Compositional Transformation of Software Connectors: Thesis Summary

Bridget Spitznagel

Thesis committee:
David Garlan (chair)
Phil Koopman
Mary Shaw
Martin Griss (UC Santa Cruz)

December 2, 2003, 1:30 PM — Newell-Simon Hall 1507

1 Introduction

Increasingly, complex software systems are being constructed as compositions of reusable software components. These components are often written independently and connected using glue code, which may have been produced by a third party. For example, in a three tier client-server system, the server, the database, and the code enabling them to communicate may have been acquired separately. The trend toward compositional software is a natural consequence of industry interest in the ability to incorporate third-party software and to reuse code.

One critical issue for such constructions is the design and implementation of the communication-based interaction mechanisms, or connectors, that permit the software components to work together. Architecturally, a connector is a discrete design element, representing a set of mechanisms that mediate interactions between components (more specifically, in this work I address connectors that enable *communication* between components); at the implementation level, however, the realization of such a connector is complex and consists of many parts. Getting connectors right is critical because otherwise the system may malfunction in subtle ways, may not give adequate performance, or may not work at all. Consider a system composed of separately developed merchants, customers, and a virtual bank, using a protocol for untraceable digital cash. A design flaw in the connectors might allow denial of service attacks, theft by a customer or merchant, violation of customer privacy, double-charging of customers, or even deadlock. Alternately, the system might function correctly, but be useless due to a sluggish response time. Connector design can have a significant impact on the system as a whole.

In addition to requirements related to the system's function, the connectors in software systems such as this example typically also have nontrivial extrafunctional requirements. These extrafunctional requirements contribute further to the complexity of the connectors. In the digital cash system, there will almost certainly be system-level requirements related to performance, dependability, and security concerns. These requirements affect both the overall system design and the connector design.

Unfortunately, it is at present difficult to create the semantically rich connectors that these software systems need. There are two commonly available alternatives: to use an existing connector, or to write a connector from scratch. Neither alternative is adequate. On one hand, it is not always possible to find an existing connector that meets the needs of the system. Simple connectors are not sufficient by themselves; more complex middleware such as CORBA might meet the functional needs but prove inadequate for extrafunctional reasons such as performance requirements. When no existing connector will satisfy the system's needs, using an existing connector is not a viable option.

On the other hand, creating a new connector is a difficult undertaking. Low-level knowledge of operating systems and auxiliary mechanisms such as stub generators is often required. In tension with this need for highly knowledgeable programmers (gurus) is the drive towards enabling less experienced, non-guru programmers to build systems from existing components. Since existing connectors will not always be adequate, these non-gurus will sometimes be faced with the task of creating new connectors in order to compose systems; while they may excel at the level of component authoring and component integration, it should not be expected that they would have the particular skills required for low-level modifications to communication mechanisms.

What is needed is a principled way to produce new connectors systematically and at low cost. This is possible if we construct new connector variants compositionally: in this approach, basic connectors are augmented with a collection of selected adaptations or enhancements to produce a more complex connector that meets the requirements. For example, an existing inadequate connector might be augmented so that it will satisfy availability and security requirements, with acceptable effects on its performance.

The central claim of this thesis is that:

> *We can define a small set of basic transformations that, when applied compositionally to simple communications-based connectors, produce a wide variety of useful complex connectors. These transformations can also be given a formal interpretation that allows us to understand their properties with respect to communication protocols. Furthermore, we can build a transformation-based tool that facilitates the generation of implementations of these new connectors.*

For brevity, this summary will not present formal interpretation results.

2 Connector Transformations

Before discussing connector transformations it is important to understand what is meant here by a connector, both when working at the level of an architectural abstraction (as it would be used at design time), and at the level of implementation (when it is actually a collection of several concrete artifacts).

2.1 A Connector

Architecturally, a connector is a discrete design element, representing a set of mechanisms that mediate interactions between components. Shaw et al. [4] describe connectors, their realization at the implementation level in terms of a collection of dispersed and varied concrete artifacts, and different kinds of interactions that connectors may embody (such as communication, resource contention, and scheduling concerns). Specifically, in this work I address connectors that enable *communication* between components. A connector has some number of *roles* (connector interfaces); each role represents a participant in the communication. For example, consider a remote procedure call (RPC) connector: it has two roles, one role for the participant that makes the call, and another role for the participant that receives the call. An architect attaches the roles of a connector to the *ports* (component interfaces) of a set of components. The roles define behavior with which these ports must be compatible; a role's specification indicates the minimum that is required of a component in order to participate in the communication represented by the connector.

