Composable Collaboration Infrastructures Based on Programming Patterns

Vassil Roussev, Prasun Dewan, Vibhor Jain
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
+1 919 962 1700
{roussev, dewan, vibhor}@cs.unc.edu

ABSTRACT
In general, collaboration infrastructures have supported sharing of an object based on its logical structure. However, current implementations assume an implicit binding between this logical structure and particular system-defined abstractions. We present a new composable design based on programming patterns that eliminates this binding, thereby increasing the range of supported objects and supporting extensibility.

INTRODUCTION
The development of distributed collaborative applications is a challenging task and requires a number of implementation solutions that are not readily available in generic distributed system toolkits. To simplify and lower the cost of development of collaboration software, a large number of collaborative infrastructures have been built. Although their solutions vary, the approaches they have taken can be classified in two broad categories.

One of the approaches emphasizes automation by supporting a fixed number of system-defined shared abstractions, for which the system provides various collaboration services. However, applications developed using these abstractions are inherently limited in scope and capabilities. For example, systems such as XTV that support sharing of the (system-defined) abstraction of a window provide a high level of automation but are fundamentally suited for WYSIWIS collaboration. The main problem here is that the infrastructure uses the sharing of the user interface of the object as a means of sharing the object itself and, therefore, the two cannot be separated.

The other approach emphasizes flexibility by providing lower-level primitives for sharing arbitrary, programmer-defined abstractions. This however, comes at the price of higher development cost, as the programmer must manually implement the sharing services. For example, broadcast methods [13] provide a basis for implementing sharing of arbitrary objects, however, it is the responsibility of the developer to implement the specific mechanism.

The outlined approaches can also be viewed as representing two extreme design points in a fundamental trade-off between the system knowledge about the shared object and the kind of support it can provide. That is, the more a system knows about the object, the more specialized service it can provide. Inversely, the less it assumes about the object, the wider range of objects it can accommodate. For example, if two users are editing the same document simultaneously, and the document is opaque to the system, any two concurrent updates are conflicting. At the same, if the system knows that a document consists of a sequence of sections, it will consider simultaneous modifications to be in conflict only if they are within the same section. Further subdivisions of document (e.g., subsections, sentences, and words) present even better opportunities to provide fine-grained collaboration services.

In an attempt to integrate the described approaches, a number of systems, such as *GroupKit* [10], provide two-level support for sharing: Applications using a set of system-defined object types get an automatic high-level sharing mechanism, whereas applications requiring programmer-defined types are given lower-level collaboration primitives to implement the desired behavior. This hybrid approach, however, does not reconcile the other two–it merely provides them under one roof. Therefore, the problem of designing a single mechanism that provides both automation and flexibility without the described limitations is still open and is the subject of this work. Specifically, we are interested in providing such a sharing mechanism for object-based systems.

The first idea that helps us address this problem builds on the fact that collaboration services can be automated without full knowledge of the object semantics. For example, the concurrency control semantics mentioned above can be implemented by simply knowing the structure of the object. Thus, from a collaboration point of view, the structure of the shared object is more important than the particular semantics it implements.

The second idea is to provide flexibility by developing a component-based architecture that permits incremental modifications to the system. The idea of component-based architectures is not new; our contribution is its application to the domain of collaboration.

We have developed an approach for creating collaboration infrastructures based on these two ideas. The approach builds on the notion of a *bean* by extracting the logical stricture of an object based on patterns in the signatures of the public methods of the object. The second idea addresses the problem of flexibility within the infrastructure itself, that is, giving the developer the ability to change every aspect of the collaboration services. To achieve this we present a component-based architecture that permits incremental modifications to the system. We have implemented the approach in a system supporting three important collaboration services: coupling, merging, and access control.

The rest of this paper is organized as follows. In the next section, we briefly survey a list of influential systems in the CSCW domain and identify the relationships among the design choices of these systems and their advantages and limitations. Next, we use these design choices to identify a set of generic requirements for collaborative and show that none of the existing systems meets all of these requirements. Following that, we describe a new approach for meeting these requirements and present two concrete implementations of the approach that support coupling and merging, respectively. Finally, we summarize our findings and outline future extensions of this work.

RELATED WORK

The first prerequisite for building any distributed application is the availability of a standard communication mechanism. The TCP/IP socket-based communication provides the basic means of remote message delivery and enables the development of any distributed application. However, its relatively low-level interface essentially forces the development of application-specific message protocols.

To illustrate the problems with using TCP/IP, consider how it can support sharing of replicas of a document object that has a title, a sequence (ordered list) of sections, and a set of keywords and is an instance of the following class:

```
class Document {
 String getTitle();
 void   setTitle(String title);
 Section getSection(int i);
 void    setSection(int i,Section s);
 int     getSectionCount();
 void    insertSection(int i,Section s);
 void    removeSection(int i);
 void     addKeyword(String key);
 void     removeKeyword(String key);
 String[] getKeyword();
}
```

Programmers using TCP/IP must implement a message protocol to encode updates performed on the object, such as changing the title or adding a new section. A separate concern here is organizing group communications to propagate the updates, and yet another concern is ensuring consistency. Solving these problems involves tedious and routine work that involves writing code for operations, such as message parsing, that are not part of the main application functionality.

