A Component Model for Internet-Scale Applications

Steven P. Reiss
Brown University
Providence, RI 02912
401-863-7641, spr@cs.brown.edu

ABSTRACt

This paper describes a component model where the overall semantics of a component is included in the interface definition. Such a model is necessary for future computing where programs will run at Internet-scales and will employ a combination of web services, grid technologies, peer-to-peer sharing, autonomic capabilities, and open source implementations. The component model is based on packages and supports static and dynamic objects, interfaces, structures, and exceptions. The interface definitions provide a practical approach to defining functional semantics and include appropriate extensions to provide semantics for security, privacy, recovery, and costs. The component model has been implemented in a prototype framework and demonstrated in an Internet-scale example.

Categories and Subject Descriptors

D. Software; D.2 SOFTWARE ENGINEERING; D.2.2 Design Tools and Techniques; Modules and interfaces, Software libraries, Evolutionary prototyping.

General Terms

Design, Languages, Reliability.

Keywords

Component models, Internet-scale applications, Interface semantics.

1. INTRODUCTION

The concept of a program is changing from a local, self-contained object into an Internet-scale, pervasive, self-organizing, omnipresent entity. We have developed a novel component model that addresses the programming issues that will make this transition smooth and manageable. This model has been implemented using a prototype framework that demonstrates its potential.

There are a number of current trends that point to the future of programs and programming. These include web services, grid computing, peer-to-peer computing, autonomic software [11], the open source movement, and more-reliable networking.

ASE'05, November 7-11, 2005, Long Beach, California, USA.

These trends are converging. Programs are written using a multitude of web services some of which use other web services. These web services are written by different, often anonymous programmers and change with little or no notice. Grid-based applications such as today's database systems run on any available machine and find new computational resources as needed. Data is shared across the Internet both through web services and through peer-to-peer connections. Instant messaging, SETI, Gnutella [6], Napster, and Electric Sheep [8] are examples of current Internet-scale applications, with more to come.

Dealing with this new reality will require us to change the way we think about programs and programming. We can't continue to think of a program as a self-contained entity with only local effects that we control. Instead, we need to work in terms of a global system where we do not control most of the components or interactions. We need a means for programming in a world where software systems will be built mainly from components designed, developed, maintained, and modified by different people, on machines that are not under the control of the software developers or designers, and where the components themselves evolve outside of the control of the software system. It is this set of problems that our research tries to address. Our goal is to ensure:

> Developing an Internet-scale application should be no more difficult in the future than developing a standalone application is today.

The remainder of this paper details the component model we have constructed to achieve this goal and an implementation of that model. We start by illustrating an Internet-scale application and discussing the lessons it provides. This is followed by a detailed discussion of the underlying component model and the prototype framework called TAIGA that implements the model.

2. AN EXAMPLE APPLICATION

As an example of an Internet-scale application we have built a system for visualizing what people are browsing on the web. The application gathers data from potentially millions of users, monitoring what pages they are currently browsing. It summarizes this information by categories and then displays the results so that users can understand browsing patterns over time and spot trends. The application and framework are available for download at *http://www.cs.brown.edu/people/spr/webview.html*.

A view of the application is shown in Figure 1. Time is represented by the concentric circles, with the outermost circle being the most recent time interval (the past 5 minutes) and the innermost circle representing the earliest interval (the previous

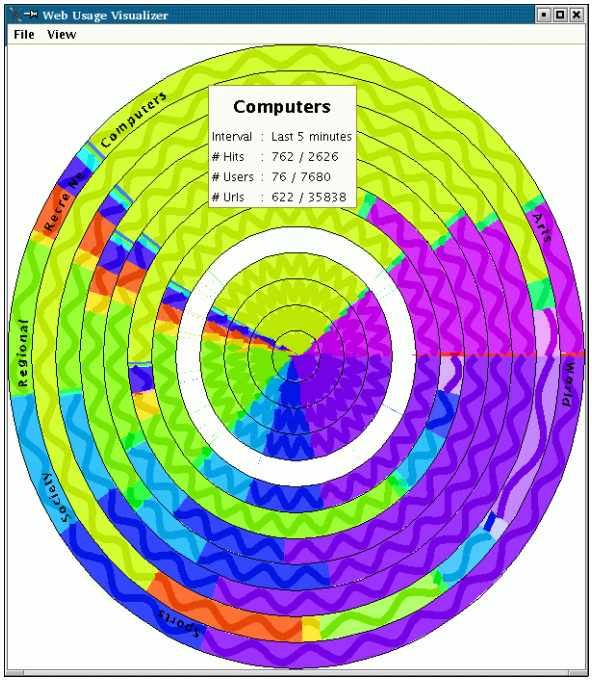

Figure 1. WebView visualization

2 days). Within each circle, hue is used to show the Open Directory (http://dmoz.org) category (labeled in the outer circle), the span of the arc reflects the number of views of that category, and color saturation shows the relative number of views (whether there were more or fewer page views than one would expect for the time interval). Additional information is shown by the wavy line that traverses each circle. The frequency of the waves within the category shows the relative number of distinct URLs that were browsed, so that a flat line indicates few and a wavy line indicates a lot. The width of the line indicates the relative number of users.

This application was developed using three external interfaces and global files. One interface is used to accept information from data miners about a web page as it is browsed by a user. The implementation of this interface gathers such information from multiple users, accumulates it by category, and periodically outputs a record to the current global file containing the number of pages, users, and distinct URLs for each category. A second interface manages the global files, creating a new file when the current file gets too large and providing an index that lets the visualizer find the appropriate file for a given time in the past. The third interface is used to provide the category for a URL. We have three implementations of this interface, one based on the Google web service, one based on MeURLin [14], and one that reads the Open Directory database and builds a decision tree.

