A COMPONENT-BASED COLLABORATION INFRASTRUCTURE

A Dissertation

by

YI YANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2005

Major Subject: Computer Science

A COMPONENT-BASED COLLABORATION INFRASTRUCTURE

A Dissertation

by

YI YANG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Du Li
Committee Members,	Richard Furuta
	Frank Shipman
	Ronald Zellner
Head of Department,	Valerie E. Taylor

December 2005

Major Subject: Computer Science

ABSTRACT

A Component-Based Collaboration Infrastructure. (December 2005)

Yi Yang, B.E., Southeast University, Nanjing, China;

M.E., Nanjing University, Nanjing, China

Chair of Advisory Committee: Dr. Du Li

Groupware applications allow geographically distributed users to collaborate on shared tasks. However, it is widely recognized that groupware applications are expensive to build due to coordination services and group dynamics, neither of which is present in single-user applications. Previous collaboration transparency systems reuse existing single-user applications as a whole for collaborative work, often at the price of inflexible coordination. Previous collaboration awareness systems, on the other hand, provide reusable coordination services and multi-user widgets, but often with two weaknesses: (1) the multi-user widgets provided are special-purpose and limited in number, while no guidelines are provided for developing multi-user interface components in general; and (2) they often fail to reach the desired level of flexibility in coordination by tightly binding shared data and coordination services.

In this dissertation, we propose a component-based approach to developing groupware applications that addresses the above two problems. To address the first problem, we propose a shared component model for modeling data and graphic user interface(GUI) components of groupware applications. As a result, the myriad of existing single-user components can be re-purposed as shared GUI or data components. An adaptation tool is developed to assist the adaptation process.

To address the second problem, we propose a coordination service framework which systematically model the interaction between user, data, and coordination protocols. Due to the clean separation of data and control and the capability to

dynamically "glue" them together, the framework provides reusable services such as data distribution, persistence, and adaptable consistency control. The association between data and coordination services can be dynamically changed at runtime.

An **E**volvable and e**X**tensible **E**nvironment for **C**ollaboration (EXEC) is built to evaluate the proposed approach. In our experiments, we demonstrate two benefits of our approach: (1) a group of common groupware features adapted from existing single-user components are plugged in to extend the functionalities of the environment itself; and (2)coordination services can be dynamically attached to and detached from these shared components at different granules to support evolving collaboration needs.

To my parents and my family

ACKNOWLEDGMENTS

First, I would like to thank my advisor Dr. Du Li. Without his inspiration, guidance and encouragement, this dissertation would not be possible. he led me into the exciting field of computer supported cooperative work and taught me how to approach problems in a rigorous way. The methodology and philosophy that I learned in my research will definitely benefit my career for life.

I want to express my gratitude to the members of my advisory committee, Dr. Richard Furuta, Dr. Frank Shipman, and Dr. Ronald Zellner, for their valuable comments and earnest help.

I also thank the fellow students in my research group Rui Li, Jiajun Lu, and CSDL members Luis Francisco-Revilla, Unmil Karadkar, Hao-Wei Hsieh and many others for their insightful opinions in the discussion.

Finally, I would like to thank my parents and my family. I could not have gone through this long journey without their constant love and support. I owe them so much.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Groupware applications such as e-mail, instant messaging, multi-user gaming, desktop conferencing, and collaborative learning systems have been increasingly gaining popularity in recent years. They allow a geographically separated group of people to work on a common task or pursue a common goal together over a computer network [1]. They generally aim to promote the productivity of human collaboration. In general groupware must be reusable and flexible to cater for the different and evolving needs of a range of collaboration tasks. However, the development of groupware applications has long been recognized as a challenging task due to a number of subtly interacting social and technical issues [2].

To reduce the costs of groupware engineering, a plethora of collaboration infrastructures have been developed over the past two decades. These approaches largely fall into two categories: collaboration transparency and collaboration awareness [3].

A. Collaboration Transparency

Collaboration Transparency aims to re-purpose existing single-user applications for cooperative work without modifying their source code, such as [3, 4, 5, 6]. The philosophy behind is simple - since there have been many popular single-user applications which are used by people in their daily activities, why not build a runtime system enabling them to do the collaborative work? The benefit is that there would be no need to re-build many futures that have already been built in single-user applications. Additionally, users do not need to learn how to use new applications. A typical ex-

---

The journal model is *IEEE Transactions on Automatic Control.*

ample of collaboration transparency system is Microsoft NetMeeting. Once a user starts a collaboration session in NetMeeting, he/she can invite other people to join this session. Then users can share specific applications or their desktops (the screen) to other users. Usually users take turns to operate on shared applications.

Some usability problems are quickly identified in collaboration transparency systems [7, 4]. First, They are rather rigid in supporting concurrent work because at any moment only one user can manipulate the shared application. This limitation effectively excludes concurrent work. Second, a related problem is that it only supports what-you-is-what-I-see(WYSIWIS) [7] kind of collaboration which forces the collaborators to see exactly the same view of an application. Third, their performance is generally poor especially in a wide-area network due to image broadcasting used for viewing sharing.

Notably, even with these problems, collaboration transparency systems are still useful for sporadic collaboration needs when specialized collaborative applications are not available. For example, the remote assistance function in Windows XP allows the system administrators to remotely assist users in configuring or diagnosing system problems.

B. Collaboration Awareness

Special-purpose groupware applications are usually built to address the limitations in early collaboration transparency systems. Since these applications are built with the intention of supporting collaboration, they are called collaboration-aware systems or simply collaboration awareness. Collaboration-aware systems focus on providing reusable coordination services (e.g., access control and concurrency control) and multi-user interface widgets to ease the development of special-purpose groupware ap-

plications, such as [8, 9, 10, 11]. Systems taking this approach can achieve improved performance and flexibility. For example, they allow collaborators to simultaneously work on different parts of a shared application. This is called relaxed-WYSIWIS [7].

However, reusability and flexibility in previous collaboration-aware systems have not reached desired levels, for the following reasons: First, they usually require the developers to follow custom programming abstractions to access the reusable coordination services provided in the infrastructures. They generally do not address the large base of third-party programs that fail to follow their programming abstractions. Second, the coordination services are often tightly bound with data objects they control and cannot be reused in many applications without refactoring.

C. A Summary of This Research and Contributions

From above, we can see that much progress can still be made towards achieving more flexibility and reusability in collaborative systems. Our research hypothesis is that flexibility and reusability are not necessarily competing goals that compromise each other. Our objective is then to investigate an alternative approach to building collaborative systems which can meet desired flexibility in supporting collaboration while reasonably reusing previous development effort without forcing developers to discard the standard practice.

In this dissertation, we propose a component-based approach to developing groupware applications that addresses the above reusability and flexibility issues of previous approaches. To address the reusability problem, we propose a shared component model for modeling data and graphic user interface(GUI) components of groupware applications. Since this model only requires that the components conform to industrial component standards such as JavaBean and .NET component, the myriad of

existing single-user components can be re-purposed as shared GUI or data components. An important implication of this is that multi-user widgets can be built almost as simply as their single user components. An adaptation tool is built to assist the adaptation process.

To address the flexibility problem, we cleanly separate the shared data, consistency protocols, and systematically model users' interaction with them in our coordination framework. A meta-service is provided in our coordination infrastructure to dynamically glue data and control components at runtime. Users can dynamically switch collaboration protocols in order to support evolving needs for coordination. As a byproduct of the data-control separation, the coordination services can also be reused.

An **E**volvable and e**X**tensible **E**nvironment for **C**ollaboration (EXEC) is built to evaluate the proposed approach. Newly adapted shared components can be incrementally plugged in to extend the functionalities of the environment itself. Coordination services can be dynamically attached to and detached from the shared components to support evolving collaboration needs.

D. Organization of Dissertation

The rest of dissertation is organized as follows: Chapter II introduces important concepts, techniques and principles related to building groupware applications and lays a research foundation for this dissertation. Chapter III introduces the share component model, its Java embodiment, and a component adaptation tool which converts existing single-user components into shared components. Chapter IV introduces a coordination services framework. For the scope of this dissertation, we focus on adaptable consistency control while briefly overviewing others coordination services.

After that, in Chapter V, we empirically evaluate our contributions on the EXEC platform by demonstrating the reuse of existing single-user components and the flexible application of coordination services. In the end, in Chapter VI, we summarize contributions of this dissertation and point out possible future research directions.

CHAPTER II

BACKGROUND

In this chapter, we explain important concepts, technologies, and principles related to building groupware applications. This lays the foundation of our own work.

A. A Groupware Taxonomy

There are many different groupware applications enabling people to collaborate with each other to finish tasks with different nature. Even though there is no common agreement on the definition of "groupware", people tend to agree that groupware applications support teams or a group of people work together towards a common goal. Based on whether the collaboration happens at the same time and whether at the same physical space, groupware applications can be largely categorized into four kinds [12], as shown in table I.

Table I. Groupware taxonomy

Place and Time	Same	Different
Same	Single-Display Collaboration Conventional Gaming	Work-Shift
Different	Collaborative Writing, Networked Gaming, Instance Messaging Team-Room	E-mail, Bulletin Board Workflow

If each user's actions are expected to be seen and responded by other collaborators close to their initiating time, the collaboration is considered as **synchronous** or **real-time**. Otherwise, the collaboration is considered as **asynchronous**. Notably,

this definition is only conceptual and there is no hard line between synchronous and asynchronous collaboration. For the same groupware application, both technical and non-technical factors could affect the delivery of remote user actions on local machine, which in turn affects the collaborators' response. For example, e-mail is usually considered as groupware supporting asynchronous collaboration. However, frequent exchanging messages between collaborators can certainly increase the degree of synchrony. Our focus in this dissertation is the collaboration happening in real time at different places. However, we need to point out that applications built with our infrastructure can support both synchronous and asynchronous styles of collaboration, depending on the nature of the collaborative task.

B. Overview of Collaborative Systems

In this section, we overview collaborative systems in two general categories: collaboration transparency and collaboration awareness.

1. Collaboration Transparency Systems

Collaboration transparency is also called application sharing due to the nature of this approach. Collaboration transparency systems are runtime systems that enable existing single-user applications to be collaborative. They generally adopt either *centralized* or *replicated* architectures.

Centralized application sharing systems, such as XTV [13] and NetMeeting [6], execute a shared single-user application on a server. The main advantage is that the infrastructure is generally reusable for sharing arbitrary single-user applications. However, the following disadvantages exist: As the single-user application is usually not designed to process multiple concurrent input streams, the users must take turns

to provide input to the application, which limits concurrent work and is often counter-productive. The application's graphics output is multicasted to all collaborating sites for display, which often generates considerable network traffic. When the interaction is not local, the response time is sensitive to networking delays. Moreover, output broadcasting makes it only possible to support a strict what-you-see-is-what-I-see (WYSIWIS) type of collaboration [7].

Typical replicated application sharing systems, such as Dialogo [3] and Disciple [5], execute a copy of the shared single-user application at all sites. Each user interacts with the local replica directly, which implies improved response time and reduced network traffic compared to centralized application sharing. Only the input is duplicated for synchronization. However, users generally still have to take turns to input locally and see exactly the same view.

As documented in [3], the following problems make it difficult for replicated application sharing systems to be as generic as their centralized counterparts: First, replicated application sharing systems generally synchronize application replicas by replaying input events at all sites. This essentially assumes that the shared applications are deterministic, i.e., they must always generate the same output in response to the same input. Unfortunately, many single-user applications are not deterministic because of time-dependent behavior.

Second, the shared applications usually need to access external resources, such as disk files, databases, system clocks, network sockets, environment variables, the window manager, and other processes. To maintain consistency among application replicas, the sharing infrastructure must be able to manage these external resources, which generally cannot be achieved without a redesign of the operating systems.

Several recent replicated application sharing systems have been developed to address the above problems by taking domain-specific approaches, such as Flexible

JAMM [4], ICT [14], CoWord [15], and Zipper [16]. However, in general flexibility is achieved at the loss of generality (or reusability) of the infrastructure.

Flexible JAMM [4] exploits the component dynamic loading mechanism in Java and is able to replace some components in the shared Java application with custom versions at run time. This achieves two important features: First, the application's accesses to external resources can be managed by the infrastructure by dynamically adding custom resource proxies. Second, some user interface components can be replaced with multi-user versions to implement relaxed-WYSIWIS and allow for more concurrent work. However, the extra flexibility is only achievable on a subset of Java Swing-based applications.

ICT [14, 17] allows the users to share heterogeneous single-user applications for cooperative work. However, the infrastructure needs application-specific knowledge to translate the input events to abstract operations in order to interoperate the shared applications. While the application knowledge is generally difficult to acquire, the problem can be mitigated in specific domains. For example, when it is known that the shared applications are single-user editors, diffing can be used to derive the editing operations, which eliminates the need to translate window events. As a result, the users' views and inputs do not need to be constrained for synchronization purposes as in early systems.

CoWord [15] aims to share productivity tools such as Microsoft Word and PowerPoint. Similar to ICT, it can also achieve unconstrained interaction by understanding and translating window events. The translation is aided by the APIs provided in the shared applications but still labor intensive and must be done on a per-application basis. Consequently the cost of adapting (reusing) single-user applications to achieve the desired flexibility is high.

Zipper [16] explores aspect-oriented programming techniques to adapt single-

user applications for cooperative work. Relaxed WYSIWIS and flexible control are achieved as a result. However, it assumes the availability of source code and requires the developers to manually find out the right places in the original programs where new code that implements the advanced features can be correctly injected. Hence the cost of reuse is also high.

Roussev et al. [18] transparently share JavaBean components such that coordination services can be applied externally. The components are assumed to follow an extended naming convention such that their logical structures can be introspected. The runtime infrastructure maintains a copy of the logic structure of each shared component and uses diffing to derive the state changes before they can be applied on the actual components. However, it does not address how to share components that fail to follow the extended naming convention. In addition, it admittedly fails to provide sufficient performance for synchronous collaboration.

2. Collaboration Awareness Systems

Application sharing is useful in adapting (reusing) familiar single-user applications for cooperative work to save engineering and learning costs. However, it does not eliminate the needs for developing specialized groupware: First, existing application sharing systems are often either inflexible or domain-specific. Second, many collaborative tasks require specialized user interfaces that are often awkward, if not impossible, to implement in collaboration transparency.

To address the flexibility limits of early application sharing systems, many researchers turned to collaboration awareness, which effectively lowers the ambition of reusing existing applications as a whole to reusing libraries of coordination services [19]. As a result, a large number of groupware toolkits have been developed, such as Suite [8, 20], DistView [21], GroupKit [11], Corona [10], and JView [9]. These toolkits

usually provide reusable functions such as multi-user interface widgets [8, 11], group communication [11, 10], concurrency control [20, 21], and bundled services [9].

These pioneering toolkits were designed when there lacked commonly accepted software development practices. Consequently the following limitations are retrospected: First, they generally define system-specific programming abstractions for accessing the provided coordination services, such as the predefined data types in [20] with embedded locking protocols. Apparently the intended reusability is undermined if their *custom* abstractions are not followed by the developers.

Second, their coordination services (see next section for detail) are generally tightly bound with the programming abstractions, as in [20, 9]. Due to the lack of a clean separation between data and control, the toolkit often has to be redesigned if the functions need to be revised or extended, e.g., for different groupware applications.

Third, little has been done in previous toolkits to adapt (reuse) third-party programs that do not follow the expected programming patterns. Hence much redundant effort is still required in developing groupware application despite the growing base of available single-user programs.

C. Groupware Architecture

In general, there are three architecture choices for building a groupware application - centralized architecture, replicated architecture, and hybrid architecture [3]. Different architectures imply different pros and cons in the resulted groupware applications. So it is important to understand the tradeoffs of different architectures. In the following, we use collaboration-transparency systems as an example to compare the tradeoffs between different architectures. Notably, these tradeoffs generally apply in collaboration-aware systems.

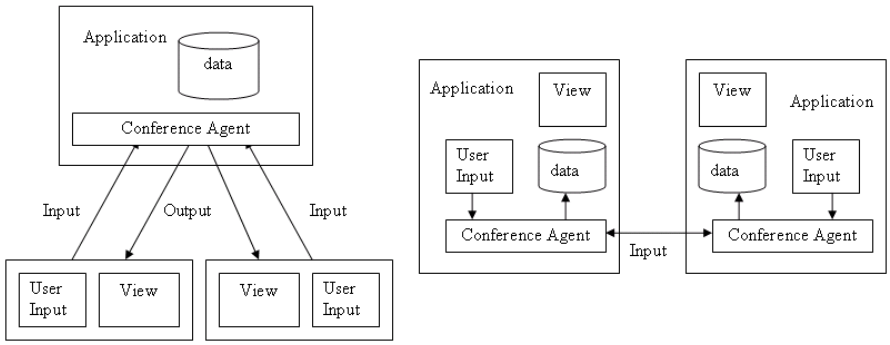

Fig. 1. Centralized and replicated architecture

1. Centralized Architecture

In a typical centralized architecture, the application or data resides in a centralized server, as shown on left side of Figure 1. On the clients, collaborators share the output of the application. A conference agent usually resides on the server in order to merge user inputs and broadcast application outputs to the views on different clients.

The strength of centralized architecture is its simplicity. Since all user inputs are first sent to the conference agent on the server, the server can naturally act as a rendezvous to serialize the concurrent user inputs. Coordination such as consistency control becomes very simple. Since there is only one copy of the shared data located on the server, it is impossible for shared data to diverge. Since all users receive the same output of the application, naturally all users share the same view of the application.

However, the price for this simplicity is degraded system performance and flexibility of collaboration. First, user inputs will have to be sent across the networks before being computed by the application. The output will again be sent across the network back to the clients for display. The response to local user action can be slow.

Second, since application output is usually graphic image and will be broadcast to different collaboration sites, it can consume a lot of network bandwidth. Third, since all users have exactly the same view of an application, it becomes difficult for different users to work on different areas of an application, which forces the users to scroll the view port up and down in order to work on their own areas. Hence only one user can work on the application at one time, which effectively reduces the concurrency of the collaborative work.

2. Replicated Architecture

On the right hand side of Figure 1, we show an example of fully replicated architecture with only two collaboration sites. In this architecture, all components of an application are replicated among all sites, including the conference agents. User inputs, instead of being sent out for computation, can be computed locally. Thus the view can be updated quickly without being affected by network delays. The conference agent only sends out the local inputs and receives remote inputs and there is no need to broadcast the application outputs anymore. Thus the bandwidth consumption in replicated architectures is lower than centralized architectures. In addition, since different users do not share the view image directly, it is possible for them to work on different area of the application independently. For example, an replicated group editor can choose not to synchronize the user actions that trigger coordinates changes of application's viewport, e.g., actions that scroll up and down the editing window. Then people can work on different portions of the document without interfering with each other, achieving improved concurrency in collaborative editing.

A replicated architecture has its own drawbacks. One of the main challenges is to maintain the consistency of application states. Since the shared data are replicated, any change to any copy of the shared data will have to be applied at all sites in a

certain order. Many different consistency protocols have been devised in the groupware domain for this purpose, as we will detail in Chapter IV. Another challenge for replicated architecture is how to support new comers in an on-going conference. This needs all active users to agree on an up-to-date image of the shared data and then some logging and process migration mechanism needs to be used in order to let the new comer catch up with the latest state of the application.

3. Hybrid Architecture

As pointed out in [22], it is generally rare for groupware application to have a pure centralized architecture or a fully replicated architecture. For centralized architecture, groupware applications might choose to replicate some parts of the functionalities to local site in order to improve the performance. On the other hand, in replicated architecture, to support long lasting collaboration, a centralized server can be chosen to store the collaboration data so that different users can leave and re-join the collaboration session anytime they want and still be able to carry on the collaboration. Traffic-wise, a centralized server can serve as relay-point for broadcasting user inputs to remote sites, thus reduce the number of communication links from between the sites and increase the scalability of the system. The downside of this is the delivery time for messages will increase due to message relay.

4. Summary of Tradeoffs of Architectures

From above, we can see different architectures have their advantages and disadvantages for the resulted groupware applications. Notably sometimes the choice of architecture is also related to the hardware . For example, for a given client with low computation power, e.g. PDA, the groupware application designers might consider avoiding replicating computation intensive components on this device. Thus

the architecture choice for a specific collaborative application is really related to the collaborative task and available hardware resources. In this dissertation, we choose a hybrid architecture to to gain the benefits of both centralized and replicated architectures. On one hand, we replicate the data and coordination services at collaboration sites for fast local response. On the other hand, a centralized server serves as the place for persisting shared data and relay messages, which simplifies the design of the system itself.

D. Coordination Services

An important difference between the single-user applications and groupware applications is the need for coordination services. Coordination services can have many aspects and a common decomposition is communication, consistency control, access control and awareness control [23].