To free the application-programmers from defining such a message protocol, the abstraction of remote procedure calls (RPC) was introduced [2]. RPC bridges the gap in the abstraction level between the application and the network interface by reducing the programming effort of remote access to a procedure call. For instance if we use *Java*'s standard RMI (remote method invocation) mechanism, we could export any of the above methods and invoke them remotely instead of encoding them as messages.

To automate group communication, many collaboration systems provide multicasting facilities. Corona [12], for example, is a server-based general-purpose multicast system that provides support for maintaining group membership and for message delivery. If we apply this approach to our document problem, we would be relieved from the burden of organizing the lower-level details of communication but we would still be responsible for designing the communication protocol among the client replicas and ensuring consistency.

Integrating the ideas of group communications and RPC, *GroupKit* [10] provides a higher-level collaboration abstraction–multicast RPC–that allows a procedure call to be automatically invoked on multiple hosts. An analogous idea has been applied to objects and has resulted in the implementation of *broadcast methods*, such as the ones provided by the Xerox's Colab [13] system. To appreciate the advantages of this system, let us consider how it can help in sharing of the example object. We could declare all of the methods that modify the state of the object (`setTitle, setSection, insert/removeSection, add/ removeKeyword`) as broadcast. As a result, if all instances in the object group start with the same state, they will be kept consistent at all times (assuming that broadcasts are atomic and excluding non-idempotent operations). Thus, we achieved one commonly used form of sharing at a very low programming cost. However, given two versions of the same document, broadcast methods alone would be of little help in restoring consistency.

Since this operation-centric mechanism is not suited for all collaboration scenarios, other systems have taken a more data-centric approach by providing a system-defined shared abstraction. In shared window systems, such as XTV [1] and DistView [9], collaboration is achieved through automatic sharing of application windows. Hence, to implement our document-sharing example, all we need is any single-user implementation of a text editor and a shared window system. Unfortunately, a shared window system is not a silver bullet for our sharing problems. It is inherently limited by the fact that it cannot allow sharing of an object without also forcing the sharing of its user interface. For

instance, it cannot allow two different users to share the document object but scroll to different parts of it.

Choosing a different shared abstraction does not help much. To illustrate this, consider *GroupKit*'s [10] *shared environments*, which are dictionary-style data structures containing keys and associated values. The main feature of these environments is that all replicas are automatically kept consistent, and the application can register callbacks to learn about events of interest, such as insertions and deletions. To use the automatic sharing of *GroupKit* in our example, we would have to cast the document structure into an environment. However, this is both difficult and unnatural as the structure is inherently recursive and requires certain ordering.

Sync [8] provides a more flexible solution by embedding generic collaboration capabilities into several base classes that can be combined with each other to create a larger variety of structures. In particular, the system provides classes for replicated records, sequences, and dictionaries. An application developed using these classes requires virtually no additional support to use the merging framework provided by the system. On the other hand, the framework exhibits the problems of shared window systems and *GroupKit* if a programmer-defined class must be shared. For instance, we cannot take our document object as it is and ask Sync to provide merging services. Instead, we must decompose it into a replicated record containing a string, a replicated sequence, and a replicated dictionary field. Further subdivisions of the document hierarchy must follow the same rules. In summary, we are faced with the choice of casting the shared data structure into a predefined set of primitives, or implementing it using a custom-built sharing mechanism.

Thus, what we need is a mechanism to flexibly share arbitrary programmer-defined types. Such a mechanism has been implemented in *Suite* [3]. It allows the developer to choose any data representation and still get fine-grained structure-based collaboration services. The problem with Suite, however is that it is a *C*-based system and, therefore, does not support objects. Hence, constructing our shared document involves redefining it in terms of *C* data types. Another concern is the fact that Suite is designed as a monolithic system and changing aspects of the provided collaboration functions is difficult and error-prone.

To address the latter problem, *Prospero* [4] uses computational reflection as a means of allowing the application to change the implementation of the collaboration toolkit. Another way to achieve this is to break the system into smaller components and define interfaces among them. In this case, modifications are introduced by replacing components with compatible ones. If all components adhere to the specified interfaces, the overall system correctness should not be compromised. For instance, *JComposer* [5] was initially implemented as a single-user CASE editing tool but due to its component-based design was seamlessly extended to a collaborative application. Furthermore, the implemented collaboration services can be independently used to create collaborative versions of other editing applications.

Each of the described systems has made its own set of design choices based on initial assumptions and target application domains. Based on our discussion so far, we can identify the following collaborative infrastructure requirements.

INFRASTRUCTURE REQUIREMENTS

Automation. As with most development environments, automation is the primary purpose of collaborative infrastructures. Typically, automation is measured by the amount of code a programmer has to write to build a particular application. In the case of a collaborative environment, it is a measure the developer's effort to achieve multi-user behavior with respect to the single-user case.

Flexibility. The flexibility requirement is so broad that we must consider different aspects of it to classify different systems. We distinguish among four different types of flexibility.

- *Semantics flexibility*. This refers to the generic capabilities of the infrastructure to offer an implementation platform for a full range of collaborative services. To evaluate a system with respect to this requirement, we ask the question: what range of collaboration behavior can be achieved using the mechanisms supplied by the toolkit?
- *Extensibility*. It is not reasonable for any system to assume that it provides an exhaustive list of features covering all possible applications and scenarios. Hence, it is important to allow programmers to devise their own extensions and to customize system behavior.
- *Composability*. This is an extension of the previous requirement–a composable system allows the developer to take any part of the system and replace it with a custom one. An important issue here is the granularity at which this process takes place–ideally, it should be restricted only to those logical components that need to be changed.
- *Abstraction flexibility.* As part of the task to lower the development cost, infrastructures must support sharing of as many types of objects as possible–in particular programmer-defined object types.

