To appear in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001).*

A Compositional Approach for Constructing Connectors

Bridget Spitznagel
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15217

David Garlan
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15217

Abstract

Increasingly, systems are composed from independently developed parts, and mechanisms that allow those parts to interact (connectors). In many situations, specialized forms of interaction are needed to bridge component mismatches or to achieve extra-functional properties (e.g., security, performance, reliability), making the design and implementation of these interaction mechanisms a critical issue. Unfortunately, system developers have few options: they must live with available, but often inadequate, generic support for interaction (such as RPC), or they must handcraft specialized mechanisms at great cost. In this paper we describe a partial solution to this problem, whereby interaction mechanisms are constructed compositionally. Specifically, we describe a set of operators that can transform generic communication mechanisms (such as RPC and publish-subscribe) to incrementally add new capabilities. We show how these transformations can be used to realize complex interactions (such as Kerberized RPC) and to generate implementations of the new connector types at relatively low cost.

1. Introduction

Increasingly, complex software systems are being constructed from reusable, independently-written software components. These components are connected by software mechanisms which allow them to communicate. Consider an n-tier client-server system; the server component and the database component may each have been acquired separately, while the middleware enabling the two to interact may have been written by yet a third party.

The design and implementation of a system's interaction mechanisms is a critical issue. Generic forms of component interaction, such as RPC, are not always sufficient for complex systems. Specialized forms of interaction are often needed simply to make the parts of the system work together, as well as to achieve desirable extra-functional properties such as performance, security, or reliability. For example, the server component and database component considered above may not agree on the data format, necessitating on-the-fly conversion. A different software system might require compensation for different control mechanisms (such as synchronous versus asynchronous), or support for system monitoring and debugging. Similarly, one might need to enforce security mechanisms such as authentication, or improve performance through caching.

Unfortunately, at present it is difficult to create the semantically rich connectors that these software systems need. Architects have two choices when selecting connector types for a system design: use an existing off-the-shelf connector, or a new specialized connector for which no implementation yet exists. Neither alternative is adequate. On the one hand, it is not always possible to find an existing connector that can meet the needs of the system. On the other hand, creating a new connection mechanism is well known to be difficult and costly, typically requiring low-level knowledge of operating systems, communication protocols, and auxiliary mechanisms such as stub generators. The situation is compounded by the need to combine multiple interaction capabilities in a connector. For example, one might want to use both caching and communication monitoring, or authenticated RPC and encryption. This leads to a combinatorial explosion in the number of useful interaction mechanisms. The cost of implementing each combination as a monolithic, handcrafted form of connection quickly becomes prohibitive; a new approach is required.

What is needed is a way to produce new kinds of connectors systematically and at low cost. This would be possible if we could construct new connectors compositionally. An architect desiring a new connector type would select a basic connector type such as RPC (representing an existing interaction mechanism implementation), which could then be augmented with selected adaptations to produce more complex connector types appropriate for the software system being designed. System generation tools would then compile those enhancements into new run time mechanisms adapted to the problem at hand.

In the remaining sections we describe steps towards re-

alizing such an approach. We begin by motivating the problem. Next we enumerate a set of operators that can incrementally add new capabilities to generic communication mechanisms (such as RPC and publish-subscribe). We show how these operators can be used in combination to realize complex interactions, giving examples in security and dependability. The contributions of this work are three-fold: first, a different, compositional way to think about connectors; second, techniques for translating the modification of a connector from architectural concept into implementation; third, convenient encapsulation of a set of patterns for changing extra-functional, quality-of-service aspects of a software system.

2. Related work

There are four main areas of related work: software architecture; protocols and their formal analysis; generating implementations; and design patterns.

The first area is software architecture, in which the treatment of connectors as first-class entities was introduced [12, 1]. When component interactions are embodied at the level of architectural design as *connectors*, this enables the system designer to make interactions explicit and easy to identify, to attach semantics, and to capture abstract relations. Some work has been done toward a conceptual framework for classifying connectors [9] to facilitate understanding of them. First-class treatment of connectors also enables their formal specification and analysis, independent of the components they connect [1]. For example, formal analysis of the High Level Architecture (HLA) for Distributed Simulation [2], a proposed connector, revealed interesting flaws; the work on the HLA is additionally of interest due to its concern for how specific connectors can be built up in a traceable and modular way. Also related are techniques for resolving architectural mismatch. When two mismatched components are unable to communicate via existing connectors, one option is to construct or modify a connector that will resolve the mismatch [13]. Sometimes component wrappers are used. Another technique, Flexible Packaging [5], separates the component's functionality (ware) from its assumptions about the communication infrastructure (packaging).

