# Integrating XML and Object-based Programming for Distributed Collaboration

Vassil Roussev, Prasun Dewan, Naveen Koorakula, Sriram Sellappa
*University of North Carolina*
*{roussev, dewan, naveen, sriram}@cs.unc.edu*

## Abstract

In this paper, we explore some of the new opportunities for distributed collaborative applications that emerge from the use of XML as a data specification language. We present two different approaches: the first one transparently adds asynchronous collaboration to applications whose persistent state is in XML format, while the second one helps build synchronous collaborative applications starting with an XML schema specification.

Although the two approaches start with different assumptions, they both lead to the same problem—the need for a generic one-to-one conversion between objects and XML constructs. Using object properties, we define two variants of a conversion scheme for the two approaches.

## Introduction

Experience shows that writing distributed multi-user applications is a non-trivial task and involves a number of issues not raised by single-user applications. The added complexity stems primarily from the need to coordinate the actions of a group of users working on a common task. Furthermore, the coordination mechanisms must be flexible enough to allow the collaborators to effectively cooperate in a number of different situations where computer aid is desirable.

To illustrate the needs of distributed collaboration, let us consider a simple example of collaborative editing. Users *A* and *B* are planning to write a paper together but would like to do it over the Internet as they live far apart. They could cooperate by sending each other emails with their respective versions of the document. However, this implies that they would have to rely entirely on social protocol to ensure the consistency of the document, and would have to manually reconcile different (and potentially conflicting) versions of it. Instead, they would like to have a collaborative environment where they could work on the paper simultaneously and have an automated mechanism that can ensure consistency and help resolve conflicts.

The development of collaborative applications from the ground up is a costly and error-prone process. Therefore, we need software infrastructures to support this process, as well as simple and intuitive abstractions that can be presented to the user. Ideally, the infrastructures should also be compatible with existing standards.

Traditionally, many distributed collaborative systems have been built around the notion of *sharing*. The basic idea behind sharing is to give each user a copy of the shared entity and to guarantee a certain level of consistency among these copies. In general, users are most familiar with two types of entities on their computers–data files and applications. Therefore, it is intuitive to present shared versions of these abstractions to the user.

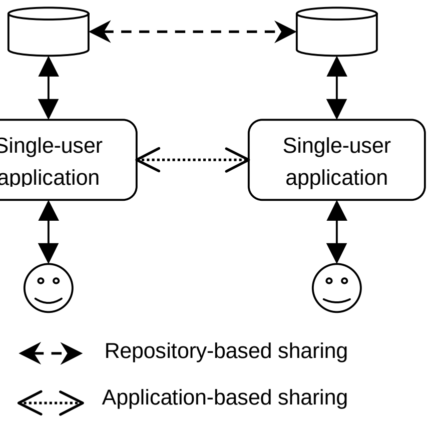

***Figure 1 Collaborative sharing***

Thus, there are two ways to achieve the abstraction of sharing. One is to create a shared file and allow collaborators to edit it independently. The other is to create a shared application, which allows the collaborators to see the effects of each other's action on the document in real-time. We refer to these models as *repository-based* and *application-based* sharing, respectively.

Our work explores new ways of implementing these two forms of sharing mechanisms based on XML document representation and its integration with object-oriented technologies. The rest of this paper is organized as follows. First, we discuss the use of XML in developing a fine-grained repository-based sharing mechanism. Next, we apply a similar idea to an application-based sharing scheme, and describe an experimental implementation of it. Finally, we summarize our results and future extensions of this work.

## Repository-based sharing

The OS has traditionally assumed the responsibility of maintaining the persistent storage of all data. This opens up the opportunity to synchronize the state of applications using the file I/O channel. Indeed, users of distributed file systems have used repository-based sharing for many years. Distributed file systems (DFS), such as *Coda*, allow multiple users to cooperate by repeatedly loading, modifying, and saving the same file. Moreover, *Coda* supports merging of data of disconnected and weakly-connected clients and its recent extension [1] handles user-level operations in an attempt to improve performance.