The web usage visualizer is coded as if it were a standalone application. The data miners save their data by simply calling a recorder method. The recorders append to a file that is opened and used as if it were local. The manager similarly reads and writes files that it thinks are local. Finally, the visualizer itself gets the current file from the manager via a method call and then uses standard Java file methods (*available* and *readLine*) to determine when new data has been added and to read that data.

Experience with this application has illustrated several points about Internet-scale programming. First, it pointed out the importance of dynamic binding of implementations to interfaces. After we had been running it for a while, Google, without notification, changed their web service so that the Open Directory category was no longer returned for all pages. Our framework detected this and choose a different implementation without user intervention. Second, it illustrated the importance of using cost models. The initial version of MeURLin was very slow and this was reflected in the costs based on the time needed to run various test cases. An improved version of MeURLin was put up (again without our knowledge), and our framework detected that the new version was practical to use. MeURLin also illustrated the importance of having multiple implementations. Its server has a tendency to be down once or twice a week; this is detected by our framework and the next best implementation is chosen. We have also seen the need for dynamic rebinding. We run the decision tree classifier on a local machine. Unfortunately, that machine has had some hardware problems and has needed to be rebooted relatively frequently. When this occurred, our system automatically found an alternative binding, either by running the classifier on another local node or using one of the other implementations.

We have also implemented several simpler applications in our framework, most notably a chat program and a N-body problem solver. Our experiences with all these programs led us to the object model and implementation we currently use.

3. INTERNET-SCALE COMPONENTS

Programming in a pervasive, Internet-scale environment requires a new approach that can handle the issues of scale, the notion of change and failure, the consequences of lack of control, and the effects of global sharing of data and code. The primary requirement is a component model that can deal directly with these issues and their corollaries.

We have developed a component model that attempts to meet this requirement. Our model ensures:

- Existing web services, libraries, and other component implementations can be used as components without modification;
- Implementations actually implement the interface as intended;
- Interfaces and implementations define and enforce security and privacy properties;
- Fault tolerance and recovery are defined as part of the interface and are inherent to the system.
- Versioning and evolution of interfaces and implementations is dynamic and built into the system [23];
- Interfaces support classes and objects and can contain static methods, constructors, and concrete methods.
- Interfaces can be bound and rebound dynamically.
- Components can be used synchronously in a threaded environment or asynchronously with appropriate callbacks.

3.1 Related Work

There are many different object-based component systems. Most of these take the approach of CORBA or Microsoft's ActiveX and utilize a separate interface definition language.

This language is then mapped into appropriate stub and skeleton code for passing and accessing remote objects. These have the advantage of being relatively language independent, but the disadvantage that the user has to define the interface as well as a corresponding implementation class with the same name. Java RMI works only for Java, but uses the Java reflection mechanisms to work directly from the implementation class, eliminating the need for a separate definition. More recent work here is reflected in the notion of a web service defined by a separate interface definition language, WSDL. The WSDL files are typically generated automatically from the particular implementation, are globally accessible, and can be used to build an appropriate programming interface for an arbitrary client. JavaBeans takes a different approach. Here the component interface is essentially the same for all components. This interface offers the ability to get and set properties, to register event listeners, to generate events, and a reflection mechanism. A reflection mechanism is provided so that beans can query the properties and events of other beans. Beans interact with other beans by knowing the types of events to generate and listen for.

An approach that is more network oriented is illustrated by Jini [19]. Here the components are services which can have multiple implementations. Servers register for a service with a lookup server. Clients can then find a relevant server through the lookup mechanism. The abstract services are effectively Java interfaces which are implemented by remote clients using Java RMI with Jini providing the binding mechanism.

There are also a variety of techniques for combining these different component systems. Web services based on SOAP are able to support .Net components directly and other components via wrappers. Similarly, legacy systems or other components are often used as components in various component models using appropriate wrappers [22]. Frameworks like the VCF [20] formalize this process by automatically producing wrappers.

One problem with these approaches is the significant commonality required between the client using an interface and the implementation. For CORBA and RMI, any objects that are being shared usually require that appropriate stubs be loaded both in the implementation and in the client. For a web service, the WSDL file defines the particular implementation of the web service complete with the URL to be used, and not a generic specification. For JavaBeans, the clients and implementations have to be implemented as beans and the different components need to understand the properties and events used by the other components. For Jini, the clients and implementations need to agree beforehand on the service interface and its semantics.

These approaches do not work at Internet scales in a pervasive world. Servers cannot be expected to load the stubs needed to support classes from all potential clients. Objects will often need to be passed through multiple services, and the intervening services might not want or need to understand the object. There will be no standard set of properties or events that can be created on the fly to cover all potential developers. Binding of interfaces to implementations needs to be dynamic and mutable to handle failure and recovery. All this is difficult with the technologies currently used. A more flexible and less tightly-coupled approach is needed.

One approach to doing this decoupling is to change to a programming model that enforces it such as Linda [1]. Here a central tuple space is used for communication and coordination among the various clients. Tuples in the tuple space follow a standard format and are easily shared. While the original version was designed for local systems, extensions such as Javaspaces [9] and TSpaces [25] provide scalable implementations. This approach has the advantage of being relatively simple and isolating the distributed aspects of the computation. However, it is a different programming model and is rather limiting for many types of applications.

Language support for components and component technologies has also been relatively common. Modula 3 and Ada both allow the definition of package-based interfaces that effectively describe the set of classes in a component. Java provides interfaces, but at the class level not the component level. Language support makes using components easier for programmers by letting them work with standard frameworks and programming models.