Another related area, in protocol research, is the area of protocol synthesis. Decompositional techniques for protocol synthesis break a complex task into subtasks, which have simpler protocols that can be created more easily and then combined, in a principled and sometimes automatic way, to form the desired protocol [16]. Some properties of safety and (given certain restrictions) liveness can be predicted in a similar incremental way using a finite state machine model [15]. Other recent work in the area of protocol synthesis includes Ensemble [17], which enables the construction of an adaptive protocol composed of stacked micro-protocol modules. The x-Kernel [11] project has also used micro-protocol composition to design and implement a dynamic architecture for flexible protocols that take advantage of operating system support for efficient layering. Conduits+ [8] also provides a framework for network protocol software, with a focus on reuse aided by design patterns; layered protocols are composed from conduits (software components with two distinct "sides") and information chunks (which flow through the conduits).

A third area of related work is in code generation as it relates to generating connectors. UniCon [14] addresses implementation issues in realizing specific connectors. The UniCon compiler enables the construction of a system from an architecture description including generation of the code and other necessary constructs that implement the system's connectors. A specific set of connector abstractions are supported. Another related area of generation is in generating variations on a single connector. For a specific type of connector, such as RPC, work has been done in delaying the binding of some design decisions (such as the level of reliability) to make the implemented connector more flexible and appropriate for a wider range of applications; the decisions are not bound until the connector is integrated into a system. One approach is to have a set of small modules [7]. The options for behavior are classified into categories, such as call semantics (synchronous or asynchronous), and communication semantics (degree of reliability); micro-protocols modules, selected from these categories, are composed as in decompositional protocol synthesis. Another approach is to use object-oriented inheritance to specialize communication class libraries [19]. Our work differs from these in its focus on producing new connector types that may be based on a variety of existing connector types. A domain-specific compositional approach is taken in GenVoca [3], which illustrates the leverage that can be gained from restriction to a particular domain; connector transformations are more generic (less domain specific) and smaller.

The final area of related work is design patterns. A design pattern is an application-independent design fragment that can be reused to solve a well-identified problem [6, 4]. For example, the problem of decoupling the producer of a piece of data from observers of its value is solved by the "Observer" pattern; the pattern allows one to change the number and type of data observers without changing the data producer. Connection transformations can be viewed as a class of pattern for adapting component interaction mechanisms; our work goes beyond pattern identification and codification to develop tools for applying the transformation.

3. Motivation

Today, when an existing connector implementation must be altered or replaced by a new hand-built mechanism, someone must take on the implicit role of connector modifier. At present this task is difficult, costly, can require guru-level expertise, and has little available automated support.

Consider the following scenario. A software development group is constructing a product using a set of components written in Java with the assumption that Java RMI (Remote Method Invocation) will be used to make them work together. In order to improve the security of the system, the system's architect decides that some or all of the component interactions should use authenticated communication. Further, the group decides to use Kerberos [10] to provide authentication.

Currently two alternatives might be used. The first is to retain the use of vanilla RMI, but have a team of implementors modify the affected components so that they make appropriate Kerberos library calls before, during, and after each communication with another component. This alternative is highly undesirable from an engineering point of view. Modifications are distributed throughout the system, and any future change (such as a new version of the Kerberos protocol) will require just as much work. It is not possible to reuse the modifications as a new connector type in another Java-RMI-based system. There is also no guarantee that the implementors will carry out the changes correctly.

The second alternative is to develop a new Kerberos-Java-RMI connector. In practice this would be done by modifying the RMI stub generator so that it produces run time code for Kerberized RMI: it would insert appropriate Kerberos library calls within the RPC stack, so that they occur at the beginning and end of all remote method invocations. This approach has the advantage that changes are localized (kept within the stub generator), and that they are more likely to be correctly applied since programmers need only make sure to invoke the appropriate stub generator.

However, the task of modifying the stub generator will not be easy; it must be done by someone who is experienced both in Java RMI and in Kerberos. Moreover, adding additional modifications to the stub generator will require, at best, the same level of effort and expertise as before. In fact, it may be *increasingly* difficult to add a second modification to the ad-hoc changes that have already been made.

Connector transformation tools would make a superior approach possible, achieving the desired result (a new connector type) with less work and many long-term engineering benefits. In this alternative, we would apply a sequence of parameterized generic *connector transformations* to an existing connector type (such as RMI) to produce one that supports additional capabilities (such as authentication). That new connector type — typically realized as a code generator or set of run time communication libraries — can then be used freely as a new interaction mechanism throughout the system, as with the modified stub generator.

