H. Gellersen et al. (eds.), ECSCW 2005: Proceedings of the Ninth European Conference on Computer-Supported Cooperative Work, 18-22 September 2005, Paris, France, 45–64.

Supporting High Coupling and User-Interface Flexibility

Vassil Roussev
Department of Computer Science
University of New Orleans
vassil@cs.uno.edu

Prasun Dewan
Department of Computer Science
University of North Carolina
dewan@cs.unc.edu

Abstract. Collaborative systems that automate the sharing of programmer-defined user interfaces offer limited coupling flexibility, typically forcing all users of an application to share all aspects of the user interfaces. Those that automatically support high coupling flexibility are tied to a narrow set of predefined user-interfaces. We have developed a framework that provides high-level and flexible coupling support for arbitrary, programmer-defined user interfaces. The framework refines an abstract layered model of collaboration with structured application layers and automatic acquisition, transformation, and processing of updates. It has been used to easily provide flexible coupling in complex, existing single-user software and shown to support all known ways to share user-interfaces. Coupling flexibility comes at the cost of a small amount of additional programming. We have carefully crafted the framework to ensure that this overhead is proportional to the degree of coupling flexibility desired.

Introduction

Collaborative environments today (such as *NetMeeting*,*Webex,* and *LiveMeeting*) typically have two components: a shared-window application-sharing system that allows sharing of collaboration-unaware applications and a set of applications, such as a whiteboard and a distributed presentation tool, that are collaboration aware. The reason for providing the shared-window system is that the cost of implementing collaboration-aware applications is high. The reason for providing a special set of collaboration-aware applications is that a shared window system provides a very tightly coupled and inflexible model of collaboration in which What You See Is What I See (WYSIWIS). Collaboration-aware applications relax/extend this model in several ways. This is illustrated by the *NetMeeting*, *LiveMeeting*, and *Webex* whiteboards. They support sharing of a subset of the user-interface objects: for example, Figure 1(b) shows that the line drawn by user 1 is shared but the line-selection mode used to draw the line is not. Moreover, they support both synchronous and asynchronous communication of changes to

shared objects. For example, text insertions are sent as they are made. On the other hand, as a user draws a new line, the other users get no feedback (Figure 1 (a)). It is only when the line is completed that others user see it (Figure 1 (b)).

From a software-engineering point of view, it is important for the application-sharing system and collaboration-aware applications to share a single set of high-level abstractions for coupling user interfaces. However, in current systems, high-level coupling abstractions limit either the coupling flexibility or the user-interface flexibility. Those that provide high user-interface flexibility, like *NetMeeting*, restrict the coupling to near-WYSIWIS sharing. Those that provide high coupling flexibility restrict the user-interface to a textual display. Thus, none of these systems can support the variety of coupling modes implemented in collaboration-aware graphical applications supporting loose coupling. These include whiteboards, *MS PowerPoint* presentation systems (e.g., *Webex* , *LiveMeeting*), structured idea-finding systems (Prante 2002), and even table-top and large-display applications (Tse 2004). As a result, all of these applications must be implemented manually. While this has been an open problem for more than a decade, it remains an active issue because of the overhead of implementing collaborative applications. In fact, in the CSCW 2004 conference, the developers' workshop, "Making application-sharing easy," was devoted to the issue of how to relax the coupling of current near-WYSIWIS application-sharing systems.

We have addressed this question by developing a high-level framework that supports both high coupling and user-interface flexibility. Section 2 describes the research work related to our own, Section 3 and 4 discuss the components of our framework, Section 5 details our experiences with coupling complex, existing, single-user code, and Section 6 presents conclusions and future directions.

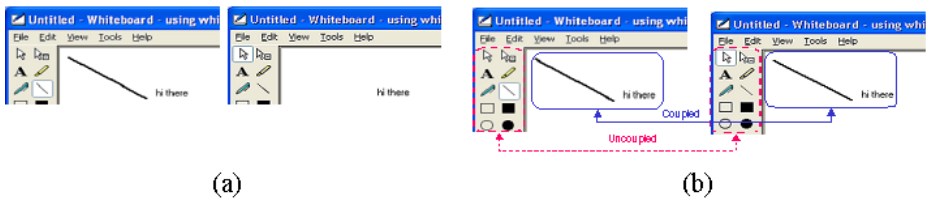

Figure 1 Non-WYSIWIS coupling in NetMeeting Whiteboard

Related Work

The coupling between two user-interfaces defines which parts of them are shared and when a change to a shared part in one user-interface is reflected in the other. Multiple coupling policies have been developed for two main reasons. First, users should be allowed to use a coupling policy that reflects their level of collaboration. For example, two users may wish to see the same or different

visualization of some data depending on whether their discussion is about the visualization or the data. Second, the system should be allowed to choose a level of coupling that gives the desired quality of service. For example, *NetMeeting* and other commercial collaboration-aware applications do not support immediate or synchronous remote updates to a graphical object being dragged (Figure 1) probably because of jitter problems in a wide-area network, while some systems that address these problem do support incremental graphical updates (Dyck 2004).

The relationship between user-interface and coupling technology is demonstrated by shared window/screen systems, which share the user interface (UI) by intercepting the I/O stream of a collaboration-unaware application. They collect the input from different users into the single I/O stream expected by the application, and replicate the single application output stream for each user. The result is that each user sees the same sequence of outputs. The degree of coupling, then, depends on the abstraction level of the output. In a screen-sharing system, the entire screen is replicated, while in a shared window system, only the shared windows are replicated. Stream-based sharing of this form is formalized in Chung and Dewan (2001), which provides sharing of an abstract I/O stream, to which the specific I/O stream of a system/application must be translated.

Stream-based sharing does not support sharing at multiple degrees of abstraction such as the ability to share either the same or different visualization of some data. The PAC (Coutaz 1987) and MVC (Krasner 1988) architectural frameworks provide a way to formally describe these two levels of sharing. MVC, in particular, divides an interactive application into a Model, a View, and a Controller, which address semantics, output, and input, respectively, of the application. The controller invokes methods in the model to inform it of input; model sends notifications to all of its views of any changes to its state. Different views can respond to these notifications in independent ways, thereby creating different visualizations of the model. The MVC framework is often simplified in later systems to the Model/View framework, in which the view and controller are combined into one object because of the many dependencies among them in editor-based applications, where input consists of editing output.