We wanted our component model to be fully integrated into the base programming languages. It is important that programmers be able to work with components using traditional programming metaphors and styles, and that components be easily integrated into existing applications. A component needs to represent the equivalent of a Java package, i.e. a set of classes, interfaces, and exceptions that are related and actually share a single implementation. At the same time, we wanted to remain language independent as much as possible and to be flexible in the way an implementation can match an interface. There might be multiple web services that perform a single function, each of which uses slightly different calling conventions. A single interface should be usable for all such implementations. Moreover, the implementations themselves should not have to be modified.

3.2 Outerfaces and Implementations

Our approach is to provide an interface definition language that essentially defines a Java package or its equivalent, providing not only the syntax of the calls, but also their semantics. To differentiate our interfaces from Java interfaces, we call ours an *outerface*. The basic outerface definitions used by the web usage visualizer are shown in Figure 2. As seen in the example, outerfaces can include class definitions with abstract static methods, traits (properties), and test cases. Outerfaces can also include definitions for classes with dynamic methods, constructors, and concrete methods; interfaces for implementing callbacks; structures for handling call-by-value; contracts; security and privacy specifications; availability restrictions; a cost model for choosing an implementation; and recovery information.

Outerfaces are hierarchical and can either extend or restrict existing outerfaces. If an outerface extends an existing one, implementations of the new outerface will be considered as candidates of the old; if an outerface restricts an existing outerface then implementations of the old are candidates for the new. When restricting an outerface the user is limited to adding new test cases, adding additional security or privacy constraints, and changing the cost model.

The outerface definitions are used by the prototype to generate Java packages that the user can code to directly. Thus the web visualizer source code contains the import line:

```
import edu.brown.cs.webview.taiga.WebManager.*;
```

```
outerface edu.brown.cs.webview.taiga.WebManager {

trait { rebind=true; }
class FileManager {
    static public String getCurrentFile();
    static public String getFileForDate(long date);
}

testcase Test0 { ... }}

}      // end of outerface WebManager


outerface edu.brown.cs.webview.taiga.WebRecorder {

trait { rebind=true; }
class Record {
    static public void record(String url,String usr,long date,int tzoff);
    static public String getUniqueId();
}
}      // end of outerface WebRecorder


outerface edu.brown.cs.webview.taiga.WebClassify {

trait { rebind=true; }
class Classify {
    static public String findCategory(String url);
}

testcase Test0 { .. }
testcase Test1 { ... }
testcase Test2 { ... }
}
```

Figure 2. Outerface definitions used by WebView (with the test case code elided).

and then just uses the class *FileManager* as if it were directly implemented. When this class is first used by an application, the framework gets control, dynamically finds a matching implementation, and does the appropriate binding. This approach to dynamic binding is more general and broader-based than previous approaches such as [16,21].

Implementations are defined using a separate but similar language, with the WebView examples shown in Figure 3. These identify the class and then define the mapping between the outerface class and the implementation. The description language is flexible enough to allow name mapping, parameter reordering, default parameters, and simultaneous implementation of multiple outerfaces. As illustrated, the implementation descriptions can also define traits, cost information, and availability restrictions.

These examples define outerfaces and specify implementations with a language based on Java, using similar keywords and constructs. The language itself is merely a front end to an internal XML description that embodies the principle elements and has been changing significantly as the system evolves.

3.3 Classes, Interfaces, and Structures

The component model needs to support a variety of programming methodologies and techniques. Outerfaces are generally built around classes that can support either object-oriented or procedural programming. The outerfaces of Figure 2 are essentially procedural since the classes provide only static methods. Figure 4 shows a more complex outerface for a chat server.

This outerface is consistent with object-oriented programming. The programmer can create instances of the class *ChatServer* and then call methods on those instances. Since the outerface specifies the implementation should be shared in the trait clause, these instances will be implemented as proxies to remote objects. If the implementation was in the form of a

```
implementation edu.brown.cs.webview.taiga.SimpleManager {
    using edu.brown.cs.webview.recorder.RecorderManager;

    implements edu.brown.cs.webview.taiga.WebManager {
    using class FileManager = edu.brown.cs.webview.recorder.RecorderManager;
    }
}

implementation edu.brown.cs.webview.taiga.SimpleRecorder {
    using edu.brown.cs.webview.recorder.RecorderMain;

    implements edu.brown.cs.webview.taiga.WebRecorder {
        using class Record = edu.brown.cs.webview.recorder.RecorderMain;
    }
}

implementation edu.brown.cs.webview.taiga.GoogleClassify {
    using edu.brown.cs.webview.classify.ClassifyGoogle;
    available edu.brown.cs.*;

    implements edu.brown.cs.webview.taiga.WebClassify {
        using class Classify = edu.brown.cs.webview.classify.ClassifyGoogle;
    }
    cost = 100;
}

implementation edu.brown.cs.webview.taiga.XmlClassify {
    using edu.brown.cs.webview.classify.ClassifyXml;

    implements edu.brown.cs.webview.taiga.WebClassify {
        using class Classify = edu.brown.cs.webview.classify.ClassifyXml;
    }
    trait { javaopts = "-Xmx1700m"; startup = 2000 }
    available edu.brown.*;
    cost = 10;
}

implementation edu.brown.cs.webview.taiga.MeurlinClassify {
    using edu.brown.cs.webview.classify.ClassifyMeurlin;

    implements edu.brown.cs.webview.taiga.WebClassify {
        using class Classify = edu.brown.cs.webview.classify.ClassifyMeurlin;
    }
    cost = 40;
}
```

