Designing Internet-Based Software

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912
spr@cs.brown.edu

Abstract

Next-generation software is going to be Internet-based. It will involve web services, peer-to-peer data sharing, grid-based execution, and open-source components, and will have to meet the continuously changing demands of a broad set of users. In such an environment software is out of the control of the developer. The requirements for the software will be continuous changing to meet user expectations and demands. Many of the components of a software system will developed independently and will change without notice during design, development and even after deployment. The underlying infrastructure, based on the Internet and on independent grid-based computers, will be inherently unreliable and out of the developer's control. One essential issue that faces software design and development is how to cope with this lack of control. We propose a component-based mechanism which separates component interfaces and implementations, includes semantics, in a broad sense, as part of the interface, checks these semantics dynamically against the implementation, and allows for dynamic selection (and reselection) of implementations. In this paper we show how different aspects of the semantics including functional, recovery, security, and economics can be encoded and checked, and argue how this approach can address many of the problems that Internet-based software will face.

1. Introduction

Software design today is considered a difficult problem. Current trends in software development, leading to the development and design of what will effectively be Internet-based software, will make this significantly more difficult. To cope with software design both now and in the future one needs to understand its unique problems and determine ways of addressing these problems. This is the goal of this paper. We start by analyzing why software design is different and what the future of software development will imply for software design. We then suggest a component-based framework for software design and development that has the potential to deal with many of these difficulties, both in current and in future software development. An overview of this framework is given in Section 2. The key feature of the framework is that interfaces include semantic information and that this semantic information is used to select and verify potential implementations. The subsequent sections detail how these semantics might be practically specified and checked. We conclude by discussing our experiences and future work.

1.1 Why Software Design is Difficult

Design can be viewed as a combination of constraint satisfaction and value maximization (or equivalently cost minimization). In doing design, one is attempting to find a solution to a particular problem that satisfies the set of constraints representing the various requirements while maximizing the value and minimizing the cost of the resultant product. For software design, this means finding a solution that meets the requirements and specifications of the software system while minimizing the cost of building the software and maximizing its quality.

So why is this so difficult. Many reasons have been offered in the past. One is the inherent complexity of a software system. Today's software systems consists of millions of lines of code, each of which is non-standard and needs to be "designed" in some way or other. Other forms of engineering typically have a limited repertoire of standard parts and relatively limited ways of connecting them. Software is much more flexible and hence has much greater potential complexity and more sophisticated design problems. Yet software designers have developed a variety of techniques such as modularization, object systems, components, and product lines, that deal directly with this issue and should make software design practical.

Another proffered reason for the complexity of design is that software design involves many different dimensions that somehow have to be reflected in the final product. Such dimensions extend beyond the basic functionality of the system to include security, privacy, efficiency, user interactions, interactions with real-world devices, etc. All these dimensions, some more concrete than others, need to be reflected in some way in the final product, the source code. Attempting to capture these dimensions in the limiting abstraction of source code can be difficult. Yet here software design is not much different from other forms of engineering design. An architect needs to consider plumbing, electrical, human factors, costs, and other such constraints when designing a house. Moreover, various techniques such as feature analysis, feature driven development, separation of concerns, and aspect-oriented programming have been developed and used somewhat successfully to address these issues.

We feel that the real differences in terms of the complexity of software design are best understood by looking at the set of constraints that one is trying to solve when doing a design. As noted by Manny Lehman [19], this set is infinite. But people are incapable of enumerating, evaluating, or even comprehending an infinite set of constraints. Instead designers try to enumerate a finite subset of the overall constraint set and design to meet those constraints. They typically select the constraint subset through an appropriate process of prioritization, choosing those that they think are the most important as the ones they will design against and then hoping this is sufficient.

Presuming designers can identify the most important constraints, they should be able to develop software designs that work. So what is the actual problem? It is not that the set of important constraints is too large or that they have great difficulties in setting priorities. Good designers, those with experience in the problem domain and experience with software systems, can relatively easily identify a manageable set of constraints that are the essential ones for the success of a particular software system.

The real problem here is that the set of constraints and their priorities are constantly changing. Moreover, while this is true in other fields of design as well, the change in the software area, especially today in the evolving information age, is so rapid that a year or two after a piece of software is originally specified, there are often very different constraints (e.g that a system originally specified to only run on Windows now has to run on a Linux box, or that a client-server application now needs to have a web interface), or the priorities of the various constraints change significantly(e.g. where security was incidental before, it is now of primary importance). While buildings and other physical entities are seen as inflexible and are not assumed to be dynamically changeable to meet the relatively slowly changing constraints of their fields, software is assumed to be flexible and is supposed to adapt to the changes in real time. Software design is put in the unenviable position of having to meet unknown and unspecified constraints and changing priorities in order to be considered successful. In this sense software design is an attempt to design the undesignable.

Despite these problems, software design has progressed substantially over the past half century. The field has developed a variety of techniques and methodologies that are sufficient to deal with a wide range of systems.

Software design has always been considered difficult. Decades ago, when a large program was a hundred thousand lines of code, programmers struggled with the planning and development of such systems. Today, with better programming languages, tried and proven modularity concepts, various forms of data abstraction leading to object-oriented programming, powerful libraries and COTS components, and the codification of various metaphors as design patterns, designing most such systems is relatively simple, and can almost be considered a solved problem.