At the implementation level, however, the realization of such a connector is complex and consists of many parts [4]. Consider again the RPC example, and suppose that this architectural RPC connector has been instantiated at the implementation level as a Java Remote Method Invocation (Java RMI [7]) connection. Part of the connector implementation appears in the source code of the communicating components: the caller must initially obtain a reference to the remote callee, while the callee must initially register itself at a known location; at the call sites, the caller makes calls to a generated "stub" that stands in for the callee. Another part of the connector implementation appears in the stub itself (and perhaps a corresponding "skeleton" at the callee) and in communication libraries provided by the operating system and/or the middleware. Obtaining the reference to the callee may require the use of registration, naming, and lookup services; in this example, a process on the remote machine must be running the standard RMI registry provided by Sun, at a location known to the caller. In addition to these "running code" artifacts, RMI requires a security policy file (to permit the callee to accept incoming connections). Finally, independent of other concrete parts of the implementation, if some aspect of the architectural connector has been formally specified, this specification exists in some format (perhaps a text file) and should remain associated with the connector implementation.

When considered at high levels of abstraction, only a handful of basic forms of interaction, or *connector primitives*, exist. For example, the feature-based classification for architectural styles given by Shaw and Clements [3] identifies a small set of abstract connectors as a part of their style discrimination framework. Commonly used connector primitives include remote procedure call, implicit invocation and message passing (multicast and unicast asynchronous events), and dataflow.

2.2 A Connector Transformation

Returning to the architectural abstraction, in which a connector is a discrete design element, a *connector transformation* is a function from connectors (plus additional parameters) to connectors. Some transformations operate on a single connector type[1], while others operate on two or more connectors to produce some combination of them; in either case, the output is a single new connector type.

At the implementation level, where a connector is a nonlocalized collection of parts rather than a single entity, a connector transformation modifies one or more parts of an existing connector C resulting in a new connector C'. For example, the transformation may add, remove, or modify lines of application-level code in the components (at the call sites or initialization/finalization). It may modify a library or stub, or replace the library or stub with a different one that presents the same interface to the application-level code. The transformation may add an infrastructure service, or replace one service with another. It may add, modify, or remove entries in data tables, such as the locaton of a communicating party. A transformation can affect the policy of connector use, perhaps by adding new restrictions as a result of changing the assumptions needed to use the connector. A transformation also affects aspects of the connector that are described by the connector's formal

[1] We will use the term *connector* loosely, when it is apparent whether a *connector type* or a *connector instance* is meant, to refer to either. "Java RMI" is an example of a connector type; "the Java RMI connector between this client and this server" is an example of a connector instance.

Name	Effect	Parameters	Typical uses
Data Translation	Apply function to payload	Function Location (sender/recipient)	Unit/format conversion Compress on the fly Encryption
Add Observer	Eavesdrop w/o responding	Traffic copied to new observer component	Audit trail
Add Redirect	Intercept, relay/reject	Traffic affected by the redirection	Authorize/confirm Add a cache
Add Switch	Act in place of existing role	Traffic affected by the switch When to switch	Load balancing Backup server
Add Parallel	Duplicates an existing role	Traffic copied to new component Merging responses	N-version Best-effort fidelity
Aggregate	A choice of connectors	When to choose How to choose	Protocol negotiation Bandwidth adaptation
Sessionize	Add state: record/replay	What/when to record When to replay	Replay lost messages
Splice	A chimera combining dissimilar roles	N/A	Overcome protocol mismatch (between components)

Table 1: Summary of Connector Transformations

specification.

2.3 A Set of Connector Transformations

As part of this thesis, I have identified a candidate set of connector transformations. Criteria used to select this set include the following: they should be generic, rather than specific to a particular domain (such as dependability or security), and applicable to multiple connector types; as a compromise between power and simplicity for transformation granularity, it must be possible to produce useful modifications with a composition of 1-3 transformations (rather than, say, 5-10); a range of desirable modifications should be possible, though the set need not be complete; the transformations should be reasonably orthogonal in effect so that selection of appropriate transformations for a particular modification is straightforward. To evaluate coverage, the thesis describes a number of *dependability* enhancing modifications in terms of connector transformations.

Table 1 lists the transformations in this proposed set, and summarizes some of their key attributes: the *name* for this kind of generic transformation; the *effect*, in general terms, that this transformation has on a connector; the *parameters* that must be fixed upon when instantiating the generic transformation; and a few *typical uses* or domain-specific instantiations of the transformation. Each kind of transformation given in Table 1 is described in greater detail in the thesis.