Figure 3. Implementation definitions for the web usage visualizer.

```
outerface edu.brown.cs.taiga.examples.chat.ChatPackage {

    import edu.brown.cs.taiga.base.TaigaTupleSpace;
    available edu.brown.*;
        trait { shared }
    class ChatServer {
        public static final ChatServer getServer(String url) {
            TaigaTupleSpace ts =
           TaigaBase.the_kernel.getTupleSpace("edu.brown.cs.spr.ChatServer");
            ChatServer cs = (ChatServer) ts.get(url);
            if (cs == null) {
              cs = new ChatServer(url);
              ts.put(url,cs);
             }
            return cs;
        }
        static private ChatServer createServer(String url);
        private ChatServer(String url);
        public Connection establishConnection(String uid,ServerCallback cb)
            throws TooBusy;
        public void dropConnection(Connection c);
        public void send(Connection c,String msg);
    }
    interface ServerCallback {
        void handleMessage(String u,String m);
        void handleNewUser(String u);
        void handleUserQuit(String u);
    }
    exception TooBusy;
    abstract class Connection;
    testcase Test1 { .... }
}
```

Figure 4. Outerface definition for a chat server.

downloadable library, the objects would have been local objects. This example also illustrates that outerface classes can have constructors and methods with a predefined implementa-

tion. The former lets the programmer use standard programming techniques when dealing with outerface objects; the latter simplifies the implementation which preserving a more sophisticated outerface.

A common technique in today's programs involves using callbacks, either as an instance of an observer design pattern [10], to simply handle calls from the implementation back to the client, or to allow for asynchronous calls. Our component model handles these through the definition of interfaces within the outerface. The *ServerCallback* interface in Figure 4, for example, is implemented by some client class and passed to the *ChatServer* object to register the client. The implementation handles the fact that this might indicate a remote object in the implementation and that the implementation will have a different name for the callback interface.

Outerfaces also support abstract classes and exceptions. Abstract classes represent private objects that are passed back from the implementation to the client and can only be used as parameters to calls to the implementation. In the chat example, the connection is represented as such a class. These are implemented as suitable remote objects where necessary. Exception declarations in the outerface are used to create corresponding subclasses of java.lang.Exception for the client. The corresponding exception class in the implementation is mapped to the client class when the exception is thrown.

Finally, outerfaces support passing objects by value rather than simply as remote objects. This is done by declaring an "interface class" rather than a standard class. Interface classes are essentially structures. They can have fields and implemented methods, but cannot have any abstract methods. An assumption is made in this case that such items are passed by value; the above restrictions ensure that no remote method calls will be needed.

3.4 Functional Semantics

Central to our component model is the fact that the *outerface includes the semantics of the implementation*. By semantics here we mean not only what the calls do, but also how the implementation addresses issues such as security, privacy, availability, and reliability. Including the semantics in the outerface lets us use arbitrary implementations with a degree of confidence, provides a rational means for choosing among implementations, and offers a basis for ensuring that implementations act as expected.

The most used approach for defining the functional semantics of an interface is to use a natural language definition such as a requirements specification document. Experiences with such informal descriptions have shown that there are many ways of misunderstanding what is really needed and it is extremely difficult to have the descriptions cover all the cases or even be unambiguous in the minds of the implementor. For these reasons, more structured approaches have been taken. The Java Community Process, for example, is a formal but open, committee-based mechanism whereby new interfaces for Java libraries can be proposed, discussed, and formally understood. However, it is cumbersome and typically takes months or years for a new proposed interface to be fully understood and accepted. For the framework we envision, we needed something that was much less cumbersome, much quicker, and that could accommodate large numbers of interfaces and implementations.

An alternative approach is to express the semantics using a mathematical language such as Z [24] or Larch [13] or a high-level specification language. While such specifications are becoming more common, they are still difficult to write and more difficult to get correct, even for relatively simple processes. When you have a complex process as many web services are, writing such specification is often more difficult that implementing the process in the first place.

A more widely used approach is to use contracts as introduced in Eiffel [17]. Contracts attempt to constrain the behavior of a process by defining what inputs are allowed and how the outputs are related. They are typically expressed in the form of preconditions and postconditions on methods or as conditions on a class. More recently, dynamic contracts have been used to specify the ordering and behavior of sets of methods of classes. For example, contracts in the form of AsmL or extended finite state automata have been used to specify more detailed behavioral properties of components [2,3,5]. Contracts are relatively easy to specify, but they don't fully capture the semantics or the intent of the interface.

Another solution can be found in the agile or extreme programming approach to development where test cases are developed first and the implementation is tested continually. The test cases provide the developers with a good sense of what the implementation should do and ensure that the resultant implementation works correctly, at least for the circumstances it was envisioned for.

The problem of semantics is related to the "component trust problem" [18]. The latter problem deals with trusting a component once you know what it does while our problem doesn't even assume we know what the component is supposed to do. The techniques used here are a combination of formal approaches, contracts, and testing [4]. The same techniques are appropriate for our problem.

Indeed, we feel the most appropriate approach is a combination. In our component model, the main semantics of an outerface are specified by a set of test cases defining the implementation's behavior. The framework guarantees that any implementation passes all the test cases of the outerface. Thus, for the *WebClassify* outerface of Figure 2, the system ensures that any implementation passes the three test cases that are included. Users can also restrict outerfaces by defining additional test cases. In this case, the user is guaranteed that any implementation chosen will pass both the original test cases and the new ones.

Test cases are used to control the binding of outerfaces to implementations. When the user requests an outerface, the system searches for implementations that specify (directly or indirectly) that they can be used for that outerface. For each such implementation, the system runs the appropriate test cases and validates whether the implementation can or cannot be used. It will then choose one of the acceptable implementations and dynamically bind the outerface to it. For efficiency, the information about allowable bindings of outerfaces to implementations is cached by the system and shared globally.

In addition, the outerface can include contracts in the form of preconditions and postconditions on methods and general constraints on the behavior of an outerface class. Checks for these are inserted into the code generated for the outerface. If a precondition fails, the call is aborted; if a postcondition fails, then the implementation is viewed as failing and a new implementation will be used. The recovery mechanism here is the same as

that described in Section 3.6. Note that contracts augment the test cases since a failing postcondition will result in a failing test case and hence will

Finally, availability constraints, part of the outerface, are used to determine who can access and use it.