The problems we face today involve designing systems that consist of tens of millions of lines of code. These systems are large enough so that they can't be understood or even designed by a single person. Moreover, there shear size and complexity overwhelms the abstraction power of the various techniques that have been developed for smaller programs. However, we are well on our way, out of necessity, to developing techniques that work for these programs as well. The main approach that is being taken here is to raise the level of abstraction. This is evident in the blossoming field of software architecture and in related areas such as architectural design patterns and large-scale components such as web services.

1.2 Internet-Based Software

While these and similar advances are necessary and important efforts, they address the past and not the future. Future programs are likely to be quite different and to face very different problems. As such, they are going to require very different design techniques. Understanding and solving yesterday's or today's design problems may be of little help when addressing future problems. For example, one probable property prevalent in future programs will be that the programmers no longer control the program. This violates a basic underlying assumption of all today's design techniques that we have the power to design what we want.

To understand this and understand what future programs will look like, we need to look at the direction that programming is taking. The concept of a program is changing from a local, self-contained object into an Internet-scale, pervasive, self-organizing, omnipresent entity. This can be seen in 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.

Web services represent a loosely-coupled component framework using the Internet as the interconnection mechanism. The components have interfaces defined using a standard description language (WSDL), and interact, either with a browser or with other programs, through a standard wire protocol (SOAP). With Microsoft pushing .NET as a framework for web services, and Sun offering similar capabilities using J2EE, it is only a matter of time before there will be large numbers of available components that can and will be used in building new systems.

Grid computing utilizes either idle cycles or idle machines to handle the variable load of a complex computation. Building on long-term experience with distributed processing frameworks, it tries to offer substantial computing power on demand by splitting an application into loosely-connected communicating components. With companies such as Oracle and IBM providing frameworks for grid computing, a wider range of applications are starting to make use of this approach.

Peer-to-peer computing involves loosely coupling large numbers of machines to share data, files, or computations. While most widely known for its capability to do file-sharing for entertainment purposes, the technology offers the promise of providing wide-ranging ad hoc networks where the individual components can easily access data or otherwise communicate with each other without needing to know all the other machines or the changing structure of the network.

Autonomic software is software that is effectively self-healing [11]. While fault tolerant hardware and software has been around for a long time, only recently have the specialized techniques used here been introduced into everyday machines and programs. As software become more and more complex, the need for fault tolerance becomes much greater. Thus, IBM and others are attempting to put together both hardware and software components so that the overall system can tolerate and recover from failures of the individual components.

Open-source software is a social framework where programmers write software that is then made available to and improved upon by other programmers. The end effect can be robust, powerful, and reliable software that benefits not only the authors, but all potential users. The success of open source in developing the widely used GNU tools, the Linux operating system, and the Apache web server has and continues to lead to additional projects being done using this framework. It is only natural that more and more software will be developed this way in the future.

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 [4], Napster, and electric sheep [7] are examples of current Internet-scale applications, with more to come.

Dealing with this new reality will require changing the way one thinks about programs and programming. One cannot continue to think of a program as a self-contained entity with only local effects that the programmers control. Instead, developers will need to work in terms of a global system where they do not control most of the components or interactions. They need a means for designing and 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 their control, and where the components themselves evolve outside of the control of the software system.

Programmers have always had to deal with outside factors. However, most of these factors were somewhat under their control. Compilers, loaders, operating systems, and tools all change over time and systems need to adapt to these changes. However, programmers typically could choose when to upgrade the operating system or when to change compilers.

Uncontrolled evolution is more difficult. The FIELD system integrated a variety of programming tools including the debugger, editors, *make*, and *rcs* using a message-passing mechanism and tool wrappers [24,25]. Here we encountered one of our first examples of external evolution. The wrapper for the debugger was the most problematic. It operated by parsing translating FIELD commands into debugger (*gdb* or *dbx*) commands and then parsing the debugger output to determine what was going on. Unfortunately the debugger output syntax was not considered part of its interface by its developers and hence changed frequently and significantly. The result was that every time a new version of either debugger became available, we had to make significant changes to the wrapper. Moreover, because we had to work on multiple platforms, these changes tended to occur relatively frequently.

A more recent experience is even more telling. For one of our visualizations we needed to get the OpenDirectory classification for a web page. The Google web service provides a programmatic way of making Google web searches with the results returned as structures. Part of this structure is the OpenDirectory category for the found page. By searching for the particular page we were interested in, we were able to get a fast and accurate classification. However, Google changed their underlying framework (but not the web service) so that the category was no longer computed for recent pages and we suddenly found that pages that previously had a category, no longer did. Our application ceased to work.

In terms of software design, this means we have to develop methodologies for designing systems that are out of our control and that evolve in unspecified ways even as we are doing the design. This is like building a house where the properties of the materials being used, such as the strength and size of the lumber, can vary without our knowledge. We are faced with an impossible task: designing such systems essentially involves designing the undesignable.

2. Semantics-Based Abstraction

Designing programs in the face of continuous change requires a flexible approach that can provide the necessary abstractions in a safe and dynamic manner. Abstraction at all levels is central to design; it is the basis for most of the current approaches to software design that are actually used. However, today’s abstractions are too rigid to deal with constantly changing constraints or software fragments that are out of the control of the programmer.

Designing Internet-based software 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 provide an appropriate set of abstractions at all levels while dealing directly with these various new issues and their corollaries.