Some collaborative systems based on the Model/View framework provide sharing of either component. Model sharing is provided by allowing the views to be created on the displays of different workstations. View sharing is provided by creating each view as a physical replica of a single logically shared object. Different systems provide different mechanisms to keep a model consistent with its views and the different view replicas consistent with each other. *Rendezvous* and *Weasel* (Graham 1996) use declarative constraints, *GroupKit* (Greenberg 1994) uses table updates, *DISCIPLE* (Wang 1999) uses *JavaBean* events, *JViews* (Grundy 1998) uses a more general `ChangeDescription` object, and *Colab* (Stefik 1987) supports broadcast methods that are invoked on all replicas. Some of these systems, such as *Rendezvous*, centralize the model while others, such as

GroupKit and *JAMM* (Begole 1999), replicate it, while addressing display synchronization, externalities, and other replication issues.

These systems allow arbitrary user-interfaces to be created by the programmer-defined view objects. However, their support for coupling is limited in several ways. They cannot support fine-grained sharing of model or view objects. For example, they cannot support the *RTCAL* collaborative calendar application (Sarin 1985), which allows sharing of public, but not private, appointments of a user. Moreover, they do not provide automatic support for asynchronous sharing of shared objects. Most of the above systems support only synchronous sharing. The exception is *JViews*, which also supports asynchronous sharing by logging events, but requires users to manually flush the logs and resolve any inconsistencies. Furthermore, they do not allow users to dynamically change between model and view sharing. The level of sharing is fixed at compile time based on whether a shared or normal view object is used. Finally, they do not support sharing at a lower-level of abstraction than views, in particular screen or window sharing.

The design of *Suite* (Dewan 1992) shows that coupling inflexibility of other (concrete) collaborative systems can be addressed if the system coupling the UIs is also the one that automatically generates them. This knowledge is used to support fine-grained, synchronous, asynchronous, and multi-layer sharing policies, and dynamic changes to them. The problem is that these policies apply only to the limited set of user interfaces generated by the system, which does not include graphical interfaces such as a whiteboard.

One way to achieve the coupling flexibility of *Suite* and UI flexibility of systems supporting programmer-defined views is to provide an architecture for easily adding new coupling implementations. This approach has been taken in several systems such as *DISCIPLE*, *AMF-C* (Tarpin-Bernard 1998), and *JViews*. While it is important to offer extendibility/composability, it is also crucial to recognize the commonality in the coupling policies supported by existing software and provide high-level support for these policies. The experience with *Suite* has shown the benefit of providing high-level support for a comprehensive set of coupling policies. Fifteen years after it was developed, as far as we know, no new coupling policy has been identified for the (textual) UI it supports.

The layered architecture model of Dewan (1998) provides a way to reason about multiple levels of sharing for arbitrary user-interfaces. It assumes that input and output are processed by a series of layers, where each layer abstracts the I/O received by the lower layer. An example of such a series of layers is the screen, window, view and model layers. The layers communicate interaction events to implement the user-interface of the application. A collaborative architecture can be modeled by a (possibly empty) series of shared layers followed by a series of "replicated" layers that communicate coupling events to share their state.

The “replicas” can diverge by sharing a subset of the objects managed by them and queuing changes to shared objects before transmitting them to peer replicas. This model is abstract in that it does not describe the exact form of communication across layers. Thus, it does not provide a system that automates multiple levels of sharing, serving only to define them informally.

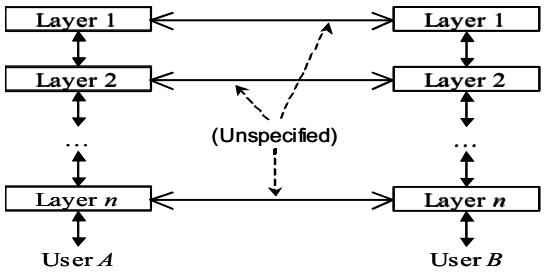

Figure 2 Layered model

Our framework combines and extends the various approaches/concepts described above. It defines an open, composable architecture for implementing coupling that can be considered as a concrete, automatable version of the abstract layer architecture above. Based on user-defined descriptions of application layering, it supports sharing of any subset of layers. More generally, it provides high-level support for a set of coupling policies that is comprehensive in that it covers all known coupling policies. Like *Suite*, it is able to support dynamic changes to the coupling policy.

Overview

To make our discussion concrete, we first consider the implementation of a multi-user Outline application in *Java* from the point of view of the application programmer. Our starting point is a single-user model/view implementation where the model has the recursive structure (of subsections) shown on Figure 3 and the view is the tree interface on Figure 4.

```
public class Outline {
 String getTitle();
 void setTitle(String title);
 void insertSection(int i,Section);
 void removeSection(int);
 Section getSection(int);
 void getSection(int,Section);
 int getSectionCount();
}
```

Figure 3 Example *Outline* object definition

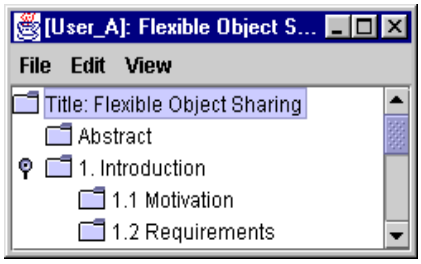

Figure 4 Example *Outline* user interface

To add collaboration support using our infrastructure, the developer must: **a)** register the roots of the shared object structure (`Outline/OutlineView`); **b)** provide specifications so that the rest of the shared object structure (title, sections, and subsections) can be extracted automatically; **c)** provide update notifications of user updates to the outline; and **d)** provide layer descriptions to enable dynamic transition between model and view coupling. Depending on the coding conventions used and the desired degree of coupling, steps b), c), and d) are optional. For the rest of this section, we briefly present each step of the process.