3.5 Security and Privacy

Security and privacy are two sides of the same coin. Security constraints specify what the implementation will do for the client based on authentication of the client. For example, an implementation might specify that a certain method can only be called only if the client has the appropriate certificate or capability. Privacy constraints, on the other hand, let the client specify what it will let the implementation do. For example, a client might specify that the implementation can perform no local writes or that a particular parameter (for example a credit card number) can only be passed to a particular client (a bank) and never saved.

In order for Internet-scale programming to be widely used, both security and privacy must be part of the outerface definitions and both must be enforced by the underlying system. We are currently researching the best way that this can be done, balancing ease of specification, ease of checking, and potential implementations. The framework implementation includes a security context to hold the various certificates and associated information. The current security context is maintained across calls, both local and remote, and is integrated with the Java security context. Research so far has been focused on formalizing different privacy and security specification languages such as P3P [7] or EPAL [15], and on creating an appropriate sandbox that can be used to check privacy and security. Currently, TAIGA lets both the implementation and outerface specify their own privacy constraints and checks that the implementation is at least as restrictive as the outerface.

3.6 Recovery

Recovery from network or component failures needs to be an essential part of an Internet-scale computing framework. This can be seen in the various experiences we have had with WebView and the implementations it was trying to use. The recovery model we use, while not fully implemented, attempts to address the problems.

First, our framework specifically supports the dynamic binding and rebinding of implementations. If an outerface fails, for example due to network connectivity or a failing postcondition, then the framework will automatically unbind and unload the original implementation, choose a new implementation that satisfies the outerface, load and bind this new implementation, and then reexecute the failing call. All this is done so as to be invisible to the user.

This works well in the simple case where the implementation does not manage any state and where there are no remote objects passed back from the implementation. This case, while restrictive, is not uncommon. Most web services, for example, follow this model. The definitions of Figure 2, note through their trait definitions that all three outerfaces can be dynamically rebound in this way.

We are currently working on handling the more complex cases. This first involves defining structures as part of the outerface that represent the information needed to recover an object or a class and requiring that any implementation be able to restart from this class data and recreate objects from the object data. Recovery would then consist of maintaining the appropriate data objects in the client's memory so that the appropriate state of a new implementation could be created. In addition, we are looking at ensuring transactional semantics for particular remote calls.

3.7 Cost Model

The final component of an outerface definition is an cost model that lets the underlying framework choose among different implementations. The cost model we currently provide lets the user define the cost of an implementation as a weighted total of cost elements. The current cost elements include the cost of using a particular implementation, the cost of a particular type of binding (e.g. library vs. web service), costs associated with various traits, costs associated with resource (time or memory) utilization over all the test cases, or costs associated with resource utilization for a particular test case.

The ability to include test case resource utilization here means that the client can define test cases that reflect their important use cases and can then have the system choose the implementation that most appropriately addresses those test cases. Note also that by restricting an existing outerface, a user can specialize the cost model to their own requirements.

4. THE TAIGA FRAMEWORK

The notion of outerfaces for Internet-scale programming is attractive and seems to be relatively comprehensive. However, in order for it to be practical, we need to demonstrate that the component model can be implemented, that it can handle appropriate dynamic binding, that it can support a variety of different types of components, and that it can enable Internet-scale programming.

To this end we have built a prototype framework called TAIGA consisting of three main elements: a package that processes outerface and implementation files, a peer-to-peer kernel built on top of JXTA [12], and a set of utility routines.

4.1 Outerface and Implementation Management

TAIGA maps an outerface description into a Java package that can be used directly by the client. The package is given the same name as the outerface and the classes, interfaces, and exceptions defined in the outerface are defined as corresponding classes or interfaces in the package. Portions of the code generated for the *WebClassify* class of the outerface in Figure 2 are shown in Figure 5.

For each class defined in the outerface, we generate an actual class that uses delegation to provide the appropriate implementation. We considered several alternatives here. One approach for an object-oriented framework is to define the remote object as an interface that is inherited both by the local class and by the implementation class. This is the approach taken in Java RMI. This doesn't work because we want to allow constructors, static methods, and outerface-defined methods which cannot be part of interfaces, and we want to let the implementation have different parameters than the outerface. Using an abstract class instead of an interface gets around some of this but not all, and raises other problems. A second approach is to use reflection and dynamic binding. Here the implementation classes and methods are determined and saved at run time and

```
package edu.brown.cs.webview.taiga.WebClassify;

public final class Classify implements edu.brown.cs.taiga.base.TaigaObject {

    private static java.util.Map object_map = new java.util.HashMap();
    private _TaigaImplements base_object;

public static Classify taigaGet(Object o) {
    if (o == null) return null;
    if (o instanceof Classify) return (Classify) o;
    Classify x = (Classify) object_map.get(o);
    if (x == null) {
        x = new Classify(((_TaigaImplements) o));
        object_map.put(o,x);
    }
    return x;
}

public java.lang.Object taigaGetImplObject() { return base_object; }

private Classify(_TaigaImplements base) { base_object = base; }

public Object getBaseObject()          { return base_object; }

static public java.lang.String findCategory(java.lang.String url) {
    for ( ; ; ) {
        Object s = TaigaBase.setupSecurityPolicy();
        if (TaigaBase.static_object_Classify == null)
            throw new java.lang.Error("Binding failed for
        try {
            return TaigaBase.static_object_Classify.findCategory(url);
        }
        catch (edu.brown.cs.taiga.base.TaigaLostBinding _e) {
            TaigaBase.static_object_Classify = null;
            TaigaBase.rebind();
        }
        catch (java.lang.Exception _e) {
            throw new java.lang.Error(_e);
        }
         finally {
            TaigaBase.resetSecurityPolicy(s);
        }
    }
}

public interface _TaigaImplements {
    java.lang.String findCategory(java.lang.String url)
        throws edu.brown.cs.taiga.base.TaigaLostBinding;
}     //end of subinterface _TaigaImplements
```

Figure 5. Excerpts from code generated from an outerface definition.

method calls use the reflection interface. A local library then could be bound in directly and standard methods for generating SOAP calls could be generated for the outerface independent of the implementation. This gets closer but still does not address the issue of allowing different calling sequences. Moreover, it makes it difficult to support different parameter passing mechanisms.