However, the impressive list of features in traditional DFS comes with a relatively high cost of deployment and maintenance. This makes them inherently unsuitable for spontaneous collaboration over a wide-area network. Currently, a number of freely-available products allow the creation of an application-transparent 'Internet drive'. The importance of these systems with respect to collaboration is that they make wide-area sharing affordable to practically all Internet users.

We see two major advantages in the use of DFS for collaborative sharing:

*Transparency*. None of the applications has to be aware of the sharing–all of the distribution functions are handled automatically at a lower level.

*Interoperability*. The sharing mechanism is independent of the application as all applications use the same I/O functions and abstractions provided by the system.

One the other hand, there are some collaboration-specific issues that DFS are not designed to deal with:

*Fine-grained sharing*. The unit of sharing is a whole file, which often is inadequate if two or more users want to work on it at the same time. Consider our editing example: the authors would like to simultaneously work on different sections of the document. Furthermore, they might want to ensure that the document is consistent by either preventing conflicting updates, or by merging non-overlapping changes.

In the physical world, our users could, for example, work on separate sheets of paper and then simply combine them to get the whole paper. Surprisingly enough, current distributed technologies do not offer such a generic solution to the above scenario in the virtual world. The problem has been that there was no agreed upon standard, such as XML, for representing arbitrary structured data, which made it infeasible to derive the data structure in a generic way and to provide finer-grain services.

*Incremental response*. In general, file systems do not provide incremental synchronization, access control, concurrency control, and fault-tolerance. The large overhead of file operations effectively prohibits the access to these services incrementally.

Thus, the lack of incremental response is an inherent problem to repository-based sharing. However, the lack of fine-grained sharing can be overcome by breaking away with the canonical treatment of files as opaque entities and assuming that at least some the applications will store their data in XML format. As we discuss in the next section, this allows us to give better sharing services to those applications.

## XML-based repository sharing

To illustrate this approach, consider the following XML document:

```
<document title="XML 1.0">
  <section title="1 Introduction">
      <p>Extensible Markup Language …</p>
      <section title="1.1 Origin and goals">
        <p>XML was developed … </p>
      </section>
      <section title="1.2 Terminology">
        <p>The terminology used … </p>
      </section>
  </section>
</document>
```

It presents a definite hierarchical structure that allows us to deduce that the main document consists of smaller units (called "section") that in turn can have even smaller subdivisions ("section" and "p"). (This information can also be derived from the corresponding DTD schema, if present.)

Note that, while we still don't understand the semantics of the data, the standard syntax permits its incremental breakdown into smaller and smaller units. Eventually, we reach the indivisible (leaf) units, which we must treat as atomic entities (in this example the "p" elements). Having derived the structure, we are now capable of providing sharing at a much finer granularity (in fact, at variable granularity).

Looking back at our original scenario, we are now ready to help our collaborators edit their document in a consistent manner by providing generic collaboration services, such as coupling, concurrency control, and fault-tolerance, that follow the document structure.

## XML serialization

The previous section showed how XML could be used to implement fine-grained collaboration services in an application-neutral way. However, we expect that applications would rarely operate directly on the textual representation of XML data. More likely, they would convert it into language-specific objects, process it, and then export it back in XML format. Hence, a marshalling/unmarshalling utility that can automate this process would be very useful.

One way to handle these conversions is to use standard representations, such as the DOM, that provide a one-to-one mapping between objects (in different languages) and XML. DOM objects are modeled after XML and any application that uses it must deal with XML-specific objects (document, schema, elements, attributes, etc.). This raises at least two problems:

- The DOM naming does not reflect object semantics in the application context and, therefore, its universal usage would distort programming style. For example, coding a text editor with `element` and `attribute` objects is not as intuitive as doing it with `title`, `section`, and `paragraph` objects.
- Existing applications need to be rewritten to comply with the DOM in order to get the benefits of automation.