Reuse. Often, the development of multi-user applications starts with a fully implemented single-user version of it. Therefore, the extent to which existing functionality is reused has a direct impact on the complexity and cost of the application development.

Standard language support. It is important to avoid (whenever possible) additional support from the language, such as specialized compilers or interpreters. Thus, the implementation of the collaboration infrastructure should not be bound to a specialized language interpreter and can be modified independently.

Table 1 evaluates each of the described systems with respect to the derived requirements. When a requirement can be met to different extents, we use high(H), medium(M) and low (L) to approximate the degree to which the requirement can be met. The main purpose of this evaluation is to establish benchmarks that will help us in evaluating the success of the proposed new design. Table 1 clearly shows that none of the presented systems meets all of the requirements to a high degree. Therefore, our goal is to design a system that satisfies each requirement at least as well as the best system in the category.

	Automation	Semantics flexibility	Extensibility	Composability	Abstraction flexibility	Reuse	Standard language support
TCP/IP	L	H	H	L	Y	H	Y
RPC/RMI	L	H	H	L	Y	H	N/Y
Corona	M	H	H	L	Y	M	Y
Multicast RPC	M	H	H	L	Y	H	Y
XTV/DistView	H	M	L	L	Y	H	Y
Sync	H	M	L	L	N	L	Y
Suite	H	M	L	L	Y	L	N
JComposer	L	M	H	H	N	H	Y

Table 1 System evaluation

PROPERTIES AND PATTERNS

Let us consider first how a high-level of automation and semantics flexibility can be provided for programmer-defined objects. To understand how this goal can be achieved, consider the two approaches we have seen above in the treatment of object semantics in current collaboration infrastructures.

The first one assumes nothing about the object semantics and leaves the development of sharing mechanisms entirely in the hands of the programmer. This leads to an open system where flexibility is not restricted but the lack of semantic information limits the automation opportunities.

The second approach takes the opposite assumption as a starting point–object semantics is fixed up front (through a set of system-defined objects) and appropriate collaboration services are provided. This leads to a much higher degree of automation but may pose flexibility problems if the supplied primitives are inadequate for a particular application, or code reuse is an issue.

The key to automation is the automated discovery of object structure. Once the structure is derived, we can use existing techniques to provide means of supporting variable-grained collaboration functions.

Our solution is based on the observation that an object is not an opaque entity but represents a logical structure. Therefore, what is needed are ways to extract this structure without violating the data encapsulation principle. As it turns out, the *JavaBeans* framework [6] provides a basis for implementing this approach.

Simple and indexed properties

The main goal of *JavaBeans* is to define a generic component-based architecture for *Java* programs. An application built according to the architecture consists of (*Java*) beans–objects that adhere to a particular programming style that permits their automated customization and composition. Beans are programmed using certain conventions that allow an external agent, called the *introspector*, to decompose them into components called *properties*.

'Properties are discrete, named attributes of a Java Bean that can affect its appearance or its behavior' [6]. This is a very broad definition that can be applied to a wide range of objects. However, for the purposes of describing the logical structure of an object we need a more precise property definition. Therefore, by summarizing the discussion in [6], we give the following definition. An (*object*) *property* is a named attribute of an object that adheres to a well-defined semantics and is implemented by a set of dedicated methods.

The main benefit of this more formal definition is that it specifies that properties are externally recognizable by the appearance of a specific group of methods, and requires that these methods implement a pre-defined semantics. Intuitively, we can think of a property as representing an abstract data type with the corresponding methods implementing the individual operations on the type.

By default, *Java* recognizes two types of properties–*simple* and *indexed*. A simple property is defined by a pair of 'get' and 'set' methods of the following form:

```
<PropertyType> get<PropertyName>()
void set<PropertyName>(<PropertyType> value)
```

The 'get' method corresponds to a *read* operation and returns the current value of the property, whereas the 'set' method defines a *write* operation that assigns a new value to the property. Thus, if a matching 'get'/'set' pair is found during the object analysis, then a read-write property named `propertyName` of type `PropertyType` is discovered. In some cases, one of the methods may be absent in which case the property is considered as write-only/read-only respectively.

Indexed properties are a natural extension of simple properties and approximate standard array semantics in procedural languages. Whenever a simple property of an array type is discovered, it is considered to be an indexed property. The object is then searched for a second pair of get/set methods that manipulate individual elements by their index.

The main advantage of this property-based view of the object is that it enables the logical decomposition of a bean object by representing it as a collection of properties with predefined semantics. However, the standard *JavaBeans* model exhibits a number of limitations that stem from the

fact that property naming conventions and semantics are hardwired into the system:

- Legacy code must either be rewritten to comply with the exact specification, or interfacing code must be added. In general, even previous versions of the *Java APIs* have not followed such strict conventions. Well-written systems usually follow similar conventions but may use other keywords. For example, instead of 'get', developers may have used other verbs to describe the reading of a property– 'read', 'check', etc. Similarly, write operations may be represented by methods starting with 'write', 'update', 'reset', etc., instead of 'set'. To accommodate such classes within *JavaBeans* framework, the developer must supply additional *BeanInfo* classes for each of the original classes.