In this example, we would produce the Kerberizing-Java-RMI (stub) generator by using a transforming-Java-RMI tool that accepts parameterized transformations to adapt RMI. Specifically, we would determine the transformations required, and then give them code fragments that apply to this situation, Kerberos authentication; these fragments are used to instantiate the new Kerberized RMI stub generator.

What have we gained by doing this? First, we have reduced the costs in developing the new connector type, since it is no longer necessary to know the details of the Java RMI run time mechanisms or stub generation. Second, by breaking down the overall connector modifications into smaller, easily understood steps, connector transformations make the resulting connector easier to understand, reason about, and maintain. For example, changing the encryption policy only requires tweaking the code fragment of one of the transformations, whereas even the modified stub generator would have to be changed in multiple locations.

While the approach described above is a nice vision, it raises critical questions. First, what is the set of connector transformations, and what kinds of connectors can the transformations be applied to? These transformations must be chosen at an appropriate level of complexity, to enable easy decomposition of desired modifications and a broad range of applicability. Second, how would one actually build the generic tool described above? The remainder of the paper will provide one set of answers to these questions.

4. Conceptual framework

In order to talk about connector transformations, it is necessary first to to be clear what we mean by a connector. Abstractly, a connector is a discrete architectural element, representing a set of mechanisms that mediate interactions. The boundary between its interfaces (or roles), and the component interfaces (or ports) to which these roles are attached, is distinct. Concretely, the connector can be thought of as a six-tuple $\{c, l, s, t, p, w\}$ (described below and shown in Figure 1). It is not possible to point to one concrete entity and identify that as a connector. Even the parts we identify here may be spread across several files (or other concrete units), and mingled with other parts of the system; as a result, generation or modification of connector mechanisms is not an easy or localized task.

c: Application-level code that appears within a component or compilation unit. This may be code at the point of communicating with another component (calling sites); also, there may be code necessary for the initialization and finalization of the connector, in "main()" or the equivalent part of this compilation unit.

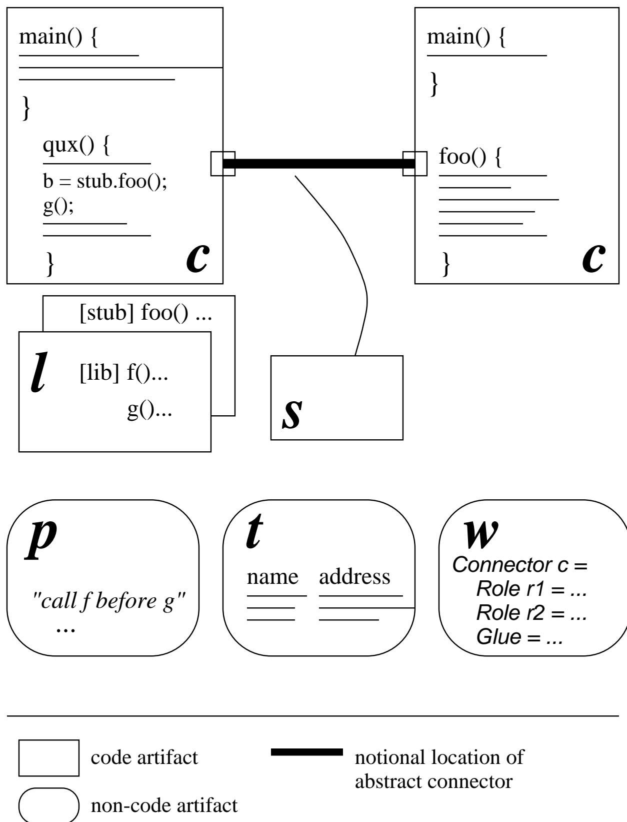

Figure 1. Concrete parts of a connector

l: Communication libraries, generated stubs, etc., below the application level.

s: Low level infrastructure services provided by, e.g., the operating system.

Other parts of the connector are non-code artifacts.

t: Data/tables, e.g., locations of communicating parties.

p: A policy documenting the proper use of these parts. For example, there may be a rule that library call x must be made before library call y.

w: Formal specification describing the connector's proper behavior.

A connector transformation modifies one or more parts of an existing connector $C = \{c, l, s, t, p, w\}$, resulting in a new connector C'. For example, the transformation may add, remove, or modify lines of code in c (at the call sites or initialization/finalization). It may modify a library or stub in l, or replace the library or stub with a different one that presents the same interface to the application-level code. The transformation may add a service in s, or replace one service with another (as in l). It may add, modify, or remove entries in t. A transformation can affect the policy p, perhaps by adding new restrictions as a result of changing the assumptions needed to use the connector. A transformation also modifies the connector's formal specification w.