Data translation operates on a single connector. It applies a function (which must be specified) to the data being communicated by the connector. The "location" (the role of the connector) at which the function will be applied must also be specified.

The *Add role* transformations operate on a single connector, plus a role. They introduce a new participant (the additional role) to the communication[2] represented by the connector. Some part (which must be specified) of the existing traffic will now be intercepted and sent to an additional destination or a different destination. The target of the redirection, and the responsiveness of this target, is constrained (differently) in each of these transformations: *Observer, Redirect, Switch,* and *Parallel replication*. In the *Add Observer* transformation, intercepted traffic is sent to the new role in addition to the original destination; the new role receives but does not respond to this traffic. In *Add Redirect*, intercepted traffic is sent to the new role, which may choose to forward the traffic to the original destination. In *Add Switch*, intercepted traffic is sent to either the new role or to the original destination. In the *Add Parallel* transformation, intercepted traffic is sent to both the new role and the original destination.

Aggregate operates on more than one connector. It combines them into a choice of connectors. The points in the protocol at which it is permissible to switch to another connector, and the means of determining which connector to use, must be specified.

Sessionize operates on one connector. It adds state to a communication session by recording and later replaying some piece of data. The thing that is to be recorded, and the appropriate moments to record, update, discard, or replay it, must be specified.

Splice operates on more than one connector. It combines, from each of the original connectors, a subset of the connector's roles, to create a new, chimeric connector that *appears* to be partly one beast, partly another. For example, given binary (two-roled) connectors A and B with roles A_1, A_2 and B_1, B_2, a splice transformation might create a new binary connector with roles identical to A_1, B_2.

2.4 Composition

These simple transformations may be composed to produce more complex modifications. A transformation may be composed with itself (e.g., two data transformations f and f^{-1}) or with a different kind of transformation. The ordering of the composition will in some cases affect the outcome. Formal determination of commutativity of transformations is discussed in the thesis.

To illustrate informally how transformations may be composed, both to achieve a single complex augmentation by using more than one transformation and to achieve multiple augmentations in a single connector, consider the following scenario. A remote procedure call connector C provides communication between a client component A and a server component B. The client sends a request to the server and expects to receive a response. Requests are not necessarily idempotent, but rather affect the state of the server, so that the same request may provoke a different response depending on the recent communication history. The server is subject to silent failure: it may crash (either between requests or while a processing a request), after which it will not respond until a human intervenes. If the server does not respond to a request within an interval t, the connector will report a timeout to the client. The communication channel, too, is not completely reliable, and a timeout (effectively the loss of a request or a response) may occasionally be reported to the client in place of a response even when the server is still operational.

In this example, we wish to increase the dependability of the system by replacing the original connector C with an enhanced connector C'. This connector has two independent enhancements (each of which is composed of connector transformations). First, rather than sending a timeout to the client, the connector attempts to re-send the request that timed out. Second, if the attempts to re-send the request continue to fail, the connector will divert communication intended for the failed server to a "warm" backup server.

[2] Recall that this work focuses on communication-oriented connectors.

For the first enhancement, because the requests are not idempotent and the reason for the timeout is not (yet) known, it is not enough to simply cache the request at the client side and re-send it; perhaps the server is still alive, and only the response was lost. We must also add a means of duplicate elimination, such as caching the response at the server side, and associating a unique id with each cached request (and corresponding response), so that a lost response can simply be resent rather than regenerated. This enhancement will require both a *sessionize* transformation (to record and replay) and a *data translation* transformation (to add a unique id).

For the second enhancement, we will be adding a "backup server" that duplicates the functionality of the existing server. Since the server has state, it is necessary either to use a "warm" backup server (that is, one that has the same state as the primary server should), or to get the backup server caught up with recent happenings. Therefore, in addition to using an *add switch* transformation to add the backup server's role, we will also require another transformation. Either the backup server must function as an *observer* while the primary is operational (essentially eavesdropping on all client requests), or the request history must be recorded with *sessionize* so that, after the failure of the primary server, sufficient history can be replayed to warm up a hitherto entirely dormant backup server.

3 Implementation Generation

The idea of connector transformations provides a different way of thinking and reasoning about a large monolithic modification to a connector. However, a more compelling contribution of connector transformations is the potential to provide a middle ground between the use of existing connector implementations (which, though convenient, may not meet extrafunctional requirements) and the laborious creation of new connector implementations by hand.

To demonstrate the claim that the use of connector transformations makes it possible to generate implementations of complex connectors, I have implemented a tool that operates on elements of a connector implementation, with additional inputs specific to a connector transformation, and produces elements of an augmented connector implementation.