We have been developing and exploring such a component model [26,27]. This current model separates interfaces from implementations while attempting to ensure:

- Existing web services, libraries, and other component implementations such as open-source libraries can be used as components without modification;

- Interfaces can have multiple, independent implementations that the system can choose between using an appropriate cost model.
- Implementations actually implement the interface as intended by the interface designer;
- 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 [31];
- Interfaces support classes and objects and can contain static methods, constructors, and concrete methods.
- Implementations can be bound and rebound dynamically.

2.1 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 [22]. 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 [30]. Frameworks like the VCF [23] 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 based on decoupling 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 [34] 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.

2.2 The TAIGA Foundation

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.

This component model is embodied in a prototype framework called TAIGA. TAIGA provides the tools needed both to interpret component interfaces and implementations and to support Internet-programming based on these components.

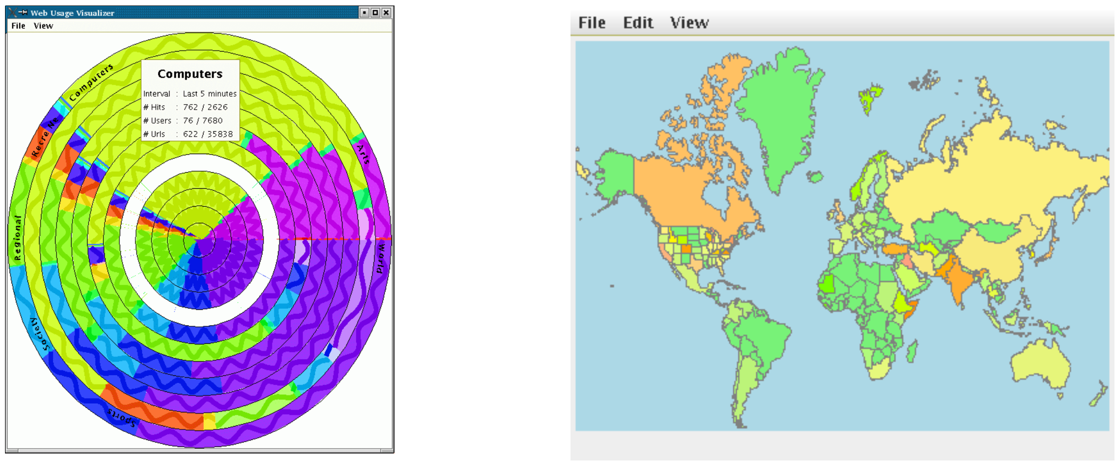
The underlying support mechanism of TAIGA is a peer-to-peer system that can work either on top of JXTA [13] or on top of our own hierarchical DHT implementation. This peer-to-peer backbone supports the sharing of interfaces and implementations and, at the same time, provides a global file system and shared data facilities. Interfaces and implementations are registered and assigned unique version numbers. When an application first tries to use an interface, the peer-to-peer system finds potential implementations and chooses the most appropriate one given the application’s constraints. The binding occurs at run time and can be changed dynamically to handle fault recovery, broken network connections, or even dynamic upgrades of the components. This approach to dynamic binding is more general and broader-based than previous approaches such as [20,28].

The prototype also provides an initial approach to global data sharing. It uses a global name space where each computer has a unique hierarchical name. On top of this it supports a simple global file system with file sharing and the ability to create and write files. It also supports shared tuple spaces based on Linda [1] and shared SQL databases.

TAIGA uses a separate interface definition language to define what is essentially a Java package. This language is designed to work in an open-source environment where implementations are done and controlled independently of the interfaces. To differentiate our interfaces from Java interfaces, we call ours an *outerface*. Our approach differs from other interface languages such as that of Microsoft ActiveX or CORBA in that *outerfaces define both the syntax and the semantics of the potential calls*. By semantics here we mean not only what the calls do, but also how the implementation addresses issues such as security, privacy, availability, economics, and reliability.

Once we have a way of checking implementations against an interface, we have a means for ensuring that changes outside the software system can be controlled. TAIGA supports four different operations. First, a user can register an outerface. This generates a jar file that can be programmed against and makes the outerface available for general use. Second, a user can register an implementation. This associates a web service, a library, or a server component with a set of outerfaces. Here a set of jar files is generated containing the code for internal components and containing the code needed to connect to external components. While implementations can be registered arbitrarily, they will not be used by TAIGA until they are bound. This, the third operation, compares the semantics in the outerface with the code in the implementation, running any test cases, doing static checking as appropriate, and generally matching the specifications. It is only when the system is assured that the implementation is acceptable, that it enables it to be used by applications. The fourth operation is internal to TAIGA. When the user first uses an outerface, TAIGA will search for an appropriate working implementation and dynamically bind it into the user’s program. If the component should fail for any reason, TAIGA can unbind the current implementation, find another working implementation, and dynamically replace the old implementation with the new one.

Evolution of the design, software, or underlying systems can be handled in this framework in various ways. As developers understand what functionality they want from a component, they can create test cases for that functionality. TAIGA lets the programmer extend an outerface with additional test cases and then finds implementations that pass not only the original test cases, but the new ones as well. This can be used to dynamically change the software requirements and then find new components that meet the updated requirements. If the existing external components fail the test cases, other implementations will be chosen. Implementations such as web services that might change in hidden ways can be tested dynamically against their interfaces to ensure they haven’t changed in significant ways. As outside components change, their developers can register a new version of them with the system. TAIGA will detect this new version and, if it passes the appropriate semantic tests, will use it.