5. A set of transformations

There are many approaches that one might use in choosing a set of transformations. At one extreme one might identify a minimal set of simple transformations with an eye toward formal simplicity and elegance. This has the attraction of providing a good basis from which to reason, but it may leave a large gap between the transformational calculus and complex connectors that are needed in practice. At the other extreme, one could enumerate a large number of powerful, specialized transformations that have direct applicability to certain areas, such as security or performance. This has the advantage of being directly usable within certain areas, but harder to reason about or to guarantee adequate coverage of the space of connectors.

We have attempted to find a middle ground. Specifically, we have identified a small number of moderately complex transformations, that have direct applicability to realistic applications, that are simple enough to reason about formally, and that are generic enough to ensure broad coverage. (These transformations have been derived from a survey of systems papers in which one or more modifications to a software connector are described.) We'll describe these transformations informally in this section. For each transformation we summarize its purpose, examples of its use, and sketch its dimensions of variability.

5.1. Data transform

Data Transform changes the format of the data being exchanged in an interaction. Format changes may occur at either end of a communication, or at both ends. It does not alter the protocol of interaction, although it may require additional information to be transmitted.

A simple example of Data Transform is conversion from little- to big-endian. Another example, in which data is transformed at both ends, is data compression: data is compressed at the sender and decompressed at the receiver. A more complex example, in which additional information is transmitted, would be to include checksums for error detection. As a final example, consider a connector that handles the situation in which a sender produces data more quickly than the receiver is able to consume it; the connector could be modified to present the receiver with the average of every n values.

Aspects of this transformation that can be varied include: Where the operation occurs (at the receiver, sender, both); How many data transforms occur (one, two, n); What is the operation on the data? Does it preserve the number of messages, decrease, or increase? Is it reversible or irreversible (lossy)?

5.2. Splice

Splice combines two binary connectors c and d into a new binary connector. The new connector has one interface from c and one from d. Unlike a Data Transform, Splice changes the protocol being used to exchange the data. This transformation will be possible for some but not all pairs of connectors. For example, c may not have access to some information required by d. This transformation is chiefly used to enable two mismatched components to interact.

Software adaptors [18], which can overcome some kinds of component mismatch, are one specific example of splicing technology. The Java Bean Box also incorporates a splice between event-based and procedure-call connectors.

5.3. Add a role

This transformation adds a new interface (or "role") to an interaction to enable a new party to be involved. Two kinds of roles can be added: observers, and participants. Observers listen to the communication between the connected components, but do not affect it. Participants may take an active part in the communication.

A simple example of an *observer* is an eavesdropper that logs all communication. Another example is an auditor that requires some information to be supplied in addition to what was originally being communicated. These would be of use for providing logs and audit trails, or for collecting data such as performance measurements.

Examples of *participants* are a local component that acts as a cache, a confirmer or authorizer that determines whether a request should proceed, and a "trusted third party" that supplies information such as encryption keys to be used by another transformation.

Choices that must be made during instantiation include the following: Whether the role is an observer or participant; What communication events are accessible to the new role; What is the effect of a participant (e.g., does it swallow some messages sent to another component and produce different responses to the sender?).

5.4. Sessionize

Sessionize makes a connectionless protocol session-oriented, or vice versa. The resulting connector will maintain state, i.e., cache some piece of information, in some way that the original connector didn't.

Examples of this transformation's use are database query refinement, and caching a proof of identity such as a session key: in the context of encryption, a session key is agreed upon by the participants at the beginning of a session, and used to encrypt their communication during that session.

This transformation can vary along the following dimensions: What "state" the connector is maintaining; Where state comes from come from, when and how it is updated, and how long is it kept; The effect that the state has on the communication.

5.5. Aggregate

The Aggregate transformation combines two or more connectors with a controller. One connector is active at a time. The controller determines how the connectors interoperate, i.e. which connector is active at what point in time, but does not change their basic protocols.

Two examples of this transformation's use are to create a connector that negotiates, and a connector that adapts during execution. A negotiating connector supports a set of protocols and attempts to determine initially what protocol(s) are acceptable to the components it connects; then it will operate using the protocol held in common. This is similar to two modems determining the fastest speed common to both.

An adaptive connector's controller monitors some aspect of a changing environment, and dynamically changes the protocol based on this information. For example, the controller might monitor communication faults and switch to/from a more fault tolerant, but less efficient, protocol.

Aspects of this transformation that can be varied include: When the controller is allowed to change active connectors (statically at the beginning, or dynamically); How the controller decides when a change should occur; How many connectors, of what types, are combined.