Registration and Initialization

Figure 5 illustrates the necessary additions to the startup code of the original application. The un-highlighted code is single-user code that would have to be written even if no coupling was desired. It creates an outline model and view, informs the view about the model, and displays the application window.

```
public static void main( String[] argv) {
  SystemBoot.initAll( argv);
  // --- Single-user initialization
  Outline outline = initOutline();
  JFrame outlineView = initOutlineView( outline);
  outlineView.setVisible( true);
  // ---
  ColabJMenu.addColabMenu(outlineView);
  PropertyRegistrar.register(outline, "Outline");
  PropertyRegistrar.register(outlineView, "OutlineView");
  ObjectBrowser.addRootObject(outline);
  ObjectBrowser.setVisible(true);
}
```

Figure 5 Initialization code for example application

The highlighted code is the added collaboration-aware code, which is external to the model and view. It adds a special collaboration menu (Figure 6) to the window that allows users to execute collaboration-aware commands to transmit pending updates. It registers the outline model and view with the infrastructure, which uses pattern specifications to decompose these objects, and assigns unique global ids to the tree of objects rooted by them. Finally, it instantiates an object browser (Figure 7) to enable flexible coupling specification by the user.

The object browser is an application-independent user interface component through which users control the sharing of application objects. The idea is to have a unified collaboration control interface in order to save development effort and to allow users to transfer collaboration experience from one application to another. The browser shows, in a dedicated window, a tree representation of the structural hierarchy of the shared objects registered with the infrastructure. To change the sharing, a user navigates to the desired object and selects a specific sharing policy from a list of predefined ones, or customize one on the fly (Section 3.5).

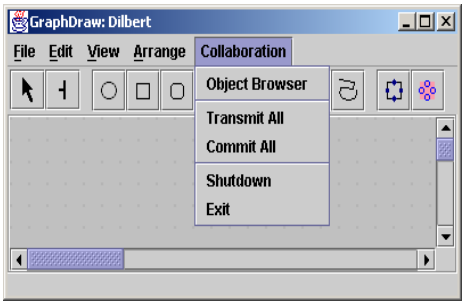

Figure 6 Collaboration menu attached to an application

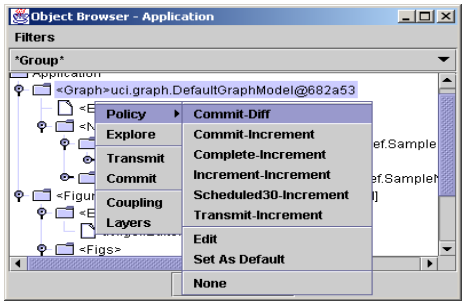

Figure 7 Object browser with policy selection pop-up menu

Object Structure Specification

To present the GUI on Figure 7, and to enable fine-grained coupling in which different layer components are coupled differently, we must derive the logical structure of the shared objects. Ideally, this should be accomplished automatically, as it is the case in *Suite*. However, *Suite* assumes that the entities are defined by concrete data types such as records and arrays, which expose their structure. Our framework assumes entities are encapsulated objects and cannot automatically decompose an object without making any assumptions about it.

Our approach to address this problem is to build on the fact that code naming conventions used to convey information to other developers can also be used by our framework to decompose an object. We support a language for describing object components or *properties* based on the relationships among the signatures of methods used to access them. This approach is more fully motivated and described in Roussev (2000); Roussev (2003)—here describe it "by example" to give a concrete idea of how it is used, based on `Outline` object of Figure 3.

The properties of instances of this type are defined by two definitions. The first definition states that a simple property <*PropName*> of type <*Type*> is defined whenever two property methods, "getter" and "setter" can be found such that the constraints on their signatures described above are met:

```
type = simple
methods
 getter = <Type> get<PropName>()
 setter = void set<PropName>(<Type>)
name = <PropName>
```

Similarly, the following definition describes a variable-sized sequence property, <*PropName*>, whose elements are of type <*ElemType*>:

```
type = sequence
methods
 insert = void insert<PropName>(int,<ElemType>)
 remove = void remove<PropName>(int)
 lookup = <ElemType> get<PropName>(int)
 set  = void set<PropName>(int, <ElemType>)
 count = int get<PropName>Count()
name = <PropName>
```

Together, these two definitions describe the two properties in the outline example—a simple property named "Title" and a sequence property named "Section". In a property definition, free variables such as <*PropName*>, <*ElemType*>, and <*Type*> must be unified to the same values in all uses. These free variables allow property definitions to describe a whole family of interfaces. In fact, an interface is a property definition with no free variables. The overhead of creating property definitions is amortized over the family of classes/interfaces that use the conventions encoded by them. For example, the *JavaBeans* convention definition is shared by all object classes that use them.

The property-description language comes with a *Java*-based introspection mechanism to dynamically determine the properties of an object and invoke the methods to read/write them. For example, it provides the following method to determine the properties found:

```
Property[] mp = Introspector.getProperties(className, specs);
```

It also provides a way to access the methods for handling a property:

```
Method getter = property.getMethod("getter");
```

The *Java* `Method` class allows runtime invocation of its instance:

```
getter.invoke(target, null);
```

In our specific example, the developer would not have to give any specifications as our implementation by default supports *JavaBeans* properties, as well as *sequence* and *table* properties. Depending on the coding style, additional specs may be necessary for other applications. We should point out that the definitions are given in separate *XML* files and are reusable across applications.

Update Notification

In order to provide automatic coupling, the infrastructure must learn of user updates to the shared structures. Ideally, the application should notify the infrastructure of each incremental update. However, this may require more effort than the developer is willing to invest and/or more than the user actually needs. Therefore, we give a range of sharing options and specify the implementation effort required for each one of them.

Asynchronous fine-grained diff-based sharing can be achieved without any notification support from the Outline application. We have developed a general property-based diffing algorithm (Roussev 2003) that can derive the fine-grained updates from successive snapshots of the object's state. The properties used in the Outline are supported by default so no additional effort is required. Thus, upon a user command (or timer expiration) the infrastructure performs diffing and communicates any discovered updates. For applications using other property types, specific diff operations may need to be defined as separate methods that can be shared across applications using the corresponding property type.