Computers

Interval : Last 5 minutes
Hits : 762 / 2626
Users : 76 / 7680
Urls : 622 / 35838

FIGURE 1. Sample applications built using TAIGA. The left one shows what people are currently browsing on the web while the right one shows where news is currently happening.

As a test case and an example of how TAIGA can be used, we have developed an visualizer of what people are looking at in their web browsers shown in Figure 1. This application has three outerfaces, one for recording web pages as they are viewed, one for managing the data files that retain the history, and one for finding the OpenDirectory category of a web page based solely on its URL. We provided three different implementations of the latter outerface, one based on the Google web service, one based on MeURLin [16], and a component we wrote that builds a decision tree from the OpenDirectory database. This latter outerface has shown both the lack of programmer's control over next generation software and the ability of TAIGA to cope. When the MeURLin server in Singapore is unavailable (which it regularly is), TAIGA would automatically switch from that to an alternative implementation. When Google changed their web service without announcing it so that the OpenDirectory category was no longer computed for new web pages, the test cases failed for this implementation and TAIGA no longer allowed it. When OpenDirectory updated their database with invalid XML and our categorizer failed, TAIGA detected the failure. When we wanted to modify our application so that specific web pages were categorized a certain way, we simply added appropriate test cases and TAIGA rejected the implementations that were not particularly accurate.

A second example program using TAIGA, again shown in Figure 1, involves a visualization of where news is currently occurring. Here we developed outerfaces to represent a map projection, a client for managing multiple news sources, a crawler that periodically looks at all stories for a particular news site such as cnn.com, a manager that keeps track of available crawlers, and a parser that takes a URL and tells what country or state the news story is about. We have developed multiple implementations for the map projection, simple crawler and manager implementations, and two implementations of the country finder, the first based on finding by-lines and the second on looking for the names of countries and principle cities. This system was designed to demonstrate the grid computing and recovery aspects of TAIGA, with the different crawlers and their URL parsers working on different grid nodes and the system able to readily recover as these nodes go down and even when the news manager goes down.

The key to TAIGA and the key in designing next generation, Internet-based software, is to include the proper semantics as part of the interface. These semantics have to be comprehensive enough to address the important issues that developers will address as the software evolves. They have to be expressive enough to address real applications. They have to be flexible enough to allow multiple implementations and evolving implementations. They have to be concrete enough so that they can actually be checked. We feel that we have started toward this goal with the current implementation of TAIGA. In the next few sections we detail what we have done and the large amount of work that remains to be done.

3. Functional Semantics

The clearest form of semantics that we need to deal with are the functional semantics that describe what the implementation for a particular outerface is supposed to do. What each method in the outerface does needs to be expressed in a practical and checkable manner. One approach is to express the semantics using a mathematical language such as Z [33] or Larch [14], or a high-level specification language such as Alloy [15]. A more widely used approach is to use contracts as introduced in Eiffel [21] or in JML [2,18]. 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.

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.

Our approach to functional semantics uses a combination of these. The main semantics of the outerface are specified by a set of test cases defining the implementation’s behavior. The system guarantees that any implementation passes all the test cases of the outerface. In addition, the outerface can include preconditions and postconditions on methods and general constraints on the behavior of an outerface class.

Test cases have the advantage that they are easy to check dynamically. One merely has to run the component see if the test case succeeds or fails. In practice, however, this has proven more difficult than anticipated. More formal approaches, such as contracts, are both much more difficult to write and much difficult to check, generally being checkable statically only in limited circumstances. Test cases also have the advantage of offering a broader range of specifications. While it is possible to specify test cases that check a method that purports to tell what country a particular web page is about by providing sample web pages where the answer is known, it would be virtually impossible to specify the behavior of such a method formally for any particular input.

Test cases however, even combined with contracts where this is feasible, do not address all the problems that designing next generation software will face. One problem is that, just as it is difficult to describe formally what a method should do, it is often just as difficult to specify a set of test cases for a complex application. A second problem is that the test cases themselves can be fairly complex and will tend to obscure the outerface definition. A third and potentially more serious problem is that the test cases need to be run in a testing environment where they don’t affect live data or active systems.

We have tried to use test cases to define the semantics of the two example applications described above. While some of the items are easy to test, many others are quite difficult. For example, consider testing for URL classification, which should be relatively easy. Our first test case here chose a predefined set of reasonably well-known pages (e.g. www.ibm.com, www.nytimes.com, www.cs.brown.edu, our home page), and had the classifier check whether these were classified correctly. The first problem was what is “correct”. Should the Brown computer science home page be listed as reference (which is the closest category for a top-level university page) or should it be listed as computers since it contains a lot of computer-related material? Different categorizers labeled it differently. Another example was in the manager for the news visualizer. This component is supposed to keep track of the different crawlers that are available and return instances of those crawlers to the caller. Creating a test case for this, however, can only be done if one has the crawlers working and even then, success might depend on the nature of the crawlers.