The solution we finally settled on is to define a subinterface, *_TaigaImplements*, of the outerface class to define the implementation. This can be seen at the bottom of Figure 5. The outerface code then uses this interface. For each method, the outerface manages security, calls the appropriate implementation method, handles the return type, and traps any exceptions. An example of this can be seen in the *findCategory* method in the figure.

The code for an outerface class includes several other bookkeeping methods. The *taigaGet* method ensures that there is a unique outerface object for each implementation object. This ensures that an outerface object passed multiple ways from a one application to another yields the same object even when the applications are independent. The static initializer is used to force implementation binding before the class is used. The *getBaseObject* method is used when an object of this class is passed remotely.

The generated package includes two additional classes. The first, *TaigaBase*, is used to do the binding. It includes a static initializer invoked the first time any of the classes in the outerface are used. This initializer handles finding an implementation and binding the implementation to the outerface. The binding is represented by creating objects that implement the implementation interface for each class. These objects are the ones that will be used for static calls and constructors. This class also provides calls to handle dynamic rebinding of the outerface. In addition, it manages the security and privacy policies for the particular outerface and class. The second generated class, *TaigaTest*, manages testing by invoking all the test cases defined in the outerface and returning appropriate status.

4.2 Implementation Mappings

The decision to use delegation in the code generated for outerfaces means that all the real work of binding the outerface to the implementation has to be done using code generated from the implementation definition. This code has to manage the actual binding of outerface calls to local or remote calls; generate the appropriate local call, SOAP message, or other call sequence; reorder and add default parameters; map return values; and map exceptions. For each implementation our approach generates a new class for each outerface class. This class implements the corresponding *_TaigaImplements* interface to manage this binding. It also generates classes to represent interfaces. If the implementation can be used in multiple forms (for example a library might be usable either locally or as a server or grid computation), then TAIGA generates a set of such classes for each form. This approach has the benefit that no modifications are needed in the implementation code.

The class generated to do the binding for each outerface class only has to consider the outerface and the particular implementation. As such, it is straightforward to generate code to manage the mappings and generate the appropriate type of call. For a local binding, this code takes the initial parameters and uses them to construct the actual call to the local method. For a remote binding, it builds and uses a remote call using extended version of the SOAP protocol. For web services with standard argument types, this is precisely the call that is needed to invoke the service. For calls to other servers, the call will be handled by our own SOAP interpreter which we include as part of a TAIGA implementation. This interpreter uses the Java reflection interface to find the actual methods and thus assumes that the correct names and bindings are generated by the caller. As an example, the local and server implementations for *findCategory* are shown in Figure 6.

Interfaces in the outerface have to be handled as special cases. These will have implementations in the client, but will not in general have implementations in the server. Moreover, the implementation will define its own matching interface and will expect to receive a class that implements this interface. To handle this case, our system generates a stub class that implements the interface defined by the implementation and is available to and included in the implementation. This stub class is specialized according to the type of implementation. For a local implementation we create a bridge class that maps the calls from the implementation interface to the outerface one. A remote implementation provides a more sophisticated class that translates the callbacks into SOAP calls that will be passed back to the calling object.

```
public java.lang.String findCategory(java.lang.String _a_0)
    throws edu.brown.cs.taiga.base.TaigaLostBinding {
    edu.brown.cs.taiga.base.TaigaMessage m = TAIGA_setupMessage(
  "edu.brown.cs.webview.classify.ClassifyXml.findCategory","java.lang.String");
    m.addStringParameter("_a_0",_a_0);
    try {
        Object o = TAIGA_sendMessage(m,java.lang.String.class);
        return (java.lang.String) o;
    }
    catch (edu.brown.cs.taiga.base.TaigaException _e) {
        System.err.println("STUB: Error: " + _e.getMessage());
        throw new java.lang.Error(_e.getMessage());
    }
}

public java.lang.String findCategory(java.lang.String _a_0) {
    java.lang.String rslt = edu.brown.cs.webview.classify.
        ClassifyXml.findCategory(_a_0);
    return rslt;
}
```

Figure 6. Implementation stubs for remote and local implementations.

Thrown exceptions are handled in the mapping functions. Local mappings trap the implementation exception and throw a corresponding outerface exception. Remote exceptions are a bit more complex. These generate a *TaigaException* object with additional information about the implementation exception. The mapping code catches a *TaigaException* and then checks for the appropriate implementation exception names and maps them to outerface exceptions.

Interface classes or structures in the outerface are handled by having the implementation create an implementation version of the structure either from a corresponding SOAP parameter or directly from the client object.

The overhead imposed by this strategy dependent on the type of binding. When the component is bound as a imported library, the overhead is two levels of indirection on each call to the component. The first is to the outerface code and the second is to the implementation interface. Neither of these do any real computation. If the component call does any work at all, this overhead will not be noticed. If the component is remote, the overhead is one indirect call and then the cost of packing and unpacking the arguments in SOAP (or whatever other transport mechanism is used) format. Since this latter cost needs to be incurred anyway for a remote component, the additional cost of using TAIGA is negligible.