Incremental, synchronous sharing can be achieved by announcing update events. This can be accomplished in one of two ways—by directly using the infrastructure-defined event model, or by translating existing application events into the model. The first option requires an extra line of code at the end of each

method of the shared object that modifies the state (such as `setTitle`, `insertSection`, `removeSection`, and `setSection`). For example,

```
public void setTitle(String title) {
…
Coupler.dispatch(new PropertyOperation(this.getGID(),"title", "setter",newObject[] {title}));
}
```

The second option may be more attractive for component-based applications that already have their own events. Our implementation provides two general-purpose reusable event adapters for translating *AWT* and *JavaBeans* events.

Delayed semi-synchronous sharing that communicates changes when they reach a certain completion or correctness level requires *synchronization* events to be transmitted (Sections 4.2-4.3). A synchronization event is a meta-event that labels a preceding update as having certain level of completion and/or correctness. For example, typing a character in the title would trigger an update notification, while pressing `<Tab>` might indicate that the change is complete.

In practice, model-level objects (i.e., `Outline`) must be aware of the level of correctness of each change, as it is their job to ensure it. Hence, they only need to pass along this information by tagging the updates as `Parsed`, or `Validated`. In terms of implementation, this corresponds to one more line of code for each modifier method. Indicating a `Complete` editing operation for the Outline is slightly more complicated—in our prototype it took an additional 15 lines of code.

Application Layering Specification

Recall that, to increase coupling flexibility, *Suite* provides sharing at two levels (model and view) that can be dynamically switched at run-time. This is possible because the system builds the UI and knows the precise application layering. However, in our model, we support arbitrary layers and thus need an alternative mechanism—developer-provided layer descriptions. To illustrate, consider the layer decomposition for the Outline application shown on Figure 8 and its corresponding (partial) *XML* description given on Figure 9.

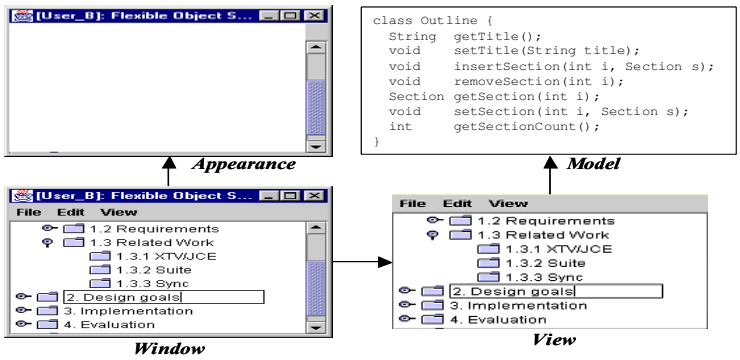

Figure 8 Layer decomposition for Outline application

At the lowest level is the *window* layer, which consists of the single application window through which all objects are edited. The *view* layer consists of the window's menu-bar and a `JTree` object through which the outline object is edited. The *appearance* layer consists of the elements of the application window that do not affect the state of the outline, such as the scrollbar. In this example, a user action may trigger one of two sequences of events. If the user performs an action that modifies the outline, the process triggers three causally related notifications at the *window*, *view*, and *model* layers, respectively. If a user action concerns only the *appearance* layer (e.g., scrolling), it triggers a sequence of two causally related notifications—at the *window* and *appearance* layers.

```
<object class = "outline.Outline">
  <layer name="model"><ALL/></layer>
</object>
<object class = "outline.Section">
  <layer name="model"><ALL/></layer>
</object>
<object class = "java.swing.JTree">
  <layer name="view"><ALL/></layer>
</object>
…
```

```
<layer_dependencies>
  <window>
    <view>
      <model/>
    </view>
    <appearance/>
  </window>
</layer_dependencies>
```

Figure 9 Layer definitions for Outline application

Thus, if *window* sharing is specified, all notifications from other layers will be suppressed. If *view* sharing is specified, then *model* and *window* notifications will be suppressed. Similarly, if *model* sharing is specified, *window/view* events are suppressed. Since the *appearance* layer is independent of both the *model* and *view* layers, its sharing can be turned on/off independently of the *model* and the *view*.

Coupling Specification

To complete the overview, we present the coupling control interface seen by the user. Using the object browser (Figure 7), the user selects a layer, an object, or an object property and then selects the desired policy from a pop-up menu. This is either a named (predefined) policy or a custom one built on the fly. The drop-down list in the browser allows different policies for the interaction with different users to be selected. The `*Group*` value shown is a default for all participants.

Policy customization is invoked by selecting 'Edit' from the pop-up menu, which brings up the policy editor (**Error! Not a valid bookmark self-reference.**). A detailed explanation of the different policy parameters is given Section 4.2 but the essential idea is to define the conditions under which updates are transmitted/received. The policy shown on the figure is asynchronous fine-grained diff-based sharing: updates are obtained using diffing, sent whenever the user chooses to commit them and are installed as soon as they are received. After editing is complete, the user has the choice of *Apply*-ing it to the target object/property, or *Save*-ing it as a named policy.

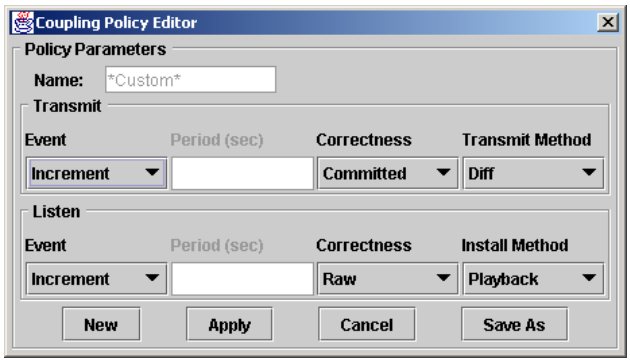

Figure 10 Coupling policy editor

In summary, depending on the desired level of support, a developer needs 20-45 lines of application code and several *XML* specifications to incorporate all features of the synchronous, event-based sharing for the Outline application. Programmers can incrementally add the code and learn the concepts behind it as more coupling flexibility is required. Users will be able to immediately take advantage of the new features using the same control interface.

Framework

Update Events

The layered model leaves unspecified the communication of (1) interaction events that go up and down layers and (2) coupling events that go across layers. The interaction events should be left unspecified in a collaboration framework to accommodate arbitrary programmer-defined UI. However, automating replica coupling implies making some assumptions about the coupling events.