6. Example

To illustrate how these transformations are used, let's return to the Kerberos example and see what the system architect would do.

Kerberos is an authentication protocol in which clients and servers are able to prove their identity to one another with the help of credentials obtained from the Kerberos server; when a client requests a service, it presents a credential to the server. The first step the system architect takes is to determine which transformations are necessary to achieve this protocol, by consulting a taxonomy of basic transformations and typical patterns of use, and comparing them to the tasks appearing in the protocol. In this case we end up with three transformations.

Add A Role - We add a trusted third party whose knowledge will enable a check that the messages exchanged by the original parties are ok.

This is a common occurrence in security and we would expect to see this pattern in some other security-related

	Calling side	Remote side
Initialization	sessionize	sessionize
Before each call		sessionize / add role
Alter arguments	data transform on sending	data transform on receipt
Alter result	data transform on receipt	data transform on sending
After each call	sessionize / add role	

Table 1. Locations requiring modification

modifications.

Data Transform - We add new information that is used to prove the sender's identity.

Using a Data Transform to add data to a communication (such as message checksums that incorporate a "secret") is another transformation that we would expect to see frequently in the context of identifying a communicating party.

Sessionize - This transformation is needed to store the credentials that the other transformations use.

Security protocols in which a third party is contacted will often mitigate the overhead of doing so by caching the information obtained. When such information is obtained, cached, and re-used, we would expect to use the Sessionize transformation.

The next step is to write or obtain the code fragments that are needed by these transformations. These fragments may make use of standard libraries[1].

Using traditional techniques these fragments would have to be inserted by hand, in the component implementation or the stub generator. This would be a time consuming and error prone process because the fragment destinations are not localized: to be specific, Table 1 illustrates some locations where code fragments must be inserted, in the case of adding Kerberos to Java RMI.

In contrast, using our tool this insertion is performed automatically. The transformations, fragments, and the component implementations are provided as inputs and the tool generates the implementation of the new connector, by creating wrappers, modifying component implementations, and/or producing non-code artifacts such as makefiles. (Referring back to the parts of a connector implementation, we produce a new connector $C' = \{f_c(c), l \cup l', s, t, f_p(p), f_w(w)\}$ with modified code, additional stubs and/or libraries, and changes affecting the policy and formal specification.)

[1] As an implementation note, since the example is in Java, and the Kerberos libraries are in C, they are used through the JGSS package (provided by the University of Illinois Systems Software Research Group), a Java implementation of the Generic Security Service API; it provides access to the Kerberos V5 libraries for Java programs.

```
// 1. New data members for server class
private security.GSS.Context ctx;
private security.GSS.Name svc_name;
...
// 2. New method for server class;
// sessionize inserts method call in initialization
protected void kerbinit() {
    ...
    ctx = new security.GSS.Context();
    svc_name = new security.GSS.Name();
    int major = svc_name.import_name (service,
                                      svc_name.SERVICE);
    ...
}
```

Figure 2. Two fragments for Sessionize

For example, Sessionize requires code fragments that declare storage for the cached information, and a method for initially obtaining this information. The Sessionize transformation has no independent knowledge of what new data members are required or what initialization library calls should be made (hence the need to supply fragments such as Figure 2), but instead has a non-domain-specific knowledge of appropriate places to insert these kinds of code.

7. Implementation

The current implementation of our tool handles the special case of RMI.

When no transformations are used, the tool naively generates a Java-RMI connector; this is the *base connector*[2]. The tool has knowledge of what initialization steps to add, etc., to prepare for and perform a remote method invocation. Each transformation that is used will insert code fragments at places that are specific to that transformation; these fragments are supplied by the person using the tool.

Given a desired transformation and associated code fragments as input, along with the source files for the components that are to communicate, the tool generates a new connector derived from the Java-RMI base connector; complex connectors can be produced by applying a series of transformations to the base connector. The prototype tool produces

[2] Though we describe no restrictions on the selection of a base connector or the set of possible base connectors, they will be best taken from a middle ground; tool implementation is more difficult for an extremely complex connector, while some time and effort may be saved by selecting an existing connector of sufficient complexity to be in common usage, rather than building up from extremely low level interaction mechanisms.

composable wrappers for the RMI stub, which are created from the interface specified for the remote object. The tool also makes some modifications to the existing source code, for example, in the client implementation, the call to the constructor method for the remote object is replaced with a call to a factory method for the appropriate wrapper object. For some transformations the wrapper objects present the same interface to the connected components as the original connector did, minimizing the impact on the component implementations; for transformations in general, this is not always the case. While the prototype tool generates only Java source files, a future version will also generate non-code artifacts such as system configuration and makefiles to simplify deployment of the generated code files.