- Typing information is not used in defining the property name. For example, given the following method signatures,

```
public void set(Parent newParent);
public void set(Editor newEditor);
```

the standard *Java* introspection will not recognize them as defining two write-only properties–*parent* and *editor*. It is natural to interpret the above signatures as shorthand for

```
public void setParent(Parent newParent);
public void setEditor(Editor newEditor);
```

which *would* be interpreted as defining properties.

Finally, and most importantly, the set of recognizable properties is limited and the system provides no means of extending it. It is not hard to argue that the existing *Java* support for properties is not general enough for our purposes. A quick look at the widely used `Vector` and `Hashtable` classes that are part of the *JDK* reveals that, according to the standard *Java Introspection*, they have *no* identifiable properties of their own. One possible solution is to replace all `Vector` and `Hashtable` occurrences in the source with appropriately written *adapter* classes. An adapter class is a direct descendant of the class it replaces and defines additional methods that adhere to the *JavaBeans* naming conventions and, therefore, has properties. However, this solution is fundamentally unsuitable, as it does not capture the dynamic nature of the `Vector` class–its ability to incrementally change its structure through the addition/removal of individual elements.

To overcome the outlined limitations, we extend the standard *JavaBeans* property model to allow for a more flexible specification.

Generalized properties and programmer-defined patterns

In an effort to gain flexibility, we split the problem of specifying a property from its interpretation. Therefore, we introduce the notion of *programming patterns* as a means of generalizing the *JavaBeans* naming conventions. A *programming pattern* (or pattern) is a set of method naming conventions, whose purpose is to expose aspects of the structure and the semantics of an object. In particular, patterns can be used to advertise object properties. Thus, *Java*'s get/set naming conventions are just a special case of using patterns.

Our next problem is to provide a mechanism that is flexible enough to accommodate the description of arbitrary (programmer-defined) properties through patterns. For that purpose, we use a declarative property specification language, which is explained below.

We adopt the de facto standard mixedCase naming convention as the basis for our pattern analysis. Specifically, we assume that method names consist of one or more tokens. The beginning of each token is marked by a capital letter following a small letter or, in case there is more than one consecutive capital letters, by a capital letter followed by a small letter. For example, `getUIGenerator` is decomposed into three tokens (`get-ui-generator`). The rest of this paper discusses pattern matches only at the token granularity.

The first step in the property specification is to define method signature patterns (method patterns) that select candidate methods. The patterns are based on a canonical string representation of method signatures of the form

```
<return_type> method_name(arg1_type, …,argN_type)
```

In addition, method patterns contain free pattern variables that are assigned string values and are denoted by enclosing them in a pair of dots. For example, to define the standard *Java Introspection* 'get'/'set' methods we use the following declarations:

```
getter = <.GetType.> get.Prop.()
setter = <void> set.Prop.(.SetType.)
```

Pattern variables are assigned upon the completion of a successful match, and contain the maximum length match (which may span several tokens). For example, if the above `getter` declaration is matched against the method `int getXPos()`, the values of the pattern variables would be as follows `GetType == "int"` and `Prop == "xPos"`.

The second step is to define the conditions under which candidate methods are grouped together to define properties. To illustrate this, consider the following declaration, which completes the description of the standard *Java* properties[1].

```
property
  type = simple
  methods = getter, setter
  constraints
    getter.Prop == setter.Prop
    getter.GetType == setter.SetType
  handler = colab.bus.SimplePropertyHandler
  name = getter.Prop
end
```

[1] The actual system implementation discussed in the case studies uses XML to represent the property definitions. For the sake of brevity, we have used simplified syntax in this discussion.

The specification states that a property of type *simple* is defined whenever two methods can be found such that they match the *getter/setter* patterns, and their respective pattern variables satisfy the given constraint equalities. Recall that the reason for extracting properties of an object was to perform several structure-based functions such as diffing and merging automatically. This processing tends to be done recursively, with a different object handling each level of the structure. The type of this object depends on the property. The *handler* of a property is the class of the object that performs the structure-based operations for the property. It is looked up by the handler of the parent of the property. After looking it up, the handler can instantiate it and invoke appropriate (operation-specific) methods in it. The last line gives a rule for deriving a public name for the property that helps distinguish it from other properties derived by the same specification.

Recall that one of the shortcomings of the *JavaBeans* model was the inability to provide alternative patterns for the same type of property. To show how we handle this issue, consider describing the special case of boolean properties. As an exception to standard *JavaBeans* naming conventions, the system allows the 'get' method to have following form:

```
boolean is<PropertyName>();
```

Describing this exception in our model is straightforward–all we need is an alternative definition for the *getter* method and the rest of the definitions will work as before:

```
getter = <boolean> is.Prop.()
```

Our next order of consideration is identifying a set of properties that should, by default, be supported by the infrastructure. Given our goal of achieving the level of support of current systems, we choose a model that parallels that of *Sync*, which provides the largest set of structures. Since simple properties already express the same structure as *Sync*'s replicated records, we only need to define properties analogous to its replicated sequence and dictionary types.

For that purpose, we use *sequence* and *table* properties. A sequence, as already discussed, represents a mapping between natural numbers and object elements, thus introducing an implicit ordering among the elements. A table consists of a series of associated key-value pairs of objects. Both sequences and tables can have a variable number of elements. Although sequences can be viewed as a special case of tables, their widespread use warrants separate treatment.