The interaction events supported by the Model/View framework provide a basis for designing and understanding coupling events. The framework supports an asymmetric communication model, where the communications up and down are different in nature. A view informs the higher-level model layer about an input event by directly invoking a model-specific method in it. On the other hand, a model informs its lower-level view layers about state changes by sending view-independent notifications to them. A view processes the notification by retrieving the state of the model in which it is interested, and calling a view-specific method to update its own state. In our framework, we combine these approaches when defining (replica) update events, recognizing the fact that an object generating such an event is also capable of processing it. As in the notification-based approach, a replica does not directly call methods in its peers, and as in the direct-method invocation approach, it does not have to map notifications to the methods that process them. The events are symmetric in nature and are defined in terms of properties to support fine-grained coupling.

An update event encodes an operation invoked on a property of a replica and the arguments of the operation. Specifically, it contains: (1) the global identifier of the replica on which the operation is performed; (2) the name of the operation (e.g., "insert" on section); and (3) a list of arguments. For example, inserting a new section into the outline would be associated with an event of the form:

```
<"Outline","section","insert",{2, section}>.
```

This information is used by the coupling infrastructure to perform a reflective invocation on remote replicas without requiring them to translate the event. Thus, this approach has the benefit of direct method invocations in that a target object does not have to do any event processing. It also has the benefit of the indirect notification-based approach in that the event can be sent to a variable number of targets, and more important, can be "intelligently" handled by the system.

Parameterized Update-Event Handling

On the surface, there does not seem to be any need for special handling of update events, beyond translating these events to corresponding replica methods. In fact, this is all we need if (1) each replica responds to property updates by generating corresponding update events, (2) all of these updates must be made synchronously to all of the other replicas, and (3) replicas are not concurrently updated in inconsistent ways. If these three assumptions do not hold, then special acquisition, processing, and installation phases, respectively, are needed (Figure 11).

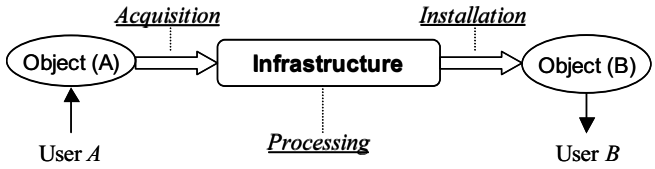

Figure 11 Phases of update handling

In the acquisition phase, a description of the update to a shared property is either received from the changed replica or generated by the infrastructure. In the processing phase, the infrastructure filters, buffers, transforms, and communicates the update to the remote parties. Finally, in the installation phase, the update is merged with the current state of the remote object to which it is delivered. Each of these phases is controlled by user-specified, interrelated parameters.

These parameters are associated with (properties of) each replica of a shared object to allow users to autonomously control event handling. As the acquisition and installation are local operations performed on the source and target objects, respectively, these are controlled by the corresponding parameters of these two objects. The processing operation, on the other hand, involves both objects as it determines what is shared by the two objects and when it is shared. As the users owning these objects should be allowed to independently specify the nature of sharing, our framework uses processing parameters of both the sending and receiving objects which place restrictions on *outgoing* and *incoming* events, respectively. We use a *Suite*-like reconciliation mechanism based on conservative

matching. By conservative we mean that of the two versions of each processing parameter (outgoing and incoming), we pick the one that supports less sharing and, thus, come up with an *effective* coupling policy. For example, if one replica wishes a property to be shared while the other does not, the effective sharing policy does not share it. Realizing that an outgoing policy that is more liberal (sends out more events) than its incoming counterpart would lead to the communication of events that will be held at the receiver site for delivery, we perform the matching at the sending site to avoid sending such events in the first place. Hence, the receiving site does not perform filtering of incoming events but proceeds directly to install them. Performing the policy matching at the sender, as Suite does, implies that policy changes must be sent to the corresponding user(s) every time a user modifies the incoming policy. This, however, is a good trade off because policy updates are infrequent relative to object updates.

With each shared property, we associate four different parameters: *Acquisition, Transmission, Correctness,* and *Installation*. The second and third parameters generalize the semantics of corresponding *Suite* parameters to arbitrary user-interfaces, while the first and last form our extension to the model to handle existing objects and concurrent updates, respectively. Below, we describe the meanings and values of each parameter in our model, and the pros and cons of choosing different values.

Acquisition. This parameter controls the method used to obtain a replica property update. Currently, we distinguish among four different acquisition methods: *Read, Log, Effective Log,* and *Diff*, as well as the special value of *None* which indicates that property updates should not be acquired (or shared).

To illustrate the differences among the acquisition methods, consider the simple scenario of a user inserting a new section in the list of sections in the outline. The object may respond to the update in three ways:

1. It may conform to our event model and announce an update event exactly encoding the operation and its parameters. In this case, the acquisition parameter must be set to *Log* or *Effective-Log*. In the former case, each update event is logged until the processing parameters require it to be transmitted. Over time, the log of operations can become rather long if, for example, one of the users is off-line for a prolonged period of time. Therefore, log-based systems provide mechanisms for compressing the log by removing operations whose effects will be undone by subsequent operations. If such compression is desired, then the acquisition parameter must be set to *Effective Log*. We have developed a generic scheme that performs log compression for static and dynamic properties based on the operations defined on them. The disadvantage of this acquisition method is that the remote user does not see each operation invoked by the local user and may not see the rationale behind the changes.
2. It may conform, not to our event model, but instead to the more general model-view model, which only requires that a notification be sent that the object has been changed. In this case the acquisition parameter must be set to *Read* to tell the infrastructure to record the end result of the user action by obtaining the complete state of the object – in this example, the section list. While this

approach is attractively simple, it does not work well when different parts of an object are concurrently edited by multiple users and we would like to merge their work by combining the edited components (e.g. different sections).

3. It may not announce any update event. In this case, the acquisition parameter is set to *Diff* to ask the infrastructure to derive the fine-grained operations from successive snapshots of the object's state. We have developed a general property-based diffing algorithm to support this acquisition method (Roussev 2003), however, it cannot support incremental coupling.