Both problems indicate that this is not an approach that can be expected to gain widespread acceptance. In general, we believe that any object-to-XML conversion scheme should fulfil several requirements in order to be successful.

*Generality*. The conversion scheme must handle generic objects–not just the ones from a particular inheritance tree.

*Automation*. The conversion should be done automatically with minimal need for program modifications.

*Reversibility*. It must be possible to execute the reverse XML-to-object conversion. In other words, enough information must be retained to reconstruct the original object in a different address space, and enough information must be stored to resolve any potential ambiguities in the reverse mapping.

*No new language support*. The solution must work with any implementation of the underlying language. In particular, it should not depend on specialized translators, such as (pre-) compilers or interpreters.

To devise a solution that satisfies the above requirements, especially the last one, we have chosen *Java* as our implementation language.

## A *Java* solution

In an ideal solution, developers' effort would be reduced to a single method invocation of the form:

```
XMLSerializer.saveAsXML(rootObject,fileName)
```

with the expectation that the `XMLSerializer` would take the `rootObject`, save it in the file with the specified name (in XML), and then find all other objects to which `rootObject` has a reference and would apply recursively the same procedure. The recursion terminates whenever it reaches a node in the document hierarchy that does not refer to any objects that have not been saved already.

The idea of serializing objects to a persistent store has been explored extensively. In *Java* however, the idea has been brought a step further by embedding it into the language in the form of a standard API.

At a high level, *Java*'s serialization mechanism works as follows. Given an object tagged as `Serializable`, *Java* allows the programmer to store it to a file, or send it over a network connection with a single method call. Moreover, it finds all other objects to which the original object has references and applies the same procedure recursively until the whole graph of related objects is stored/transmitted.

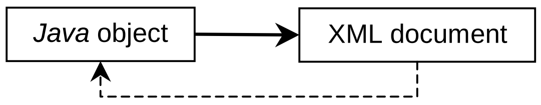

***Figure 2 Starting with Java object***

This is very similar to what we want to achieve with the exception that objects are not stored in XML format. To be fair, we must say that *Java* provides a mechanism that allows objects to take responsibility of their own serialization. During the serialization process, if an object tagged as `Externalizable` is encountered, *Java* will invoke its `writeExternal` method to export its state. Conversely, when the object is deserialized, its `readExternal` method will be invoked to restore its state. Thus, by implementing the `Externalizable` interface, we can achieve conversion to/from XML and solve our problem.

Unfortunately, this solution does not satisfy our requirement for low programming cost as it necessitates the modification of each and every class whose instances must be serialized. Thus, we are a looking for a solution similar to the standard serialization mechanism but one that can be implemented outside the *Java* virtual machine.

Therefore, we must be able to derive the structure of the object from its external appearance. In general, this is an unsolvable problem so we restrict ourselves to a set of objects called (*Java*) *beans* [2].

A *bean* is a collection of *properties* with well-defined semantics. A *property* is a distinct, named attribute of an object. It can be accessed and/or modified through a set of dedicated methods and its value may be of any type.

Bean objects adhere to a set of programming patterns (naming conventions) that permit the automated discovery of their properties. Furthermore, they may expose standard interfaces through which other objects can receive notifications about specific changes in their state.

The standard bean conventions recognize two types of properties: *simple* and *indexed*. A simple property is characterized by a pair of 'get' and 'set' methods whose purpose is to read and write the property. An indexed property is a simple property of an array type (e.g., an array of strings). In addition to the standard get/set methods, which deal with the whole array, it may have and additional pair of methods for manipulating individual elements by their index.

## A property-based solution

Let us now use the property-based view of the application objects to support our automated object-to-XML conversion. We assume that only the values of the publicly accessible properties represent the serializable state of the object. At first glance, this may seem like a severe restriction, but a closer look at good programming practices reveals that it is a valid approach.