The crawler for the news visualizer and the implementation that determines the country for a given news story pose other problems. In both cases, a full test requires accessing an unknown set of web pages and doing the analysis. We could test the crawler by creating a snapshot of a news site hierarchy and then crawling that, but this requires that the test cases have access to the sample hierarchy. To be safe, this means that the sample hierarchy has to be packaged as part of the test case and somehow made web-accessible from wherever the testing is being done. A second test case for the URL classifier illustrates a related problem. Here we wanted to test the performance of the classifier so the test case provided it with several thousand different URLs to check. These URLs could be kept in a data file, but then the data file would have to be part of the test case. Alternatively, the test case could be large enough to contain the necessary data internally. This, however, makes it too large to be a convenient part of the outerface specification.

Testing an implementation should not be the same as running it. It should be possible to test a banking application without removing all of ones money (or providing some account with large sums). Current test cases achieve this by running on special accounts or in a simplified testing environment. However, when one wants to test web services on other hosts and other components that are outside of ones control, this is often not possible. Even in our simple applications, to test the news crawler manager, one needs to actually run crawlers on various grid nodes and those crawlers are actually going to go out and find web pages, record history, etc. To do testing correctly one needs to provide a distributed test environment that isolates testing an implementation from actually running the application.

Our current work in TAIGA is attempting to address these issues. We have already provided facilities that let the outerface reference external test case files. We have developed the mechanism needed to support the inclusion of data files as part of test cases. These let us create more sophisticated test cases that can be applied by remote users without cluttering or obscuring the outerface definitions.

Next, we developed an underlying framework whereby TAIGA creates, maintains, and passes a context for all calls. This context includes the notion of a test case and is designed to interface with a sandbox implementation that will simulate external files and databases. We developed a sample implementation of such a sandbox [3] using interposed libraries, and have been working on developing a more realistic implementation based on virtual machines. This type of an approach can provide a safe environment for testing applications that are self-contained and

TAIGA-based without requiring that the application be aware of the test cases of the fact that it is being tested. It does not address the much more complex issues involved in testing black box, remote web services in a safe manner.

The most important problem from a design point of view, that we have yet to deal with, is defining semantics for methods that are inherently untestable, for example, testing programs where the output can not be prespecified or where determining the proper output can only be done by running the implementation. For example, how would one check if a map service actually returned a image or map of a specified region without knowing in advance exactly what map should be returned. While we support dynamic (and eventually static) checking of contracts, these often do not help in these difficult cases. The question here is what other forms of functional semantic specification might be used and can these be made both practical to specify and practical to check.

4. Recovery Semantics

Recovery will be an essential part of an Internet-based application. Designing such applications requires understanding that components will fail and evolve, and that the application needs to continue processing. Because of this, we felt it was important to include information on how to recover from the failure of a component as part of the outerface definition. This lets TAIGA throw away an implementation that violates its contract, detect and recover when remote web services are inaccessible or change, and generally handle errors in components that are outside of the control of the developers.

Because of the way that web services have typically been defined and used, we found it convenient to distinguish between two types of recovery. The first occurs when the application can think that it is dealing exclusively with a web service. Here the effect of calls to the web service are only dependent on the actions of this application. Web services that provide information such as the location of a zip code or a interest calculation or Internet search or a map visualization are some examples here. This characterizes many of the current web services and affords a relatively simple recovery model. The second occurs when the web service supports interactions among multiple clients. This is a significantly more difficult problem.

To handle the first case, we allow outerfaces to provide a recovery model. This model is defined as part of the outerface to reflect the state of the implementation. This state is represented as a set of model variables. Each method call in the outerface is then accompanied by a description of how the state is affected by the call through code that modifies these variables based on the calling parameters if the call succeeds. In addition, we require that the outerface provide a "recovery" method, that is, a method that can restart the outerface given the state of the model variables.

An example of such a model for the client outerface which supports weighting different news sources is shown in Figure 2. Here the model is defined for the outerface class Client in the clauses starting with "model". The model consists of one variable, a map of source names to weights. Creating a new Client object instantiates a model for that client by initializing the variable. Adding a source, calls the implementation add function and updates the model by adding the source and weight to the model. Removing a source similarly invokes the implementation method and removes it from the model. The existence of the model implies that the implementation has to

```
outerface edu.brown.cs.newsview.taiga.NewsClient {

description {{
        This outerface provides an interface between a client and the news
        crawlers and parsers.  The interface serves lets the particular
        user select and weigh different news sources.  It also handles the
        manipulations needed to merge multiple sources into a single value
        set.
}}

import java.util.*;
requires edu.brown.cs.newsview.taiga.NewsCrawler;
requires edu.brown.cs.newsview.taiga.NewsManager;

trait {
        rebind=true;
}

class Client {
        model {
                Map<String,Number> source_set
        }

        public Client()
                model {
                        source_set = new HashMap<String,Number>();
                }
                public void addSource(String name,double weight)
                        model {
                                source_set.put(name,weight);
                         };

                public void removeSource(String name)
                        model {
                                source_set.remove(name);
                        };

                public ClientValueMap getValues();
}

interface class ClientValueMap {
        public Map<String,Number> world_values;
        public Map<String,Number> state_values;
}
}
```

FIGURE 2. An example of an outerface with a recover model.

provide a constructor for Client that takes a single argument which is the map representing the model variable.