1. Communication

In order to coordinate multi-user activities, communication is a must. Other coordination services are based on communication service . In early systems, groupware developers build their communication services on top of the TCP/IP protocol stack, e.g., the group multi-casting service provided in GroupKit [11]. Later, more advanced communication capabilities are provided by different platforms, e.g. Sun RPC, Java RMI, .NET remoting, CORBA, and XML based RPC etc. These new communication capabilities enable developers to build other coordination services more easily without dealing with the error-prone details of encoding and decoding message protocols.

2. Consistency Control

Consistence control is an important coordination service in groupware. Its basic function is to ensure the consistency of the shared data. Since different users operate individually, their actions have to be "sorted out" in some way to prevent the divergence of shared data. The most common way of consistency control is to use lock. Before the shared data is modified, a lock is first applied to exclude other users' actions on the shared data. The locking scheme is straightforward, but it sacrifices concurrency of collaborative work. More advanced consistency control such as operation transformation (OT) [24] and its variations have been devised to support higher degree of concurrent work while still maintaining consistency of shared data. We will give more detailed discussion of consistency control in chapter IV.

3. Awareness Control

In general, awareness information provides the context for collaborative work, which is critical for coordinating user activities [25]. It answers the question of "who is doing what at where?" It then can be divided into several basic questions as "who is collaborating, what they are doing, and where they are working" [26]. Presence awareness is used to indicate whether an user is present in the collaborative environment or not. Location awareness indicates where the user is working at in the shared environment. User state indicates current state of user, e.g. "idle","available", or "busy" etc in the collaborative task. Many different awareness gadgets have been created to answer one or more aspects of awareness, e.g. radar view, multiple-user scrollbar, online user list, telepointer.

4. Access Control

Access control answers the question of "who can access what question" in a multi-user environment. The canonical discretion access control model used in file systems is also brought in groupware applications. A matrix of (subject, object, permissions) is used to describe the access rules for different users to different objects. Dewan and Shen [27] extends this basic model by introducing access right inheritance, the viewing right of interface object and negative right in accessing Suite active variable.

5. Session Control

Session control determines if a user can join a collaborative session and what role(s) he will take in the session. Session control can be explicit or implicit, depending on whether or not there is an explicit notion of sessions. According to Edwards [28, 29], session control must be so flexible that collaborators can dynamically join and leave sessions, and change their roles in a session.

E. Component Technologies

Collaboration transparency systems focus on the reuse of existing applications. This can be considered as the coarsest granule of reuse of previous development effort. Unfortunately, collaboration transparency system can not address how to achieve reusability in developing new groupware applications. On the other hand, reuse, as one of the most important qualities of software, has been extensively studied by researchers in the domain of software engineering. Component-based development (CBD) has been considered as the latest break-through in building reusable systems. The popularity of different component models in industry has demonstrated the attractions and power of CBD. Further more, modern object-oriented languages

such as Java and .Net provide direct support on component-based development at the language level. Component-based development provides new opportunities both groupware developers.

Component technologies can be categorized into two kinds - the local component model [30] and server component model [31]. Popular local component models include Microsoft COM (Component object Model), and ActiveX control(Based on COM), Borland Delphi VCX(Based on ActiveX), and Sun JavaBean. Server component models, also called distributed component technologies, include Distributed COM or(COM+), CORBA component model, and Enterprise JavaBeans.

Both component models emphasize component composition and replacement using well-defined interfaces. The idea is that the development process of an application could be accelerated by purchasing third-party components and integrating them into the applications. Additionally better versions of these components could replace the old ones as long as the interfaces do not change. Local component models focus on building user interface widgets. Server component models emphasize enabling the communication among objects residing on different networked machines. Usually a middleware infrastructure is provided to enable remote object invocation transparently. Besides this basic communication capability, additional services are usually built atop, e.g. data transaction service, naming and directory services, and security auditing.

Our focus in this dissertation is client component model, which mostly contribute to the construction of graphic user interfaces of applications. Traditionally these components are developed for single-user applications. There are subtle implications when they work with coordination services, as we will detail in the next chapter.

F. Summary

In this chapter we overview different groupware architectures and different approaches to developing collaborative systems: collaboration transparency and collaboration awareness. Then we introduce the important coordination services which differentiate groupware applications from single-user applications. In the end, we introduce component technologies which aim to reuse previous development effort.

Collaboration transparency has the main advantage of reusability. Collaboration awareness, on the hand, can achieve better performance and flexibility. Our objective is to develop a novel approach which can combine the merits of these two approaches. On one hand, instead of reusing single-user applications as a whole , we try to reuse the single-user components in developing new groupware applications. On the other hand, to achieve the improved flexibility of coordination, the data and coordination functions will be separated and dynamically coupled in our collaboration infrastructure. The immediate next two chapters address these two directions, respectively.

CHAPTER III

SHARED COMPONENT AND ADAPTATION

A. Introduction

To address the flexibility and reusability limitations summarized in the previous chapter, we propose a novel approach that combines the merits of collaboration transparency and collaboration awareness. Our infrastructure provides reusable coordination services (e.g., consistency control) as well as an adaptation tool for reusing third-party single-user components. This is achieved by defining a clean interface between data and control components and providing a runtime system to dynamically "glue" them together. Our programming abstractions follow a well-established industry component standard, or more specifically, JavaBean. Hence a large and growing base of components can be reused for developing flexible special-purpose groupware applications with no or only minor adaptation effort.

Figure 2 shows our abstract model of groupware applications: A groupware application mainly consists of (graphic) user interfaces, shared data objects, and coordination services such as access control and concurrency control. Conceptually all these parts are replicated for responsiveness and availability reasons. Data objects and coordination services are assumed to follow well-defined interfaces. Flexibility is mainly exemplified by allowing the users to dynamically attaching coordination services to different data objects in the same workspace.

The shared component model, its runtime system, the adaptation tool, and a library of coordination services have been implemented. The runtime system and adaptable coordination services will be presented in the next chapter. This chapter presents the shared component model and the adaptation tool.

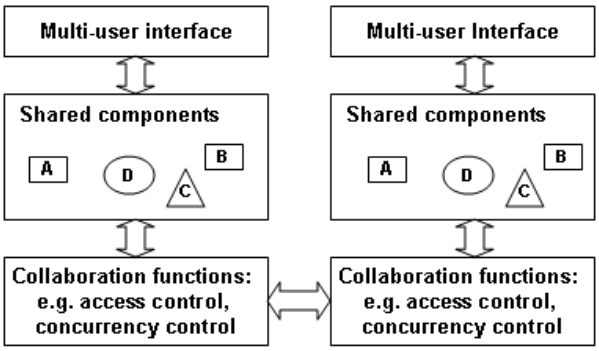

Fig. 2. Separating data objects and coordination services to promote reusability.

B. Shared Component Model

We first motivate and define the shared component model and its JavaBean embodiment, and then summarize the runtime system. In the next section, we discuss how third-party components can be adapted and shared.

1. Modeling Shared Data

First, shared data in a multi-user environment will have to be controlled by different coordination services to maintain their security and consistency. Coordination services, e.g., access control and concurrency control, answer questions such as whether or not specific property changes can happen on a data object and how to apply the changes to other replicas of the same object. To enable flexible sharing of data objects, the collaboration infrastructure must be able to allow for a range of control policies with different levels of optimism. To achieve so, we need to intercept property changes before they actually take effects on the data objects.

Secondly, for performance and flexibility reasons, different types of property changes often have to be distinguished. Even though all property changes can be

modeled as replacing an old value with a new one, some property changes that are incremental by nature should be modeled differently, such as the increase and decrease of texts.

Thirdly, data objects often have structures. A composite object may recursively contain many other objects. Objects have different structures and applications can have their own way of expressing the relationships between objects. However, the way of object composition must be standardized for coordination services that access and control the data objects to be reusable.

Last but not least, the infrastructure must be able to globally identify different replicas of the same object. Eventually property changes at one site must be applied on all data replicas at other sites to maintain consistency.

To summarize, a shared data model and its component embodiment must address the following requirements:

1. A shared data component instance must identify itself using a global unique id. This id is assigned when the instance is created and it is immutable afterwards.

2. A shared component must provide a well-known interception point for any of its shared property changes. By hooking up to this point, coordination services can intercept property changes before they take effects.

3. A shared component must define a well-known way for the coordination services to apply desired property changes.

4. Shared components that have composite structures must provide a well-known way for coordination services to access subordinate components.

5. A shared component is encouraged to provide a well-known way of applying shared property changes atomically as well as incrementally, if it contains shared

properties that are incrementally changeable.

Currently no client component models mentioned in previous chapter could accommodate these requirements. This is natural since these components are supposed build the single-user application. Our objective is then to patch existing component model to accommodate our requirements. In the following, we address these requirements in a specific component model. In the scope of this dissertation, we use JavaBean component model as our testing component model, even though the same techniques can be used on other client component model such has .NET controls.

2. Java Embodiment

In this subsection, we give Java interface definitions for our share component model. It includes a shared component interface and a shared container interface, inside which methods signatures differentiate atomic and accumulative property changes. The reason to define interfaces instead of default class implementation is because Java only allows single inheritance in sub-classing. Using interfaces allows more flexibility when the user wants to adapt existing components as shared components. Nevertheless, we include a default implementation of shared components, called *DefaultSharedComponent*, which extends the JDK *JComponent* class and implements the *ISharedComponent* interface. It provides a starting point for developers to build fresh new shared components.

a. Shared Component

Figure 3 gives a Java specification of the shared component interface. The methods defined in this interface fall into three groups. The first is a method, *getOid*(), that returns the global unique id of the shared component. By this method, each shared

```
public interface ISharedComponent {
    //(1) to return the global unique id of shared component
    public String getOid();

    //(2) to modify shared properties
    public void insert(String propertyName,
            int offset, Object value);
    public void delete(String propertyName, int offset);
    public void update(String propertyName,
            Object oldVlaue, Object newValue);

    //(3) to hook coordination services
    public void addSharedPropertyChangeListener(
            ISharedPropertyChangeListener p);
    public void removeSharedPropertyChangeListener(
            ISharedPropertyChangeListener p);
    public void fireSharedPropertyChange(
            SharedPropertyChangeEvent e);
}
```

Fig. 3. Shared component interface

component instance identifies itself globally. The second group of methods is used to insert, delete, and update shared properties of this component. They are abstract operations on shared properties and indirectly define the shared properties. The third group contains three methods: the first two are for the runtime system to hook up coordination services with the shared component to intercept its property changes. The third method is used for happen-before notification of shared property changes.

The *SharedPropertyChangeEvent* class wraps up three event types and their corresponding parameters into one common class definition. Whenever an application invokes insert, delete, or update method of a shared component, a corresponding *SharedPropertyChange* event will be fired out from this shared component by its invoking of the *fireSharedPropertyChange* method. This method iterates all registered *ISharedPropertyChangeListener* instances and then invokes the well-known notifica-

tion method defined by the *ISharedPropertyChangeListener* interface. The runtime is itself a *ISharedPropertyChangeListener* instance which is registered to be listener of all shared component instances. Whenever a *SharedPropertyChange* event is fired, the runtime will be notified before the property change takes effects on the shared component. Then the runtime delivers the event to corresponding coordination services.

b. Shared Properties

We distinguish two types of shared properties. The first type is called **atomic** property. The value of an atomic property is only dependent on the last operation on this property. For example, the foreground color of a circle only depends on the last *setColor* operation on this circle. Apparently, it is enough to use method *update* in Figure 3 to characterize the value changes of an atomic property. All JavaBean properties can be treated as atomic properties.

The second type is called **accumulative** property. The current value of an accumulative property may depend on not only the last operation, but also all the other operations in its operation history. For example, in a text component, its content can be changed by characterwise insert and delete operations. The final content depends on all the insertions and deletions that have been executed on the component.

Atomic and accumulative properties are directly mapped to the atom and indexed properties, respectively, in JavaBean and .Net. In this sense our shared property model does not lose any expressive power of the original host component model (JavaBean or .NET). While all properties can be treated as atomic properties, distinguishing some of them as accumulative properties can sometimes implement more fine-grained control or achieve better system performance.

```
public interface ISharedContainer extends ISharedComponent {
    public void insertChildren(int offset,
                               ISharedComponent child);
    public void deleteChildren(int x);
    public ISharedComponent getChildren(int x);
}
```

Fig. 4. Shared container interface

c. Shared Container

Figure 4 specifies a shared container interface for modeling data objects that have composite structures. A shared container is also a shared component. Hence it extends the *ISharedComponent* interface. Additionally it defines three methods for retrieving, inserting and deleting subordinate components. By this definition, coordination services such as concurrency control can be applied on more efficiently based on the knowledge of the logical structure of the shared component. For example, as in databases, when the component hierarchy is known, locking can be applied on specific components as well as branches of the tree structure.

Note the pair of methods, *insertChildren* and *deleteChildren*, effectively define a shared property called "Children" which is an accummulative property. They directly correspond to the *insert* and *delete* methods in the *ISharedComponent* interface. The other methods in the *ISharedComponent* interface are inherited.

3. Runtime System

Implementing the shared component interface does not automatically give Java components the capability of being shared. Sharing these components and their property changes relies on the additional support from the environment that these components live in. This environment forms the sharing context and provides additional runtime

support. Figure 5 shows a simplified view of the collaboration infrastructure that we developed for sharing components. For the purposes of this chapter, we only show the most relevant modules which support the identified requirements of the shared component model.

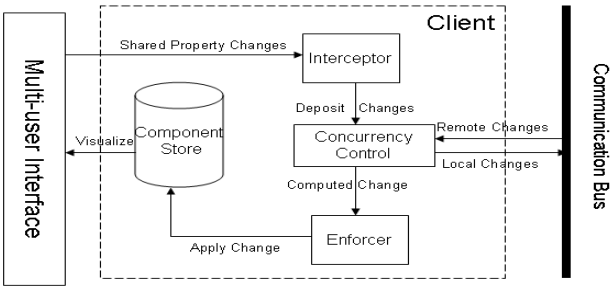

Fig. 5. The EFG runtime system.

The runtime system takes a client/server architecture. The server maintains persistent shared components and performs session control. It contains three modules: the component store, concurrency control, and the property change enforcer. In addition to these three modules, the client has an additional module called the property change interceptor. All clients in the same session communicate with each other and the server through a conceptual communication bus. The component store at each site maintains a copy of all shared data components. Each component instance is assigned a globally unique id when it is instantiated. The concurrent control modules at all sites together decide how to apply property changes on all data replicas.

The property change interceptor and enforcer modules are the most relevant to this chapter. The interceptor only presents on the client, because the server does not interact with users directly to trigger property changes. The interceptor registers itself as the shared property change listeners of all shared components. Whenever the application triggers shared property changes, the interceptor will be notified first.

Then the property changes are pushed into the concurrency control module for necessary computation. The computed property changes are then pushed into the enforcer module to be applied on the shared components. Because the property changes are intercepted before taking effects, both pessimistic and optimistic concurrency control can be implemented.

The shared property change event is a tuple of (*OID*, *PropertyName*, *PropertyChangeType*,*Parameters*). Based on *OID*, the enforcer locates the shared component instance. Based on *PropertyName* and *PropertyChangeType*, the enforcer decides the execution method signature for changing the property. Based on the execution method signature, the enforcer converts the generic object types of *Parameters* into the specific parameter types required by the execution method. Finally the enforcer invokes this execution method on this component property dynamically. The whole process requires explicit support of introspection and dynamical invocation. The enforcer also requires that the shared components follow the standard JavaBean naming conventions.

4. Summary

The shared component model and the runtime system together fulfill the requirements identified in Section 1. Once a Java component conforms to the shared component model, it can be used for developing groupware applications within our runtime system. Due to the clean separation between data and control, the runtime is table-driven: The association of data and control components is stored in a table. As a result, different control protocols can be dynamically associated with different data objects in the same workspace, and the same object can be dynamically associated with different control protocols over time. This level of flexibility has never been achieved previously in other collaborative systems [32].

Notably the mechanisms required to implement the shared component model, e.g., introspection and dynamical method invocation, are widely available in modern industry component technologies such as JavaBean and .Net. Hence although our shared component model has only been prototyped in Java, the same results can be achieved on other platforms as well.

C. Component Adaptation

There have already been a number of client component technologies, e.g., ActiveX control (based on COM) and .Net control on Microsoft platforms and JavaBean on the Java platform. Vendors of these component technologies themselves as well as third parties provide an ever-growing base of reusable components for developing applications. Unfortunately, those components generally cannot be used directly as shared components in collaborative systems. The key requirements for sharable components, as discussed in Section B.1, are generally not satisfied in those components. Thus adaptation is necessary in order to reuse them for developing groupware applications. If an adaptation tool is available for converting them into shared components, the myriad of existing and emerging components can be reused for developing groupware applications with little effort.

1. Adaptation Tool Design

We developed an adaptation tool for converting components that follow the standard JavaBean naming conventions into components that additionally conform to our shared component model. It is worth noting that the tool does not need the component source code. Instead, it generates a subclass of the original component directly from its byte code. The subclass implements the *ISharedComponent* interface. By

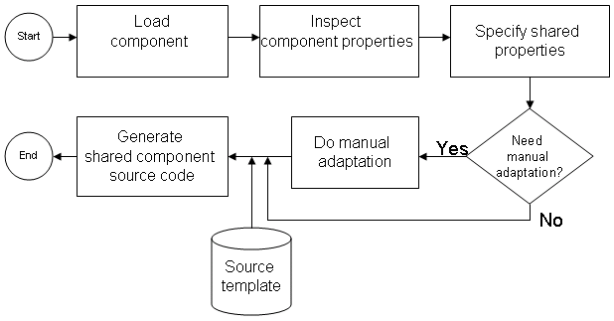

Fig. 6. The component adaptation process

default, all component properties are declared as shared properties. However, this does not always make sense. For example, the background color of a shared text component might be shared in some applications. But it might become a personal preference that should not be shared in some other applications. Hence the adaptation tool should allow the users to decide which properties are to be shared.

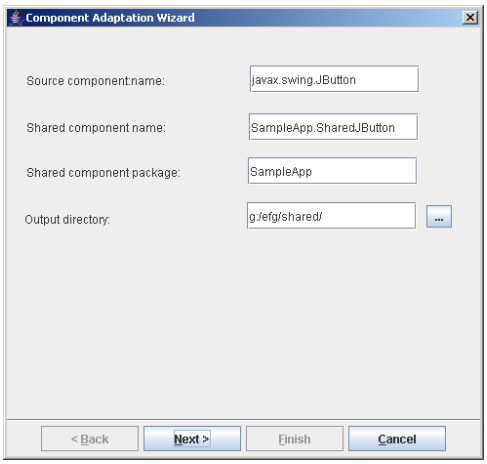

Fig. 7. Interface for setting adaptation parameters

Component adaptation generally goes through the process as shown in Figure 6. The adaptation tool implements a graphical user interface to provide guidance at each

step. The user (developer) is first prompted to provide name of the source component and the name, package, and output directory of the target component, as shown in figure 7.

Next, properties of the source component are introspected and presented to the user. Then the user selects the set of properties to be shared. The default option is to share all properties of the component. Manual adaptation is allowed to achieve more flexibility, as shown in figure 8

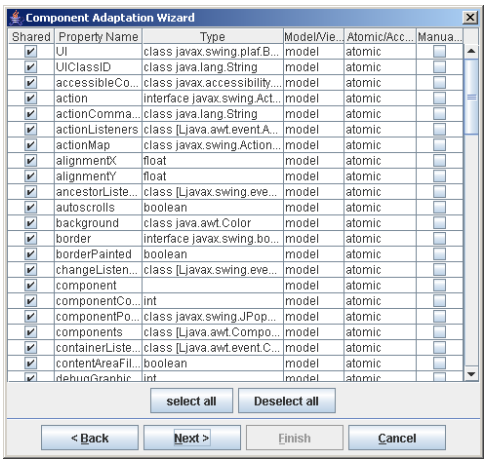

Fig. 8. Interface for selecting shared properties and types

In the end, source code of the target shared component is generated and user can add additional code and do the compilation to check if there is any error, and then output to the specified directory, as shown in figure 9.

Code to implement the shared component interface is actually very simple. Because Java does not allow for multiple inheritance, however, we have to provide the implementation source code in templates. As shown in Figure 10, these templates are used in the final step in the adaptation process. Specifically, there are four templates, namely, the template *SharedComponent*, *SharedContainer* and their corresponding

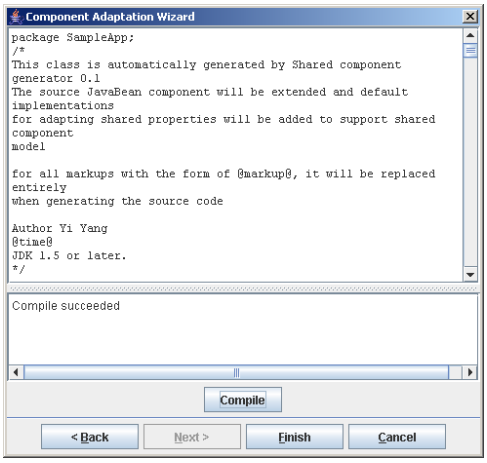

Fig. 9. Editing and compiling generated shared component code

BeanInfo classes. A *BeanInfo* class is used to describe the set of shared properties of a corresponding shared component, which is the convention in JavaBean.