Transmission. This parameter controls the transmission of updates based on the communication operation performed on the shared entity and has four possible values: *Increment, Complete, Scheduled,* and *Transmit*. Each individual update such as insertion of a character and the dragging of a line is an *Increment* operation. A *Complete* operation is executed whenever the user has indicated that he is "finished" editing the value, e.g., hitting `<tab>`, releasing the mouse. The semantics of this application-dependent operation is defined by synchronization events (Section 4.3). A *Scheduled* operation is triggered by timer expiration and has two parameters: execution time and a period. The operation is first triggered at the specified wall-clock time, and it is then triggered periodically. The *Transmit* operation is executed whenever the user explicitly requests it by pressing a `<Transmit>` command provided by the infrastructure. This generalizes the *send* command provided by a mail client by requiring the sender to explicitly indicate when updates must be sent. The transmission parameters correspond to the notification parameters identified in Shen (2002) for text editors and interaction parameters supported for document editing in *PREP* (Neuwirth 1994).

Correctness. One of the problems of incremental coupling is that a user's mistakes are seen by others. That, however, may be desirable in some cases. For example, a tutor may help a student with fixing semantic errors in a program. This parameter lets collaborators choose the degree of correctness of shared changes. Its possible values (in increasing order) are *Raw, Parsed, Validated,* and *Committed*. By default, any updated value is *Raw*, unless it has undergone a successful syntactic check after which it is elevated to *Parsed*. If the value has also passed a check for semantic correctness, it becomes *Validated*. The exact semantics of syntactic and semantic checks are defined by synchronization events discussed below. *Committed* values are explicitly designated by the user by executing an infrastructure-provided `<commit>` command.

Installation. Once an update is acquired and processed, it needs to be installed on the remote object. We distinguish among three different ways in which this can be achieved: *Replay, Real-time Replay, Merge*, and *None*. The *Replay* option is the simplest choice—the operations are replayed one after the other, with the user likely to have a "fast forward" experience in which minutes of collaboration are compressed into seconds. *Real-Time Replay*, replays the events at the rate at which they were executed at the transmitting site. The *Merge* installation option refers to the use of a merge procedure (Munson 1997) that integrates the update with the current version of the object. This is necessary when the simple replay of the updates is not sufficient. *None* ignores remote updates entirely.

Synchronization Events

The user command that completes a series of incremental changes (e.g. mouse button release), as well operations to parse/validate a value, are application-specific. We define standard synchronization events that allow such information to be passed along to the infrastructure in an application-independent fashion.

A synchronization event redefines the values of the transmission and correctness attributes of the property operations that are still in the buffer. Such an event is a four-tuple consisting of: **(a)** a global identifier of the object on which the operation(s) are performed; **(b)** the name of the property affected where null implies all properties of the object; **(c)** a new value for the transmission parameter (*Increment, Complete, Scheduled,* or *Transmit*); and **(d)** a new value for the correctness parameter (*Raw, Parsed, Validated,* or *Committed*). Thus, the tuple `<"Outline","section","Complete","Raw">` specifies that all property operations on the *section* property of the object named "Outline" should be relabeled as *Complete* and *Raw* unless they have higher values already. Once relabeled, all affected events must be reevaluated with respect to the current sharing policy and sent out if necessary.

Application Layer Model

Update and synchronization events provide a basis for flexible coupling but their naïve use raises some correctness issues. Suppose that the outline title is edited through a text field and both the text field and the outline object provide notifications about changes to their state to the infrastructure. An infrastructure that does not take into account the dependency between the states of the text field and the outline would produce incorrect results by replicating the notifications at both the text field and the Outline object. Thus, a character insertion by one user would be lead to a duplicate insertion on other replicas. The causally related notifications occur as a result of user actions being translated from a less abstract to a more abstract layer, with each step triggering a separate notification.

The most common solution among existing infrastructures is to provide sharing at one fixed layer (e.g., shared-window systems provide sharing at the window layer). The shared layer processes events received from remote replicas the same way it processes local events and propagates the results to upper (more abstract) layers, thereby achieving the sharing of those layers as well. This approach automatically eliminates the correctness problem but limits coupling flexibility.

To overcome this, we introduce an *XML* layer description language. An application layering definition consists of two parts: layer mapping and layer dependencies. The layer mapping is a set of tuples of the form `<class, property, layer>` specifying that the named *property* of all object instances of the given class belongs to the given *layer*. If the *property* is `NULL`, then all properties in the class are implied. If no explicit definition is given for a particular class, we

recursively lookup the definitions for the superclasses until an appropriate one is found. Once layers have been defined, the dependencies between them are specified as shown on Figure 9.

In addition to correctness, this generic layer model also gives us a high-level mechanism for sharing specification by dynamically changing the shared layer. Sharing a higher layer is a trivial task because we move from tighter sharing modes to more relaxed ones. However, the reverse process is non-trivial in the general case. Consider the following scenario: Initially, users are using asynchronous sharing of the model (i.e., they may have different versions of the shared object). If they want to switch to *view* sharing, the infrastructure must first bring the outline model versions into a consistent state using a merge procedure. To switch to *window* sharing, both the *view* and the *appearance* must be consistent beforehand. In general, to switch the sharing from a higher (more abstract) layer L_k to a lower one L_m, the infrastructure ensures that all layers that depend on L_m must be brought into consistency first.