The tool incorporates knowledge of a base connector implementation so that the tool user doesn't have to be a guru in that area. The tool knows how to find abstract locations, such as "initialization of the connector", within software systems that use a specific base connector implementation. (I identify a collection of such "hook-in" points later in Table 2.) Transformation applications are performed in terms of these abstract locations. This provides a useful degree of separation between connector-specific knowledge and domain-specific knowledge.

3.1 Approach

In modern connector technology, it is common to have mechanisms (such as stub generators or connection factories) that produce proxies, and lookup services that provide references to proxies. A stub generator operates statically on an interface description to produce proxies for a remote component corresponding to the interface description; connection factory produces proxies for multicasting. The local caller uses a lookup service to obtain a reference that is needed in order to use such a proxy. For example, this might be a reference to a stub for a named remote component, or it might be references to a connection factory and a destination (such as a queue, for point-to-point events, or a subscription topic, for broadcast events).

Figure 1 illustrates such a system. Here, a sender process is about to communicate with a receiver process. The sender process includes both the sender's real implementation, and auto-

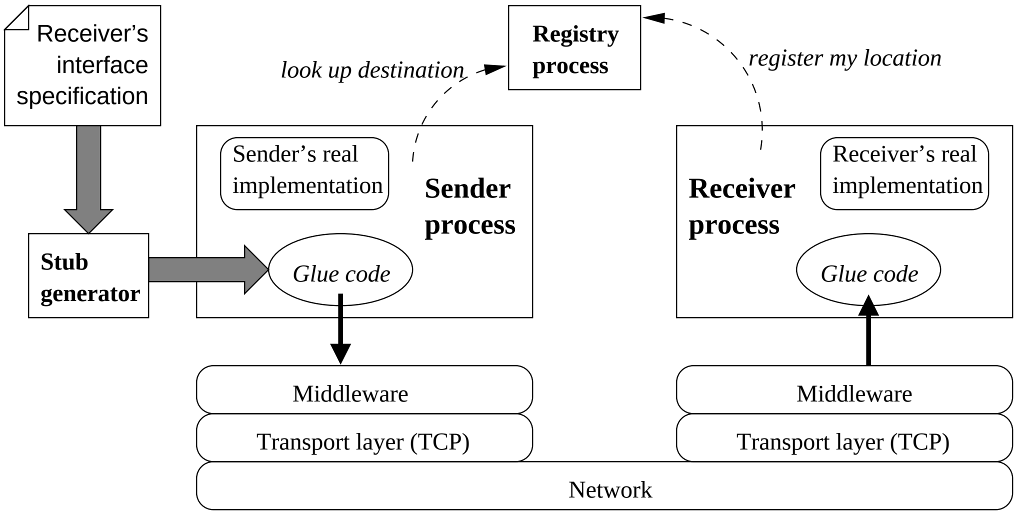

Figure 1: Runtime view of common mechanisms

matically generated "glue code" (a stub). This stub was created earlier by giving a specification of the receiver's interface to the middleware's stub generator. (The stub was either placed on the sender's host at that time, or at a location from which the sender is now able to download it.) The receiver process has registered its own location at a "well-known" registry service; here, well-known simply means that the sender process knows how to find the registry service. The sender process has obtained a reference from the registry service (and, if the stub was not placed on the sender's host initially, the sender process also downloads the stub); this reference will enable the middleware to direct the communication to the sender process. To communicate with the receiver, the sender's real implementation makes a local call to the glue code. The call traverses a set of layers from the middleware to the network; at the receiver's host, the call goes up from the network to the middleware, and the glue code in the receiver process makes a local call to the receiver's real implementation.

Transformations are introduced into the connector using the *Proxy pattern* (one of the patterns identified by Gamma et al. [1]). (Figure 2 illustrates this pattern with a simple before/after scenario; in the original system, at runtime, "MyClient" has a reference to "OriginalStub"; in the modified system, two proxies have been added between them.) The proxy that is closest to a component (e.g., "FirstProxy" in Figure 2) is introduced by performing a one-time *modification* of the application-level source code. This modification is made at specific restricted locations, where the application code ordinarily looks up a name and obtains a reference; the tool replaces[3] the lookup call, and return, in its place, a reference to this proxy.

3.2 Using The Tool

When creating a new domain-specific augmentation as a composition of supported transformations, the tool user still needs to have domain-specific knowledge (as well as non-domain-specific knowledge of how to operate the tool). For example, to create a security-enhancing augmentation it is necessary to have knowledge of security, such as: familiarity with commonly used means of authentication, existing packages that provide implementations, and how these libraries are properly employed.