Following our pattern approach, we detect a sequence property by the presence of a pair of *get/set* methods that access/assign individual sequence elements, a *size* method that returns the number of elements, and a pair of *insert/delete* methods that introduce structural changes by inserting/removing elements.

Similarly, to distinguish a table property, we search the object for a *get* method, as well as a pair of *put/remove* methods. The only notable difference with sequences is that we need an additional *elements* method that returns an enumeration of all current elements.

Thus, our model provides default implementations of four types of properties–simple, indexed, sequence, and table. (Indexed properties are entirely subsumed by sequences but are included for compatibility reasons.) However, the programmer is free to modify/extend the set of supported properties by providing appropriate specifications and implementing the property-specific part of the collaboration services. To incorporate new types of properties in the collaborative application, the developer invokes the system-provided *PropertyIntrospector*, which reads in the specifications, registers the corresponding handler classes, and performs the pattern matches on the shared objects.

The discussion so far showed how object structure can be used to reconcile the requirements of abstraction flexibility and automation. Our evaluation table, however, shows that to meet all requirements we also need to resolve the conflicts among the requirements of composability, automation, and reuse.

Composition requires that objects communicate using a well-defined generic protocol. Furthermore, communication should be indirect to avoid unnecessary object dependencies that limit composability. Thus, we adopt an event-based approach that satisfied these two requirements and discuss its implication for our collaboration model.

EVENTS

Once again the original *JavaBeans* framework provides the basis for our event model. Recall that, in *JavaBeans*, there is only one type of operation (write) that leads to state changes. Therefore, there is a single type of event (`PropertyChangeEvent`) that covers all updates and, by convention, is issued by the bean object after each property modification. However, given our extended set of recognizable properties, we need an appropriate extension of the event model. For each new property, we need a new event object that can encode the specific operations performed on the property. In general, such an event includes the type of the operation, e.g., *insert*, and its arguments, e.g., {"Bob Smith", 1}. Thus, the event model closely matches the supported properties.

Another requirement that bean objects must fulfill to enable event-based communication is to maintain a current list of listeners and to propagate each update event to all of them. For that purpose, beans export a pair of add/remove methods through which interested listener objects can register to receive events.

Thus, a well-behaved bean in our model (as opposed to in the *JavaBeans* model) must meet two obligations–event announcement and event distribution that, strictly speaking, are not part of its main functionality. This leads to a conflict with both the reuse and automation requirements–legacy code cannot be expected to follow the bean event

model and additional programming effort is needed to build this communication model into objects.

The event distribution can easily be automated by delegating it to a specialized object that maintains the listener list and propagates event notices (*JavaBeans* provides similar functionality). As it turns out, the event announcement is a stronger requirement and is harder to automate. Nevertheless, one of our case studies shows that, in asynchronous collaboration scenarios, it is feasible to fully relieve the bean object from its event announcement and distribution commitments by using an external agent to *generate* the events on its behalf. The key idea here is to exploit the property structure of the object and to deduce update operations from consecutive snapshots of the object's state.

The following section illustrates the use of our overall framework in the implementation of two concrete collaboration services–coupling and merging. It also refines the architecture for these specific services.

CASE STUDIES

The goal of our coupling experiment was to design a generic property-based coupling service, and to use it to provide different levels of coupling to an application based on the popular model-view-controller (MVC) paradigm.

We chose an existing *Java* implementation of a (single-user) drawing editor and used it to understand how the property-based architecture can handle reuse. The original application was organized as follows. Its *model* consist of a (hash)table where all of the shape objects (ovals, rectangles, text, etc.) in the current drawing are stored. Individual shapes are accessed through globally unique identifiers assigned at creation time. The *view* maintains an up-to-date graphical representation of the objects on the screen by refreshing the image every time a modification to the model occurs. The *controller* keeps track of user actions and translates them into operations on the model.

Our first experiment was to add model coupling to the application without making changes to the original code. In particular, we could not assume that the model object provides event notification. Therefore, we used a property-based implementation of an object diff-ing service similar in idea to the UNIX *diff* tool. The main difference is that *diff* works on text files, whereas our service works on objects: given two versions of a (property-based) object–old and updated–the object diff returns a list of operations such that, if applied to the old version, the two objects become consistent. We encoded the operations as event notices identical to the ones a proper bean would generate to describe them.

The overall architecture of the system is depicted in Figure 1. We used a generic implementation of a coupling service that keeps a group of distributed bean objects consistent as follows. Each site has a local coupling agent that listens for property update events from local object replicas and transmits them to all processes in the session. At the remote sites, the events are translated back into (property-based) operations that are applied to the corresponding objects and their shadow copies (previous versions).

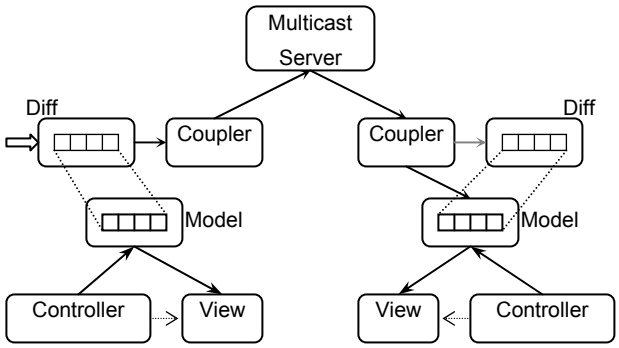

Figure 1 Diff-based model coupling

The underlying communication service uses an RMI-based multicast server that delivers events to a group of clients. However, the coupler interacts with the communication layer using the bean event model, and, therefore, is not bound to this particular implementation.