TAIGA implements such a model by compiling the model code as part of the bridge code that maps calls to outerface routines into calls (either direct or indirect through a SOAP interface or other mechanism) to the implementation. This bridge code also provides for a handler that is called if any of these calls fail. This handler checks if the outerface supports the rebind trait (which is also specified in the figure), and, if it does, will create a new object using the current recovery model and repeat the call using the new object. Note that this technique also extends to the simpler case where the implementation is stateless and it is sufficient to simply rebind the outerface to a new implementation. This is indicated by a rebind trait with no model specification.

This type of recovery has proved itself quite effective. We have been able to terminate implementations (either on purpose or because the machine there were running on rebooted) and not affect the overall system. Unless we look at the logs, we don't even notice that the implementation had migrated or changed. Moreover, most web services and many other remote components are written so that this type of a recovery model is appropriate.

The model is not complete, even for the single-user case. It currently does not take into account calls that might throw declared exceptions where changes to the model are dependent on the exception thrown. It does not allow the model code to make use of the return value from the call. While it handles state-based recovery for classes such as Client where there is an explicit constructor, it only handles stateless recovery for static classes. All of these extensions would be relatively easy to add, probably using a JML-like syntax and a non-constructor initialization method with a standard name.

The model also assumes any data stored on the back end is recoverable either by restarting the object or by the back end itself. This assumption is probably not the safest one to make, especially when the application might change implementations associated with the outerface. What is actually needed here is a transactional model where transactions work across calls. This could be built on top of something like the Java Transaction Service [5] in conjunction with Java-code based transactions [8].

Finally, the model does not handle the more complex case where the component being implemented maintains a state that is shared among multiple clients, for example if it is providing a communications substrate between users or shared access to a resource. This requires a recovery model that is kept as part of the implementation rather than being kept separately for each client. Doing general purpose recovery in this framework is much more difficult and is another question that has to be addressed before a system like TAIGA can be really practical.

5. Security Semantics

Another essential feature of Internet-based applications will have to be their attention to security and privacy. Before web services and open source can be used on any scale, they have to offer a high degree of trust. This can come from the service being provided by a trusted agent such as a bank or other large company. However, in the long run, this trust is going to have to be built into the components themselves.

Security and privacy relate to what an implementation, be it a web service, library, or other component, can and should do for an application. In the simplest terms, security or access control refers to restrictions that the component will make based on who the application can prove they are, while privacy issues typically are restrictions that the user wants to impose on the implementation component.

TAIGA includes security and privacy specifications as part of both outerface and implementation definitions. The security model is an extension of the Java security model [10,12,17]. There are a set of permissions that include the various Java permissions such as *FilePermssion*, *SocketPermission*, or *SQLPermission*. Each method in the outerface or implementation can either require or ensure that certain permissions are available. In both cases, this can be done either globally for all methods or can be specialized for a particular method.

An example of an outerface and implementation with security semantics is shown in Figure 3. The outerface first says that the default access for the implementation is to have no access to any files except those in */tmp* and to have no network access. Then it qualifies this for the *checkBibTex* method to allow the file designated by the first argument to be read, the file designated by the second argument to be written, and sockets to be created to two specific sites. Finally, the check

```
outerface edu.brown.cs.taiga.examples.refs.ReferenceChecker {

description {{
        This outerface is designed to check bibtex references for consistency
         and completeness, possibly using an external source such as CiteSeer.
}}

security ensures {
        FilePermission("<<ALL FILES>>","none");
        SocketPermission("*","none");
        FilePermission("/tmp/-","read,write");
}

class ReferenceChecker {
        static void checkBibTex(String file,String out)
                security ensures {
                        FilePermission(file,"read");
                        FilePermission(out,"write");
                        SocketPermission("www.citeseer.org","connect");
                        SocketPermission("www.researchindex.com","connect");
                };
        static String checkReference(String ref)
                security ensures {
                        DataPermission(ref,"nowrite");
                };
}   // end of class ReferenceChecker
}       // end of outerface ReferenceChecker


implementation edu.brown.cs.taiga.examples.refs.SimpleChecker {

implements edu.brown.cs.taiga.examples.refs.ReferenceChecker {
        using class ReferenceChecker = edu.brown.cs.taiga.examples.refs.SimpleCheckerImpl {
        using checkBibtex = checkBibTex(String in,String out)
                security requires {
                        FilePermission(in,"read");
                        FilePermission(out,"write");
                        SocketPermission("www.citeseer.org","connect");
                };
        using checkReference = checkReference(String ref)
                security ensures {
                        DataPermission(ref,"read");
                };
        }
}
}
```

FIGURE 3. Outerface and implementation showing security specifications.

reference method has a separate privacy constraint that the string passed in cannot be written to an external device. These specifications try to ensure that any implementation of the outerface respects the privacy of the caller by only reading and writing the appropriate files and by not recording the incoming requests. The security specifications associated with the implementation state what permissions the implementation needs from the caller. Here it says it needs to be able to read and write appropriately the passed in files, and it needs to be able to connect to a particular host to do its work. The specifications go on to state the privacy constraints that the implementation satisfies. In this case stating that it will never write the argument to *checkReference* to and external device, either explicitly or implicitly.

This security model is currently partially implemented in TAIGA using the Java security model. Java maintains a current security policy and most operations that are security sensitive are checked against the current security policy before they are permitted. TAIGA provides its own Java security policy. This policy actively changes the sets of valid permissions on each outerface call. Moreover, it creates protection domains that are passed along on remote calls so that remote implementations can also enforce the security policy. Each call to an outerface method is executed explicitly in the appropriate TAIGA security context. Remote calls include information about the caller’s security context and this information is used to build the actual security context in which the remote call is executed.