[3]We can make the tool automatically recognize the lookup because it is a particular (middleware-specific) library call documented in the middleware's API.

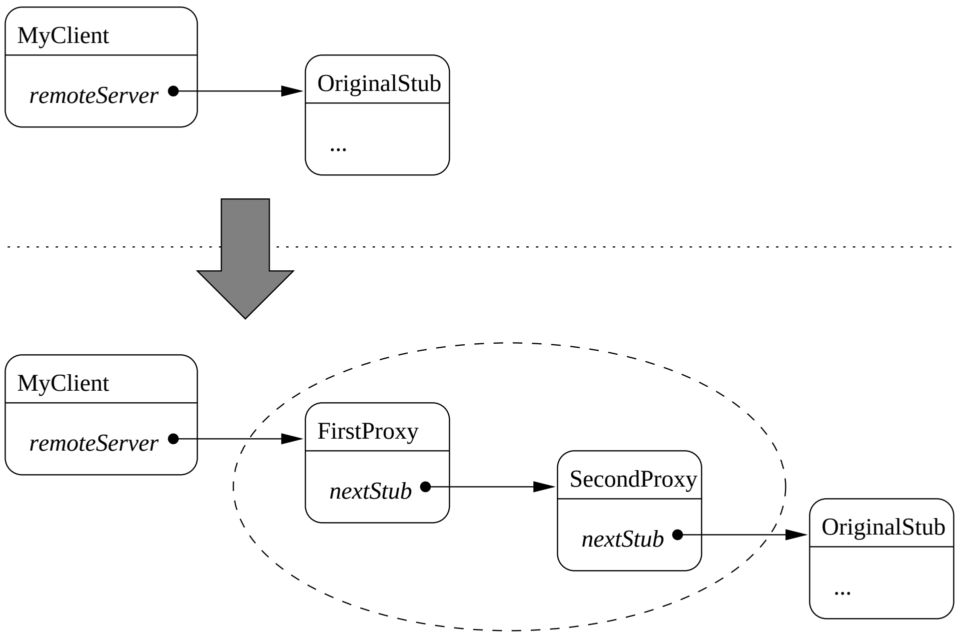

Figure 2: A chain containing two proxies

The domain-specific augmentation might later be reused by a subsequent tool user with a lesser degree of domain-specific knowledge.

A typical user of this tool will: choose the transformation that is to be used, provide code fragments and other transformation-specific parameters that instantiate the generic transformation as a domain-specific modification, point the tool at an existing connector implementation, and push a button. From these inputs, the tool produces an implementation of the new, augmented connector.

As an example of the kind of "code fragments" the tool requires, Figure 3 shows the code fragments for a *data translation* transformation that marshals arguments into a byte array, calls the data translation's *function* to calculate a tag, and prepends the tag to the byte array. We may supply (as a transformation-specific parameter) any class that implements a ByteArrayTagger interface.

3.3 Currently Supported

The tool support alleviates the need for its user to have an intimate understanding of the connector implementation from which the variant connector type is to be created. Three such "base" connector types are currently supported: Java Remote Method Invocation (Java RMI), a remote procedure call connector type; Java Message Service (JMS [6]), an event-based connector type; and the Inter Process Communication (IPC) library developed at CMU as part of the Task Control Architecture project [5]. Table 3 gives an overview of transformations currently supported, per base connector type. (Note that for unidirectional event-based connectors, data translation of a return value is not meaningful.)

The transformations can be broadly characterized in terms of the abstract "hook-in" locations for which they require support. For example, the observer transformation in the simple example added code both *before* the call and *after* the result was obtained. Table 4 illustrates some of these.

A data translation transformation introduces a function implementation (which may be a new

Abstract Location	Example
Initialization	In a role-addition transformation, we need to obtain a reference to the new component when the system starts up.
Around a communication (before, after, instead, on error)	A role-addition transformation might add a call before, and introduce a conditional around, the communication.
Altering data	A data translation transformation usually alters the communicated data.
Altering response	A data translation transformation may alter the response also. A role-addition transformation may capture (and perhaps modify) the response.
New data members	(A proxy can have data members.)
New methods	(A proxy can have more methods than those required by its interface.)

