to perform an application action).

The second interface is called the *meta* interface. This is the interface which the programmer uses to express application details, and to tailor the toolkit structures to application needs. This second interface is implemented using a *metaobject protocol* [Kiczales et al., 1991]. Normally, tailoring is done by specialising Prospero structures and then providing new, tailored methods for the specialised classes. For instance, to change the synchronisation strategy for streams in a particular case, the programmer would create a new type of stream and then associate with it just those methods needed to express the new behaviour. Prospero then integrates these new mechanisms into its own operation.

Unlike the traditional split between "mechanism" and "policy", this approach keeps the encoding of policy at the level of the toolkit, rather than the application. However, it provides the means for programmers to override or modify elements of the policies which the toolkit uses, and to create new ones. Multiple policies can co-exist in Prospero at the same time. For instance, different types of streams are predefined, such as streams which automatically synchronise after some number of operations (called bounded streams), and streams which only synchronise when explicitly requested to by the user (called explicit-synch streams).

The examples provided in the next section will show how these ideas work in practice. First, though, the rest of this section will outline how Prospero represents the guarantees and promises on which the consistency mechanism is based.

5.2 Semantics-Free Semantics

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

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

5.3 Class-based Encoding

The mechanism that Prospero uses to accomplish this is *class-based encoding*. That is, the semantic properties for an application are named as classes in an object-oriented framework. Particular operations are represented explicitly as command objects [Berlage, 1994]; that is, invocations of any operation are represented explicitly as objects within the system. Each instance of a command object represents a particular invocation, along with relevant parameters and contextual information. Command objects multiply inherit from the classes which represent their semantic properties.

The use of explicit command objects is, in itself, a useful mechanism for representing sequences of action and arriving

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

(defmethod grant-guarantee (stream object (operation <safe-write>))
  ;; ... ok...
  (let ((guarantee (construct-guarantee <auto-consistent> stream object)))
    (record guarantee *guarantee-table*)))

(defmethod grant-guarantee (stream object (operation <write>))
  ;; ... restricted ....
  (if (granted-entry? <auto-consistent> <write> *guarantee-table*)
      (construct-guarantee <refused-guarantee>)
    (record (construct-guarantee <auto-consistent> stream object)
            *guarantee-table*)))

(defmethod grant-guarantee (stream object (operation <read>))
  ;; ... ok ...
  (record (construct-guarantee <consistent> stream object)
          *guarantee-table*))
```

Figure 1: Methods defining access to the shared bibliographical database in Prospero.

at appropriate mechanisms for resolving conflicts which might arise; but encoding semantic properties in the inheritance structure of the command objects yields two particular benefits for the problems which Prospero needs to address. First, the mechanism is inherently extensible; the application developer can create new semantic properties from existing ones within the same mechanism as she uses to create application structures and objects (i.e. subclassing and specialisation). Second, class-based encoding allows semantically-related behaviours to be defined in a declarative style. In the application, behaviours related to different semantic properties (or combinations of them) are written separately as methods specialised on the relevant classes, rather than in a complex, monolithic synchronisation handler. This relies on the object system's dynamic dispatch mechanism to match semantic properties (classes) to associated behaviours (methods) for particular command objects.

6 Using Consistency Guarantees

To provide a more detailed illustration of the use of consistency guarantees in collaborative applications, this section presents two more extended examples, along with the framework Common Lisp code which implements them. Since our concern here is with the use of the meta interface to encode application semantics and specialise the toolkit, the code examples focus on the use of Prospero rather than the design of the applications themselves. The examples show how application-specific semantic properties can be used *within a toolkit* framework to manage concurrency. Clearly, semantically-informed concurrency control could be hand-coded into applications, on a case-by-case basis; the issue here is the way in which these application-specific features can be exploited within a generalised toolkit.

6.1 A Shared Bibliographical Database

A simple example of an application whose collaborative performance can be enhanced by exploiting semantic information is a shared database for bibliographical information. The key property which we want to exploit in this example is that updates to the database are normally non-destructive. In a conventional locking approach, all updates would be seen as equally likely to conflict, and so locks would be used to prevent any simultaneous updates. However, one of the features of this particular application is that most updating is in adding new information, rather than removing or changing information already present. Simultaneous appends are much less likely to cause conflicts that simultaneous revisions, and this can be used to specialise conflict management in this particular application.

We can take advantage of this feature by introducing a class of actions which correspond to *non-destructive* writes (those which add new information, rather than changing anything). A standard access mode for the collaborative database during disconnected operation, then, would be the combination of reads and non-destructive writes, and could be encoded in Prospero as shown in figure 1.

So, in the initial code fragment, the editing actions are bracketed by a request/synchronisation pair. The generic function `request` requests[3] a guarantee for the local data stream on the whole database, specifying that the expected behaviours will be of types `<read>` and `<safe-write>` (non-destructive writes). The guarantee that it receives is later used as part of the synchronisation process, once the edit actions have taken place.

The rest of the code in figure 1 handles the other side of the transaction—evaluating the promise and granting the guarantee. Taking advantage of the specifics of this application, the code can adopt the policy that, like read capabilities, safe-write capabilities can be granted to multiple clients at a time.

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

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

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

(defmethod salvage-synchronise-action ((stream <stream>) (action <editor-action>))
  (if (action-conflict? action (stream-history *stream* :relative-to stream))
      ;; ... definite conflict ...
      (syntactic-local-apply-action action)
    (if (guarantee-conflict? action *guarantee-table*)
        ;; ... potential conflict ...
        (tentative-local-apply-action action)
      (local-apply-action action))))
```

Figure 2: Methods for synchronisation of the collaborative writing example in Prospero.