4.3 The TAIGA Kernel

The second element of TAIGA is the kernel. The kernel provides services on the current machine and uses a JXTA-based peer-to-peer backbone to access and share global services.

The first service that the kernel provides is global lookup and sharing of outerfaces, implementations, and implementation binding information. The code for an outerface or implementation is generated when the user registers that component with TAIGA. At this point, the kernel also generates an internal description of the component and assigns the component a version number. Where possible, implementations are combined into jar files or shared libraries that cannot later be changed.

Each kernel maintains a local repository for known outerfaces and implementations. Information requests for a particular outerface or implementation, with or without a specific version number, can be broadcast and the results cached throughout the peer-to-peer backbone. Similarly, requests for implementations of a particular interface can be broadcast. The kernel also keeps a cache of known acceptable outerface-implementation bindings to avoid having to rerun the test cases all the time.

Next the kernel provides information about and remote access to servers that implement particular interfaces which are running on the local machine. A remote host can query where the service is running through the backbone This mechanism also has the ability to run implementations on multiple nodes in a grid configuration.

The kernel also provides access to global files. Each node can define a set of directories that will be exported under a global name. This information is shared across the peer-to-peer backbone. If a user attempts to open a global file, the local kernel will find a node that supports that file system and establish a connection with its local file manager. This connection can then be used to support the basic file I/O operations. Finally, the kernel provides an implementation of Linda-like tuplespaces across the peer-to-peer backbone. These are implemented as a TAIGA outerface with a standard implementation.

4.4 The TAIGA Core Library and Object Model

The third element is the TAIGA core library. This is a set of routines that is automatically bound into any Java application that uses an outerface or other TAIGA features.

This library serves several functions. First, it provides a simple Java interface to all the services offered by the kernel. It provides methods to find outerfaces, implementations, and services. It provides implementations of *InputStream* and *OutputStream* that access global files.

Second, the library handles the dynamic binding and rebinding of outerfaces to implementations. The first time an outerface is used, the system queries the kernel bindings to find the most appropriate implementation. If needed, it asks the kernel to download the appropriate jar file containing the implementation bindings. Finally, it uses a class loader to add that jar file to the application and sets up the internal bindings to use the loaded classes. If the outerface fails due to a bad connection or other detectable error and the outerface is marked as rebindable, then the library asks the kernel for an alternative implementation, gets the appropriate jar files, and then replaces the original binding with the new binding.

Finally, the library provides support for remote objects and SOAP calls. To provide a consistent programming model and to achieve our goal of making Internet-scale applications easy to code, we needed to provide an object and type model that provides natural support for remote objects. In particular, our model ensures:

- An object received by process A that originated in process B is the same regardless of the way that the object was passed from B to A (even if it goes through another process C).
- An object passed from A back to A corresponds to the original object.
- An object passed to process A will have its proper type if A knows that type.
- Types for remote objects obey global namespace conventions.
- Objects passed from an outerface to an implementation through a binding will have the implementation type.

- Collections of objects are supported.
- All types passed remotely act as if they were passed locally; the application is not be able to determine, using standard means, whether the implementation is local, remote, a web service, or a grid node.

There are several cases that the library handles. First, the implementation can be a web service that uses SOAP calls and is completely out of our control. Such services use a limited set of parameter types, all passed by value, which we generate correctly as part of the implementation mapping. The more interesting case occurs when the implementation is a service defined as a set of Java classes. Here the implementation is generated by building a jar file of all the relevant Java files and including in that jar file our own SOAP-based interface to receive calls. In this case we let objects be passed by reference between the client and the implementation. These then become remote objects.

Objects that are or could be remote are represented by the class *TaigaStub* or its subclasses. (All classes generated for remote bindings inherit from this class.) This class contains four pieces of information about the object. First, it contains a remote id. This is a string that uniquely identifies the object and provides information about the server or process that holds the original object. Second, it contains a connection structure containing the communications channel to the original object. This is needed if calls are made on the object and it is reconstructed from the remote id if not previously known. Third, it contains the type name of the original object. Because remote objects must have types that are defined in a global namespace, this should uniquely identify the object type. Finally, it contains information about the implementation if the object is derived from an outerface class. This includes the name of the implementation type both locally and remotely and the remote id of the implementation object.

Managing remote objects is done by two different routines. The first constructs a SOAP parameter for a remote object, essentially creating the wire representation of the object. This includes a flag indicating the object is remote, the remote id of the object, the name of the source type, and information about the implementation. The second routine is given this information and the desired type and creates an appropriate remote object.

Before creating a wire representation of an object, the binding routine has to determine which object to send. Normally it just sends the object passed in. However, if the object being passed is an outerface object and the receiver expects an object of the corresponding implementation type, then the routine will actually send the implementation object associated with the outerface object rather than the outerface object itself.

The routine to pass an object to a remote server first checks if the object is a standard one within the SOAP framework (e.g. *String* or *Integer*). If so, it uses standard SOAP conventions to pass the object by value. Next it checks if the object is declared to be passed by value. If so, it creates an appropriate SOAP structure object and passes it that way. Otherwise, it checks if the object being passed is derived from a *TaigaStub*. This can either indicate that is already a remote object or it is implementation object for a remote server. In either case, it passes the information associated with the stub in the SOAP call.

If none of these cases holds, the system next checks if the object is derived from an outerface class. If so, it find the associated implementation object and builds an implementation string to match it; otherwise the implementation string is left empty. Next the system finds the existing remote id, or, if this is the first time the object is used remotely, creates a new remote id for the object. The resultant information is then passed in the SOAP call.

This scheme allows any object to be passed remotely, preserves the properties of remote objects even if the receiver doesn't know the types involved, and preserves information about implementations for outerface objects.