Table 2: Abstract Locations

Transformation	Java RMI?	JMS?	IPC?
Data trans.:			
Arguments	$\surd$	$\surd$	
Return value	$\surd$	(N/A)	(N/A)
Add Observer:	$\surd$	$\surd$	
Add Redirect:	$\surd$	$\surd$	
Add Switch:	$\surd$	$\surd$	$\surd$
Add Parallel:	$\surd$	$\surd$	$\surd$
Sessionize:	$\surd$	$\surd$	$\surd$
Aggregate:	$\surd$		$\surd$
Splice:	(limited)		

Table 3: Implementation Coverage of Transformations

method of the proxy or a method of a new object class). The method that implements the translation function is applied to the *arguments* and/or *result* of the remote call.

A role-addition transformation (add observer, add redirect, etc.) introduces a reference to a second stub[4] which may present a different interface than the original stub, or may present the same interface. It is created at *initialization*. A call to the new stub may be made in addition to (either *before* or *after*) the original call, or, if presenting the same interface, may replace the original stub as the *target* of the call. The *result* of the call (ultimately returned by this proxy's method) may be replaced by the result of the added role's call.

An aggregate transformation introduces a second stub (with the same interface as the original stub) and a decision-making apparatus. These are created at *initialization*. The decision-making apparatus is consulted *before* a call, and determines which proxy is to be the *target* of the call; to inform its decisions, calls to the apparatus may also be made *after* a call or *on error*.

[4]The proxy embodying this transformation will thus contain two references to downstream "stubs or proxies." I refer to them here as stubs, to avoid confusion with the proxy embodying *this* transformation.

New Members fragment

```
private NEXT_INTERFACE next_stub;
private ByteArrayTagger data_translator;
```

Method Calls fragment

```
b1 = MARSHAL
byte[] b1_old = b1;
if (data_translator == null)
  data_translator = new UniqueIDCreator();
byte[] tag = data_translator.createTag(b1_old);
b1 = new byte[b1_old.length + tag.length];
System.arraycopy(tag, 0, b1, 0, tag.length);
System.arraycopy(b1_old, 0, b1, tag.length, b1_old.length);
RESULT:
next_stub.THIS_CALL
```

Method Results fragment

```
b_array = RESULT
real_result = UNMARSHAL
return real_result;
```

```
public class UniqueIDCreator implements ByteArrayTagger {
  int counter;
  private static byte[] toByteArray(long x, byte[] array) {
    for (int i = 0 ; i < array.length ; i++)
      array[i] = (byte) ((x >> (i*8)) % 0xFF);
    return array;
  }
  public UniqueIDCreator() { counter = 0; }
  public byte[] createTag() {
    return toByteArray(counter++, new byte[4]);
  }
}
```

Figure 3: Code fragments for Data Translation to prepend a tag

A sessionize transformation introduces additional storage for some value which is to be recorded and later replayed. The *arguments* and/or *result* of the call might be recorded; arguments may be replayed *on error* to retry the call, or in place of arguments of a subsequent call.

4 Validation Results

As part of the thesis validation, I performed two case studies in which I applied connector transformations to existing systems. One of the case studies is discussed here. The other (modifying Java Message Service publish/subscribe connectors in Sun's J2EE "Pet Store" web-commerce blueprint application) is detailed in the thesis.

In addition, I also measured performance of the set of generated connector transformations used in the first case study; I measured the time taken by a "novice" user to instantiate and apply a connector transformation to a software system familiar to him; and I gave a coarse task breakdown comparing the time taken to modify a connector "by hand" as compared to using connector transformations. These results are detailed in the thesis, along with a small experiment (adding Kerberos to a Java RMI connector) that shows that the connector transformation tool can also be used for enhancements outside the domain of fault tolerance.

Transformation	Parameter	Hook-in
Data trans.	Function implementation	New method of proxy, or method of a new class (new data member of proxy)
	... What the function is applied to	Arguments of call and/or result of call
	Location	(Tool input)
Add { *Observer, Redirect, Switch, Parallel* }	New role	A second proxy (new data member of proxy)
	... Created at	Initialization
	Traffic director	Before call or after call or replacing call target
	Traffic set	(Tool input)
Aggregate	Second connector	A second proxy (new data member of proxy)
	... Created at	Initialization
	Selector	New method of proxy, or method of a new class (new data member of proxy)
	Selection points	On error, or before or after call
Sessionize	Recorder	New data member of proxy, or local variable
	When to record	Record call before/after, or record result after call
	Replayer	On error or in place of call

Table 4: Hook-ins used by some transformations

4.1 Case Study: Java RMI and VisAD

In this case study, I modified Java RMI connectors in collaborative VisAD applications.