First, we believe that the majority of current applications are being developed using the document/view paradigm in which the abstract data structure is separated from its UI representation. For example, Microsoft's development environments advocate this approach and generate skeleton class files to facilitate the process. Thus, it is not overly restrictive to consider the existence of 'data' objects whose main purpose is to store the application's state. Therefore, other application objects must be given access to this information. If we follow the data-encapsulation principle, access must be provided through dedicated methods, and not by exposing the internal structure of the objects. Hence, we should be able to retrieve the state of these objects using only public methods.

Second, the use of method naming conventions (which form the basis for property discovery) is highly desirable from a software-engineering point of view–it greatly simplifies the understanding, maintenance, and further development of the application.

Finally, the serialization of public properties solves the 'fragile serialization' problem. That is, it allows two different versions of the same object to use the same serialized state. We believe that this is not possible to achieve with the standard *Java* serialization process, as the compiled classes do not contain enough data for the system to make an informed decision.

Having adopted properties as our way of describing the object state, we are now ready to define a generic mapping between the object and its XML representation. We map each object to an entity with a name derived from its class name. Next, we map all properties of the object to nested elements. To preserve the semantics of object-to-object relationships, we need to distinguish between literal and reference property values. The former include numbers and strings, whereas the latter include all object references. Literal values can be converted directly to XML's PCDATA, but doing so for references may not preserve the intended application semantics. To illustrate this, consider two `paragraph` objects, each of which has a *style* property, which is a reference to the same `style` object. If we treat objects as literal values, we would end up with two `paragraph` elements, each with a `style` nested element. If we now do the reverse conversion, we would have little choice but to create two equivalent but separate objects for the `style` elements. Therefore, if the application changes one of them, the other one would remain intact, which is probably not the intended semantics. Hence, we need to encode references differently so that object relationships can be properly reconstructed. We encode references, using XML's ID/IDREF attributes, which essentially provide the reference semantic (an IDREF attribute refers to an element with the given ID attribute), and we can expect a standard XML parser to handle these automatically.

Given this background, we can now follow a depth-first-search-based algorithm, similar to the one implemented by the standard serialization process, to serialize the graph of related objects. The only difference here is that we only consider property values when looking for referenced objects.

The reverse conversion from XML is straightforward given the one-to-one mapping defined above and is omitted for the sake of brevity. The only subtlety we will point out is that, in order to faithfully recreate the object, we need to have the binding schema, which maps object classes to XML elements. This information can be stored either as part of the XML file, or as a separate entity. We advocate the second approach as it allows the generated XML data to be interpreted independently of the source object. Furthermore, this leaves the option of binding the serialized state with another (version of the) object.

Let us now focus on a couple of implicit restrictions that result from the fact that we only use the public properties of the object and not its internal state.

First, there is no explicit ordering among the properties of an object. This does not present a problem in the conversion to XML but is a potential setback in recreating the object. A problem may occur if a given property is dependent on another property in the same object and setting the value of the first property before the second would cause an error, or an inconsistency. This is an inherent problem with using object properties and cannot be solved without specific, per-class information. For that reason, we assume that object properties are autonomous and can be assigned independently of each other.

Second, the unmarshalling of the object must start with creating an instance of it. The question is: which constructor should be used, and what argument(s) must be supplied? Again, class-specific information could be used to resolve the issue but we want to avoid the costly effort. Therefore, we always use the constructor with no arguments. To justify this we make two points here:

- Constructors with arguments are a matter of convenience and not of necessity–all classes could be written so that they always have a constructor with no arguments.
- Wherever such constructors are missing, a trivial (empty) implementation can be added automatically either at the source-code level, or at class-load time by using object instrumentation [3].