7.1. The Kerberos example

We investigated a "real world" modification, described earlier in section 3: adding Kerberos authentication to a Java RMI connector. Recall that in order to realize this modification as a composition of simple connector transformations, the first step was to determine what transformations should be used (and in what order). These were sessionize, data transform, and adding a role.

The next step was to determine the code fragments that should be inserted by each transformation. Existing code can be utilized for this; in the case of Kerberos, some code fragments were taken from sample code distributed with JGSS. The code fragments and sequence of transformations to apply are given as inputs to the general tool; to create a specific Kerberizing-Java-RMI tool they could be hardwired in. In either case the tool operates on components' source code and produces a connector implementation.

With such a tool the connector transformation approach results in a significant savings in effort as compared to the alternatives seen in section 3. It took about two days to perform the decomposition and write the code fragments. Some additional time (about a week) was spent in learning about Kerberos. Had the stub-generator approach been used instead, in addition to the time spent learning Kerberos, some time would have been spent learning about the internals of the stub generator. Examination of the sun.rmi.rmic package suggests that the additions required would affect four (of nine) classes, and would require modification of ten or more existing methods. The connector transformation approach saves this time; in the actual implementation of the connector, new code may be distributed in several locations, but this insertion is done automatically. The connector transformation approach also requires fewer lines of code to be written than the first alternative of cutting and pasting from an example or template. For example a code fragment, associated with a data transformation, that alters the arguments of a method call, can be written once and automatically applied to all methods exported by the remote object, providing a multiplicative benefit in the number of methods (or, in some cases, the number of calls), a form of commonality not easily exploited by naive cut and paste.

	Total lines	Reused lines
Initialization		
(Kerberos)	33	18
(Java RMI)	12	0
Client makes call		
(Marshalling request)	15	0
(Transforming)	18	11
(Untransforming)	13	6
(Unmarshalling result)	6	0

Table 2. Adapting existing code for use with the tool

Furthermore in the Kerberos example it was possible to reuse existing chunks of code, which indicates that the code fragments needed by the tool resemble what one would normally have written, and therefore do not require a additional learning curve beyond the basic understanding of Kerberos. Table 2 illustrates some places where code fragments are inserted, in the example of adding Kerberos to Java RMI; the last column shows what portion of the Kerberos-specific code was reused from an existing demo program.

7.2. Dependability patterns

Security is not the only domain to which connector transformations are applicable. We have begun application of this technique in the domain of dependability, to produce compositions of connector transformations that can be used to incorporate common dependability enhancing techniques into a connector.

A composition of transformations.

Our initial exploration in this domain was to compose a transformation that enables replication of a server component. The replication is essentially transparent from the point of view of a client component: when the client makes a request of what it perceives as "the server", the request is actually relayed to the replicated server components, the results they issue are tabulated, and finally the requesting client receives the best or most popular result.

Server-role replication is a modification that is composed from a single kind of transformation: the "add a role" transformation is used to add an array of similar roles $r_0 \ldots r_n$ which will be attached to the n replicated components, and to add a distinct role r_v which the tool will attach to a voting mechanism. The replicant roles all listen to client requests.

Replicants' replies are intercepted and sent to the voter role. The voting mechanism must be provided by the user of our tool; in this respect the replication modification is similar to the Aggregate transformation which uses a provided controller to determine the best action. Upon receiving and tabulating the replies, the voting mechanism may make a simple decision such as majority vote, or it may exercise some understanding of the requested task which enables it to detect and disregard some incorrect results. We do not address the task of producing the actual replicated components (given the original component) because, though potentially an interesting problem itself, it has no direct relation to the interaction mechanism.

This particular modification introduces a form of redundancy to address particular kinds of failure (such as a server failing), in order to increase the probability of obtaining a correct result. There are other possible modifications following the same pattern to achieve slightly different goals. For example, one may increase the probability of obtaining a *timely* result by enabling the voter to terminate the results collection early and make a decision based on the results that have been gathered so far, in order to meet a deadline.

A composition of modifications.

As argued in the introduction, often it is desirable to make several modifications to a single connector. These modifications are not always independent. We constructed a pair of cooperating dependability modifications, in which the first connector modification adds a simple form of error detection such as a CRC, and the second modification re-issues the last request once when an error is believed to have occurred. Composition of this pair required care because of the need to predicate the action of the second connector modification upon the outcome of the first.