The full solution works by combining the coupling and the diff-ing services–the diff object maintains a shadow copy of the shared object and, whenever prompted, performs a diff operation on the shadow copy and the new version of the object. The result is sent to the local coupler in the form of property update events and processed as described above.

The only open question is how to trigger the synchronization process. One simple solution is to let the user do it explicitly by pressing a button, or selecting a menu item. Another is to use a timer and initiate the process periodically. In our case however, the application provided an interesting option–to add the diff object as another view and, hence, be indirectly notified whenever the user performs an update. Thus, we effectively simulated a scenario where the user initiated synchronization after each operation.

Although this scheme performed reasonably well in that there was no noticeable performance degradation in our case study, it is obvious that it is not efficient–the diff object must traverse the whole object structure every time it is invoked. Consequently, this approach is inherently suitable for a more asynchronous collaboration scenario.

In our second experiment, we wanted to add view coupling to the drawing tool by reusing the same coupling service. Fundamentally, view coupling implies a more synchronous collaboration scenario and, therefore, requires explicit event notification. The initial intuition was that the widespread use of events in UI toolkits would make the task easier. Indeed, the standard *Java* user interface library–the AWT–promptly provides event notification about any UI events of interest, such as resizing a window or pressing a button. Unfortunately, its UI components do not follow the *JavaBeans* naming conventions and cannot be directly attached to our coupling service.

To address this mismatch, we built adapter classes for each of the UI objects used in the application such that they correctly follow the bean naming conventions and translate AWT events into property change events. For example, a `COMPONENT_RESIZED` event is translated into a change to the object's *size* property and communicated accordingly. Figure 2 illustrates the new configuration.

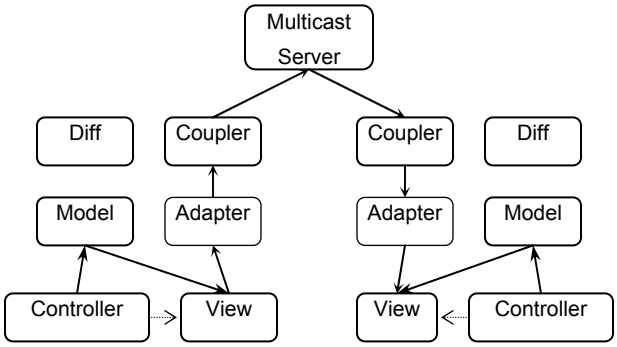

Figure 2 View coupling with bean adapters

Since the adapter classes can be used with any AWT-based *Java* application, the only application-specific developer effort was to attach the adapters to the base components. In practice, this translates into one additional method invocation per application window as the subcomponents are automatically handled in a recursive manner.

It is interesting to measure the effort involved in writing the adapter classes, as it can help in estimating the effort in converting a conventional object-based application into a bean-based one. Overall, it took us 430 lines of code distributed over eight classes. To a great extent the task was simplified by organizing the adapters into an inheritance hierarchy, parallel to the one of the original objects. Thus, the bulk of the code (~150 lines) was concentrated in one base class dealing the generic beans. The rest were relatively easy to implement, as most of the code was routine.

Another potential concern for the described implementation is the cost of performing the property matches for the shared objects. Indeed, the time to discover object properties for classes with a lot of methods can be significant. For example, this process can take 1.5s for the java.awt.Frame class. Fortunately, all the information is class specific and , therefore, can be derived off-line. Thus, the application is only burdened by the cost of loading it from a file at startup time.

In a separate experiment, we modified the original version of *Sync* [8] to show the reuse of existing infrastructure components, and to explore the issues involved. Recall that the architecture of *Sync* relies on the inheritance mechanism to provide sharing of application objects. In other words, for an object to be shared automatically, it must be subclass of the system-defined `ReplicatedSequence` or `ReplicatedDictionary` classes. The main problems of this approach become apparent when it is applied to existing code. Due to the single inheritance model of *Java*, adding *Sync*'s services to a single-user application is a cumbersome procedure, as it typically involves a major overhaul of the inheritance hierarchy of the whole application.

More generally, *Sync*'s problem is that it ties the supported object structures (record, sequence, and dictionary) to particular class abstractions. Hence, to support abstraction flexibility, we must implement a model that permits the separation of the object structure discovery from the rest of the system. To illustrate this transition, let us consider the implementation of our document example under the original and updated versions of *Sync*.