As an example, Figure 10 shows the template source code of the shared component class. The *listeners* vector is used to store the registered *IShareProperty-ChangeListener* instances. As shown in Figure 5, it only includes the runtime, or more specifically, its property change interceptor module in our current implementation. Variable *oid* is the global unique id of a shared component instance. Both of *oid* and *listers* will be initialized when the shared component is instantiated. The default constructor invokes *getUUID()*, which is provided in a *Utility* class in the the *Framework* package to generate the global id.

The template contains special markups in the form of @*markup*@. Each of these markups will be replaced by the adaptation tool using the actual values when the target component source code is generated. These markups are configured in the beginning of the adaptation process, including the package name, shared component name, and base component name. The newly generated shared component class

```
@packagename@

import framework.*;
import java.util.Vector;
import @BaseComponentName@;

public class @SharedComponentName@
    extends @BaseComponentName@ implements ISharedComponent {

  @VirtualProperties;@

  public @SharedComponentName@(){
    this(Utility.getUUId());
  }

  public @SharedComponentName@(String oid) {
    super();
    this.oid = oid;
    listeners = new Vector<ISharedPropertyChangeListener>();
  }

  public String getOid(){
    return oid;
  }

  public void insert(String propertyName, int offset, Object value){
    fireSharedPropertyChange(new SharedPropertyChangeEvent(
        SharedPropertyChangeEvent.INSERT, oid, propertyName, offset, null, value));
  }

  public void delete(String propertyName, int offset){
    fireSharedPropertyChange(new SharedPropertyChangeEvent(
        SharedPropertyChangeEvent.DELETE, oid, propertyName, offset, null, null));
  }

  public void update(String propertyName, Object oldValue, Object newValue){
    fireSharedPropertyChange(new SharedPropertyChangeEvent(
        SharedPropertyChangeEvent.UPDATE, oid, propertyName, 0, oldValue, newValue));
  }

  public void addSharedPropertyChangeListener(ISharedPropertyChangeListener l){
    if(!listeners.contains(l)) listeners.add(l);
  }

  public void removeSharedPropertyChangeListener(ISharedPropertyChangeListener l){
    if(listeners.contains(l)) listeners.remove(l);
  }

  public void fireSharedPropertyChange(SharedPropertyChangeEvent e){
    for(ISharedPropertyChangeListener l: listeners){
      l.sharedPropertyChange(e);
    }
  }

  protected Vector<ISharedPropertyChangeListener> listeners;
  protected String oid;

 }
```

Fig. 10. Template source code of the shared component implementation

inherits the original component class and implements the *ISharedComponent* interface whose boilerplate code provides the default implementations.

2. Experiments and Analysis

Using our adaptation tool, we successfully and automatically adapted all Swing components (derived from the *JComponent* class) and all AWT components (derived from the *Component* class) that come with JDK. The experiments used all default settings, e.g. all original properties are treated as shared properties and all shared properties

are atomic. All the generated shared component source passed Java compilation without any exception.

However, we note several problems with automated adaptation that entail manual intervention by the developer. These problems are not necessarily limitations of the adaptation tool. Some of them may disappear as new conventions are established in JavaBean, while some others must involve context-sensitive decisions by human users. We document our experience as follows.

First, default adaptation typically results in an overwhelmingly large number of shared properties. For example, a converted *JButton* component contains as many as 87 properties. In most cases, not all the properties of a component need to be shared when used in a groupware application. Transmitting a lot of unnecessary property changes across the network can degrade application performance and network efficiency. Current component models such as JavaBean have no standard convention for describing the usage of properties. It is impossible at this stage to automatically decide shared properties. In fact, even the same property of a component may be shared in one application while not shared in another. Only the developer knows whether a property should be shared in a particular groupware application. The adaptation tool allows the developer to check shared properties via a simple spreadsheet-like GUI.

Second, manual adaptation is also necessary when the developer wants to add additional shared properties that do not exist in the original component. Sometimes even although the properties do exist, the developer still needs to create virtual properties to simplify control. For example, each Swing component has properties *X*, *Y*, *Width*, and *Height*, which describe the relative coordinates of the component in its container. Instead of using these four properties directly, we may create one virtual property called *boundbox* that logically congregates them. As this kind of adaptation often happens on user interface components, we provide the virtual property and its

implementation in separate templates for the developer to choose during adaptation.

Third, manual adaptation is generally required to support accumulative properties for fine-grained sharing. Take the "text" property of the *JTextPane* component as an example. By default it is adapted as an atomic property implemented by an *update* method. The adaptation tool can create additional *insert* and *delete* method signatures to implement it as an accumulative property. However, the body of these two methods must be filled in by the developer manually because their actual implementation is type-specific. For example, Java types such as *Vector* and *String* provide different functions for inserting and deleting elements.

Fourth, manual adaptation is also required when the desired property changes happen in an inner component but are not exposed by the original outer component developer. For example, the *JTextPane* component uses the *StyledDcoment* component as its model (due to the well-known model-view-controller or MVC paradigm). Incremental changes of the document, e.g., the insertion and deletion of characters in the document, are emitted as model events, which are not exposed by the *JTextPane* component. Thus interception of these incremental changes has to be done at the document model level. Fortunately, the *StyledDocument* class provides a hooking point through function *setDocumentFilter*. The developer can set a custom document filter that implements the *insert* and *delete* methods.

Fifth, manual adaptation is mandatory when the shared use of a component causes subtle side-effects on the user interface. Consider again the *JTextPane* component. When the model-level insertion and deletion are intercepted, transformed, and eventually applied say by a concurrency control protocol, the caret is not moved automatically as usual. This may cause subsequent insertions and deletions by the user to happen at the wrong position. This is because, in the *JTextPane* component, the view is implemented by the *JTextPane* class itself, while the model is implemented

by the *StyledDocument* class. When property changes work around the *JTextPane* methods, the caret position maintained in *JTextPane* is thrown out of sync. Therefore, in the adapted *insert* and *delete* methods, we need to calculate the right position of the caret and adjust the caret every time after an incremental property change is applied.

The amount of work increases progressively in the above five cases. However, manual adaptation mostly can be done via well-documented APIs and does not require analysis of source code. Even in the fifth case, it only takes about 100 lines of code in total. The kind of indepth plumbing work is only required on a few (text editing related) JDK components when fine-grained sharing is really needed. This represents a general tradeoff in collaboration transparency: adapting (reusing) existing applications and components saves the overall engineering costs but often at the loss of flexibility. When truly advanced features are desired, extra effort cannot be avoided. Nevertheless, the manual adaptation in the fifth case results in a group editor, which would require months of work if it were built from scratch.

D. Discussions

In this section we first show the flexibility achieved in our work and then compare our results with related approaches in terms of flexibility and reusability.

1. Achieved Levels of Flexibility

When a single-user component is not adapted to provide the shared component interface, as in [4, 5], we can still implement a component sharing mechanism by the transparent window techniques to intercept low-level user input events before they take effects. This mechanism can be provided as part of the runtime system for users

to share specified components. Due to the lack of object id, the runtime system has to do extra bookkeeping to track different replicas of the shared component. In the simplest case, when there is no component-specific knowledge for the mechanism to understand the low-level events, we have to maintain consistency by replaying these events verbatim to all the component replicas. As in typical replicated application sharing systems [3, 5], however, a turn-taking protocol must be followed by all participants to manipulate the shared component. Nonetheless, concurrency control is allowed at the component level, which means different components in the same workspace can be locked by different users. Furthermore, if the low-level events can be understood and translated as in [14, 15], more concurrency is allowed on the same shared component.

Using mechanical (or default) adaptation, a shared component can be automatically generated out of an existing single-user component. There is no extra coding effort required from the developer except some simple configuration in the beginning of the adaptation process, e.g., to provide output component name. Despite the large number of shared properties, much flexibility can be achieved with the generated component. Now that the results of happen-before interception are high-level property change events instead of the low-level window events, concurrency control can happen at component as well as property levels. That is, different users are allowed to work on different properties of the same shared component at the same time.

With manual adaptation, much more flexibility can be achieved. For example, if some shared property is turned into accumulative, only the incremental changes are transmitted over the network. Moreover, sophisticated concurrency control methods such as operational transformation (OT) [33, 34] can be applied on this property to allow for even more concurrency. With OT, multiple users are allowed to manipulate the shared accumulative property simultaneously without being blocked. Any user

can edit any part of the content at any time and all modifications are preserved in the final result.

Therefore, with no or limited adaptation, our work allows collaboration-transparent components to be shared under a range of strict and relaxed WYSIWIS policies. Concurrency control can be applied on individual components as well as properties. With support from the runtime system, control protocols can be dynamically switched [32].

2. Comparison with Related Work

In general the presented work takes a middle ground between collaboration transparency and collaboration awareness. It provides an adaptation tool for transforming third-party components to implement a shared component interface. It also provides middleware services such that the adapted components can be used for constructing groupware applications that allow for flexible sharing at component and property levels. The adaptation is done without modifying source code of the original components.

By comparison, traditional collaboration transparency systems such as [13, 4, 3, 5, 14, 15] aim to share a single-user application as a whole instead of individual components. Typical application sharing systems such as [13, 3, 5] can only allow for strict-WYSIWIS mode of collaboration. Although [4] is able to replace certain components in an application with custom multiuser versions, relaxed WYSIWIS is only limited to these custom components while the rest of the shared application is still strict-WYSIWIS. Other domain-specific application-sharing systems, such as [14, 15, 16], can also implement flexible collaboration at the application level. However, they require much higher engineering effort to adapt and reuse single-user applications. Moreover, due to reliance on application-specific knowledge, reusability of their infrastructures is generally low.

In [18], it is also possible to adapt components to support component and property level sharing policies. However, it requires the source components follow an extended naming convention (which is different from the JavaBean standard) but does not address how to translate the myriad of components that fail to observe their naming convention. Moreover, it relies on object diffing for implementing happen-before interception of property changes, which suffers from performance problems. It is admitted in [18] that their work is not suitable for synchronous collaboration.

Traditional collaboration-aware approaches, such as [8, 9], define custom programming abstractions that tightly couple data and control, without addressing the large base of components that do not observe their programming abstractions. By comparison, in our work, the programming abstractions strictly follow the standard JavaBean component model, data and control are separated components, and a tool is provided to translate components that do not observe our shared component model. Hence more flexibility and reusability are achieved.

Specification-based approach has been used to explore automatic component retrial and adaptation for reuse, e.g. [35], [36]. However, these approach based on the assumption that a component has the needed specification, e.g. the state based specification, for both the problem and component. However, this assumption does not hold for most of the existing popular components. They also usually didn't address the specific adaptation needs of component for reuse in groupware applications.

E. Conclusions

We claim the following two contributions in this chapter: First, we propose a new shared component model for building flexible groupware applications. Based on this model, coordination services are decoupled and dynamically associated with shared

data objects at different granules. Secondly, a new adaptation tool is provided for re-purposing third-party components into shared components without modifying their source code. Notably all the source components are only required to follow standard industry component models. Much flexible sharing is achieved with no or minor manual work. Techniques required to implement the proposed model and system are generally available in modern component technologies such as .Net and JavaBean. As a result, the large and ever-growing base of components can be reused for constructing collaborative systems.

CHAPTER IV

ADAPTABLE COORDINATION SERVICES

A. Introduction

In the previous chapter, we motivate the shared component model and provide its Java embodiment. A component adaptation tool is also provided to convert existing single-user component to be sharable components. In this chapter, we look into the collaboration infrastructure providing reusable coordination services. Our main focus is the adaptable consistency control issue. As noted in [37], consistency control in interactive groupware is both a technical problem and a human problem. Traditional approaches cannot be applied directly in groupware because the distributed system in question must include support for human social protocols. Specific consistency control methods impact groupware interfaces and ultimately groupware users. Therefore the choice of consistency control must reflect the way people actually work together. The human and technical issues must be considered together because the design of user interfaces and the choice of consistency control algorithms often compromise each other.

Significant progress has been made in the groupware field over the past decade in devising consistency control mechanisms that appear more effective for people, e.g., [38, 39, 37, 33, 20, 40, 41, 42]. The rich variety of consistency maintenance methods in the literature is testament to the fact that no single approach is applicable in all systems or application domains. For example, turn-taking protocols are generally established in application sharing systems [19], one of the most accepted collaboration technologies. However, studies [4, 43] show that they are not as effective for intellectual work due to the low level of concurrency. Operational transformation algorithms

[44], on the other hand, are widely implemented in group editors because they are able to achieve high responsiveness and concurrency. However, they may not be effective for large or unfamiliar groups due to heavy reliance on the users' conscious following of specific social protocols for coordination [1, 37]. Specific consistency protocols each have their own niche where they are more effective than in other situations. Ideally we would like to use the most effective protocol for each collaboration scenario, and apply a protocol only when the scenario matches its design tradeoffs.

Early collaborative systems generally focused on exploring the innovative aspects of a specific consistency control algorithm or framework. As a result, design considerations are usually biased towards some collaboration scenarios and are not sensitive to the situated and dynamic nature of cooperative work [45]. The lack of flexibility in these systems has two consequences. First, when a consistency protocol designed for one scenario is used with another, system efficiency may be undermined because tradeoffs differ significantly from expectation. Second, a consistency protocol that is effective for one user group may be disastrous for another in which the participants have drastically different cultural background or personalities. "Fascist" groupware that fails to achieve the desired level of flexibility often suffer from low acceptance or organizational resistance [46].

Therefore the consistency control mechanism of collaborative systems should be implemented such that it is possible for the users to choose the "right" consistency protocols when the needs emerge, instead of trying to prescribe possible protocols and scenarios [29]. Technically, consistency control requires maintaining consistency among replicas of shared data. When the same set of data objects is manipulated in different scenarios, being able to switch between consistency protocols necessarily implies a clean separation between data and control. That is, only when consistency protocols are decoupled from the data or interfaces they control, is it possible to

implement a reusable library of consistency protocols that can be chosen to apply in specific collaboration scenarios. The question is how.

Most previous collaborative systems are only able to provide limited adaptability in consistency maintenance. The only type of adaptation allowed at run time is often through setting parameters to choose between different policy variations within the provided consistency protocols. This is the case in most groupware frameworks, e.g., [20, 11], and groupware applications, e.g., [47, 48]. The reason is generally that they tightly bind the consistency protocols, such as locking, serialization, and operational transformation, to the shared data or interfaces in the system which they control. As a result, they are not able to address the needs for applying different consistency protocols on different shared data objects at the same time or on the same objects over time, as have long been motivated in the literature, e.g., [38, 37, 7].

This chapter proposes a novel framework that supports adaptable consistency protocols. Due to the separation between data and control, consistency control protocols can be dynamically associated with the data or interface objects that they control at various levels of granularities. The framework provides services to facilitate the dynamic association and switching of protocols. As a result, different objects in the same workspace can be controlled by different consistency protocols, and the same objects can be controlled by different protocols as the collaboration needs change. All these features are achieved at run time without modifying source code.

Our implementation follows the established component-based software engineering practices, e.g., [18, 31, 49, 9, 50, 51, 52]. Adaptable consistency control entails a componentized design of data, protocols, and the "gluing" code that facilitates the interaction between data and protocols at run time. These three types of components, on the other hand, can often be reused across different collaborative applications. Following the groupware engineering principles noted in [23], our approach has been

prototyped over the past three years in a groupware framework called EFG (Evolvable Framework and Groupware) and a collaborative application environment called EXEC (see next chapter for detailed description of EXEC and its extension). Both the framework and the groupware applications developed atop are component-based such that they can evolve together.

The rest of this chapter is organized as follows: In Section B we discuss how to model user interaction in a shared workspace application and the behavior of consistency control. After that, Section C describes the reusable coordination services provided in the EFG framework. This is followed by a comparison with related work in Section D. Finally, Section E concludes contributions of this chapter.

B. Consistency Protocol Modeling

Adaptable consistency control entails a well-defined interface between shared data and protocols. Not to deviate from our main research focus, we assume that all shared data objects are replicated in the system. Users collaborate by interacting with a collaborative workspace that visualizes the shared data in some way. In this section we model consistency protocols, and data-protocol their interaction.

1. User Interaction

The finite state machine (FSM) in Figure 11 depicts the typical lifecycle of a consistency protocol as a three-stage process: At stage one, the user applies a protocol to a shared data object; at stage two, concurrent changes are made on the object under the control of the protocol; and at stage three, the protocol is detached from the object so that it no longer controls concurrent manipulation of the object. In most existing systems, stages one and three are performed by the system developer

at design time. If it turns out that a protocol is no longer appropriate for the application scenario, the developer often has to make a major system redesign such that the object is controlled by a different protocol. Due to the tight coupling between data and control in traditional systems, it is generally difficult for them to support the dynamic switching between different types of consistency protocols.

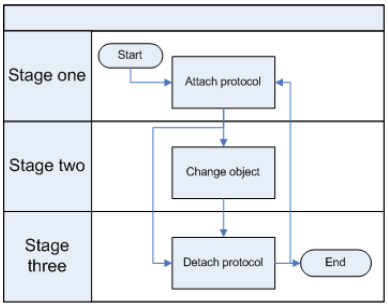

Fig. 11. Lifecyle of a consistency protocol.

However, if data and control are separated, stages one and three will become easier, even without incurring source-level modifications or redesign. A consistency protocol can be attached to a shared object at stage one when the needs arise, and then detached from the object when it is no longer needed. The attach and detach actions at stages one and three can be either implicitly (automatically) triggered by the system or explicitly (manually) triggered by the user. Since consistency protocols are often parameterized for the user or the system to choose between different policy variations, the parameter settings during their lifecycle can also be implicit or explicit. If the triggering is explicit, we say the consistency control mechanism is *adaptable*; or if the triggering is implicit, we say it is *adaptive*.

From the user's perspective, stages one and three each have only one action to attach (enable) or detach (disable) a protocol. At stage two when the protocol is

effective, however, the user may perform arbitrary number of operations to cause state changes on the object. Any action (attach, detach, or operation) the user performs may take some time for the system to respond and for the user to perceive its (visual or auditory) feedback on the user interfaces. Before the user moves on to issue the next action, he may willingly or unwillingly be blocked until the system response to a previous action is perceived. Or the user issues the next action in a nonblocking manner, i.e., without being held back for a response to previous actions. This observation is consistent with the model of [38] which suggests that, during a user's expected response time of 50-100ms, he may often issue several actions in a row before perceiving response to the first action. In other words, it may not be necessary or effective to execute every action in a blocking manner in an interactive system. Some flexibility should be allowed.

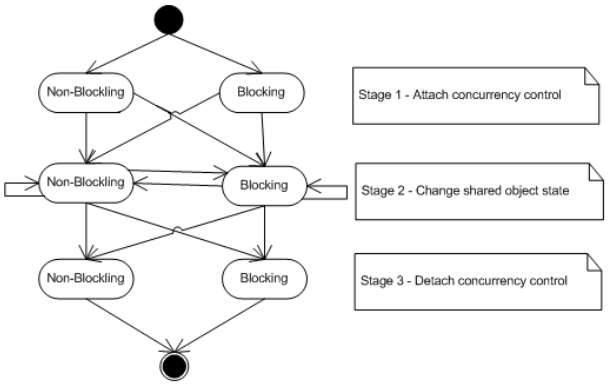

Fig. 12. B/NB actions at each stage

Hence we refine the three-stage lifecycle FSM into the form of Figure 12, which highlights the blocking (B) and nonblocking (NB) semantics of user actions. Based on this new FSM, we can formulate four major action paths in regular expressions in Table II.

Table II. A model of four major action paths

Stage one	Stage two	Stage three
B	(B|NB)*	B
B	(B|NB)*	NB
NB	(B|NB)*	B
NB	(B|NB)*	NB

Stage two can contain arbitrary number of user actions and each of them can be blocking or nonblocking. In a regular expression this is represented as (B|NB)*, standing for a sequence of zero or more interleaved blocking or nonblocking actions. It is general enough to generate numerous paths, e.g., (B)*, (NB)*, (3NB·B)*. Specifically (B)* represents the case that all actions are uniformly executed in a blocking way. (NB)* means that all actions are nonblocking. (3NB·B)* means that, after every three nonblocking actions in a batch, a fourth action will be executed in a blocking way. If we take the (B)* mode as a pure pessimistic policy and (NB)* as a pure optimistic policy, then between these two extremes (B|NB)* effectively expresses a continuous spectrum of policies of different degrees of optimism.