This implementation works for permissions that are actually checked by Java, including file and socket access, for implementations which are local, and for implementations that are remote and implemented within the TAIGA framework. It currently does not handle data access permissions or permission checking inside web services and other black box implementations.

Extending the security specification to handle data permissions requires that we actually check that data access obeys the security specification. Our initial plan here is to match data access limits in the outerface with data access specifications in the implementation. For the example in Figure 3, the data access required by the outerface is NOWRITE, which is then satisfied by the implementation specification of READ. This level of checking is already done in the current implementation. However, to make this secure, we need to check that the implementation actually obeys its specifications. This can be done using a conservative static analysis that looks at how the particular data element flows from the called routine throughout the system and ensures that the specifications are obeyed for all possible flows [6,29]. Such a check would be part of the testing procedure for a potential implementation.

Handling security in black box applications is more difficult. Here we are investigating of requiring validated implementations to run in virtual machine sandboxes that would be security-aware. The socket interface to the application would be interpreted and validated by the virtual machine which would take responsibility for ensuring the validity of any actions taken by the application when processing the remote call. This type of approach could handle a limited set of security issues, but would represent a step in the right direction. Moreover, it could be built on the same type of sandbox that would be needed for safe testing.

6. Economic Semantics

When there are multiple implementations of an outerface available, the programmer will want to choose the one that best fits the application. There are a lot of factors that go into “best fit” in this situation. One might want to choose the least expensive implementation; one might choose the implementation that is fastest for the particular types of inputs that the application is going to provide it; one might want the implementation that has the strictest privacy policy; one might prefer an implementation that is downloadable rather than one that only exists as a web service. The choice here needs to be made in an open-source world where the programmer might not know of all the potential implementations or their characteristics. Moreover, the choice might have to be made dynamically, for example if different users running the same application are in different locations or if the primary implementation fails or becomes unavailable.

To accommodate all these options, we include an economic model as part of the outerface definitions. This model describes the factors that are important in selecting an appropriate implementation and gives weights to those factors. Currently these factors include the cost of the implementation which can be specified in the implementation definition, costs associated with different types of bindings (library, component, grid, web service), costs associated with traits that the implementation may have, and costs associated with resources used by the implementation.

While most of these are straightforward, one interesting aspect is determining the resources needed by the implementation. Here we rely on the fact that outerfaces include standard test cases which are run for each implementation and we can record resource utilization when the test cases are run. Currently we track the run time and the memory usage associated with each test case. The economic model can then provide weights for different test cases and for the different resources. This lets developers create a new outerface with a suite of test cases that are typical of the use of the component they are interested in, and then state that the run time (or memory) used by the component in running these particular test cases should be the dominant factor in choosing the implementation.

This cost model demonstrates that it is possible to include an economic framework for choosing implementations as part of the semantics of a component. It is not a completely workable solution at this point. There are several difficulties. The first is that the current implementation forces the outerface to specify how to value what are basically incomparable costs, for example the cost of a the licence to use a particular implementation versus the resources used by that implementation. The second is that some costs, for example, the cost of using a grid node of some outside user, will not be known at the time the implementation is being chosen and might affect the choice. Finally, the costs right now are just numbers and these need to be tied to real economic values, i.e. money, in order to ground the model in a reasonable way.

7. Future Work

TAIGA was developed to demonstrate that it is possible to design and implement dynamic and Internet-based programs in a disciplined manner even in the face of all the difficulties that such programs face, that they are inherently undesignable. Changes to the constraints for an application can in many cases be translated into changes to the outerface that do not affect the caller but rather cause a different implementation to be used. If one assumes a vast library of open source components that can be used, and one can rely on the components working the way they should as defined in the outerface, then it should be possible for many applications to adapt without major rewrites. Whether this is actually possible depends to a large part on how well the application is designed to make use of outerface-based components. What can be done here is something that won’t really be understood until we have substantial experience designing and developing such systems.

Where TAIGA provides a better footing is in handling the difficulties inherent to Internet-based programs, particularly the lack of control over the component implementations and how and where they run. TAIGA has demonstrated that it can handle components that change without the user’s knowledge, that it can handle component failures due to network outages, bad data files, or just the failure of the remote machine. It has demonstrated that one can dynamically choose among different implementations without affecting the original application. It has demonstrated that one can specify ones security and privacy needs and use these as constraints in choosing an appropriate component implementation.

Beyond the many problems already described, there is much that can be done along these lines. First, the notion of semantics, while quite broad, still does not encompass everything that one would like to take into account when choosing applications. Other things that one could conceive of including as part of an outerface definition might include accuracy or error ranges on floating point computations, a composable performance model to provide predictable performance for a complex system [32], mathematically checkable and composable specifications, authentication and access control, and monitoring and debugging aids.

One also needs to consider practical issues such as when and how should testing be done, how to define and test assemblies of components that themselves should be viewed as a component, and how to build up a repository of interfaces and implementations. The first question involves balancing attempting to continually detect when web services or outside components might have changed so that they can be retested with having to do what might involve expensive testing very frequently. The second involves developing strategies for building more complex abstractions on top of the abstractions that TAIGA currently provides. The third involves combining something like Google's code search facility with some social process for specifying interfaces and implementations such as that provided by RentACoder or similar sites.