One might expect to achieve the first modification with only a Data Transform (adding checksum bits at the sender and stripping them at the receiver). From a semantic standpoint, however, it is evident that such an approach is inappropriate. The purpose of the Data Transform is to transform the data carried (e.g. by applying a function such as unit conversion) without altering the kind of communication-event (in this case "method x returned this result"). In contrast, the purpose of the first modification is actually to conditionally alter the kind of event (from "successful result" to "error") with the expectation that the recipient's behavior may be very different as a result; this modification follows a pattern that one might describe as "confirmation". The Add Role transformation is more appropriate for confirmation modifications than Data Transform, because it permits the introduction and/or substitution of different communication-event types. In implementation terms, the Data Transformation expects a piece of code re-

```
// 1. In addition to fragments, we must provide
// an implementation of this new "checker" object.
// Its interface might look like this.
public interface ErrorDetector {
    // Return a new copy of b, with checksum appended
    public byte[] appendCRC(byte[] b);
    // Return bAndCRC with checksum confirmed and
    // removed, or raise an exception
    public byte[] removeCRC(byte[] bAndCRC) throws ...;
    ... }
// 2. We supply fragments, used to build the method calls
// of the stub wrappers. A caller side fragment:
    ErrorDetector chk = new ErrorDetector();
    // In this case we've no interest in the individual
    // arguments of the wrapper's method calls...
    MARSHAL
    // That tells the tool to place this part after the
    // arguments are marshalled in byte array b:
    b = chk.appendCRC(b);
    // Finished intercepting arguments. Tell the tool
    // to proceed with the call to the wrapped stub:
    SEND
// 3. The second caller-side fragment intercepts
// the result of stub methods.
    b = RESULT
    b = chk.removeCRC(b);
    UNMARSHAL
    // In this case we've no further interest in the
    // result; the tool will just arrange to return it.
// The callee-side fragments are similar.
```

Figure 3. Fragments for Add Role

alizing some function which is to be applied to the communicated data. The Add Role transformation, in contrast, allows us to supply a "checker" object to which we forward the received data and which is permitted a range of possible responses. The checker that we would supply for this modification may return the data (perhaps modified), or may raise an exception indicating an error it cannot correct.

Code fragments for an Add Role transformation will chiefly appear within the method implementations that are generated as part of the stub wrappers. Figure 3 shows how such code fragments can request to be interleaved before and after marshalling, sending the request, and unmarshalling; the callee side fragments would be similar.

The second modification makes use of the Sessionize transformation to store and reuse information within a communication session; in the simplest variation of the "retry" technique, the information is the request itself, and a session is comprised of one successful request or one request and one resend.

In summary we have built a tool to perform some of the

composable connector transformations described in this paper. It produces implementations of a variety of complex connectors derived from a single basic connector, Java RMI; the tool enabled us to create and apply complex modifications such as Kerberization with less effort than would have been required using a more traditional approach.

8. Discussion

By describing a set of connector transformations, and implementing a tool to apply transformations to a particular connector type, we have taken a crucial first step toward realizing the vision outlined in section 3. One of the interesting questions that arises is whether we have picked an appropriate level for defining transformations. As indicated earlier, we have sought a middle ground, whereby the transformations would be semantically rich, but simple enough to combine in many ways and for many domains. (Further, applicability of the approach to a useful range both of base connectors and of domains is a not unreasonable expectation since the transformations were derived by examining similarities between examples of modifications made to different kinds of base connectors in several different domains.) While more work will be needed to decide this question, our experience in the domains of security and reliability suggests that the transformations can be easily applied to a wide variety of connectors.

A second important issue is the construction of tools to aid in the application of the transformations: without such tools, the approach is largely an academic exercise. While the abstract transformations should be independent of any particular connector, the implementation of a transformation tool is dependent on knowledge of the base connector type. Our initial efforts have focused on being able to perform multiple transformations for one connector type, where we might instead have worked to implement one kind of transformation for multiple connector types, because this enables investigating compositionality in practice as well as more interesting and complex examples of new connectors. The ability to apply this approach to a range of base connectors is also important. Such extension of the implementation will be addressed in future work.

The context of the work discussed in this paper has been code generation (though the design-time versus run-time distinction is not intrinsic to the notion of connector transformations, and one could instead contemplate taking a dynamic approach that makes use of reflection). A formal semantics for connector transformations is an orthogonal problem, out of the scope of this paper, but is equally important and is being addressed in ongoing work. In the future we expect formal analysis of these transformations to aid in determining a number of important properties, particularly those related to compositionality of individual transformations. For example, formalism can be used to show that the ordering of a composition matters or doesn't matter, and that a particular composition would result in a deadlocked connector (and thus should be avoided). Some findings are generally applicable in nature, e.g. if composition of two particular transformations is always contraindicated, whereas others are specific to the goals that a sequence of transformations is intended to achieve, e.g. when composing encryption and a new role in a particular order, the interpretation as a "good" or "bad" composition depends on whether the component attached to the new role is intended to receive encrypted or unencrypted data.