Note the *actual behavior* of consistency control protocols are usually beyond the expressive power of regular expressions. For the purposes of this dissertation, however, it suffices to use regular expressions only for modeling the *observable behavior* of protocols from the framework perspective.

a. A Taxonomy of Policies

To testify the generality of our interaction model, here we examine some consistency protocols that are frequently referenced in the literature and implemented in many

collaborative systems. Example protocols are locking, serialization, and operational transformation. Each protocol has a number policy variations. Our taxonomy extends that of Greenberg and Marwood [37].

Locking - The locking protocol maintains consistency by excluding actions from users who are not holding the lock. In this way, only operations from the current lock holder are allowed to cause state changes to the shared object. Turn-taking or floor control protocols [19] are but coarse-grained locking protocols that lock the whole application [4]. Three variations of locking are given in [37] based on the level of optimism in requesting and releasing the lock, namely, pessimistic, semi-optimistic, and optimistic locking. A pessimistic policy blocks the user's further actions when he requests or releases the lock. A semi-optimistic policy only blocks the user's actions when he releases the lock. An optimistic policy blocks neither request nor release actions. Table III maps these policies into our interaction model. Apparently the taxonomy in [37] does not address user actions at stage two.

Table III. Mapping of locking policies

Locking	Stage one	Stage two	Stage three
Pessimistic	B	B	B
Semi-optimistic	NB	NB	B
Optimistic	NB	NB	NB

Serialization - The serialization protocol maintains consistency through a total ordering of concurrent operations targeted at the same object. There are basically two policy variations [37]: optimistic and pessimistic. While pessimistic serialization blocks every user operation until it is guaranteed to be in the right order, optimistic serialization allows user operations to execute locally and then undo their effects after

they are detected out of order. Table IV shows how these two policies map into our interaction model. The taxonomy in [37] only addresses actions in stage two.

Table IV. Mapping of serialization policies

Serialization	Stage one	Stage two	Stage three
Pessimistic	-	(B)*	-
Optimistic	-	(NB)*	-

Operational Transformation - Operation transformation (OT) [44] is a pure optimistic consistency protocol. Any local operations are allowed to execute in a nonblocking manner. OT differs with optimistic serialization on how to repair inconsistencies. Optimistic serialization always undoes operations that are out of order and then redo them by a global order. OT transforms remote operations such that concurrent operations can be executed in any order and at the same time their effects relation is preserved [33]. Table V maps OT into our model.

Table V. Mapping of operational transformation

OT	Stage one	Stage two	Stage three
Optimistic	-	(NB)*	-

In the above analysis, locking seems to map closer to our model because locks have to be requested (attached) and released (detached), implicitly or explicitly, as a convention. Traditional systems that implement serialization and OT protocols do not address stage one and stage three issues. The reason is that, in general, they only consider application-wide concurrency control policies. The protocols automatically take control of the shared data once the application is launched. Attach and detach

actions are taken at the source level. In previous work, the protocol and policy decisions are usually made by developers before hand, not by users at run time. Obviously we can consider B/NB policies at both stages one and three for any protocols under our framework.

In our model, all known protocols can be adapted to support a continuous spectrum of policy variations. Policies can also be chosen at stage two. Even under a pure pessimistic policy, the user may not wish to synchronize every single operation. Usually a batch of operations can be accumulated and propagated together to save bandwidth and reduce interferences between users [38]. On the other hand, an optimistic policy does not necessarily mean that users do not want to be blocked at all. For example, discussions in [47, 53] reveal a variety of synchronization policies in optimistic serialization and OT protocols. For awareness reasons, a balance is often sought in collaborative systems such that synchronization should not be delayed for too long. Hence policies such as (3NB·B)* may often be more effective in practice than pure (NB)* policies.

It is also worth noting that our taxonomy is intended to model consistency protocols for the purpose of supporting adaptable control. The above protocols each have different merits and application domains. Boundaries between them are often not as distinctive as they might appear. For example, although allowing for highly concurrent and interactive collaboration, OT is applicable only when operations are commutative after transformation. As revealed in [54, 34], serialization is used in OT to handle conflicting or non-commutative operations that intend to change the same property of the same object. Locking can also be integrated with OT to achieve more flexibility, as shown in [41]. In addition, other types of consistency control protocols exist in collaborative systems, such as merging [55] and multi-versioning [56]. Nonetheless these facts do not really change the way these protocols and their varia-

tions interact with the framework, although some of them may have more complicated user interfaces in actual implementation.

b. General Implications on Design

The interaction model provides an important guideline for designing our adaptable concurrency control framework such that it is able to accommodate a spectrum of consistency protocols. It will serve as a contract between the framework and specific consistency protocols, for devising services to support the plug-n-play of protocols, and for devising interfaces for the protocols to receive such services. The following analysis in turn motivates the needs for meta protocols, the interfaces between framework and protocols, and undo mechanisms.

In a distributed workspace, any user action in the above three stages can clash with concurrent actions from peer users. For example, two users may concurrently attach different consistency protocols to the same data object. While operations at stage two to change object states are controlled by specific consistency protocols, the attach and detach actions are apparently beyond the duties of these consistency protocols. Therefore, in addition to the "ordinary" consistency control protocols, "meta" protocols must be implemented in the system for maintaining consistency and resolving conflicts at stages one and three.

The B/NB semantics at different stages are interpreted by protocols at different levels. Consistency protocols are on their own to process actions that are directed to them. They are also responsible for providing the interfaces for users to choose between policy variations, e.g., by setting protocol parameters. In other words, it is the specific consistency protocols rather than the framework that interpret the regular expression of (B|NB)* at stage two. For example, optimistic serialization and operational transformation protocols usually have explicit rules for controlling

operation synchronization among collaborating sites, as noted in [47, 53]. When certain conditions are satisfied, the system needs to synchronize a batch of nonblocking operations which are probably combined. However, it is difficult and inefficient to further externalize these conditions and their checking mechanisms from specific consistency protocols and formalize them at the framework level. It is also dangerous for the framework to bind itself with specific semantics of data objects and consistency protocols.

To interpret the semantics of B and NB actions, we can conceptually imagine an input queue inside each of the ("meta" and "ordinary") consistency protocols. Actions at stages one and three are queued by the meta protocols, while actions at stage two are queued by the ordinary protocols. B means the current action is processed by the corresponding protocol synchronously in collaboration with remote sites, and the next action in the queue will not be processed until the processing of the current one is finished. NB means the current action is dequeued and executed immediately at the local site while the protocol is still processing it asynchronously with collaborating sites. However, the next action in the queue is not blocked by this background processing.

In addition, to support nonblocking interaction, it is necessary to provide "undo" mechanisms in the framework. Nonblocking policies can in general achieve better local response than blocking policies. However, while nonblocking actions are allowed to proceed before consistency is ensured, it may turn out in the consistency protocol (e.g., optimistic serialization or locking) that a nonblocking action should not have happened. In that case the system must be able to restore the object state to a previous one by undoing the effects of wrongly presumed actions. This requirement has also been confirmed in previous work, e.g., [37]. Note the different purposes between the system-generated undo here and the user-initiated undo in [42].

2. Interface of Consistency Protocols

From the above analyses, we conclude that the framework (or more specifically, the meta protocol) is responsible for resolving actions in stage one and stage three in our user interaction model. In this subsection, we look into the ("ordinary") consistency control protocols themselves, which are responsible for resolving stage-two user actions. When an action gets deposited into a consistency protocol, the protocol can treat the action differently based on the blocking (B) or nonblocking (NB) semantics it poses on the action.

If the protocol interprets that this action should be treated in a nonblocking way, then it should be executed immediately by the framework. However, to indicate that it is really being resolved, a "pending" notification should be emitted to notify the framework. For example, the application receiving the pending notification can somehow indicate on the user interface that this action is still being resolved, although it has been seemingly executed. This "pending" notification could be useful for users to make sense of what is going on when this action is eventually "vetoed" by the consistency protocol and its effects are undone from the user interface. If the resolution is a success, however, a "confirmed" notification will be sent to the framework to indicate that this NB action does not need undone.

On the other hand, if the protocol determines that this action should be treated in a blocking way, no subsequent action shall be taken until this blocked action is resolved by the consistency protocol. If the resolution is a success, the action will be executed and a confirmation notification will be sent. Otherwise, a veto notification will be sent. Notably, there is no action to be undone in this case since this action has not been executed yet.

Figure 2 defines the Java interface that all (meta and ordinary) consistency

```
public interface IConsistencyProtocol {
        //(1) to get the unique id and description of this protocol
        public String getProtocolId();
        public String getProtocolDescription();
        //(2) to deposit user actions and to register/deregister the protocol
        public void   deposit(String protocolOwner, AbstractUserAction action);
        public void   addUserActionResolutinListener(IUserActionResolutionListener l);
        public void   removeActionResolutionListener(IUserActionResolutionListener l);
        //(3) to control the resolving of user actions
        public void   startResolution(int queueId);
        public void   suspendResolution(int queueId);
        public void   resumeResolution(int queueId);
        public void   stopResolution(int queueId);
        //(4) to undo all actions resolved by this consistency protocol
        public void   undo();
}
```

Fig. 13. Consistency protocol interface

protocols are assumed to implement in our framework. The first group of two methods are to return the unique id and the high level description of a consistency protocol, respectively. In the second group, the *deposit* method is called by the application to deposit user actions into the consistency protocol. The other two methods are for the application to register/deregister itself as an action resolution listener. The application can communicate with the protocol only when it is registered as a listener. Methods in the third group and the fourth group are called by the meta protocol when the application attaches or detaches an consistency protocol. We will give more details of how these methods are used in Sections 3 and 4.

a. Collaborative Workspace Applications

We model a collaborative workspace application as a container containing shared components. This conceptual model is compatible with modern (form/window based)

interactive applications. For example, in Java, almost all form-based applications have a root container *JFrame* which contains other Swing components. The contained components can also be containers that contain other Swing components. Similarly in .NET, window-based applications also have class *Form* (different .NET languages may have different names) as the root container.

These components often form a hierarchical structure. There are mainly two ways to construct components: inheritance and composition. For example, Java allows developers to build a new JavaBean component by extending the JComponent class or its subclass and by composing other JavaBean components. In .NET, the *Component* class serves as the base class of all components. Inheritance and composition naturally form the containment relationship between components. With a root container, all components in an application form a component tree.

```
public interface ISharedContainer extends ISharedComponent {
        public void insertChildren(int offset, ISharedComponent component);
        public void deleteChildren(int offset);
        public ISharedComponent getChildren(int offset);
}
```

Fig. 14. Shared container interface

As shown in Figure 14, we model a shared container as a shared component with an accumulative property named "children". Three methods are defined such that a shared container can add a subcomponent into itself by method *insertChildren*, remove a subcomponent from itself by method *deleteChildren*, or get a subcomponent from itself by *getChildren*.

With this interface, the framework runtime system will be able to traverse a shared container and dynamically reflect all its descendant components. For the purposes of this dissertation, we assume that all shared data components implement the

ISharedComponent interface. In future work we will consider how to transparently adapt third-party components to provide the same interface.

C. Framework and Meta-Services

In the previous section we have discussed the modeling of shared data and consistency control protocols in collaborative workspace applications. In this section, we describe our adaptable consistency control framework, which at run time dynamically binds the separated data and protocol components and facilitates their interaction. We will discuss in turn its architecture, the support of multi-granularity protocols, the processing of user actions, and the working of meta protocols. This framework is actually part of a larger initiative to build evolvable framework and groupware (EFG) using a component-based approach. In this dissertation, we refer to the part in EFG that supports adaptable consistency control as EFG for brevity.

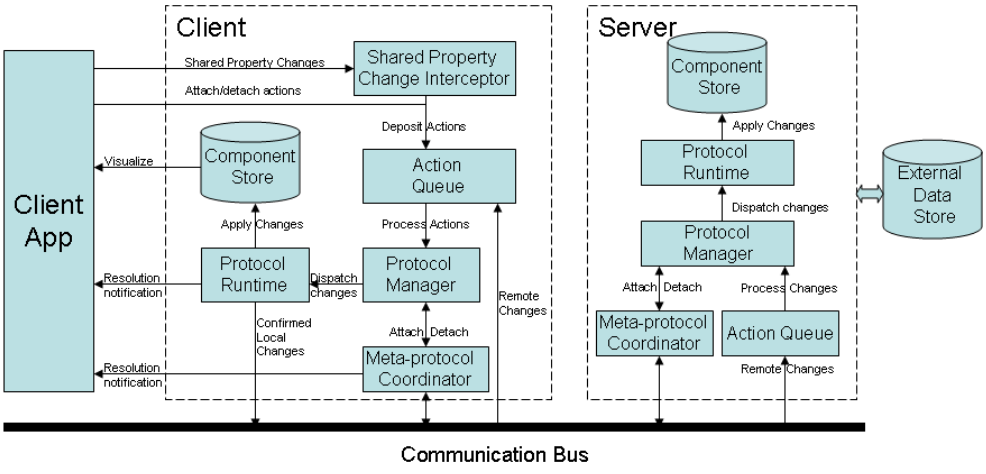

Fig. 15. Architecture of the EFG framework

1. Architecture Overview

EFG takes a client/server architecture as shown in Figure 15. The server maintains persistent shared data and performs session control. There are five modules in the server that are relevant to implementing adaptable consistency control: component store, protocol manager, protocol runtime, meta-protocol coordinator, and action queue. While sharing these five modules with the server, the client has an additional module called the shared property change interceptor. All clients in the same session communicate with each other through the server on an abstract communication bus. The users interact with each other via the groupware application that uses services provided by the EFG client at each site.

The data distribution service replicates the shared data and collaboration functions from the server to the client each time a client is launched. Currently we use Java serialization and Java RMI as the replication mechanism. (Notably similar mechanism could be found .NET framework). Java serialization has known problems of version compatibility in byte-code formats. More general serialization mechanisms will be used to replace Java serialization, e.g., XML-based serialization. The shared data must persist across multiple collaboration sessions. Similar to data distribution service, we use the Java serialization as for implementation. In the following we describe these modules and their interaction.

Component Store: it maintains all the shared data components in a table with the schema of $<UID, LocalID>$, where UID is a universal unique id of the component instance and $LocalID$ is the local reference to the component instance.

Protocol Manager: it maintains a property-protocol table that associates shared component properties to consistency protocol instances. The schema of this table is $< ComponentId, PropertyName, ProtocolId >$, where $ComponentId$ is the

universal id of a shared component, *PropertyName* is a shared property of the component, and *ProtocolId* identifies a consistency protocol instance from a table maintained by the protocol runtime module. The protocol manager also maintains several other tables for implementing multi-granularity protocols and default protocols, as will be discussed in Section 2.

Action Queue: it stores user actions, including attach, detach, and property changes. The action queue in the client stores both local and remote user actions. Since the server does not have a local application instance, the server action queue only store remote actions.

Shared Property Change Intercepter: it intercepts shared property change events such as insertions, deletions and updates that are emitted from shared components in the client, right before the changes really take effects on the component states. Then it pushes these events into the action queue.

Meta-Protocol Coordinator: it runs in both the server and the clients to implement meta protocols that will be described in Section 4. The application calls the client meta-protocol coordinator to issue consistency protocol attach and detach requests, which are resolved by the meta protocol. The server meta-protocol coordinator acts as the moderator in the meta protocol execution.

Protocol Runtime: it maintains the consistency protocol instance table, the fields and their meanings being shown in Table VI. The *state* field reflects the meta-protocol resolution of actions to attach/detach this consistency protocol, where vetoed and detached protocols can be safely removed from the table.

The groupware application built with the EFG framework resides in the same address space as the client at each site. It talks to the client through the service APIs provided by client. Actually these services are merely an aggregation of ser-

Table VI. The consistency protocol instance table

Field	Meaning
ProtocolId	global Id of this protocol
ProtocolInstance	local reference of this protocol instance
State	**Active** - the attach of this protocol is a success and it is valid for resolving property changes **Vetoed** - this protocol is inactive because the attach of this protocol is vetoed **Pending** - attach of protocol is being resolved nonblockingly **Detached** - inactive because it has been successfully detached
Owner	ID of the client that attaches the protocol
$<Seq_1, Seq_2, ..., Seq_n>$	The numbers of *local* property changes from different clients that have been confirmed by this protocol where n is the number of clients
Lease	Time to expire

vices provided by the client's internal components (modules). They largely fall into the following four categories: First are methods for retrieving the shared data components, querying the available consistency protocols and protocols attached to a given shared property. The second type of methods are called to attach or detach consistency protocols. The third type of methods are called by the application to register or deregister itself as the user action resolution listener to get notified of resolution results of user actions, including attach, detach, and property changes (see the *IConsistencyProtocol* interface in Fig. 2). The fourth type of methods are called to

register or deregister the property change interceptor as the shared property change listener of shared components (see the *ISharedComponent* interface in Fig. 3). In addition, we also provide APIs in the client such that the application can proactively push the property changes into the action queue directly, instead of via the change interceptor. All these APIs are just wrapper methods of the corresponding component functions.

Now we briefly describe how the system is started and how the server, client and application modules interact at run time. The EFG server initializes first and reads in the persisted shared data components from the external data store. After that, when an application is launched, it first starts the client. The client connects and registers itself to the server and replicates the shared data components from the server. After that, the application visualizes the shared data components and available consistency protocols on its GUI. After the application initialization finishes, the user can then start to issue commands to attach/detach consistency protocols and change the shared component properties. For attach/detach requests, the client meta-protocol coordinator will talk to the server coordinator to resolve the requests. The resolution results will be sent to the application. For shared property changes, the property change interceptor will catch the changes and push them to the action queue. Then it will be dispatched to corresponding consistency protocol instance. Confirmed changes will be propagated to remote collaboration peers for execution. At the same time, the property change resolution will be sent to the application. Note that the visualization of shared data and resolution results is application-specific and beyond the scope of this disseratation.

2. Multi-Granularity Protocols and Performance Issues

The property-protocol table in the protocol manager module is extended for supporting multi-granularity protocols (MGP). In this subsection, we discuss how MGP is implemented and how performance issues that ensue are addressed. However, due to the similarity between MGP the concept of multi-granularity locking (MGL) in databases [57], our discussions will only be conceptual.

a. Aggregate Properties

As shown in the previous subsection, the property-protocol table in the protocol manager module maintains the mapping between shared properties and consistency protocols. To support MGP, we introduce three special built-in properties: "component", "workspace", and "subtree". Respectively, they are used when a user wants to attach a consistency protocol to (1) all shared properties of a shared component, (2) all shared properties of a shared container itself and all shared properties of its children components, and (3) all shared properties of a shared container and all shared properties of all its descendent components. By definition, a descendent component of a container is (recursively) a child component of the container or a child component of its descendent container.

The reasons of supporting these aggregate properties are performance and scalability. First, attach and detach of protocols invoke the execution of meta protocols, which may involve expensive communications between multiple parties. Using aggregate properties, we can reduce such communication costs. Secondly, the use of aggregate properties also reduces the size of the property-protocol table, which in turn saves the time to read and update the table as its size grows. This eventually translates to improved local response at run time.

b. Intention Protocols

When user wants to attach a protocol to control the whole subtree of a shared container, one part of performance penalty is addressed by introducing the "subtree" property to reduce the number of meta-protocol requests. The other part of the performance penalty comes from that the protocol manager has to scan the component subtree to find out if there exist any attached protocols that conflict with the protocol to be attached. To reduce this cost, we maintain an Intention Protocol table in the protocol manager module. Each entry in this table takes the form of $< ComponentId, ProtocolType, DestComponentId, DestPropertyName >$, where *ComponentId* is the component which is associated with an intention protocol, $ProtocolType$ is the type of the consistency protocol, and *DestComponentId* and *DestPropertyName* together point to the property of the component that was attached wth this consistency protocol.

Each time a consistency protocol is attached to a shared property, an intention protocol entry is added to each of its ancestor components. Accordingly, each time a consistency protocol is detached from a property of a component, the intention protocol entries will be removed from the ancestor components. Whenever a user attaches a consistency protocol to a property, the protocol manager will check the intention protocol table to see if it conflicts with any intention protocols along the path leading to that property. If a conflict is detected, the current protocol is not attachable and the request is vetoed without going through the meta protocol.