VisAD, or "Visualization for Algorithm Development", is a freely available Java library that enables interactive visualization of numerical data. Two examples of applications that use VisAD are "Galaxy", which allows astronomers to manipulate a simulation of the Milky Way and view its appearance from Earth, and "GoesCollaboration", which allows atmospheric physicists to manipulate weather satellite data [2].

Suppose that we would like a VisAD-based system to be more resilient to server crashes and to high-variance communication latency. The raw Java RMI support provided in the VisAD library does not have facilities for diagnosing and dealing with situations in which the client is (briefly or otherwise) unable to contact the server.

To deal with the potential failure of the server, we would like to use a connector that transparently incorporates *failover* to a backup server. Since servers in a VisAD-based system generally have state, we will also need some means of bringing the backup server's state up-to-date before it is called into service; we might do this by maintaining *checkpoints* and/or logs for the primary server and replaying them to the backup server. Finally, because we expect that, in the final deployment environment of our proposed VisAD-based system, sometimes a timeout might be due simply to a brief spike in communication latency or a transient failure in the transmission channel (rather than

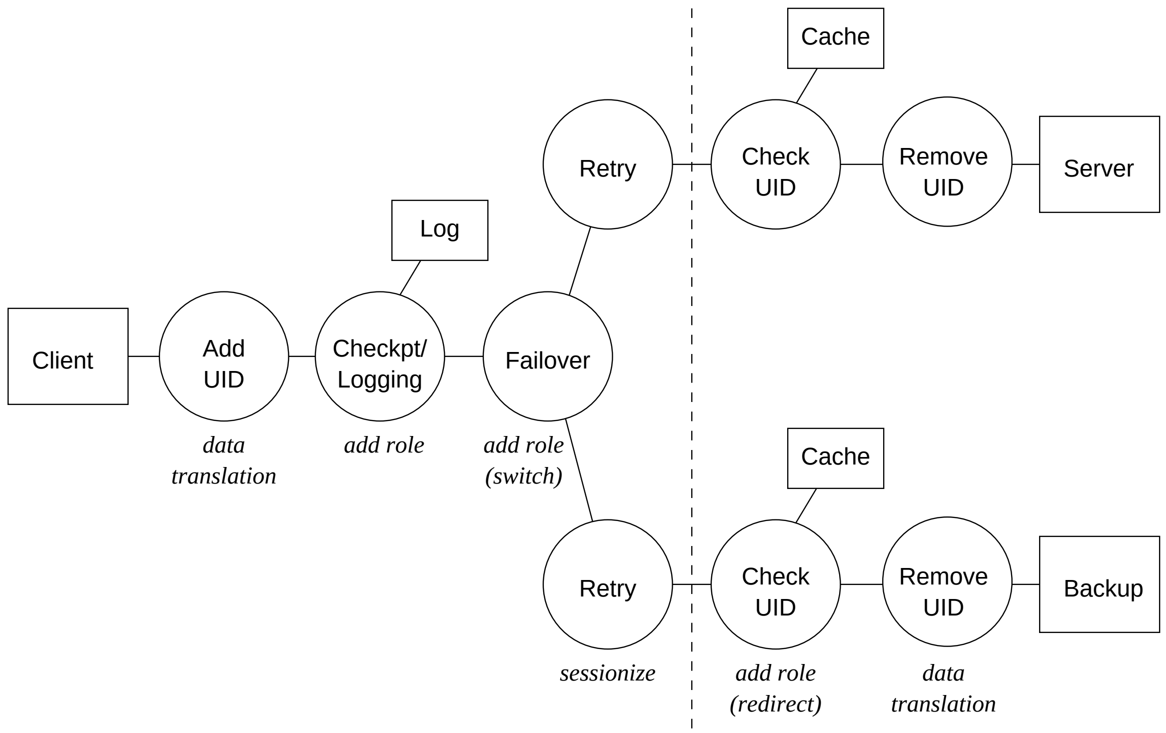

Figure 4: Ordering of transformations

actual server failure), we would also like to have the connector automatically *retry* any request that times out, before resorting to the backup server.

For this case study, I composed the above set of modifications and created an enhanced RMI-based connector; I used the tool support to generate the connector implementation, and tested it in an existing collaborative VisAD system. Let's consider the modifications, and their interactions, in more detail.