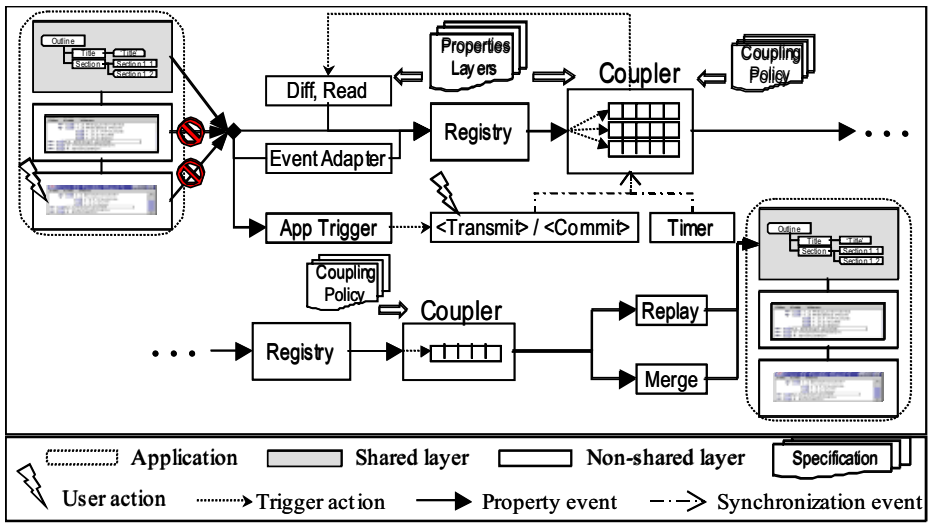

Figure 12 Event flow model of the infrastructure

Putting It All Together

Our infrastructure is not involved in communicating interaction events that go between layers but handles communication of events that are passed across layers (Figure 12). These events may be created explicitly by the programmer as update events or implicitly by a *diff* or *read* operation in response to a synchronization event. In both cases, they are delivered through a static, 'well-known' replicated object called the `Coupler`. An event passes through the registry first to register dynamic additions/deletions to the shared object structure. Next, the event is filed

with each of the outgoing queues associated with individual users, and it is then evaluated with respect to each of the corresponding policies based on user and the target object. Events that meet the minimum requirements set by the policy are immediately sent to their respective recipients using an event multicast service.

After the remote site receives the incoming event, it is processed along the same lines as outgoing events. First, the local registry and dependency tables are updated, then the local `Coupler` looks up the installation policy, selects the installation method, and applies the updates to the application object. The difference between Figure 2 and Figure 12 graphically illustrates how we have refined the abstract layer model by defining the communication between peer replicas and a generic mechanism for dynamically selecting the shared layer.

The programmers are not concerned with the details of the low-level event flow described above, which is driven by high-level code and the overhead of specifying this code is proportional to the degree of flexibility desired.

Case Study Evaluation

To understand how well we can add flexible coupling to existing complex single-user applications. One of these is *GraphDraw*, which is a *Visio*-like application provided as part of the *GEF* (Graph Editing Framework) developed at the University of California by *Jason. Robbins*. The basic goal of *GEF* is to provide a UI toolkit for the development of various applications requiring graph editing, such as circuit design, or a *Petri* net editor. *GraphDraw* is fairly simple; it consists of 9 *Java* classes that define two types of graph nodes and two types of graph edges and registers them with the framework, which handles everything else. *GEF,* consist of 171 *Java* classes totaling over 26,000 lines of code.

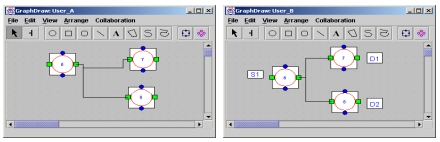

Figure 13 *Graph* layer sharing

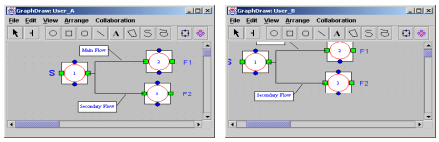

Figure 15 G*raph view* and *figure* sharing

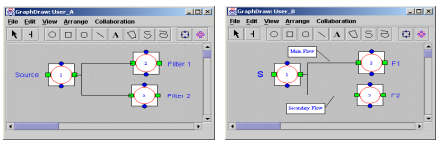

Figure 14 *Graph view* layer sharing

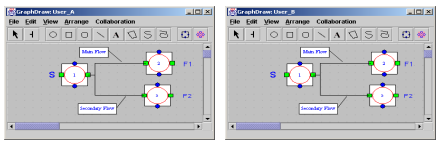

Figure 16 *Window* layer sharing

A *GEF* graph diagram consists of three basic layers—a *graph* layer, a *graph view* layer, and a *figures* layer. The graph layer represents the abstract graph (nodes and edge) being edited. The *graph view* provides the specific graphic representation through which users can manipulate the nodes and edges of graphs.

The *figures* layer consists of a number of standard shapes, such as ovals, rectangles, and text boxes that can be used to annotate the graph. We also define a *window* and an *appearance* layer much like we did for the *Outline* application.

This layer decomposition can be used to support multiple combinations of the layers. It is possible to share only the *graph* layer, that is, the graph but not its visual appearance and annotations (Figure 13). Sharing the *graph view* layer (Figure 14) leads to the sharing of the topology *and* its graphic presentation but not the annotations. To also share the annotations, we must as the infrastructure to couple both the *graph view* and *figures* layers (Figure 15). In all of the above cases, users retain the freedom to navigate autonomously and edit the graph concurrently. Sharing the window layer *window* layer (Figure 16). This is the WYSIWIS sharing supported by shared-window systems. As Table I below suggests, it was fairly easy to interface *GraphDraw* and *GEF* with our infrastructure. They employ only three basic patterns in the object structures we want to share—standard simple (*JavaBeans*) properties, as well as two versions of *set* properties very similar to the *sequence* property given earlier. The bulk of the interfacing effort was concentrated on three event adapters that translate *GEF*-defined events into property and synchronization events. All changes were linear code with a total of only four `if` and six `for` statements; almost 20% of the code changes were `import` statements.

We similarly interfaced our infrastructure with a graphical editor and the *Java Swing* toolkit. The latter experiment resulted in an application sharing system that allows the sharing of collaboration-unaware *Swing* applications. The interfacing effort in both cases was very similar to the presented *GraphDraw/GEF* case.

Action	*GraphDraw*	*GEF*		*Total*
	Modified	*Modified*	*Added*	
Total Lines of Code	26	76	102	204
Affected Classes	5	13	4	22
Code Complexity				
`import` statements	11	15	12	38
`if` statements	0	3	1	4
`for` statements	0	5	1	6

Table I Code statistics for *GraphDraw* application adaptation

These and other experiences detailed in Roussev (2003) confirm our claim that the infrastructure is high-level. In addition, as Sections 2 and 4 have shown, it includes the coupling modes of all known high-level infrastructures and supports programmer-defined user interfaces. Thus, it satisfies the requirements of providing higher coupling and user-interface flexibility.