At current stage of this work, we only allow one data object (property, component, workspace, or subtree) to be attached with one protocol at a time, although we allow unrelated objects to be controlled by different protocols. The scopes of two protocols conflict if they overlap or intersect on the same set of objects that they con-

trol. For example, two different properties of the same component can have different protocols. If one user say Alice has attached a locking protocol to the color property of a circle, a second user say Kathy is prevented from attaching any protocol to the circle at the component level. However, if the second user is also Alice herself, the attach will be allowed and the original protocol on the color property will be detached. This is called protocol upgrading. In future work we will allow for more flexibility in defining whether or not two protocols are considered as conflicting and flexibility in resolve conflicts. For example, as confirmed in [54, 41], locking, serialization and operational transformation protocols can often coexist, rather than conflict, with each other.

c. Default and Implicit Protocols

As discussed in Section B, a consistency protocol must be attached to a shared data component before they can be modified by any user. After the intended changes are made, the protocol should be detached. The attach and detach actions can be either explicit or implicit. Given the flexibility provided by MGP, some obvious performance problems need be addressed. First, non-expert users may not be able to, or not willing to, decide which protocols should be used on which objects. Second, users may feel distracted if they have to explicitly press buttons to issue attach/detach actions all the time.

To mitigate the first problem, we allow the (expert-) users to configure default protocols in a default protocol table via a spreadsheet user interface. This table is persisted on the server and loaded into the client (more specifically, the protocol manager) at initialization time. The default protocol table keeps information in the following format: $< \mathit{ComponentType}, \mathit{PropertyType}, \mathit{DefaultProtocolType} >$. For example, collaborative editing component by default use operational transformation for

unconstrained group editing. A "meta" default protocol (e.g., locking) can be configured as the default protocol for any property if its default protocol is not specified explicitly.

The default protocols eliminate the needs for users to make protocol decisions in "typical" situations. However, as users' experience with the system accrues, they may be willing to learn and explore more advanced features (e.g., adaptable consistency control) to get extra benefits [58, 59]. We provide a simple user interface to enhance the learnability of our adaptable consistency control mechanism. As shown in Fig. 2, we assume every consistency protocol implements a method *getDescription* to describe itself, e.g., how it works, where it should be used, and what the user experience and interface effects will be like. At initialization time, the client provides the application with a listing of available consistency protocols with their descriptions. The application user interface can display the description of a consistency protocol in tool tips or balloon when the user points his cursor to the protocol. The description is expected to help the user make more informed protocol decisions.

To address the second problem, we allow the user the specify a "Lease" parameter for each consistency protocol, as shown in table VI. If a user explicitly attaches a consistency protocol to an object, the protocol lease can be set to "forever" by default such that the user must explicitly detach the protocol later. However, there are times when the user may just want to make some casual changes, for which it would be an overkill to do explicit attach and detach. Instead, a default protocol can be implicitly attached in a nonblocking manner to the affected property once a user starts to make changes without explicitly attaching a protocol. The lease of this protocol will be set a default value, say 30 seconds. If the user continues to work on this property, the lease will be renewed automatically. When the lease is eventually expired, the consistency protocol is automatically detached.

3. User Actions: Attach, Detach and Property Changes

In the following we describe how to implement the attach and detach of consistency protocols and object state change operations. The application is ultimately responsible for providing user interfaces for triggering these operations. As discussed in Section B, the execution mode of each user action can be either blocking or nonblocking. The mode of property change actions are determined by consistency protocols. Now the question is who decides the execution mode of attach and detach. In general, the application should allow users to configure the attach and detach policies of consistency protocols.

In fact, the execution modes of attach and detach are not only application specific but also situation dependent. For example, when a user explicitly attach a lock to an object, the mode is set as blocking by default. However, the mode is nonblocking by default if the locking protocol is attached automatically by the system, e.g., when the user attempts to modify an object without explicitly attaching a protocol first. We omit further details of the configuration since it is application specific and out of the scope of this dissertation.

a. Attaching a Consistency Protocol

When a protocol attach action in the action queue is to be processed, the client first checks whether or not the protocol is attachable by checking the intention protocol table. If it is not attachable, e.g., due to the existence of attached conflicting consistency protocol(s) in the scope of this protocol, the client simply vetoes this attach action by sending a veto notification. If the protocol is attachable, the client checks if it is blocking or nonblocking.

If the attach is blocking, the meta protocol is invoked to attach this protocol

in all clients synchronously. The client will not proceed to the next action in the action queue that operates on the same object until the attachment is resolved by the meta protocol. The resolution result will be sent to notify corresponding listeners (the application). If the attach resolution fails, the protocol state will be set as "vetoed". If the resolution is a success, an instance of the consistency protocol is created in the protocol runtime of all clients and the server. In the protocol instance table (Table VI), the Owner field of this instance is set as the client id, the "Seq#" vector is zeroed since no stage-two action has been performed yet by this protocol, and the protocol state is set as "active". A tuple will also be added into the property-protocol table in the protocol manager at all sites with the corresponding object id and protocol id. Then the *startResolution* method of this protocol will be invoked in order to start resolving stage-two user actions (refer to Figure 2). At the same time, a tuple will be added into the intention protocol table for each of the ancestor components of this component property.

If the attach is nonblocking, the client will first create an protocol instance in the protocol runtime and set its state as "pending". A pending notification of this attach will be sent out. At the same time, a tuple is inserted into the property-protocol table for resolving user property change actions. The intention protocol table will be updated accordingly. While the client asynchronously resolves the attach action, it proceeds to process subsequent actions on this property in the action queue. If the resolution of the attach eventually succeeds, the state of this protocol instance will be set as "active" and a confirmation notification will be sent out. The property change actions confirmed by this protocol instance will be sent out to the server and other peer clients for delivery since these actions are sure not to be undone later.

However, if the nonblocking attach fails, the client will first stop the resolution of user actions by this protocol. It does so by invoking the *stopResolution* method of

this protocol (refer to Figure 2). Then the client needs undo all local executed actions which are based on the assumption that this attach would succeed. The processing is complicated if the detach of this protocol is also nonblocking. In that case, the client will also need to find consistency protocols that are nonblockingly attached (to the same object) after this one. Then it invokes the *undo* method of this and all the other protocols to undo actions they have executed, in the reverse order of their attachment. Finally the state of this protocol will be set as "vetoed". Corresponding entries in the intention protocol table will be removed accordingly.

Apparently, nonblocking attach of protocols can achieve fast local response but at the risk of cascading undo when the detach is also nonblocking. In this chapter we investigate the feasibility of implementing adaptable consistency control mechanisms and try not to foresee how these mechanisms will be used. Some of the B/NB combinations of attach and detach may be found useful in some situations and not in some other situations. Application developers and end users will be at their discretion as how and where to use these mechanisms.

b. Detaching a Consistency Protocol

The detach action can also be blocking or nonblocking. If it is blocking, the client does not proceed to the next user action (on the same object) until the resolution result comes back. If it is a success, the state of this protocol is set as "detached", the corresponding tuples in the property-protocol table and the intention protocol table are removed, and a "confirmed" notification will be sent out. If it fails, however, these tables and the state of this protocol remain except that a "vetoed" detach notification is sent out.

If the detach is nonblocking, the client will set the protocol state as "pending" and then proceed to the next user action (on this object) without waiting. The user

can attach a new protocol to the same object. If meta-protocol resolution comes back positive, the protocol state is set as "detached" and the property-protocol and intention protocol tables are updated accordingly. If the detach is vetoed, the assumed attach of new protocol(s) will be undone, similarly to the undoes in nonblocking attach. The resolution results will also be emitted.

c. Component Property Changes

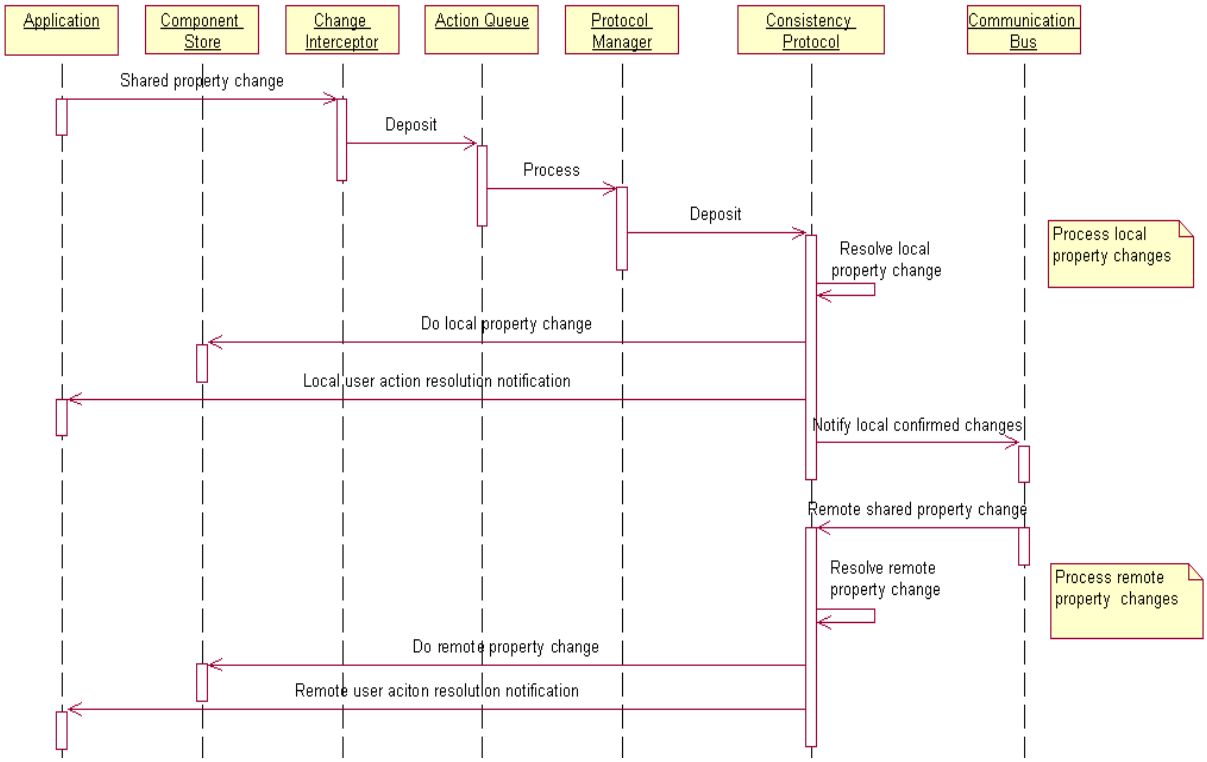

Fig. 16. Shared property change propagation path

Figure 16 illustrates the propagation path of a shared property change. It is first intercepted by the change interceptor and then pushed into the action queue. When a shared property change in the action queue is processed, the client (or more

specifically the protocol manager) checks the existence of a consistency protocol on this property and its state.

If no entry exists in the "property-protocol" table for this property, the client will try to attach a default protocol to this property. If the attach fails, this property change will be vetoed. If there exists consistency protocol entry but its state is neither "active" nor "pending", this action is vetoed immediately since no effective protocol is available for processing this action.

If a corresponding protocol exists and its state is either "active" or "pending", the client immediately deposits this action into the protocol instance for processing. If the action is determined to be blocking, the consistency protocol will not send out any resolution until after it resolves this action. If the action is confirmed, the protocol will emit a confirmation notification (to the application) and get it executed by the component store by forwarding the property change. Otherwise, a veto notification will be sent out. In this case, the property change does not need undone because it has not been executed yet.

For an action determined to be nonblocking, the protocol sends out a "pending" notification immediately and has it executed by forwarding the property change to the component store. After the action is eventually resolved positive, a confirmation will be sent to the application. Otherwise, the action will be undone and a veto notification is sent. In this case, the consistency protocol will also generate a compensation property change for the component store to undo the property change that it executed nonblockingly.

Confirmed actions of an "active" consistency protocol will be propagated to remote clients via the communication bus. After being received, they are pushed into the action queue for resolution and execution at remote sites. To avoid cyclic processing, these remote actions will not be propagated again.

Every time a property change is confirmed (and executed), the client will update the Seq# vector of the corresponding consistency protocol by increasing the element that corresponds to the originator of this property change. This is similar to the maintenance of state vectors in [60].

4. Meta Protocols

As shown in Figure 15, the protocol manager is fully replicated for local responsiveness. There is a need to maintain consistency when multiple users concurrently modify the property-protocol table. For example, two users may concurrently attach different consistency protocols to the same shared object. Therefore consistency protocols have to be used to determine which attachment wins. Since these protocols maintain consistency among replicas of framework objects instead of application objects, we call them meta (consistency) protocols. They control the attach and detach of (ordinary) consistency protocols to and from application data objects. For simplicity we assume that there are no communication failures.

Protocol attachment uses a two-phase commit (2PC) protocol to attach a consistency protocol. The EFG server acts as the protocol coordinator.

Phase one: On receiving an attach action from a client, the server sends all clients a query message to determine if they are "ready" for this new attachment. On receiving this query message, each client first checks the intention protocol table for conflicts. If a conflict is detected, the client will reply "not ready". If there is no conflict, the client will add an entry to the property-protocol table and create a protocol instance with "pending" state. After that, the client sends back a "ready" answer to the server.

Phase two: If all clients answered "ready", the server sends a "commit" message to all clients. On receiving the "commit" message, each client changes the state of

the corresponding protocol from "pending" to "active" and then invokes the *startResolution* method (refer to figure 2) of this protocol instance to enable processing of property changes. If not all clients answered "ready" within a prespecified time interval, however, the server sends an "abort" message to all clients that answered "not ready". On receiving the "abort" message, each client sets the protocol state to "vetoed" and removes the corresponding entry from the property-protocol table. In either cases, the intention protocol table will be modified accordingly, as explained in the previous subsection.

Protocol detachment is more complicated than protocol attachment. The complexity results from the requirement that a "quiescent" state must be reached within the protocol instances before the protocol is detached from all peer objects. That is, all peer objects must have executed exactly the same set of "confirmed" actions that have been generated at all sites during the lifecycle of this protocol. We use a three-phase commit (3PC) protocol for protocol detachment. By assuming no communication failures in the system, the 3PC protocol works similarly to 2PC as described above, except that in 3PC the consistency protocol has to internally reach a quiescent state first.

Phase one is for information collection. The process starts when a client sends the server a detachment request which piggybacks the "Seq#" vector of the data object. Recollect that "Seq#" is the number of state change actions that have been *confirmed* by the consistency protocol. On receiving this request, the server sends a "collect-info" message to all other clients. On receiving this message, each client will stop enqueuing further local operations on this object (into the protocol instance) by invoking its *suspendResolution* method (refer to figure 2). But the consistency protocol does not stop processing remote actions. After all local actions are processed by the current consistency protocol, the client sends its own "Seq#" back to server.

Phase two is the preparation phase. On receiving the "Seq#" from all clients, the server broadcasts them to all the sites. On receiving these "Seq#" of all other clients, a client can determine if it has executed all the actions originated from all other sites. If not, it will wait for the arrival of these remote actions, and then execute them. After a client finishes executing all these actions, it sends a "ready" message to server. When all clients finish executing all the required actions, they (and the protocol) reach the "quiescent" state.

Phase three is the commit phase. On receiving the "ready" message from all clients, the server will send all clients a "commit" message. On receiving "commit" message, each client will detach a protocol from an object. First client stops the consistency protocol from resolving user actions by invoking *stopResolution* method of this consistency protocol. Then it removes the entry from the consistency protocol manager and set the state of corresponding consistency protocol as "detached" in consistency protocol runtime.

D. Related Research

Greenberg and Marwood [37] are the first to our knowledge who motivate to support object-level consistency control such that different objects in the same workspace can be associated with different protocols. However, their work does not address how to achieve so. Specifically, there is no similar models of data and control, which we consider key to achieving the objective. Although data and control are separated in their work, the developer has to program in the application how the separated (locking) protocols work together with the data objects. Application-independent runtime mechanisms for "gluing" data and control are absent.

COAST [61] resembles our work in that it also models shared data objects in a

collaborative system as visual user interface objects that can be directly manipulated. However, it hard-codes consistency control protocols and does not address adaptable control as we do.

The ideas of multi-granularity and intention locking have long been established in databases [57]. Munson and Dewan [20] are the first to our knowledge to adapt these ideas into collaborative systems for flexible concurrency control. We further extend these ideas into general multi-granularity and intention protocols for the purposes of improving the flexibility and performance of adaptable consistency control. Note our use of multi-granularity protocols is at the user interface level. By comparison, the concept of multi-granularity locking in databases is in the database kernel and not exposed to the users.

Similar to multi-granularity and intention protocols, two-phase and three-phase commit protocols are also adapted from databases [57]. We are not claiming any innovation on these concepts. They are included for this chapter to be self-contained and for examining the feasibility of implementing adaptable consistency control, which is the main contribution claimed in this chapter.

In terms of supporting adaptable consistency control, Suite [20] provides parameterized access to the underlying (locking) protocols through a spreadsheet-like interface. Protocols are coupled with a custom data model (called active variables), which provides predefined data types such as sequence and record with embedded locking tables. Applications developed under the Suite framework automatically come with locking protocols if they construct shared data objects from these data types. Different locking policies are chosen at construction time by setting parameters in the locking table. However, Suite does not address how to dynamically adapt the system to use different consistency protocols that are not prescribed. Due to the lack of component-based programming support in its implementation language (C/C++),

data and control are tightly coupled.

Trellis [62], DCWPL [63] and our previous work on COCA [64, 65] also separates control from data: Data objects are wrapped by the application code, and the control part implements collaboration protocols including consistency control. These two parts communicate by exchanging custom messages. Protocols are specified by a custom language. It achieves adaptable system behavior by dynamic replacing protocols, at a different level of flexibility as compared to traditional spreadsheet-like approaches. However, it focuses more on modeling and enforcing general collaboration protocols, with limited automation and reusability. It requires the developer to carefully craft the application such that it is able to voluntarily notify the control part when the object state is changed, and to execute external commands from the control part to cause the desired object state changes. By comparison, the presented approach separates and externalizes the code for detecting and causing object state changes from specific data components. These becomes general services in the framework and can be reused with any application components that follow an industrial standard like JavaBean. More flexibility is achieved with significantly less programming efforts.

Roussev et al. [18] resembles our work in that it also takes a component-based approach and separates data and control. The shared state of a data component is modeled as JavaBean properties and state changes as property change events. However, their work in general has a different focus and does not address adaptable consistency protocols in particular. It also differs from ours in that the system has to periodically compare object states in order to detect state changes, which appears less efficient than our method of intercepting property changes.

Litiu and Prakash [51] and Hummes and Merialdo [50] provide system services to support the dynamic migration of application components between collaborating

sites. However, their goals are generally not to support adaptable consistency control and they do not deliberately differentiate data and control components.

Grundy and Hosking [9] is another recent framework for developing component-based groupware applications and it supports the plug-n-play of components. However, data and control are tightly coupled in those components. For example, the distributed editing component in [9] embeds a locking protocol. It is impossible to have different protocols coexist on different objects without a major redesign of the editing component.

DICIPLE [5] uses glass panes to intercept mouse and keyboard events generated from single-user applications (that are hosted by its runtime environment) and replicates these events to remote peers for synchronization. While DICIPLE focuses on transparent sharing of single-user (Java) applications, it does not address runtime plug-n-play of consistency protocols.

Chung and Dewan [66] have an observation on the subtle differences in object attributes that is similar to our differentiating of shared property changes in Section B. In their work, object attribute changes are logged differently based on the ways how these changes affect the object attributes – either "replacing" old value or doing "cumulative" changes. However their work largely focuses on supporting efficient later comer joining and does not address shared data modeling and dynamic consistency control issues.

E. Conclusions

Object-level consistency control is a feature motivated as early as in [37]. As also confirmed by numerous other researchers, to name but a few, [29, 7, 45], the capabilities to allow for different policies on different objects in the same workspace and

to support evolutionary policies in collaborative systems are important towards addressing the dynamic and situated nature of cooperative work. However, this has not been achieved in previous collaborative systems to our knowledge. Our hypothesis in this dissertation is that it is feasible to implement adaptable consistency control mechanisms in a range of collaborative workspace applications.

To test this hypothesis, we first propose a novel model in Section B, which cleanly separates data and consistency control protocols and defines their interfaces and interaction. Second we devised a novel consistency control framework in Section C that at run time "glues" together the data and protocols that are mutually transparent to each other. As a result, consistency protocols can be dynamically attached to data objects at the property, component, workspace, and subtree levels. We also addressed some performance issues that come with the new level of flexibility it achieves. The services provided in the framework are neutral to specific applications and thus reusable in a range of collaborative workspace applications that follow our data models. In immediate next chapter, we provide the evaluation for our approach.

CHAPTER V

AN INTEGRATED EMPIRICAL EVALUATION

Chapter III proposes a shared component model which forms the cornerstone for the groupware infrastructure as well as groupware applications. Based on this model, we provide a shared component converter tool for converting existing single-user components into shared components complying with the shared component model. Its objective is to maximize the reusability of existing single-user components in building groupware applications. In Chapter IV, we propose a coordination services framework that it built atop the shared component model. We focus on adaptable consistency control and explain in detail how to achieve dynamically switching of consistency protocols.