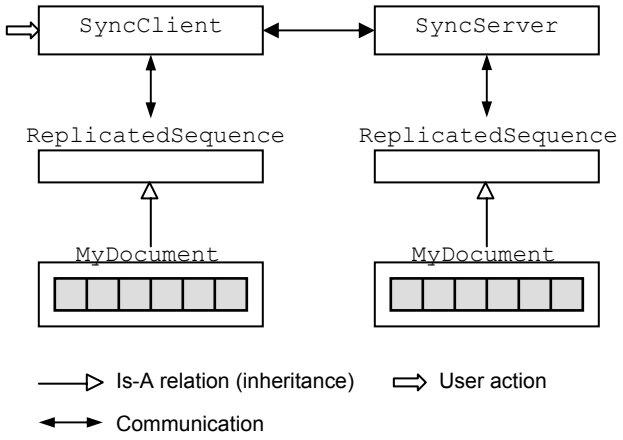

Figure 3 Original Sync architecture

Figure 3 presents the implementation of a shared document (`MyDocument`) according to the original *Sync* architecture. For the sake of brevity, we assume that the document consists simply of a sequence of sections. Thus, it is sufficient to make `MyDocument` a subclass of the system-provided `ReplicatedSequence`, which will keep a log of all updates to the sequence elements (represented by the shaded rectangle). Synchronization takes place whenever the user initiates it through the SyncClient, which queries the client replica for the log since the last synchronization. The log is then communicated to the SyncServer (which maintains the master copy) and eventually reaches the server replica. Depending on the current merge policy in place, the server selectively accepts the client updates, and sends back to the client a list of changes that must be applied to the client replica so that the two version reach a consistent state (the list may include updates from other clients).

Figure 4 shows how the original version of *Sync* is transformed into a pattern-based one through introduction of an adapter class. The main difference is that the shared object is no longer tied to a particular inheritance hierarchy. Instead, it implements a set of methods that are characteristic of a sequence pattern, such as `insertElementAt` and `deleteElementAt`. In addition, it issues events whenever updates to its elements are performed.

The other new element in the architecture is the presence of the `SequenceDelegate` object. Its function is to provide

means of reusing the existing *Sync* implementation. In essence, it mediates the synchronization process by maintaining a shadow copy of the local replicated object. As suggested by the figure, the `SequenceDelegate` is a direct descendant of one of *Sync*'s base classes—`ReplicatedSequence`—and, as such, is automatically shared with its peer objects. It holds a reference to the user object—`MyDocument`—and is registered to receive update notifications from it.

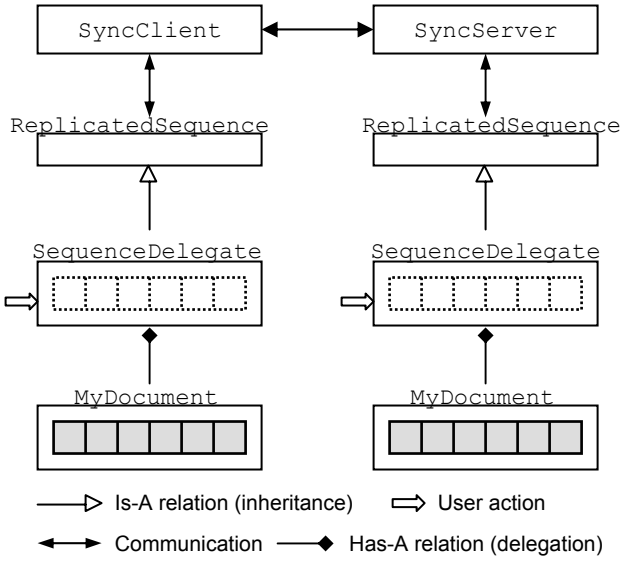

Figure 4 Pattern-based Sync architecture

To illustrate how the synchronization process works, let us follow it step by step. Suppose that a new element is inserted into the client replica of `MyDocument`. The replica issues an event informing its listeners about the change. Thus, the local `SequenceDelegate` object is notified and performs a corresponding insert in its own shadow copy. The operation is automatically logged by *Sync* and later transmitted to the server. As a result, the server shadow copy is automatically updated. This triggers an invocation of the 'insert' method on the server replica of the shared object, which brings the two replicas to a consistent state.

Here we showed how an adapter class for the sequence property was added. Similarly, we added adapter classes for simple properties and hashtables for a total of about 900 lines of code in the adapter classes, which allowed us to reuse, without modification, approximately 9000 lines of *Sync* code.

A potential drawback of adding the adapter code is that its overhead might noticeably degrade the system performance. Therefore, we did an experiment with a drawing application and took measurements to quantify the effect of the added code on the overall cost of merging. As it turns out, in our experiment, merging took the same time in both the original Sync and its pattern-based extension with adapter classes (about 70 milliseconds.). It is not clear if the adapter classes will add negligible overhead for collaboration functions such as coupling that are less expensive than merging.

Thus, the merging experiment shows that it is possible to implement our approach using existing implementation of an infrastructure.

EVALUATION

To convince ourselves that the presented architecture indeed meets the identified requirements, let us examine them one by one and compare our accomplishments with that of other systems.

Our first requirement was to support a high degree of automation in application development. In the coupling example, we demonstrated the addition of collaboration services that is achieved at minimal development cost. Furthermore, our approach supports the incremental development of collaborative applications where the level of support rises as a function of how much information the application provides. As our example showed, it is possible to add asynchronous collaboration just by using standard patterns in the object design. Adding the standard *JavaBeans* support for properties is sufficient for some objects to get synchronous sharing. Extending this support to include dynamic properties enables the full range of collaboration functions to be used.

Recall that the first aspect of infrastructure flexibility was that of semantics flexibility. To satisfy this requirement, we modeled the supported logical structures after that of *Sync*, which subsumes the logical structures provided by other systems. Thus, we do not claim any contributions with respect to this requirement, and provide the same expressive power as *Sync*.

With the respect to abstraction flexibility, like Suite, we support programmer-defined types. While Suite supported conventional programmer-defined types, we support object types. Our approach cannot handle arbitrary programmer-defined types–only those only those that have logical structures supported by us, and follow conventions that can be encoded in our pattern-based language. Thus, it supports more object types than existing systems but not all possible object types.