## An example

To illustrate the use of XML-based serialization in distributed collaboration, consider the potential use of the *XMLDiff and Merge Tool* [4] for enhancing repository-based collaboration. The essential diff/merge functions can, for example, be embedded as part of a client/server system:

Whenever the server receives a put request for a particular XML file, the new version is automatically compared with the existing copy and any non-conflicting changes are merged. Similarly, the client must now take into consideration that its copy may not be not be up-to-date, and after saving its copy must immediately request the document back and merge with its own version.

To outline the benefits of using fine-grained XML services over generic file-based solutions, we compare the above approach with the UNIX *diff3* tool. *diff3* takes as input an original text file and two versions of it. It then produces a script that, when applied to one of the versions, reconciles it with the other.

Consider the integration of *diff3* at the server as a means of achieving consistency between, say, two XML documents. It can lead to two types of problems:

• Discovery of irrelevant formatting differences. The simplest example is differences in the spacing of XML constructs. In general, those are ignored with the exception of some data blocks.

• Discarding of non-overlapping changes. For example, very often multiple attributes of an element are defined on the same line. Thus, if one user changes one of them, while the other changes another, *diff3* will detect a conflict and will choose one version of the line over the other. However, a fine-grained diff/merge tool will discover that the updates are independent and will proceed to automatically reconcile them without discarding user updates.

## Application-based sharing

Recall that repository-based sharing is inherently unsuitable for synchronous collaboration. Therefore, support must be included in the application at development time. As it turns out, XML can play an important role in this case, too.

The idea is that the data schema of the application is not given in terms of *Java* classes. Instead, it is defined by an XML schema, for which the *Java* classes are derived automatically.

The use of XML for data specification has several advantages over other options:

*Intuitive*. XML is simple enough so that even non-programmers can produce XML specifications.

*Formal*. Using a formal language significantly reduces the possibility for ambiguities and misinterpretations. This is an important advantage over using, for example, a description written in plain English.

The described data-centric approach leads us to explore the generation of (parts of) the collaborative application from an XML specification. That is, we start with a DTD schema and we generate a class representation of it. From there on, we are free to follow the standard object-based development process.

Note that this is a different problem than the problem of recreating objects from their XML-serialized state (Figure 2). The differences stem from the fact that now the schema is fixed and object representation can vary. Therefore, we must decide on a suitable class representation of the DTD schema, specifically its elements and attributes. Given our property-based approach, we have little choice but to map them both to corresponding properties. Thus, it is no longer possible to distinguish nested elements and attributes in this class representation. In our view, this does not pose a problem, unless we impose the additional requirement that object instances of the generated classes be convertible into XML structures conforming to their initial schema.

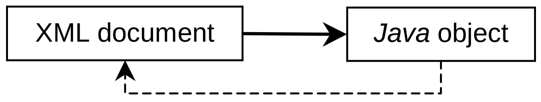

***Figure 3 Starting with XML specification***

This optional requirement allows us to get two additional benefits from using XML for data specification:

*Interoperability*. The data output of the application can easily be shared with other XML-enabled applications.

*Standard rendering*. The XML data can be displayed using a standard viewing utility, such as a web browser.

Given these advantages, we must address the ambiguity mentioned above. To resolve the issue, we use the initial DTD specification by reading the object properties in the given order, and then mapping them to elements, or attributes, according to the schema. As in the previous mapping, we need to store the binding between the DTD and the resulting object(s) to enable the correct XML serialization of the object.

Our XML-to-Java binding schema has a different goal and takes a different approach than most other implementations. While other schemes are concerned with fixing a standard *Java* representations of XML documents, we are more interested in providing a generic two-way conversion between XML and *Java* objects that preserves the natural style of programming. Another difference is that we are not using XML as a means of declarative programming as it is the case with *BML* [5]. To the best of our knowledge, there are no other schemes that provide a model for XML serialization of generic objects.

### XML-based application sharing

In this section, we outline the major steps involved generating a collaborative application from an XML data specification. We also use a simple collaborative text editor that we have built to demonstrate a concrete implementation.

• We start by building an appropriate DTD schema for the application. In our case, the basic data structure is the document so we use a simple hierarchical description of nested sections, and paragraphs.