In this chapter we do an empirical evaluation of this work. The evaluation consists of two parts: (1).reusing single-user components to build groupware features and (2).supporting adaptable consistency control in collaborative tasks. For demonstrational purposes, we build a platform which can be used to test the shared components and the flexibility of coordination services such as adaptable consistency control. The organization of this chapter is as follows: in Section A, we introduce the demonstration platform - an **E**volvable and e**X**tensible **E**nvironment for **C**ollaboration (EXEC). In Section B, we evaluate the shared component model by leveraging single-user components into shared components and plugging them into the collaboration platform. These groupware components represent typical collaboration features that can be found in popular groupware applications as will be surveyed in this section. Then in Section C we use exemplar tasks to study how different consistency protocols can be applied in different collaboration scenarios.

A. Overview of the Collaboration Environment

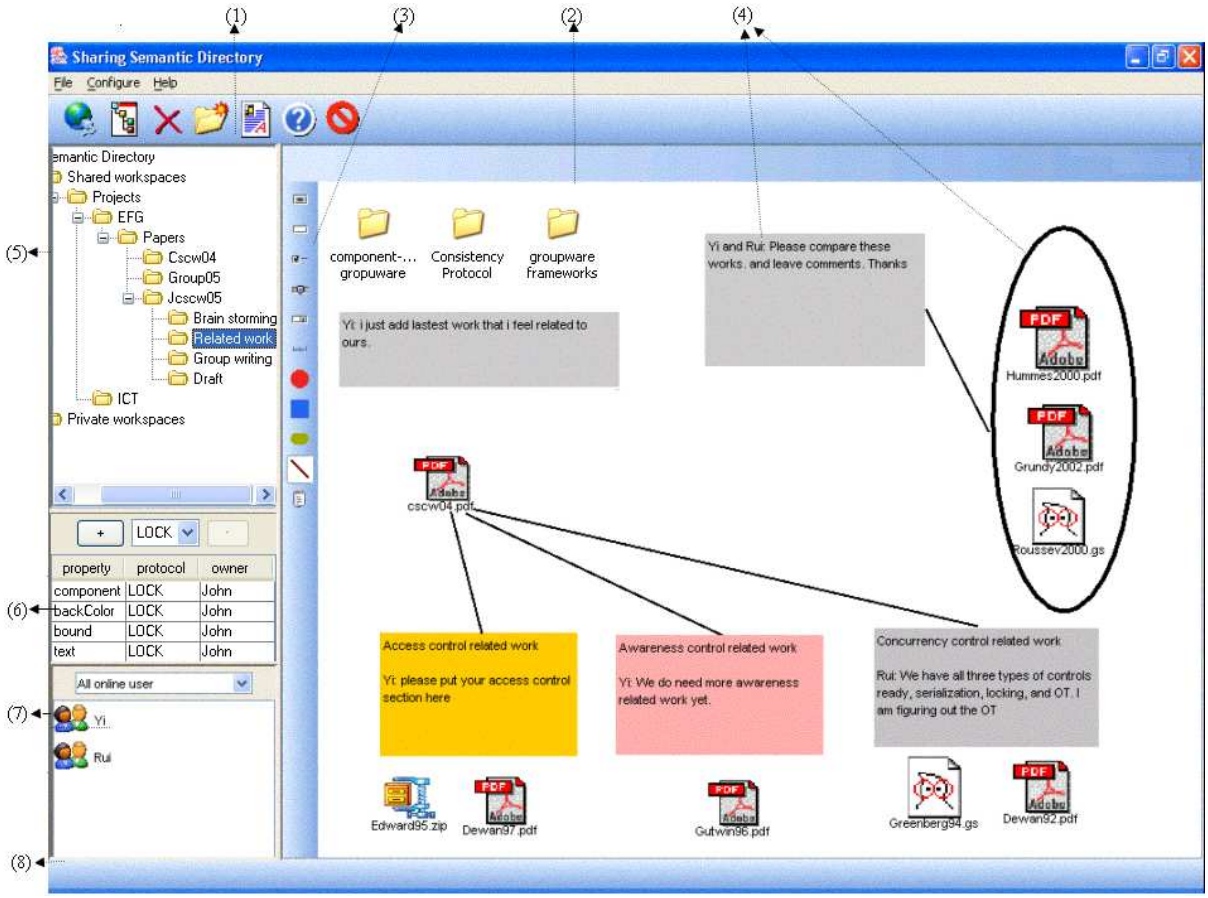

Fig. 17. EXEC screen shot: (1) general functions toolbar (2) workspace view (3) shared components toolbar (4) shared component (5) workspace hierarchy (6) component property-protocol table (7) presence awareness (8) status bar

The EXEC project aims to develop an extensible collaborative user interface. Figure 17 shows a screen shot of EXEC. On the upper left, a tree view displays the shared workspace hierarchy. On the right, a workspace view component displays the content of the currently selected workspace in the hierarchy. An implicit session control is provided by EXEC. By entering different workspaces in the hierarchy, users

implicitly exit one session and join another. The property-protocol table on the middle left displays which property of a selected shared component is currently controlled by which consistency protocol. The user can switch between different protocols on a shared property (component, workspace) by manipulating this table at run time. When the user selects a shared property in this table, the corresponding default consistency protocol for this property will be displayed on the protocol selection list. Users can accept the default protocol or switch to other available protocols by using the protocol list control. A presence awareness panel on the bottom left lists the current online users. For each user, a telepointer is provided to display his mouse trace on the shared workspace. The telepointer is a simulated mouse pointer tagged with the user name.

The main area of the EXEC GUI is the workspace view on the right-hand side. Its main function is to display the shared components in the selected workspace. Workspace view is a complex component with multiple layers of panels. The bottom of the workspace is a content panel which displays all shared components. A transparent glass panel is laid atop to intercept user mouse and keyboard inputs and then to re-dispatch these inputs to underlying shared components. The re-dispatching of events is important for these shared components to function properly. Another usage of glass panel is to display the components that must float above any other shared components in the workspace. For example, in order to indicate a component is "selected" after the user clicks on a shared component, several rectangular boxes are laid around this shared component to indicate the "selected" mode. A user can also drag these boxes to change the location and size of the selected shared component. The telepointer, which has the same requirements, is also displayed on the glass panel.

There are two toolbars in EXEC. The top horizontal toolbar serves as shortcut for a few general functions in the menu, e.g. displaying help information. The ver-

tical toolbar is called shared component toolbar, which displays all available shared components in this collaborative platform. Whenever the user clicks any icon on this toolbar, a new instance of corresponding component will be inserted into upper left corner of the selected workspace.Each time when EXEC starts, the shared component toolbar loader will try to load the available shared component description from an external XML configuration file. This capability is useful when testing the new shared components. Plugging-in new shared components does not need to modify the source code of the platform itself.

The **S**hared **S**emantic **D**irectory (SSD) is a concrete groupware application built atop the EXEC platform. In SSD, the shared components are also called semantic components. The semantic components toolbar contains document(file), text and graphics components. The look and feel of SSD mostly resemble familiar file system interfaces such as Microsoft Windows Explorer and Linux Konqueror. Shared Workspaces are organized in a hierarchy, as in [67]. Users can annotate documents with semantic objects, including notes (texts) and graphics. Semantic relationships between documents can be expressed explicitly by using semantic objects or implicitly as in spatial hypertext [68]. SSD allows the user to drag and drop files from local file system into the SSD workspaces. The files will be uploaded into centralized SSD server. A simple caching mechanism is provided to replicate the file locally when the user tries to open this file using a local file opener.

Traditional distributed file systems (e.g., [69, 70, 71]) provide limited support for cooperative work. While familiar LAN-based file systems such as NFS and Samba rely on locking for consistency control and coordination, SSD affords awareness (e.g., who are present and working on which objects [26]) and allows the users to experiment alternative consistency control protocols on shared objects.

The user experience of SSD in part resembles other typical workspace systems.

For example, the use of persistent hierarchical workspaces for fluid session control follows [72, 73], and the way users interact with shared objects follows [74]. One of the main focuses of our work is on supporting adaptable consistency control at property, object, and workspace levels, which is not addressed in previous work. The resemblance of SSD to classic workspace metaphors suggests the generality of our approach.

At its current stage, SSD is not intended to support the concurrent editing of the same file. Currently real-time group editors are available only for limited document types, e.g., [34]. We model the contents of documents as a content property in the file component and by default only allow locking protocols to be applied for exclusive access. Theoretically alternative consistency protocols are possible if corresponding group editors are available.

B. Shared Component Model Evaluation

The major design objective of shared component model is to enable the reuse of existing single-user components in building groupware application. In Chapter III, we have shown it is possible to blindly adapt all JDK components as shared components. In this section, we will adapt components which could be potential more useful in a practical multi-user environment. Our approach is to survey existing groupware applications and their common groupware features. Then we try to build these features by leveraging existing single-user components. However, not all needed single-user components corresponding to those needed groupware features can be found. Our solution to this problem is to build the single-user component by ourselves and then adapt them as the shared components. In this process, we record the effort and issues that arise when adapting components. This approach saves engineering costs because

developing single-user components is significantly easier than developing multi-user components.

1. A Survey of Groupware Products

There have been many groupware products and research prototypes in the market. The "groupware yellow page" [75] gives a comprehensive lists of up-to-date groupware products available. Our survey is based on this list. Below, we only list the most representative products and their features. Appendix A includes a complete list of groupware products that we surveyed.

a. Groove Virtual Office

The Groove Virtual Office is the latest groupware product by Groove Networks(recently acquired by Microsoft). It takes a replicated architecture to achieve better performance. The supported groupware features are as follows:

1.Co-Editing: Groove supports multi-user editing using Microsoft Word. An explicit turn-taking protocol is used for consistency control. The users take turns to make any changes to shared document. The document changes made by a user are transmitted to remote users when the user explicitly synchronizes the document.

2.Co-browsing: multiple people can browse web pages together. The multi-user browser supports relaxed-WYSIWIS mode, in which different users can view different portions of the same page. When a user clicks on a URL link or explicitly types in the destination URL, peer browsers are updated accordingly.

3.Group Sketching - multiple users can insert or delete graphic objects together. It serves as the most basic brain-storming tool. This tool uses a serialization protocol for concurrency control.

4.Picture - multiple users can upload and view images together. This tool does

not allow concurrent editing on the same image objects. It only uploads or removes the pictures from the image repository.

5.Note - similar to the picture tool, multiple users can add their own notes. Notably this tool does not support concurrent notes-taking. It only allows multiple users to insert or remove notes from the note repository.

6.Presentation - allow collaborators to use Microsoft PowerPoint to do shared presentation. When the turn-holder changes the current slide, all the collaborators will update the current slide. However, it does not support concurrent slides editing.

b. LiveMeeting/NetMeeting

Microsoft LiveMeeting is successor of Windows NetMeeting. It is an application sharing platform closely integrated with Microsoft products like office applications. It can share any windows applications using screen sharing mechanism, as in NetMeeting. Notably, a centralized architecture is used in collaboration. However, serialization protocol is used as the basic concurrency control mechanism for Microsoft Office applications, which is different from the turn-taking in NetMeeting. LiveMeeting also provides several built-in groupware features, including:

1.Whiteboard - multiple users can insert or delete graphic objects on the shared canvas. This tool is similar to sketching tool in Groove

2.Text - Multiple users can concurrently edit plain text in the text editor. Similar to the Groove co-editing tool, turn taking-is used to for consistency control.

3.Browser - Similar to the Groove Co-browsing tool, multiple user to view a web page together.

4.SnapShot - Similar to the Groove Picture tool, it allows multiple users to view the images together.

c. InstaColl

InstaColl converts Microsoft Office applications, e.g. Microsoft Word, Excel and PowerPoint - into collaborative applications. Different from Windows LiveMeeting, it takes a replicated architecture, and requires the Office applications installed on the local machines. Explicit turning taking is used as the collaboration protocol. For non Microsoft office applications, screen sharing is used. There is no additional built-in tools coming with InstaColl as in Groove or LiveMeeting.

d. CoWord/CoPowerPoint

Like InstaColl, CoWord[15] leverages existing popular Microsoft office applications into collaborative applications. Similar to InstaColl, CoWord takes replicated architecture for fast local response. The major difference between Co-word/and InstaColl is that CoWord uses the operational transformation [44] as the fundamental consistency maintenance mechanism, which allows for unconstrained collaboration.

e. Communiqu/Web-Ex

Web-Ex is a typical Web-based conferencing tool that supports synchronous collaboration among multi-users. It has very similar functions as Windows LiveMeeting, e.g., allowing for sharing a certain application or the host desktop. Turn-taking is the consistency protocol. There are many other similar Web-based conferencing products having similar functions, e.g. Gotomeeting, Helpmeeting, and Antaya BoardRoom 2.0.

f. CommunityZero

CommunityZero is a Web-based communityware. We purposely choose CommunityZero because collaboration in CommunityZero is asynchronous. However, it shares many similar features to synchronous groupware products. Its main collaboration features include:

1.Discussions - a threaded discussion board.

2.Note Board - announcements and informal discussions.

3.Shared Lists - collect and share structured information.

4.Calendar - to store time-sensitive information including meetings, reminders and project milestones.

5.File Sharing - store and share file.

6.Chat - allows any number of community members to have a group text meeting in realtime.

7.Community Messenger - A toolbar at the bottom of the screen includes a Who's on indicator that shows how many community members are online and accessible.

8.Member List - The Member List area is used to track and review community membership. The list indicates when members joined and their most recent visits. Access to detailed membership profiles is provided here.

g. Lotus Notes

IBM Lotus Notes provides an enterprise groupware platform. Its primary strength is to model the business working flow. Other important features include content management, web-conference, document sharing and white board session. Products with similar functions include Microsoft Exchange server and SharePoint server.

2. Building Example Groupware Features

There are many groupware products (see Appendix A for all surveyed products). However, they can be categorized along different dimensions, e.g., centralized v.s. replicated architecture, asynchronous v.s. synchronous collaboration. Based on their functions, they can be also categorized as meeting software, content management software, and business process management.

Most of these groupware products share a list of common features, e.g. **group editor**, **group sketch**, **group calendar**, **group browser**, and **group todo-list**. Hence we first evaluate how to build these groupware features by leveraging single-user components. For each of these features, we examine its usage in groupware applications and the strategy to build it, either from scratch or by adapting existing components.

a. Group Editor

Group editing is a classic research topic of groupware. Many specialized group editors, e.g., Groove [60], DistEdit [76], ShrEdit [43], Reduce [77], CoWord [34], have been built as research vehicles of different CSCW issues like consistency control.

Our objective is to adapt the existing Java Text component into a sharable component which comply with the shared component model. There have been three built-in text components coming with JDK(Java Development Kit): JTextField, JTextArea, and JTextPane. JTextField has the very basic editing function which supports one-line plain text editing. JTextArea supports multiple-line plain text editing. JTextPane is the most sophisticated component which has the capability of supporting both plain text document and styled text document such as HTML and RTF etc. In our experiement, we choose JTextPane.

Using the component adaptation tool introduced in Chapter III, we can directly convert the JTextPane component into the simplest form of shared component. Original properties, e.g. *x*, *y*, *width*, *height*, *background*, and even the *text* content itself, can be easily adapted as shared properties automatically. Then we plug this component into EXEC platform for testing.

A preliminary testing of this component reveals that some usability problems. First, different properties might need customized ways to change the value. For example, the *Background* or *Foreground* color property usually needs a special color selector to allow the user to choose its color intuitively. String-based properties might need a text editor to change their values.

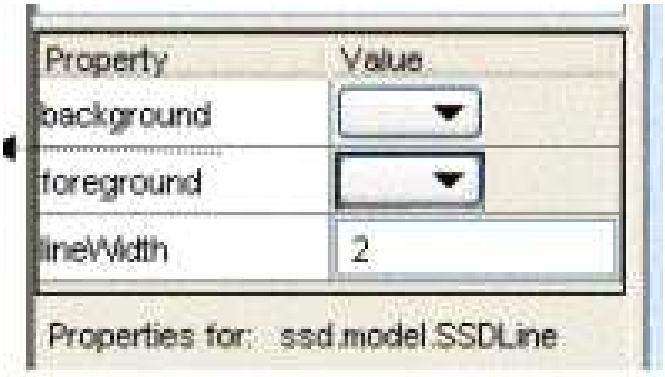

Fig. 18. General property editor

To address this problem, we provide a "Shared Property Editor" component (see Figure 18, which is adapted from Sun BeanBuilder), which incorporates editing capability for a few common properties, e.g. numeric, string-based, and object-based properties like Color. Nonetheless, there are two drawbacks in this approach: the shared properties might not be in the list of supported properties, and the property editor, mostly design for developers, is not always intuitive for end users. However,

this problem can not be solved generally at the framework level. Refined design could be applied in an application specific fashion.

The second problem is that changes to the shared *text* property can not be caught prior to its happening. Even though we successfully declared the *text* as an accumulative property by default adaptation, the shared property interception service seemingly failed to catch the user actions in a happen-before fashion, and more importantly, in an accumulative way. A deeper look into this issue reveal that the users' keyboard actions trigger text changes that actually did not happen directly at JTextPane component level. Instead, it first happened at the document model of JTextPane component. JTextPane component will be notified by its document model for the changes and actively retrieves the value from its document model as the value of text property. Hence the value of text property of JTextPane component is always changed whenever we catch the property notification. Moreover, the value that JTextPane retrieved from its document model is always a string, instead of the incremental character-wise change that we expect.

Fortunately, Java provides an indirect way for developers to catch the document changes before they actually happened. To do so, developers implement a class which implements an interface called *DocumentListener* and register this class to be the document change listener of the model of a JTextPane component. This interface contains three public methods, *insertString* and *remove*, and *update*, which provide a niche for developers to inject the code which will be executed right before the execution of actual user actions. If the injected code exits those three functions before user actions are executed, the model will never be changed. In our adaptation, the injected code only needs to fire out shared property change ("text") in these functions and then exit these functions. Because whether or not these document changes are allowed is subject to the collaboration protocols.

The above "hacking" process is complex because it requires the developers to open up the implementation detail of the JTextPane component. This process is seemingly contradictory to the design principles of CBD which emphasize on component encapsulation, that is, the components should interact with each other through well-defined interfaces and avoid relying on knowledge of internal implementation. In Java, however, this can not be achieved in some cases. The reason is that many JDK components separate their model and GUI delegates on purpose in order to reuse both. For example, the same JTextPane component can use different document models, e.g., RTF, HTML, and plain text. This separation makes it difficult for GUI delegates to wrap up the model events as their own events. Otherwise, the GUI delegates will be bound with one model and can not be reused by others. Notably, in the case of adapting JTextPane component, the actual manual adaptation is not difficult. The majority part of the manual adaptation of JTextPane component only takes around 60 lines of Java code to implement the DocumentListener class for the happen-before interception. Since *text* is declared as an accumulative property, a few lines of code is manually added to implement function signatures of insert and delete *text*.

b. Group Sketch

Group Sketch is another common collaboration tool. There are many groupware prototypes and products, e.g., GroupSketch [78], the graphic editor in RENDEZVOUS [79], NetDraw [80], collaborative white boards in COCA [81] and Grace [56]. In general, a group sketch tool allows multiple users to draw different types of graphic shapes on the canvas of a collaborative workspace. In our approach, implementing the group sketch is no different than implementing any other shared component. The real problem, however, is to find the corresponding single-user components. There are no built-in

Java graphic components coming with JDK. So our strategy is to build single-user graphic components that we need and then leverage them as shared components.

The building of graphic components in group sketch takes three steps. First, since all shape components share a lot of common properties. A base shape component is implemented with these properties such as *lineStyle*, *lineWidth*, *lineColor*, *filledColor*, and *bounds*. Bound property defines a minimum bounding box which can cover the shape. By manipulating the bounding box, position parameters such as location, width, and height of a shape can be changed in a shared workspace. Second, we then build different shape components by inheriting this basic shape component. The difference between them is small. The major difference is that different shapes have different paint function in order draw corresponding shapes. Of course, designer can add special properties to individual shapes. In the end, we directly leverage the shape components using the component adaptation tool.