Some transformations change the original connector's roles. If the components that will be communicating were written to this original interface, the components will have to change. Formal semantics of transformations can demonstrate whether a transformation must necessarily result in changes to components, although ease of tool implementation might present a reason for a tool to require access to component source code even if it is not strictly necessary in terms of the formal model. In short the full range of transformations may not be available when some source code is not available, but we do not think this sufficient reason to restrict the set of transformations to what can be done without access to source. First, when source is available, that additional leverage is desirable. Second, transformations that change roles can be used to resolve mismatch, when one or more components do not match the original interface and the connector is being transformed to match the components' expectations.

Finally, section 2 mentions work such as [7] which strives to produce a highly customizable connector: Why, then, are connector transformations needed? The problem is that the usefulness of a connector which has been designed with a set of options and alternative modules is still limited by the foresight of its designer. A fixed set of choices may not address the issues or extra-functional properties that are relevant to a particular system architect of the future, and the customizable connector itself, though carefully designed to be flexible in ways that its designer considered important, may be no more amenable to actually being altered than any other connector. In contrast, though "stock" connector transformations (such as "add Kerberos" or "replicate and vote") may be available, the intent of connector transformations is rather to enable the composition of new modifications.

9. Conclusions and future work

In this paper we have argued for an approach to connector construction based on incremental transformation. To support this notion we have identified a set of basic transformations and illustrated how they can be used to create com-

plex forms of interaction, such as Kerberized RMI. We also described a prototype tool that can be used to apply these transformations in the case of Java RMI-based interactions, generating implementations of new connector types.

As indicated earlier, we view this work as a step towards a more comprehensive engineering basis for component integration. In particular, as indicated above, more research is needed to extend the initial set of transformations that we have identified. They need to be demonstrated in the case of other base interaction mechanisms (beyond RMI), and for other development platforms (beyond Java). Ongoing work addresses the application of these ideas to event-style connectors such as Java Message Service. In addition, preliminary work toward a formal semantics for transformations suggests considerable opportunities for exploiting formal theory to carry out detailed analyses about transformation composition and compatibility, and to better express the transformations' dimensions of variability. Finally, more case studies are needed.

Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency and Rome Laboratory, USAF, under Cooperative Agreement F30602-00-2-0616, by DoD Space and Naval Warfare Systems Center under N66001-99-2-8918, by the National Science Foundation under Grant CCR-9357792, and by a grant from HP Labs. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Rome Laboratory, the US Department of Defense, or the National Science Foundation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. This research was also facilitated in part by a National Physical Science Consortium Fellowship and by stipend support from Xerox. We would like to thank Mary Shaw, Jeannette Wing, and the members of the ABLE group Bradley Schmerl, Joao Sousa, Jichuan Chang, and Owen Cheng.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] R. Allen, D. Garlan, and J. Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM.

[3] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions of Software Engineering and Methodology*, pages 355–398, October 1992.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[5] R. DeLine. *Resolving Packaging Mismatch*. PhD thesis, Carnegie Mellon, School of Computer Science, 1999. Issued as CMU Technical Report CMU-CS-99-141.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

[7] M. A. Hiltunen and R. D. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS-15)*, May 1995.

[8] H. Hueni, R. E. Johnson, and R. Engel. A framework for network protocol software. *Proceedings of OOPSLA'95*, pages 358–369, 1995.

[9] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 295–304, Limerick, Ireland, June 2000.

[10] B. C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications*, 32(9):33–38, Sept. 1994.

[11] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[12] M. Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[13] M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proceedings of the Symposium on Software Reuse (SSR'95)*, April 1995.

[14] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*, May 1996.

[15] G. Singh and Z. Mao. Structured design of communication protocols. In *IEEE International Conference on Distributed Computing Systems*, May 1996.

[16] F. Stomp and W. de Roever. Designing distributed algorithms by means of formal sequentially phased reasoning. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, 1989.

[17] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. Technical report, Cornell/TR97-1638, 1997.

[18] D. M. Yellin and R. E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. *Proceedings of OOPSLA'94*, October 1994.

[19] M. J. Zelesko and D. R. Cheriton. Specializing object-oriented RPC for functionality and performance. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, Hong Kong, May 1996.