Here, we grant guarantees for read operations and for safe-write operations (although they receive different levels of consistency, which are also class-encoded). However, for general write operations, a guarantee is only issued in the case that no other guarantee has been granted to another writing client. Guarantees are recorded so that they may be used as the basis of later decision-making, as well as for synchronisation purposes later.

The use of guarantees as part of the coordination strategy is part of the framework which Prospero provides. What the programmer has done, in this case, is to specialise the toolkit's structures to take account of semantics of the application being supported.

6.2 Collaborative Text Editing

The previous example showed the selective granting of consistency guarantees based on characterisations of expected behaviour—the semantics of activity during the period of divergence. This second example illustrates the use of semantic properties in synchronisation.

Consider a collaborative text editing system in which multiple authors work on a single document, obtaining guarantees at the level of paragraphs or sections. As in the previous example, the guarantees obtained before divergence are passed along at synchronisation-time. At this point, the guarantee must be examined to verify that only promised actions were performed.

So there are two cases (distinguished in figure 2 by the predicate `valid?`). In the first case, the predicate indicates that the actions performed by the client are indeed those which were given in the promise. The promise has been upheld. In this case, synchronisation should be straightforward since the server was in a position to know what actions were expected beforehand. At this point, then, the type of the stream can be used to determine the appropriate synchronisation method (discussed in more detail elsewhere [Dourish, 1995b]).

In the second case, however, the client has broken its promise. There are various ways in which this situation could have arisen; and, critically, since a number of them are important features of naturalistic work practice, we would like to provide as much support for them as possible.

To handle this situation, the programmer calls the generic function `salvage-synchronise` to provide fall-back synchronisation. In this case, this involves stepping through the actions attempting to apply them one-by-one. By comparing the classes of the operations (that is, their semantic characterisation) with the activities of other streams, their compatibility can be determined. Actions compatible with activities performed (and guarantees granted) since the divergence point can be applied directly; other actions must be processed specially.

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

7 Mechanism and Policy in Open Implementations

The systems community recognises the problems arising from the commitments which implementations make to particular forms of application. One way to avoid these problems is to "split mechanism and policy". Mechanism is that part of the system which provides basic functional elements; policy is that part which deals with how they will be put together and used.

Some CSCW systems have adopted the mechanism/policy split. Examples include COLA [Trevor et al., 1995] and the "open protocols" approach in GroupKit. How does Prospero's approach compare with these?

A full discussion of the relationship between Open Implementations and split mechanism and policy is beyond the scope of this paper (but see Kiczales et al., 1991; Kiczales, 1996). However, there are two basic differences.

1. In the Open Implementation approach, policy remains in the implementation. Control is available from the application level, but it is exerted over policy in the implementation. In the mechanism/policy split, on the other hand, policy decisions *migrate* from the lower-level implementation to the application.

   This allows clients to reuse aspects of policy control, changing them to their own needs without having to rebuild them from scratch.

2. Analogously, since the policy control is retained in the implementation, the level of abstraction used to manipulate it is higher (closer to the application's level). The system does not have to reveal its basic components, for programmers to put applications together; instead, it can interpret a higher-level modification interface, and map it onto whatever implementation lies below. This also make the modification interface itself portable across implementations.

8 Summary and Conclusions

One of the principal distinctions between CSCW systems and purely distributed systems is the interactional component. The need to distribute the interface as well as data and application must be taken into account when considering common distributed system issues such as concurrency control. Traditional algorithms typically maintain consistency by restricting concurrency; however, this approach is unsatisfactory in general, as it often interferes with the flexible management of group activity.

The problem is that the mechanisms which are encoded in the software make premature commitments to particular styles of working. This problem compounded in designing toolkits for CSCW, since the (toolkit) mechanisms and the application are designed in isolation. This is not simply a case of a poorly-designed toolkit. The problem is inevitable, because providing working software means that decisions of this sort have to be made, one way or another, in the course of implementation.

What we have observed here is that the semantics of specific applications can be exploited to increase concurrency while maintaining adequate consistency in a collaborative data store. By looking in detail at the semantic properties of particular actions in a CSCW system, we can find operations which can be performed in parallel without leading to inconsistency. This approach has been exploited in Prospero, a toolkit for collaborative applications which uses metalevel techniques to allow application developers to reach in and tailor toolkit structures and behaviours to the particular needs of applications. The use of explicit semantic representations to tie toolkit-level consistency management to the specifics of application functionality is one aspect of this use.