The base shape component takes around 110 lines Java code to implement the setter and getter methods for common properties. Each inherited shape component merely overloads the *paint* function of the base shape component, which costs around 10 lines of code. In the end, the conversion adds up the template code to the shape component automatically. The building of group sketch tool demonstrates the case that when needed single-user components are not available, we can build them and then leverage them as the shared components. The main benefit is that developer still build the components the same as how they do for single-user applications.

c. Group Browser

Group browser provides a convenient way for multiple users to do web browsing together. It is essential to produce the same discussing context in a meeting when its content is online. Java components such as JEditorPane and JTextPane can support

Fig. 19. Group browser

displaying HTML documents either from local file system or online URLs. They both fully support HTML 3.2 and now are migrating to support HTML 4.0. However, there is no built-in Java web-browser component, which typical includes an address bar which allows the users to type in the new URLs to go to new web-pages or go back to previous visited pages. Our approach is to extend the JTextPane component to include these basic functions and leverage it as a shared component. Figure 19 gives a screen shot of a simple web browser built with JTextPane component.

The leveraging merely declare two properties, the *bound* and *currentURL*, as the shared properties. Whenever a user types in a new URL in the address bar, or clicks on the web links in a page, or clicks on the back button, the *currentURL* property will

be set to corresponding URL. For this prototype browser, we do not enforce different users watch the exact same position of a page, thus allow for relaxed-WYSIWIS mode of co-browsing. Building a single-user browser takes around 100 lines of Java code and in total adds up to around 220 lines of Java code after leveraging it to be a shared browser.

d. Group Calendar

Group calendar is another useful collaboration tool in groupware application. It has been studied since early 90's [82]. Now almost all meeting software and enterprise applications have some form of group calendar. The functions of group calendar is similar to its single-user counterpart, e.g., the calendar in Microsoft Outlook and various PDA organizers. Basically they allow collaborators to browse a list events in a selected date and collectively add or remove events. This is convenient for collaborators to detect the schedule conflicts. Notably group calendar is also a basic building block for advanced collaboration tools such as project tracking and web-blogging.

There is no in-built calendar component coming with JDK. So we switch to online resource for the single-user calendar component. We expect it could allow the user to select date to browse the tasks or to add events to or remove events from that date. A list of available event titles should be displayed for a selected date. If the user selects a interesting event title, the event detail should be displayed. It is quite easy to find a Java-Based calendar component with the most basic function, e.g. choosing a date. However, it is rather difficult to find a Java calendar component with additional functions we required. So our strategy is to extend the basic calendar component and incorporate events management functions.

Figure 20 shows the outlook of this component. The left hand side of this com-

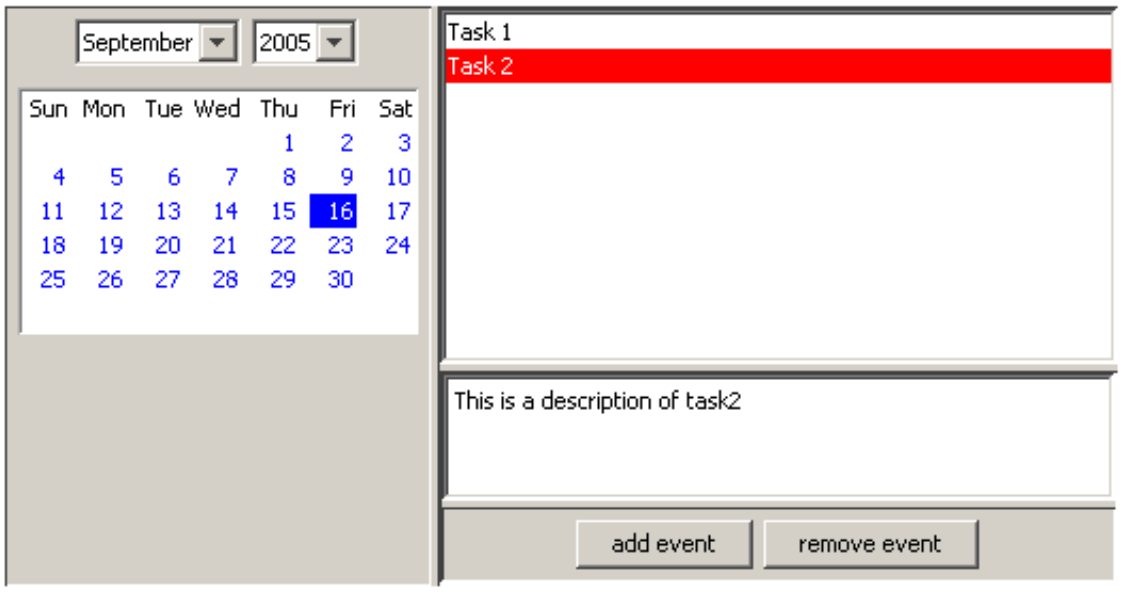

Fig. 20. Group calendar

ponent is the calendar that we adapted from third-party Java calendar. The right hand side is the event management functions that we developed. This component is relatively complex in its functions, and consumes around 700 lines of Java code. Half of the code (around 350 lines) is from third-party Java calendar component which implements date-picking function. Around 250 lines are for newly added event management functions such as displaying, adding, removing events associated with a specific date. The adaptation declares the *events* property of the self-built single-user component as an accumulative property, which requests developers manually adding a few lines codes to implement the function signatures of inserting and deleting *events*.

e. Group Todo-List

Group todo-list is another popular tool to organize people's activities with different form. The basic function of a todo-list component is similar to group calendar -

multiple users can browse the task list and add tasks to or remove tasks from the task list collectively. These tasks might or might not associate a date with an due date. Unlike calendar, which organizes the human activities by date, todo-list organizes the activities by themselves. The tasks of todo-list tends to be more emergent and less formally scheduled than the events in group calendar. Todo-list can be found on many different applications, e.g. organizer tools on PDA, smartPhone, and many personal and group productivity software.

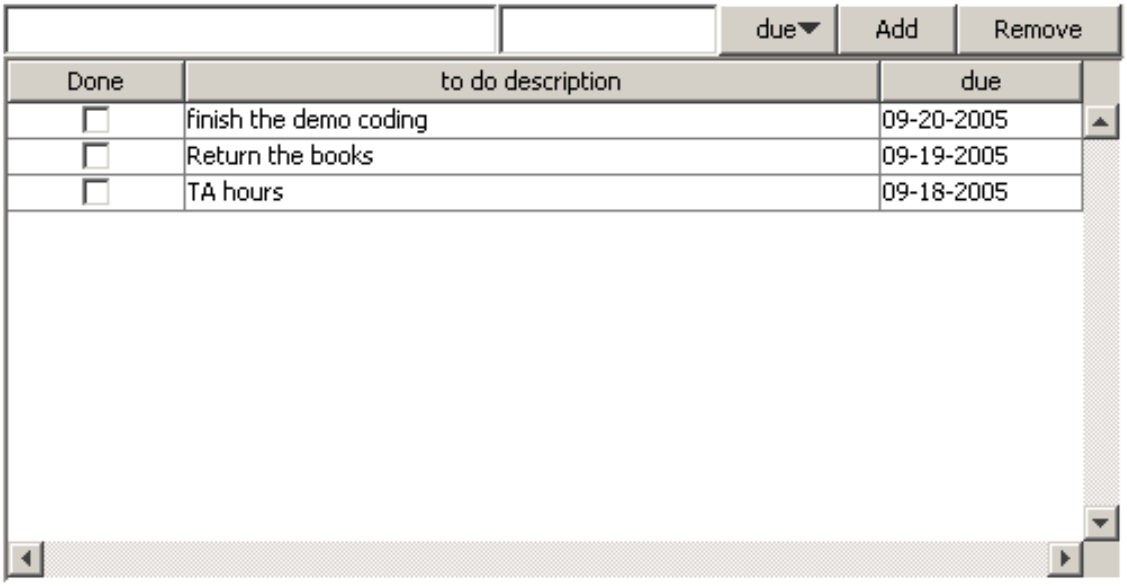

Fig. 21. Group todo list

Like calendar, there is no build-in components coming with JDK or a third party component that we match the functions that we expect. So we built group todo-list component from the scratch. Figure 21 shows a screen shot of this component which mostly resembles the todo-list in outlook. Building this single-user todo component used around 300 lines of Java Code. Then after the leveraging, it adds up total around 450 lines Java code. Similar to *events* property in group calendar component,

the adaptation declares the *tasks* property of todo-list component as an accumulative property. Developers needs to manually add a few lines code to implement the function signatures of inserting and deleting *tasks*.

f. Discussion

So far, we finished building the listed common group features(components) that we abstracted out from the survey of existing groupware products. These components can be used as individual workspace, thus extends the EXEC to become a full-feathered groupware application. They can also be used as individual components which resides in a shared workspace to form "compound" shared workspaces, as displayed in Figure 17. Table VII summarizes our adaptation results.

Table VII. Adaptation results

Group Feature	Single-user Component Source	Lines of Code manually added
group editor	JDK built-in (JTextPane)	120
group sketch	self-built	100 for base shape 10 additional lines for each shape
group calendar	extended from third-party component	250
ground browser	Extended from JTextPane	100 100
group todo	self-built	300

From above, we showed it is possible to adapt existing components to be shared components. The key is to find proper single-user components. However, if a needed

component is not available, building a shared component is usually divided into two steps: building a normal single-user component, and adapting it into a shared component. The adaptation process is mostly trivial unless the shared property resides in inner component of a container, e.g. the adaptation of JTextPane component. Currently we are extending the adaptation tool such that it can "look" into the component and adapt inner component properties as the shared properties more easily. Compared to the ad-hoc approach of building collaboration-aware components in Flexible-JAMM [4], our approach gives a clear guideline as how to develop the share component. Comparing to the approach of JView [9], the developer does not have to follow a heavy-weight class framework in developing share components. Instead, developers build their components as what they do for single-user application. These components can then be leveraged into shared components. Compared to [18], which also extends JavaBean naming convention, our approach has the benefits of happen-before event interception and higher system efficiency.

These groupware components are ready for flexible consistency controls. First, different shared properties of a shared component can be attached with different protocols by different collaborators. For example, the *bound* property of shared components, which reflects the position and size for these components in a shared workspace, can be attached with locking protocol by a collaborator in order to fix the layout of a shared workspace. Other properties, such as the *text* property in GroupEditor, the *children* property in GroupSketch tool, the *currentURL* property in group browser can be controlled by other collaborators to allow them to change the contents of the component. Also, consistency control can be applied by collaborator on different granularity, e.g., individual property, component, and workspace level. Second, different consistency protocols can be exercised on same component properties in different scenarios. For example, for all accumulative properties, such as the *text* property in

Group Editor, the *events* property in group calendar, and the *tasks* property in Group todo-list, can be attached with constrained protocols like lock for exclusive control or unconstrained protocols like operation transformation for concurrent manipulation.

All the above group components are finished using one person and in two weeks, which, of course, does not include the effort of building the EXEC infrastructure itself. Notably these groupware components serve as fast-prototyping purpose and testify the feasibility and validity of our shared component model. They would certainly take more lines of code if more comprehensive functionalities and polished user interfaces are required. However, these does not undermine the validity of the model.

C. Adaptable Consistency Protocol Evaluation

EXEC, as a collaboration platform, can be extended to support a range of collaboration tasks. In this section, we use SSD, an extension of EXEC environment, to describe how adaptable consistency protocol can support different collaboration scenarios in collaborative work.

1. Teaching Activities

Teaching is an interactive process including many collaborative activities and involving different roles such as instructors, teaching assistants, and students. These activities, depending on their natures, can be asynchronous or synchronous collaborations. Many artifacts such as course syllabus, discussion notes, reports, assignments, and exams, are generated in the course of the collaboration. In this subsection, we discuss how adaptable consistency protocol in SSD(see Figure 17) can be used to support these activities. SSD serves as content management tool as well as collaboration tool in this process.

First, instructors can set up corresponding root shared workspaces for courses they are teaching. In the root workspace for each course, a "course information" workspace is created. Inside, a calendar component is inserted. Important dates and events will be added into the course calendar. A web-browser component is inserted and displays the official description of this course. Other relevant information, e.g. notes and slides from the instructors, can be inserted to give detailed information about the courses. These material can be downloaded and viewed with local application opener by students. However, no one except the instructor can make changes to them. Thus Locking protocol is applied on this workspace by the instructor for protecting purpose.

A workspace named "Discussion" can be created underneath the root workspace for each course. This workspace serves as a free discussion board. Students can post any ideas, questions, even complaints on this board. Since it severs as a free discussion forum, a unconstrained consistency protocol is attached to workspace and its hierarchy. Students or instructors can create threaded discussion by creating additional workspaces inside the "discussion" workspaces. Graphic components can be used to indicate the relationship of the discussion notes. The collaboration mostly happen in a asynchronous manner which is similar the discussion forum.

Many courses have team projects, which require collaborative effort of a group of students. For such courses, a workspace named "team projects" is created inside of the root workspace of these courses. Multiple workspaces will be inserted inside "group projects", each of which corresponds to a specific group. Different names will be assigned accordingly. Each of these workspaces serves as the root team workspace for individual group. Inside of their own root team workspace, different group can create additional workspaces as needs arise. For example, each group member can have their own workspaces.

Normally a team member will be chosen as the group leader of a team project and all members will be assigned with roles and corresponding responsibilities. A group todo-list component can be inserted into by the leader into root team workspace. A face to face or online discussion will generate possible events and their due date in the todo-list. The group leader is responsible for tracking all the progress of the projects and making changes to the to-do list. A locking protocol is attached to this component by the group leader to make sure that only he can change the list contents. The same protocol can be attached to individual team member's workspace corresponding member so that they can work individually.

As the project deadline approaches, the group must come up with the final report for the team project, which should different sections assigned to different team members. A workspace called "final report" is created by coordinator and a unconstrained consistency protocol is first attached to this workspace. Every team member can copy their section of report into this workspace. Different note component can be be used to display the content of different sections. The group leader will compose them into an integrated report. Different team members will review the report at the same and make the necessary recommendations. Sometimes they can make direct changes to the report which presumably are small. unconstrained consistency protocol allows them working together on the same note component.

In the end, if every team member agrees, the integrated report will be sealed by the group leader attaching a locking protocol to it. After the deadline, the whole team project workspace hierarchy will be applied with locking consistency protocol by instructor so no one can make any change any more.

2. Paper Writing Activities

Paper writing is another typical group activity in the research community. It usually involves multiple people in the process. The writing process is often divided into several stages such as kickoff, brainstorming, related work analysis, individual writing and revision [83, 84]. Some of phase needs the people closely collaborate with each other and some phases do not. One of the coauthors may take the lead by assuming a moderator role to coordinate the whole process. Using SSD, different consistency control protocols can be applied to address the evolving needs in different stages of paper writing.

During the kickoff phase of the paper writing, a workspace for this paper writing project is first created, with the name of corresponding conference or journal. This workspace serves as the root workspace of this writing task. Every collaborator joins in this workspace. A calendar component is first inserted by the moderator into this workspace which is going to mark all the important dates and corresponding milestones for this writing project. For example, When different phase should be finished and who should do what in these phases. This calendar defines the writing time line and individual responsibilities. In this process, only moderator can change the schedule so the calendar will be locked by the moderator. Collaborators exchange their thoughts and feedback through instance messaging software. After setting up milestones of the paper writing task, the collaboration moves on the next phase - brainstorming.

During the brainstorming stage, a "brainstorming" workspace is created in the root workspace. Depending the task itself, a todo list component might be inserted into this workspace and more detailed activities which should be done in this phase are listed and initially marked "not finished" yet. The main focus of this phase is

to determine what to write. In order to come up with as many ideas as possible, an unconstraining consistency control protocol (e.g., operational transformation [44, 33]) is attached to this workspace so that any coauthor can contribute at any time [43, 4]. Then, after carefully discussing and evaluating all candidate ideas, the topic and abstract of this paper are chosen and the brainstorming phase is ended. To this point, the whole "brainstorming" workspace together with artifacts generated during the course of discussion are attached with a constraining consistency control protocol (e.g., locking) to prevent unintentional damages. Only the moderator might be able to make changes in this workspace.

Then the task moves to the next phase in which coauthors collect and analyze related works. A research paper is typically related to the literature from multiple different aspects. A "related work" workspace is created to allow coauthors to collect materials and perform analyses in this shared workspace. Since the relevance of related works is often a result of articulation work, this workspace is also attached with an unconstraining consistency protocol so that collaborators are not refrained from contributing. Eventually the resulted materials are grouped into different categories based on how they are related and sub-workspaces are created under the "related work" workspace. For example, to write this paper itself, we created such related work workspaces as "component-based groupware", "consistency control", "groupware frameworks" to address the various aspects of this work. At this point, coauthors agree to each work on some (but not all) of these different aspects. As a result, a more constraining protocol is attached to each sub-workspace.

In the "group writing" phase, the moderator sets up an outline (plan) of the paper and then assigns sections to coauthors, e.g., as a result of negotiation. A "draft" workspace is created which initially only contains a note object holding the outline and a few empty documents to hold individual sections. Only the moderator

is allowed to modify the paper outline since any change to it may impact the sections that have been agreed upon. Similarly only "owners" of the sections can modify the contents of corresponding documents so that coauthors work in parallel and do not interfere with each other. However, all the coauthors are allowed to create new objects in the "draft" workspace, e.g., to comment on other people's write-ups and to create figures. Meanwhile the whole "related work" hierarchy is open to all coauthors for them to evolve their understandings of related works abreast of the paper writing. At this stage, modifications to any part of the "related work" hierarchy will unlikely be intensive. An unconstraining protocol here would save the coordination costs of constraining protocols.

After coauthors mostly finish their assigned sections, the task enters a revision phase in which the individual pieces are smoothed out. A constraining protocol is posed on the "draft" workspace such that coauthors take turns to make modifications. During the course of revision, however, the "draft" workspace or some objects may be re-openned from time to time for other coauthors to contribute, e.g., when a section needs rewriting or comments from its original author. Sometimes it is necessary to have discussions or help from colleagues if the current floor holder needs to reorganize sections. Editing actions as such are often highly situated and it is generally difficult to predict before hand how the control policies should be like [45, 29].

3. Summary

Even though we only use typical teaching and research activities to demonstrate the usage of different consistency protocols in different collaboration scenarios, their nature is shared by many other group activities. These activities contain collaboration scenarios which require different consistency protocols. Due to the separation of data and control in our infrastructure, applications like SSD could accommodate these

changes easily. Compared to existing research prototypes and commercial products, SSD demonstrates unique flexibility in its usage. First of all, SSD can accommodate any shared components. Second, different components in the shared workspace can be associated with different consistency protocols. The granularity and association of consistency protocols can be dynamically changed in order to satisfy different collaboration needs. This flexibility has not been seen in any research prototypes to our best knowledge.

CHAPTER VI

CONCLUSION AND FUTURE WORK

A. Summary of Dissertation

Previous collaboration transparency systems reuse existing single-user applications for collaborative work. They generally achieve high level of reusability since no source code of the single-user applications needs to be modified. However, they often suffer from flexibility and performance problems. Previous collaboration awareness systems provide reusable coordination services and multi-user widgets to reduce the costs of developing specialized groupware applications. However, they often do not provide guidelines as how to reuse existing single-user components in constructing groupware widgets. The tight binding between data and coordination services often leads to degraded flexibility and reusability.

In this dissertation, we propose a component-based approach to developing groupware applications. We propose a shared component model for modeling data and graphic user interface (GUI) components of groupware applications(Chapter III). Due to the simplicity of the share component model, the myriad of existing single-user components can be re-purposed as shared GUI or data components. An adaptation tool is built to assist the adaptation process.

We propose a coordination service framework(Chapter IV) with several reusable coordination services such as data distribution, persistence, and adaptable consistency control. The key for our coordination services to achieve improved flexibility over previous work is the clean separation of data and coordination services and the capability to dynamically "glue" them together. By doing so, users can dynamically switch collaboration protocols in order to support evolving coordination needs.

An **E**volvable and e**X**tensible **E**nvironment for **C**ollaboration (EXEC) is built to evaluate our approach(Chapter V). In the first part of evaluation, we survey popular groupware products and their common features. Single-user components, either acquired from open source or built in-house, are adapted into shared components with relatively minor effort. By plugging in these shared components into EXEC, the environment is extended to support different collaborative tasks. In the second part of evaluation, we evaluate the adaptable coordination services by showing that in different phases of exemplar collaborative tasks, different consistency control policies can be applied to support evolving collaboration needs.

B. Future Directions

There are several possible directions to extend our current work. First, there are still some system performance issues to be addressed in the presented work. Currently consistency protocols are implemented in threads. This does not scale well with the number of protocol instances in the system. When many fine-grained protocols are running, e.g., at the property level, the system resources may be exhausted. Although the use of aggregate properties mitigate this problem, there is a space for improvement. One possible way is to only attach consistency protocols to objects that are really shared (viewed or modified) by at least two users, which ultimately depends on the timeliness of awareness information. Another direction is to run only one protocol instance for each type of consistency control protocol, which may pose new requirements on the design of consistency protocols.