Composition was another focus of this work and our two experiments show that we provide a high degree of composability comparable to that of *JComposer*. The main difference is that, in addition to addressing composition of the application and the collaboration infrastructure, we also address composability within the infrastructure itself. In our coupling example we showed how a generic coupler can be attached to two different application modules. At the same time, the coupling service implementation can be easily substituted without affecting other aspects of the system. This is possible because the application does not know specifics of the services that can be attached to it. It is unaware that it is being coupled or merged.

Our case studies show that our approach is coherent with the goals of code reuse. In the first case, we were able to provide collaboration functions at virtually no cost to the application—the specialized adapter code written can itself be reused for any AWT-based user interface. In the second

case, we addressed the reuse of existing collaboration function in the infrastructure and were able to achieve that at a modest cost.

Finally, we did not go outside the limits of the *Java* language by adding non-standard features that require specialized processing. We based our work on the standard *JavaBeans* architecture and designed our extensions so that existing bean components can be directly integrated with our framework.

CONCLUSIONS AND FUTURE WORK

In this paper, we described a novel approach to building collaborative infrastructures. By reviewing previous work in the area, we derived a set of generic infrastructure requirements, related to automation, flexibility, and reuse that have not been simultaneously addressed by existing systems. Furthermore, we discussed the relationship between the design choices of these systems and their limitations. We showed that using the structure of shared objects is important in supporting flexible collaboration services but current systems have not been successful in providing a flexible mechanism for specifying that structure.

To overcome this problem, we described an infrastructure design that is based on using programming patterns as a means of extracting the logical structure of shared objects without exposing details of their implementation. This approach is complemented by the adoption of an event-based communication model that promotes a component-based implementation of both the application and the collaboration framework.

We concluded our discussion by showing example applications of our approach to two specific collaboration service–coupling and merging–and evaluating the results with respect to the initial requirements. In fact, we have implemented another service, access control, using this architecture and the *JavaBeans* vetoable events. We did not describe it because of lack of space.

Our ongoing research efforts are aimed at extending this work in several directions. In the short term, we are working on integrating the described diff-ing and merging services, as they naturally complement each other. We also plan to fit other collaboration services, such as concurrency control and undo/redo into our model.

Another line of study is to apply this pattern-based approach to the integration object-based programming and data specification languages, such as XML. Specifically, we would like to create an XML structure directly from an object based on the logical structure we extract [11].

In the longer term, we would like to also explore the use of predefined patterns to generate objects. That is, given a pattern specification, the user could fill in the pattern variables and the system could create a skeleton class definition that relieves the developer from routine work and guarantees a certain style of programming.

So far, we implicitly assumed that objects behave nicely and their properties implement the expected semantics. An open issue here is the ability of the system to test the advertised object behavior. One possible solution is to use axiomatic specifications, as described in [7].

Finally, in this paper, we have not addressed architectural flexibility–the ability to support multiple architectures and dynamically adapt the architecture of an application. This is the subject of another project of our group, and we would like to eventually integrate it with this work.

ACKNOWLEDGEMENTS

The insightful comments of the reviewers helped improve the presentation of the paper. This research was sponsored in part by National Science Foundation grants IRI-9627619, IIS-9977362, and CDA-9624662.

REFERENCES

1. Abdel-Wahab, H., Jeffay, K., *Issues, problems, and solutions in sharing clients on multiple displays.* Internetworking: Research and Experience, 1994. **5**: p. 1-15.
2. Birrel, A.D. and B.J. Nelson, *Implementing Remote Procedure Calls.* ACM TOCS, February 1984. **2**(1).
3. Dewan, P. and R. Choudhary, *A High-Level and Flexible Framework for Implementing Multiuser User Interfaces.* ACM Transactions on Information Systems, October 1992. **10**(4): p. 345-380.
4. Dourish, P., *Open Implementation in CSCW Toolkits (Doctoral Dissertation)*. 1996, University of London.
5. Grundy, J. *Engineering component-based, user-configurable collaborative editing systems*. *EHCI*. 1998.
6. Hamilton, G., *JavaBeans specification*. 1997, Sun Microsystems.
7. Hughes, M., Stotts D. *Daistish: Systematic Algebraic Testing for OO Programs in the Presence of Side-effects*. in *ISSTA*. 1996.
8. Munson, J., Dewan, P., *Sync: a Java framework for mobile collaborative applications*, in *Computer*. 1997. p. 231-242.
9. Prakash, A., Shim, H. *DistView: Support for building efficient collaborative applications using replicated active objects.* in *Proceedings of the ACM Conference on CSCW*. 1994.
10. Roseman, M., Greenberg, S., *Building real-time groupware with GroupKit, a groupware toolkit.* ACM Transactions on Computer-Human Interaction, 1996. **3**(1): p. 66-106.
11. Roussev, V., Dewan, P., Koorakula, N., Sellappa, S. *Integrating XML and Object-based Programming for Distributed Collaboration*. in *WET ICE*. 2000.
12. Shim, H., Hall, R., Prakash, A., Jahanian, F. *Providing flexible services for managing shared state in collaborative systems*. in *Proceedings of the Fifth European Conference on CSCW*. 1997.
13. Stefik, M., et al., *Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings.* CACM, January 1987. **30**(1): p. 32-47.