Processing also needs to be done to convert the SOAP wire representation into an object within the receiver. The library routine uses both the SOAP representation that is being passed and the expected type of object. The latter is available both for calls since the binding routine fully identifies the routine being called and for return values where the binding routine knows the resultant type.

This routine first checks if the remote id of the object corresponds to a known local object. If so, the local object is used. Next it checks if the remote id has been used before for a remote object. If so, it returns the same object.

If neither of these cases holds, then it looks at the type name of the object being passed and uses reflection to see if this class is known to the receiver. If so, it looks at the expected type of the parameter. If this is a Java interface, then it replaces it with the actual type. If the received type is an instance of *TaigaStub* the routine similarly replaces the expected type with the actual type. Finally, if the received type is an outerface class and the object has implementation information and the implementation matches the implementation associated with the outerface, then the framework will find the unique outerface object associated with the implementation and will return this, building a new such object if necessary.

In any case, the system next checks whether the passed in type and the desired type match and are both stub types. If so, it builds a new instance of the desired type using the stub information passed in. Otherwise, it creates a new stub object of a generic stub type again using the information passed in. In either case it associates the remote id with the new stub object.

This mechanism is sufficient to ensure that objects passed to the implementation have the type that the implementation expects and that objects can be passed through multiple servers and still be correct.

5. STATUS AND FUTURE WORK

This paper has outlined a component model and its implementation suitable for Internet-scale programming based on the merger of web services, grid computing, peer-to-peer computing, and the open source movement. The underlying framework uses a peer-to-peer backbone to support a global registry of independent outerfaces and implementations and does dynamic binding between the two, guaranteeing that an implementation embodies the intent of the interface.

The component model allows the independent development of outerfaces and implementations while preserving type consistency, a standard programming model, and the use of existing components. Its novel contribution is the inclusion of the semantics required of an implementation as part of the interface. Semantics here is taken broadly to include functionality, security, privacy, economics, and recovery. The practicality and usability of this model is demonstrated by the TAIGA implementation that provides appropriate global services and

defines a suitable type model for remote objects that is consistent with the component model and works for Internet-scale programming.

We are continuing to work on extending both the component model and the underlying framework embodied in TAIGA. In addition we are currently working on additional applications that illustrate its potential and test its capabilities. We are also working on some of the additional enhancements that will be needed to simplify Internet-scale programming such as automatic and invisible transactions.

Acknowledgements. This work was done with support from the National Science Foundation through grants CCR9988141 and CCR0086057. Margaret Benthall helped with the JXTA kernel.

6. REFERENCES

1. Sudhir Ahuja, Nicholas Carriero, and David Gelernter, "Linda and friends," *IEEE Computer* Vol. **19**(8) pp. 26-34 (August 1986).
2. Mike Barnett and Wolfram Schulte, "Spying on components: a runtime verification technique," *Workshop on Specification and Verification of Component-Based Systems*, (October 2001).
3. Mike Barnett and Wolfram Schulte, "The ABCs of specification: AsmL, behavior, and components," *Informatica* Vol. **25**(4)(November 2001).
4. Antonia Bertolino and Andrea Polini, "A framework for component deployment testing," *Proc 25th ICSE*, pp. 221-231 (May 2003).
5. Ana Cavalli, Bruno Defude, Christian Rinderknecht, and Fatiha Zaidi, "A service-component testing method and a suitable CORBA architecture," *Proc 6th IEEE Symp. on Computers and Communications*, (2001).
6. Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker, "Making Gnutella-like P2P systems scalable," *Proc. ACM SIGCOMM 2003*, (Aug 2003).
7. World Wide Web Consortium, "The platform for privacy preferences (P3P) specification," http://www.w3c.org/TR/P3P (2002).
8. Scott Draves, "Electric Sheep," *http://electricsheep.org*, ().
9. Eric Freeman, Susanne Hupfer, and Ken Arnold, *Javaspaces Principles, Patterns, and Practice*, Addison-Wesley (1999).
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley (1995).
11. A. Ganek and T. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal* Vol. **42**(1) pp. 5-18 (2002).
12. L. Gong, "JXTA: a network programming environment," *IEEE Internet Computing* Vol. **5** pp. 88-95 (2001).
13. J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch family of specification languages," *IEEE Software* Vol. **2**(5) pp. 24-36 (March 1985).
14. Min-Yen Kan, "Web page classification without the web page," *Proc 13th WWW Conference*, (2004).
15. International Business Machines, Inc., "The enterprise privacy authorization language (EPAL) specification," http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification (2003).
16. Nenad Medvidovic, "On the role of middleware in architecture-based software development," *SEKE '02*, pp. 299-306 (July 2002).
17. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall (1988).
18. Bertrand Meyer, "The grand challenge of trusted components," *Proc. 25th ICSE*, pp. 660-667 (May 2003).
19. Sun Microsystems, "The Jini Architechture Specification," *http://www.sun.com/software/jini/specs/index.xml*, (June 2003).
20. Johann Oberleitner, Thomas Gschwind, and Mehdi Jazayeri, "The Vienna component framework: enabling composition across component models," *Proc. 25th ICSE*, pp. 25-35 (May 2003).
21. Ran Rinat and Scott Smith, "Modular Internet programming with cells," *Proc. ECOOP 2002, Springer-Verlag LCNS 2374*, (2002).
22. Ashish Shah and Dennis Kafura, "Symphony: a Java-based composition and maniuplation framework for distributed legacy resources," *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems*, pp. 2-12 (May 1999).
23. Clemens Szyperski, "Component technology - what, where, and how?," *Proc 25th ICSE*, pp. 684-693 (May 2003).
24. J. B. Wordsworth, *Software Development with Z*, Addison-Wesley (1992).
25. P. Wyckoff, "T Spaces," *IBM Systems Journal* Vol. **37**(3)(1998).