Figure 17 shows the practical benefits of satisfying these requirements. In current systems, application sharing systems and collaboration-aware applications do not share any high-level abstractions, making the cost of implementing them high. This is probably the reason why more collaboration-aware applications such

as a spreadsheet have not been developed. Moreover, changing the coupling is a heavyweight operation requiring manually switching between two different systems. To illustrate, suppose we wish to share the *NetMeeting* (*LiveMeeting*, or *WebEx*) whiteboard in a WYSIWIS manner so that we can browse the drawing together. This requires its addition to the set of applications shared by the application-sharing system. Now suppose we wish to switch to a collaboration mode in which we can scroll independently. This requires us to (1) remove the whiteboard from the set of shared applications and (2) use its native collaboration support to re-establish the conference. If we do not take the first step, then two different systems would try to simultaneously support sharing without coordination. As Figure 17 shows, we allow application-sharing systems and collaboration-aware applications to share a common-set of high-level abstractions. Moreover, we can dynamically switch the coupling in a collaboration-aware application, with the system ensuring causally dependent events are not duplicated. Given the low cost of developing collaboration-aware applications, we expect application-sharing to be used only for closed systems that we cannot introspect or whose events we cannot intercept.

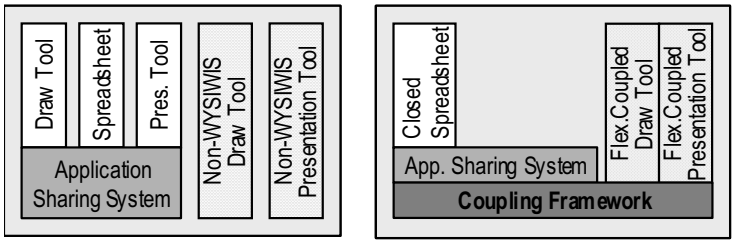

Figure 17 Coupling framework impact

Conclusions and Future Work

While there has been much effort in developing and automating general software architectures (such as MVC) for single-user interactive applications, there has been relatively less attention paid to multi-user applications. We have taken an important step to address this problem by refining/formalizing the abstract/informal layered architecture with several novel concepts for supporting all known coupling modes without making assumptions about the user-interface. These include **(1)** property-based decomposition to support fine-grained coupling, **(2)** update events, unifying method- and notification-based communication, **(3)** flexible acquisition, processing, and installation of these events, **(4)** synchronization events to support multiple degrees of update synchronization, and **(5)** layer definitions allowing users to dynamically choose shared layers.

Our preliminary experience has shown that changing complex, existing code to interface with our infrastructure requires a few mundane changes to it consisting mainly of writing event adapters. Further research is needed to verify that this

holds for a larger set of existing single-user applications/toolkits and adapt the framework in response to problems uncovered by this research. It would also be useful to automate other abstract collaboration architectures.

Acknowledgements

This research was funded in part by *Microsoft* and NSF grants ANI 0229998, EIA 03-03590, and IIS 0312328.

References

Begole, J. e. a. (1999). 'Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems'. *ACM TOCHI* **6**(2): 95-132.

Chung, G. Dewan, P. (2001). *Flexible Support for Application-Sharing Architecture*. Proceedings of the European Conference on Computer-Supported Cooperative Work (ECSCW), Bonn.

Coutaz, J. (1987). *PAC, an Object Oriented Model for Dialog Design*. Proceedings of Interact.

Dewan, P. (1998). 'Architectures for Collaborative Applications'. *Trends in Software, special issue on CSCW* **7**: 169-194.

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

Dyck, J., Gutwin, C., Subramanian, S., Fedak, C. (2004). *High-Performance Telepointers*. Proc of the ACM Conference on Computer-Supported Cooperative Work (CSCW), Chicago, IL.

Graham, T. C. N., T. Urnes, et al. (1996). *Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware*. ACM Symposium on User Interface Software and Technology.

Greenberg, S., Marwood, D. (1994). *Real-Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface*. CSCW, Chapel Hill, NC

Grundy, J. (1998). *Engineering component-based, user-configurable collaborative editing systems*. Proc of Conference on Engineering for Human-Computer Interaction (EHCI).

Krasner, G., Pope, S. (1988). 'A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80'. *JOOP* **1**(3): 26-49.

Munson, J., Dewan, P. (1997). Sync: a Java framework for mobile collaborative applications. *IEEE Computer*. **30:** 231-242.

Neuwirth, C. e. a. (1994). *Computer support for distributed collaborative writing: Defining parameters of interaction.* CSCW, Chapel Hill, NC.

Prante, T. e. a. (2002). *Developing CSCW Tools for Idea Finding - Emperical Results and Implications for Design*. CSCW, New Orleans, LA.

Roussev, V. (2003). *Flexible Sharing of Distributed Objects Based on Programming-Patterns*. Ph.D. Thesis, Department of Computer Science, Chapel Hill, Univeristy of North Carolina.

Roussev, V., Dewan, P., Jain, V. (2000). *Composable Collaboration Infrastructures based on Programming Patterns*. CSCW, Philadelphia, PA

Sarin, S., Greif, I. (1985). 'Computer-Based Real-Time Conferencing Systems'. *IEEE Computer* **18**(10): 33-49.

Shen, H., Sun, C. (2002). *Flexible Notification for Collaborative Systems*. CSCW,New Orleans,LA.

Stefik, M. e. a. (1987). 'Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings'. *Communications of ACM* **30**(1): 32-47.

Tarpin-Bernard, F., David, B.T., Primet, P. (1998). *Frameworks and Patterns for Synchronous Groupware : AMF-C Approach*. (EHCI), Heraklion, Greece.

Tse, E., Histon, J., Scott, S., Greenberg, S. (2004). *Avoiding interference: how people use spatial separation and partitioning in SDG workspaces*. CSCW, Chicago, IL.

Wang, W., Dorohonceanu, B. and Marsic, I. (1999). *Design of the DISCIPLE Synchronous Collaboration Frameworks*. IMSA, Nassau, Grand Bahamas.