• Next, we generate 'data' object classes that will contain the actual run-time data on which the application will operate. We emphasize the use of naming conventions that closely match the names in the original DTD to facilitate any subsequent modifications to the code.

In the example, we generate a `Document` class, which has a simple *title* property, as well as a list of authors and a list of sections (both indexed properties). Similarly, the `Section` class is generated with a *section* and *paragraph* indexed properties. Finally, the `Paragraph` class has (for the sake of simplicity) a single *text* property containing the actual text of the paragraph.

• Next, using generic window components, we generate a UI that allows the user to create, edit, and store/load the data objects (in XML).

• Finally, we combine the generated objects with generic implementations of collaboration-specific services, such as coupling, access/concurrency control, merging, and fault-tolerance, to achieve the desired collaborative behavior.

In our implementation, we have provided coupling and fault-tolerance as example collaboration functions. We should note that the discussed application was developed manually. We are currently working on an infrastructure that will permit the automation of collaboration services.

## Related Work

Let us outline some of the main differences between our approach and other solutions. *CORBA* [6] provides a framework for inter-operating objects that are implemented in different languages. However, its mechanism is relatively heavyweight and requires the individual description of every shared object.

On the other hand, platforms, such as *XML-RPC* [7] and *SOAP*, simply extend the original idea of RPC by using open standards (XML/HTTP) for their implementations.

KOML [8] is most closely related to our work, however, its approach is tied to the specifics of the *Java* serialization format.

## Summary and future work

In this work we showed two different ways to utilize XML for the purposes of distributed collaborative applications. The first one enhances existing asynchronous, repository-based sharing by adding fine-grained collaboration services. The second one enables the generation of synchronous collaborative applications from a generic XML schema. We argued that using XML for data specification provides users with an intuitive, yet formal way of starting application development. Furthermore it can help in inter-operating different applications and displaying application data with a generic viewer.

We showed that the successful implementation of both of these schemes hinges on the solution of a more general problem—the generic mapping between XML data constructs and objects. We gave two such mappings, both based on the bean property model. We motivated the differences between the two approaches by showing the different requirements they must satisfy.

In this paper, the presented implementation work is limited to the generation of class files based from a DTD schema. We would like to extend this and build an implementation that allows instances of these classes to be serialized in XML. We would also like to build a prototype implementation of the presented XML-based repository sharing mechanism.

Another direction for our ongoing research is concerned with overcoming the limitations imposed by the bean property model. Space limitations do not permit us to discuss them here in detail but we should not that, currently, many objects cannot be adequately described with simple and indexed properties. For example, the standard `Vector` and `Hashtable` classes have *no* identifiable properties.

Therefore, we are working on extending the bean model to incorporate other types of properties. Our extension is based on the idea of flexible specification and recognition of properties using programming patterns.

## Acknowledgements

The reviewers gave several helpful suggestions for improving the quality of this presentation. This research was sponsored in part by National Science Foundation grants IRI-9627619, IIS-9977362, and CDA-9624662.

## References

1. Lee, Y., Leung, K., Satyanarayanan, M. *Operation-based Update Propagation in a Mobile File System*. in *USENIX*. 1999. Monterey, CA, USA.
2. Hamilton, G., *JavaBeans Specification*. 1997, Sun Microsystems.
3. Cohen, G., Chase, J. *Automatic Program Transformation with JIOE*. in *USENIX Annual Technical Symposium*. 1998.
4. Birsan, D., Sluiman, H., Fernz, S., *XML Diff and Merge Tool (http://www.alphaWorks.ibm.com)*. 1999.
5. Weerawarana, S., Duftler, M., *BML (http://www.alphaworks.ibm.com/tech/bml)*. 1999.
6. *http://www.corba.org*.
7. *http://www.xmlrpc.com*.
8. *http://www-sop.inria.fr/koala/koml/*.