Second, this research tackles more fundamental usability issues such as flexibility of coordination. These issues are at the system level and we have performed corresponding evaluation experiments. However, we have not done any usability study at

the application level. Our current support of default and implicit consistency protocols is a promising yet initial step towards addressing usability issues of the adaptable consistency control mechanisms. We plan to refine and extend this design in future work. Ideally the system should be adaptive by making some protocol decisions for the users in some situations and suggesting some decisions to the users in some other situations. We will explore these issues in specific applications.

Third, other coordination services such as access control and awareness control are critical in building real-world groupware applications. How to model these services in our shared component model and how these new services interact with existing services are important issues to be investigate in future work.

REFERENCES

[1] C. A. Ellis, S. J. Gibbs, and G. Rein, "Groupware: Some issues and experiences," *Commun. ACM*, vol. 34, no. 1, pp. 39–58, 1991.

[2] J. Grudin, "Groupware and social dynamics: Eight challenges for developers," *Commun. ACM*, vol. 37, no. 1, pp. 92–105, 1994.

[3] J. C. Lauwers, T. A. Joseph, K. A. Lantz, and A. L. Romanow, "Replicated architectures for shared window systems: A critique," in *Proc. of the Conf. on Office Information Systems*, Cambridge, MA, 1990, pp. 249–260.

[4] J. Begole, M. B. Rosson, and C. A. Shaffer, "Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 6, no. 2, pp. 95–132, 1999.

[5] W. Li, W. Wang, and I. Marsic, "Collaboration transparency in the disciple framework," in *GROUP '99: Proc. of the International ACM SIGGROUP Conf. on Supporting Group Work*, Phoenix, AZ, 1999, pp. 326–335.

[6] Microsoft Corporation, "NetMeeting," http://www.microsoft.com /windows/netmeeting/, Nov. 2005.

[7] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar, "WYSIWIS revised: Early experiences with multiuser interfaces," *ACM Trans. Inf. Syst.*, vol. 5, no. 2, pp. 147–167, 1987.

[8] P. Dewan and R. Choudhary, "A high-level and flexible framework for implementing multiuser user interfaces," *ACM Trans. Inf. Syst.*, vol. 10, no. 4, pp. 345–380, 1992.

[9] J. Grundy and J. Hosking, "Engineering plug-in software components to support collaborative work," *Journal of Software Practice and Experience*, vol. 32, no. 10, pp. 983–1013, Aug. 2002.

[10] R. W. Hall, A. Mathur, F.Jahanian, A. Prakash, and C. Rassmussen, "Corona: A communication service for scalable, reliable group collaboration systems," in *CSCW '96: Proc. of the 1996 ACM Conf. on Computer Supported Cooperative Work*, Boston, MA, 1996, pp. 140–149.

[11] M. Roseman and S. Greenberg, "Building real-time groupware with groupkit, a groupware toolkit," *ACM Trans. Comput.-Hum. Interact.*, vol. 3, no. 1, pp. 66–106, 1996.

[12] J. Grudin, "Computer supported cooperative work: History and focus," *Journal of IEEE Computer*, vol. 27, no. 5, pp. 19–26, 1994.

[13] H. Abdel-Wahab and M. Feit, "XTV: A framework for sharing x window clients in remote synchronous collaboration," in *Proc. of IEEE Tricomm '91*, Chapel Hill, NC, Apr. 1991, pp. 159–167.

[14] D. Li and R. Li, "Transparent sharing and interoperation of heterogeneous single-user applications," in *CSCW '02: Proc. of the 2002 ACM Conf. on Computer Supported Cooperative Work*, New Orleans, LA, 2002, pp. 246–255.

[15] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen, "Leveraging single-user applications for multi-user collaboration: The coword approach," in *CSCW '04: Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, Chicago, IL, 2004, pp. 162–171.

[16] L. Cheng, S. L. Rohall, J. Patterson, S. Ross, and S. Hupfer, "Retrofitting

collaboration into uis with aspects," in *CSCW '04: Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, Chicago, IL, 2004, pp. 25–28.

[17] J. Lu, R. Li, and D. Li, "A state difference based approach to sharing semi-heterogeneous single-user editors," in *the 6th International Workshop on Collaborative Editing Systems, Jointly with ACM CSCW'04*, Nov. 2004, (4 pages).

[18] V. Roussev, P. Dewan, and V. Jain, "Composable collaboration infrastructures based on programming patterns," in *CSCW '00: Proc. of the 2000 ACM Conf. on Computer Supported Cooperative Work*, Philadelphia, PA, 2000, pp. 117–126.

[19] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, "MMConf: An infrastructure for building shared multimedia applications," in *CSCW '90: Proc. of the 1990 ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, 1990, pp. 329–342.

[20] J. Munson and P. Dewan, "A concurrency control framework for collaborative systems," in *CSCW '96: Proc. of the 1996 ACM Conf. on Computer Supported Cooperative Work*, Boston, MA, 1996, pp. 278–287.

[21] A. Prakash and H. S. Shim, "Distview: Support for building efficient collaborative applications using replicated objects," in *CSCW '94: Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, 1994, pp. 153–164.

[22] J. B. Begole, "Flexible collaboration transparency: Supporting worker independence in replicated application sharing systems," Ph.D. dissertation, Department of Computer Science, Virginia Tech. Blacksburg, VA, 1998.

[23] P. Dewan, "An integrated approach to designing and evaluating collaborative

applications and infrastructures," *Computer Supported Cooperative Work: The Journal of Collaborative Computing (JCSCW)*, vol. 10, no. 1, pp. 75–111, 2001.

[24] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, 1998.

[25] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *CSCW '92: Proc. of the 1992 ACM Conf. on Computer Supported Cooperative Work*, Toronto, Ontario, Canada, 1992, pp. 107–114.

[26] C. Gutwin and S. Greenberg, "A descriptive framework of workspace awareness for realtime groupware," *Computer Supported Cooperative Work: The Journal of Collaborative Computing (JCSCW)*, vol. 11, no. 1-2, pp. 411–446, 2002.

[27] P. Dewan and H. Shen, "Controlling access in multiuser interfaces," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 34–62, 1998.

[28] W. K. Edwards, "Session management for collaborative applications," in *CSCW '94: Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, 1994, pp. 323–330.

[29] W. K. Edwards, "Policies and roles in collaborative applications," in *CSCW '96: Proc. of the 1996 ACM Conf. on Computer Supported Cooperative Work*, Boston, MA, 1996, pp. 11–20.

[30] W. Emmerich and N. Kaveh, "Component technologies: Java beans, com, corba, rmi, ejb and the corba component model," in *ICSE '02: Proc. of the 24th International Conf. on Software Engineering*, Orlando, FL, 2002, pp. 691–692.

[31] W. Emmerich, "Distributed component technologies and their software engineering implications," in *ICSE '02: Proc. of the 24th International Conf. on Software Engineering*, Orlando, FL, 2002, pp. 537–546.

[32] Y. Yang and D. Li, "Separating data and control: Support for adaptable consistency protocols in collaborative systems," in *CSCW '04: Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, Chicago, IL, 2004, pp. 11–20.

[33] D. Li and R. Li, "Preserving operation effects relation in group editors," in *CSCW '04: Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, Chicago, IL, 2004, pp. 457–466.

[34] D. Sun, S. Xia, C. Sun, and D. Chen, "Operational transformation for collaborative word processing," in *CSCW '04: Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, Chicago, IL, 2004, pp. 437–446.

[35] J. Penix and P. Alexander, "Toward automated component adaptation," in *Proc. of the 9th International Conf. on Software Engineering and Knowledge Engineering*, Madrid, Spain, 1997, pp. 535–542.

[36] B. Morel and P. Alexander, "Automating component adaptation for reuse," in *Proc. of the 18th IEEE International Conf. on Automated Software Engineering*, Montreal, Canada, Oct 2003, pp. 142 – 151.

[37] S. Greenberg and D. Marwood, "Real time groupware as a distributed system: Concurrency control and its effect on the interface," in *CSCW '94: Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, 1994, pp. 207–217.

[38] S. Bhola, G. Banavar, and M. Ahamad, "Responsiveness and consistency tradeoffs in interactive groupware," in *PODC '98: Proc. of the 17th Annual ACM Symposium on Principles of Distributed Computing*, Puerto Vallarta, Mexico, 1998, p. 324.

[39] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer, "Designing and implementing asynchronous collaborative applications with bayou," in *UIST '97: Proc. of the 10th Annual ACM Symposium on User Interface Software and Technology*, Banff, Alberta, Canada, 1997, pp. 119–128.

[40] A. Prakash and M. J. Knister, "A framework for undoing actions in collaborative systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 4, pp. 295–330, 1994.

[41] C. Sun, "Optional and responsive fine-grain locking in Internet-based collaborative systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 9, pp. 994–1008, Sept. 2002.

[42] C. Sun, "Undo as concurrent inverse in group editors," *ACM Trans. Comput.-Hum. Interact.*, vol. 9, no. 4, pp. 309–361, 2002.

[43] C. M. Hymes and G. M. Olson, "Unblocking brainstorming through the use of a simple group editor," in *CSCW '92: Proc. of the 1992 ACM Conf. on Computer Supported Cooperative Work*, Toronto, Ontario, Canada, 1992, pp. 99–106.

[44] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *CSCW '98: Proc. of the 1998 ACM Conf. on Computer Supported Cooperative Work*, Seattle, WA, 1998, pp. 59–68.

[45] Lucky Suchman, *Plans and Situated Actions*, Cambridge University Press, Cambridge, UK, 1987.

[46] J. Grudin, "Why CSCW applications fail: Problems in the design and evaluation of organization of organizational interfaces," in *CSCW '88: Proc. of the 1988 ACM Conf. on Computer Supported Cooperative Work*, Portland, OR, 1988, pp. 85–93.

[47] D. Li, L. Zhou, R. R. Muntz, and C. Sun, "Operation propagation in real-time group editors," *IEEE MultiMedia*, vol. 7, no. 4, pp. 55–61, 2000.

[48] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris, "Computer support for distributed collaborative writing: Defining parameters of interaction," in *CSCW '94: Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, 1994, pp. 145–152.

[49] T. C. N. Graham, C. A. Morton, and T. Urnes, "Clockworks: Visual programming of component-based software architectures," *Journal of Visual Languages and Computing*, vol. 7, no. 2, pp. 175–196, July 1996.

[50] J. Hummes and B. Merialdo, "Design of extensible component-based groupware," *Computer Supported Cooperative Work: The Journal of Collaborative Computing (JCSCW)*, vol. 9, no. 1, pp. 53–74, 2000.

[51] R. Litiu and A. Parakash, "Developing adaptive groupware applications using a mobile component framework," in *CSCW '00: Proc. of the 2000 ACM Conf. on Computer Supported Cooperative Work*, Philadelphia, PA, 2000, pp. 107–116.

[52] C. Szyperski, "Component technology: What, where, and how?," in *ICSE '03: Proc. of the 25th International Conf. on Software Engineering*, Portland, OR,

2003, pp. 684–693.

[53] H. Shen and C. Sun, "Flexible notification for collaborative systems," in *CSCW '02: Proc. of the 2002 ACM Conf. on Computer Supported Cooperative Work*, New Orleans, LA, 2002, pp. 77–86.

[54] A. H. Davis, C. Sun, and J. Lu, "Generalizing operational transformation to the standard general markup language," in *CSCW '02: Proc. of the 2002 ACM Conf. on Computer Supported Cooperative Work*, New Orleans, LA, 2002, pp. 58–67.

[55] J. P. Munson and P. Dewan, "A flexible object merging framework," in *CSCW '94: Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, 1994, pp. 231–242.

[56] C. Sun and D. Chen, "Consistency maintenance in real-time collaborative graphics editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 9, no. 1, pp. 1–41, 2002.

[57] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, New York, NY, 1987.

[58] A. Clement, "Cooperative support for computer work: A social perspective on the empowering of end users," in *CSCW '90: Proc. of the 1990 ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, 1990, pp. 223–236.

[59] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, Cambridge, MA, 1993.

[60] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in

SIGMOD '89: Proc. of the 1989 ACM SIGMOD International Conf. on Management of Data, Portland, OR, 1989, pp. 399–407.

[61] C. Schuckmann, J. Schümmer, and Peter Seitz, "Modeling collaboration using shared objects," in *GROUP '99: Proc. of the International ACM SIGGROUP Conf. on Supporting Group Work*, Phoenix, AZ, 1999, pp. 189–198.

[62] R. Furuta and P. D. Stotts, "Interpreted collaboration protocols and their use in groupware prototyping," in *CSCW '94: Proc. of the 1994 ACM Conf. on Computer Supported Cooperative Work*, Chapel Hill, NC, 1994, pp. 121–131.

[63] M. Cortes and P. Mishra, "DCWPL: A programming language for describing collaborative work," in *CSCW '96: Proc. of the 1996 ACM Conf. on Computer Supported Cooperative Work*, Boston, MA, 1996, pp. 21–29.

[64] D. Li and R. Muntz, "COCA: Collaborative objects coordination architecture," in *CSCW '98: Proc. of the 1998 ACM Conf. on Computer Supported Cooperative Work*, Seattle, WA, 1998, pp. 179–188.

[65] D. Li and R. R. Muntz, "Runtime dynamics in collaborative systems," in *GROUP '99: Proc. of the International ACM SIGGROUP Conf. on Supporting Group Work*, Phoenix, AZ, 1999, pp. 336–345.

[66] G. Chung, P. Dewan, and S. Rajaram, "Generic and composable latecomer accommodation service for centralized shared systems," in *IFIP Working Conf. on Engineering for Human-Computer Interaction (EHCI)*, Knossos Royal, Crete, Sept. 1998, pp. 129–147.

[67] P. Dourish, J. Lamping, and T. Rodden, "Building bridges: Customisation and mutual intelligibility in shared category management," in *GROUP '99: Proc. of*

the International ACM SIGGROUP Conf. on Supporting Group Work, Phoenix, AZ, 1999, pp. 11–20.

[68] F. M. Shipman, C. C. Marshall, and T. P. Moran, "Finding and using implicit structure in human-organized spatial layouts of information," in *CHI '95: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, Denver, CO, 1995, pp. 346–353.

[69] R. Guy, J. Heidemann, W. Mak, G. Popek, and D. Rothmeier, "Implementation of the ficus replicated file system," in *Proc. Summer 1990 USENIX Conf.*, Berkeley, CA, 1990, pp. 63–72.

[70] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," in *SOSP '91: Proc. of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1991, pp. 213–225.

[71] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *SOSP '95: Proc. of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, 1995, pp. 172–182.

[72] M. Roseman and S. Greenberg, "Teamrooms: Network places for collaboration," in *CSCW '96: Proc. of the 1996 ACM Conf. on Computer Supported Cooperative Work*, Boston, MA, 1996, pp. 325–333.

[73] J. H. Lee, A. Prakash, T. Jaeger, and G. Wu, "Supporting multi-user, multi-applet workspaces in CBE," in *CSCW '96: Proc. of the 1996 ACM Conf. on Computer Supported Cooperative Work*, Boston, MA, 1996, pp. 344–353.

[74] Jr. Dan R. Olsen, Thomas G. McNeill, and D. C. Mitchell, "Workspaces: An

architecture for editing collections of objects," in *CHI '92: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, Monterey, CA, 1992, pp. 267–272.

[75] Foraker Design Cooperation, "Groupware products survey," http://www.usabilityfirst.com/groupware/cscw.txl#products, Nov. 2005.

[76] M. J. Knister and A. Prakash, "DistEdit: A distributed toolkit for supporting multiple group editors," in *CSCW '90: Proc. of the 1990 ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, 1990, pp. 343–355.

[77] Y. Yang, C. Sun, Y. Zhang, and X. Jia, "Real-time cooperative editing on the Internet," *IEEE Internet Computing*, vol. 4, no. 3, pp. 18–25, 2000.

[78] S. Greenberg, M. Roseman, and D. Webster, "Issues and experiences designing and implementing two group drawing tools," in *Proc. of the 25th Annual Hawaii International Conf. on the System Sciences*, Maui, HI, Jan. 1992, pp. 139–250.

[79] T. Brinck and R. D. Hill, "Building shared graphical editors using the abstraction-link-view architecture," in *Proc. of the European Conf. on Computer Supported Cooperative Work (ECSCW'93)*, Milan, Italy, Sept. 1993, pp. 311–324.

[80] D. Qian and M. D. Gross, "Collaborative design with netdraw," in *Proc. of CAAD Futures 1999 Conf.*, Georgia Institute of Technology, Atlanta, GA, 1999, pp. 213–226.

[81] D. Li, "COCA: A framework for modeling and supporting flexible and adaptable synchronous collaborations," Ph.D. dissertation, Department of Computer Science, University of California, Los Angeles, CA, June 2000.

[82] D. Beard, M. Palaniappan, A. Humm, D. Banks, A. Nair, and Y. Shan, "A visual calendar for scheduling group meetings," in *CSCW '90: Proc. of the 1990 ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, 1990, pp. 279–290.

[83] I. R. Posner and R. M. Baecker, "How people write together," in *Proc. of the 25th Annual Hawaii International Conf. on System Sciences*, Maui, HI, 1992, pp. 127–138.

[84] S. Noel and J. Robert, "Empirical study on collaborative writing: What do co-authors do, use, and like?," *Computer Supported Cooperative Work: The Journal of Collaborative Computing (JCSCW)*, vol. 13, no. 1, pp. 63–89, 2004.

APPENDIX A

GROUPWARE PRODUCTS SURVEY LIST

Action Technologies modeling methodology and design tools for BPR

Antarya Web collaboration application suite

Attask Project Management Software

Avalon Business Systems Lotus Notes groupware development

Axista.com, Xcolla web-based collaborative project management software

bizOA messaging and groupware solution

Blackboard web-based courseware, support for collaborative classrooms

BPS Project management software, business process automation

Communique Web Conferencing Web conferencing solutions

CommunityZero web-based community development and hosting services

Cybozu web-based office groupware running on a LAN, a variety of applications

DCASoft makes BrightSuite KM and collaboration software that allows a corporation to deploy its entire knowledge base

Deep Woods consulting firm specializing in organizational technology and culture

Foraker Design usability consulting firm with ample experience in groupware and website design, offering both user interface design and usability evaluation

eBeam turns whiteboards into digital collaborative workspaces using infrared and ultrasonic technology

eGroupware Enterprise collaboration suite

Business Collaborator collaborative knowledge management system

EPIware an efficient portal solution allowing any size organization to easily share information and effectively collaborate on documents in a browser-based environment

Facilitate.com virtual internet meeting area supporting discussions with various tools such as brainstorming, organizing, voting, surveying, or chat.

Ferris Research publications on messaging

GFI Communications email based workflow software

BSCW a web-based shared workspace, The Social Web

Group Systems E-collaboration software to enable workgroup success by combining technology, methodologies, and expert services

GroupMind Express set of online work tools that connect people across geography, functions and time

GroupVille a web-based collaboration solution

HelpMeeting.com data conferencing service

iCohere provides a collaborative web environment that integrates knowledge management and collaboration tools with principles of group dynamics and learning

ILINC a collaborative learning system

Infowit web-based project management software solution

Inovie Software TeamCenter, a real-time collaborative project management system

INTERnetOFFICE web-based GroupWare solutions for today's small to mid size companies

JDH Technologies distance learning and collaboration environment

KMtechnologies a simple and flexible environment for the instant setup of light multilingual Intranets and Extranets

Level 8 Systems messaging tools and component-based enterprise integration frameworks

Lotus Notes enterprise working flow system

Lucane Free Collaborative Platform

Microsoft Exchange, NetMeeting

OPMcreator a web based team collaboration system

phpGroupWare multi-user web-based groupware suite written in PHP which also provides an API for developing additional applications

PicturePhone videoconferencing

PictureTalk cross-platform visual conferencing

POLYCOM videoconferencing

projectplace.com Web service for project collaboration that includes shared document archives, discussion forums, task lists, shared calendars etc.

SoftArc a multiplatform electronic mail and group collaboration product

Teamsoft a cross platform group scheduler

Teamspace Online service for teamwork and collaboration

ThinkVirtual delivers advanced technology and services for implementing communication and process solutions

Tracker Suite Project management solution

WebCal group web calendar

WorkZone web-based shared documents and archived reports for advertising and other professionals

VITA

Yi Yang was born in Zun Yi, Gui Zhou Province, China. He received his B.E. in computer science at Southeast University, Nanjing, China in 1995 and his M.E. degree in computer science from Nanjing University, Nanjing, China in 1999. He began pursuing a Ph.D. degree in computer science at Texas A&M University in 2000. Since then, he has worked as a graduate teaching assistant and research assistant for Dr.Du Li in the Department of Computer Science, Texas A&M University. His permanent address is: Department of Computer Science, Texas A&M University, 408A Harvey R. Bright Bldg, College Station, TX 77843-3112.