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

Prospero provides a framework in which application functionality and semantics can be integrated directly into the toolkit. The result is a level of flexibility beyond that obtainable with traditional toolkits, in which application details can play no part in the operation of toolkit mechanisms. This flexibility takes two forms. The first is that the toolkit can efficiently support a wider range of applications than would otherwise be the case, since the toolkit structures can themselves be specialised and adapted to new circumstances. The second is that applications can be used more flexibly since they more directly accommodate a range of working styles and interactional requirements.

In developing collaborative applications it's critical that we understand the interactions between notionally "low-level" issues such as distributed data management and notionally "high-level" issues such as individual interaction and group activity. Such understandings must, in turn, lead to frameworks in which they can be applied broadly, rather than only in specific, hand-coded applications. Prospero is an attempt to tackle just these issues and create a framework for the holistic design of collaborative applications.

Acknowledgments

Dik Bentley, Jon Crowcroft, John Lamping, Ellen Siegel and Doug Terry made valuable contributions to the development of these ideas and their presentation here.

References

[Barghouti and Kaiser, 1991] Naser Barghouti and Gail Kaiser, *"Concurrency Control in Advanced Database*

Applications", ACM Computing Surveys, 23(3), pp. 269–317, September 1991.

[Beck and Bellotti, 1993] Eevi Beck and Victoria Bellotti, *"Informed Opportunism as Strategy: Supporting Coordination in Distributed Collaborative Writing"*, Proc. Third European Conference on Computer-Supported Cooperative Work ECSCW'93, Milano, Italy, September 13–17, 1993.

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

[Demers et al, 1994] Alan Demers, Karin Petersen, Mike Spreitzer, Doug Terry, Marvin Theimer and Brent Welch, *"The Bayou Architecture: Support for Data Sharing among Mobile Users"*, Proc. First IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, Dec 8–9, 1994.

[Dewan and Choudhary, 1992] Prasun Dewan and Rajiv Choudhary, *"A High-Level and Flexible Framework for Implementing Multiuser User Interfaces"*, ACM Transactions on Information Systems, 10(4), pp. 345–380, October 1992.

[Dourish, 1995a] Paul Dourish, *"Developing a Reflective Model of Collaborative Systems"*, ACM Transactions on Computer-Human Interaction, 2(1), pp. 40–63, March 1995.

[Dourish, 1995b] Paul Dourish, *"The Parting of the Ways: Divergence, Data Management and Collaborative Work"*, Proc. European Conference on Computer-Supported Cooperative Work ECSCW'95, Stockholm, Sweden, September 1995.

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

[Farrag and Ozsu, 1989] Abdel Aziz Farrag and M. Tamer Ozsu, *"Using Semantic Knowledge of Transactions to Increase Concurrency"*, ACM Transactions on Database Systems, 14(4), pp. 503–525, December 1989.

[Greenberg and Marwood, 1994] Saul Greenberg and David Marwood, *"Real-Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface"*, Proc. ACM Conference on Computer-Supported Cooperative Work CSCW'94, Chapel Hill, North Carolina, October 1994.

[Heath and Luff, 1992] Christian Heath and Paul Luff, *''Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Line Control Rooms''*, Computer Supported Cooperative Work, 1(1–2), pp. 69–95, 1992

[Herlihy, 1990] Maurice Herlihy, *"Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types"*, ACM Transactions on Database Systems, 15(1), pp. 96–124, March 1990.

[Hill et al., 1994] Ralph Hill, Tom Brinck, Steve Rohall, John Patterson and Wayne Wilner, *"The Rendezvous Architecture and Language for Multi-User Applications"*, ACM Transactions on Computer-Human Interaction, 1(2), pp. 81–125, June 1994.

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

[Kiczales, 1996] Gregor Kiczales, *"Beyond the Black Box: Open Implementation"*, IEEE Software, pp. 6–11, January 1996.

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

[McGuffin and Olson, 1992] Lola McGuffin and Gary Olson, *"ShrEdit: A Shared Electronic Workspace"*, CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, 1992.

[Roseman and Greenberg, 1993] Mark Roseman and Saul Greenberg, *"Building Flexible Groupware Through Open Protocols"*, in Proc. ACM Conference on Organisational Computing Systems COOCS'93, Milpetas, Ca., November 1–4, 1993.

[Roseman and Greenberg, 1996] Mark Roseman and Saul Greenberg, *"Building Real-Time Groupware with GroupKit, a Groupware Toolkit"*, ACM Transactions on Computer-Human Interaction, 3(1), March 1996.

[Terry et al, 1994] Doug Terry, Alan Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer and Brent Welch, *"Session Guarantees for Weakly Consistent Replicated Data"*, Proc. International Conference on Parallel and Distributed Information Systems, Austin, Texas, September 1994.

[Trevor et al., 1995] Jonathan Trevor, Tom Rodden and Gordon Blair, *"COLA: A Lightweight Platform for CSCW"*, Computer Supported Cooperative Work, 3, pp. 197–224, 1995.