The problems of designing software for dynamic, uncontrolled environments are real and are going to become increasingly more important as we move toward Internet-based software in the next decade. We have to address these problems or we will not be able to design and develop the applications that users demand and need. TAIGA and the approach it emphasizes shows one way that this might be done. The challenge is finding ways to extend this approach to make it practical or finding other approaches that accomplish the same ends.

8. Acknowledgements

This work is supported by the National Science Foundation through grant CCR0613162.

9. References

1. Sudhir Ahuja, Nicholas Carriero, and David Gelernter, "Linda and friends," *IEEE Computer* Vol. **19**(8) pp. 26-34 (August 1986).
2. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll, "An overview of JML tools and applications," *Intl. Journal on Software Tools for Technology Transfer* Vol. **7**(3) pp. 212-232 (June 2005).
3. G. Sebastien Chan-Tin, "Sandboxing programs," Brown University Master's Thesis (April 2004).
4. Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker, "Making Gnutella-like P2P systems scalable," *Proc. ACM SIGCOMM 2003*, (Aug 2003).
5. Susan Cheung, "Java Transaction Service (JTS)," *Sun Microsystems*, (December 1999).
6. Dorothy E. Denning, "A lattice model of secure information flow," *Communications of the ACM* Vol. **19**(5) pp. 236-243 (May 1976).
7. Scott Draves, "Electric Sheep," *http://electricsheep.org*, ().
8. Guy Eddon and Steven P. Reiss, "Myrhh: a transaction-based model for autonomic recovery," *Proc. 2nd Intl Conf on Autonomic Computing*, pp. 315-325 (June 2005).
9. Eric Freeman, Susanne Hupfer, and Ken Arnold, *Javaspaces Principles, Patterns, and Practice*, Addison-Wesley (1999).
10. J. Steven Fritzinger and Marianne Mueller, "Java Security," *Sun Microsystems*, (1996).
11. A. Ganek and T. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal* Vol. **42**(1) pp. 5-18 (2002).
12. Li Gong, "Java 2 platform security architecture," *Sun Microsystems* (*http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security- spec.doc.html*), (2002).
13. L. Gong, "JXTA: a network programming environment," *IEEE Internet Computing* Vol. **5** pp. 88-95 (2001).

14. 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).

15. Daniel Jackson, “Alloy: A lightweight object modeling notation,” *ACM Trans. Software Engineering and Methodology* Vol. **11**(2) pp. 256-290 (April 2002).

16. Min-Yen Kan, “Web page classification without the web page,” *Proc 13th WWW Conference,* (2004).

17. Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers, “User authentication and authoriztaion in the Java platform,” *Proc. 15th Annual Computer Security Applications Conference,* (December 1999).

18. Gary T. Leavens, Albert L. Baker, and Clyde Ruby, “JML: A notation for detailed design,” pp. 175-188 in *Behavioral Specifications of Businesses and Systems,* ed. Haim Kilov, Bernhard Rumpe, and Ian Simmonds,Kluwer (1999).

19. Meir M Lehman, “Approach to a Theory of Software Evolution,” *8th Intl Workshop on the Principles of Software Evolution,* p. 135 (September 2005).

20. Nenad Medvidovic, “On the role of middleware in architecture-based software development,” *SEKE ’02,* pp. 299-306 (July 2002).

21. Bertrand Meyer, *Object-Oriented Software Construction,* Prentice-Hall (1988).

22. Sun Microsystems, “The Jini Architechture Specification,” *http://www.sun.com/software/jini/specs/index.xml,* (June 2003).

23. Johann Oberleitner, Thomas Gschwind, and Mehdi Jazayeri, “The Vienna component framework: enabling composition across component models,” *Proc. 25th ICSE,* pp. 25-35 (May 2003).

24. Steven P. Reiss, “Interacting with the FIELD environment,” *Software Practice and Experience* Vol. **20**(S1) pp. 89-115 (June 1990).

25. Steven P. Reiss, “Connecting tools using message passing in the FIELD environment,” *IEEE Software* Vol. **7**(4) pp. 57-67 (July 1990).

26. Steven P. Reiss, “A component model for Internet-scale applications,” *Proc. ASE 2005,* pp. 34-43 (November 2005).

27. Steven P. Reiss, “Evolving Evolution,” *8th Intl Workshop on the Principles of Software Evolution,* pp. 136-139 (September 2005).

28. Ran Rinat and Scott Smith, “Modular Internet programming with cells,” *Proc. ECOOP 2002, Springer-Verlag LCNS 2374,* (2002).

29. A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas of Communications* Vol. **21**(1)(January 2003).

30. 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).

31. Clemens Szyperski, “Component technology - what, where, and how?,” *Proc 25th ICSE,* pp. 684-693 (May 2003).

32. Eno Thereska, Dushyanth Narayanan, and Gregory R. Granger, “Towards self- predicting systems: What if you could ask ’what if’?,” *3rd International Workshop on Self-Adaptive and Autonomic Computing Systems,* (August 2005).

33. J. B. Wordsworth, *Software Development with Z,* Addison-Wesley (1992).

34. P. Wyckoff, “T Spaces,” *IBM Systems Journal* Vol. **37**(3)(1998).