- Retry with duplicate detection; this is composed of two simpler modifications:
  - Naive retry. If a request times out, send it again; perhaps this time it will get a response. (Since the error could have occurred during the transmission of either the request *or the response*, this could result in sending duplicate requests to the server. Thus it assumes that all services are idempotent.) We use a *sessionize* transformation.
  - Duplicate detection, to protect non-idempotent services. Recent requests (and corresponding responses) are cached at the server role; if a request is repeated, the cached response is returned. (This requires some means of uniquely identifying a request.) We use a *data translation* transformation to introduce a unique identifier, and an *add redirect* transformation to redirect incoming requests to the cache of recent requests.
- Failover. If the primary server is unresponsive, redirect the client's requests to a backup server. (This particular failover modification assumes that the backup server is stateless or up-to-date; the purpose of the next modification given here, logging/playback, is to make the backup server's state up-to-date.) In this composition of enhancements, if "retry" doesn't obtain a response, then we'll consider the server unresponsive. We use an *add switch* transformation to add the backup server.

- Logging/playback. This is a variant of checkpoint/rollback. Client requests are logged (between checkpoints, if any). The log may be played back or cleared. In this composition of enhancements, we'll use playback to update the backup server[5], before switching from the unresponsive primary to the backup. The failover modification provides the diagnosis that playback is needed. We use an *add redirect* transformation, similar to the form used for a cache, to add the logger.

The transformations used for these modifications interact in two ways. First, some modifications perform "diagnosis" for subsequent modifications: if retry is not successful in masking a problem, for example, then we assume that the server is unresponsive and that it is appropriate to try failover. This manner of cascading several techniques is common in dependability. Second, some transformations may operate on data that was introduced by another transformation: the unique identifier used by the duplicate-caching role was added to the communication by a data translation. The overall transformation ordering, shown in Figure 4, follows directly from these interactions: first, cascading modifications must be ordered so that, if X is subsumed by Y or provides a diagnosis for Y, X must be tried before Y; and second, the unique identifier data must be introduced before (and used after) all modifications that re-send a request. The task of ordering the transformations will be intuitive for someone who understands the domain, but is not time consuming (though it may require a little thought) even for someone who is not a domain expert and is reusing existing domain-specific transformations.

The code fragments for this set of transformations were written previously and applied to a simple "Java RMI tutorial" client-server system. To "reuse" them on a VisAD system it took under 30 minutes to apply the transformations, compile, and run the working modified system. For this system, the tool generated 962 non-blank non-comment lines of code. Then I applied the same transformations to another of the sample applications distributed with VisAD; this took under 10 minutes (again, generated code was about 1 KLOC), and no further problems were discovered. As an end-to-end check of whether the transformations operated as advertised, I tested the effect on the demo system of deliberately dropping requests and responses, and the effects on the demo system and the VisAD systems of killing their primary server; the transformed connectors performed as desired, first retrying the request and then (if the primary was still unreachable) failing over to the backup.

I also investigated the time required to make minor changes to the connector transformations. First, I altered the call-logging policy so that outgoing calls are recorded before their results are returned (rather than waiting for the result). This took 5 minutes including recompilation of the tool-generated code. Next I altered the "retry" transformation to retry a timed-out call up to three times (rather than only once) if it continues to generate an exception. This task took 2 minutes including recompilation.

5 Conclusions

5.1 Contributions

This work's contributions to the field of computer science fall into the following areas.

[5]The logging approach assumes that all state changes to the server are reflected in, and/or result from, communication. Further, placing the log at the client may be undesirable in the case of multiple clients; an alternative approach, using a different connector transformation, would be to make the backup server an "observer" of the traffic to the primary.

A new approach for constructing modifications to component interactions. This thesis describes an approach of separating a monolithic modification into a set of smaller composable connector transformations.

A new kind of generation technology. This thesis describes an approach in which a diverse range of new connector implementations can be semi-automatically generated from a small set of base connectors by applying domain-specific instantiations of connector transformations.

An approach to formalism that incorporates reusable parameterized fragments. In order to provide leverage both for people who have a sufficiently firm grasp of formalism to be able to *synthesize* formal descriptions, and for people whose current ability is limited to *application* of existing descriptions, this thesis provides parameterized templates for connector transformations.

A formalism that is targeted at software engineers. The formal work of this thesis focuses on analyses that are intended to address engineering concerns.

A new view into the space of software connectors. By considering complex connectors in terms of compositions of connector transformations and simpler connectors, the space of connectors appears simpler and more orderly, revealing hidden similarities between variants of connector types.

5.2 Future Work

The connector transformation approach offers opportunities for interesting future research directions.

Dynamism. A dynamic approach to connector modification, enabling connectors to be reconfigured or changed "on the fly" in ways that were not foreseen when the system was deployed, would support current research topics such as self-healing systems. A vital piece of this work would be to resolve or mitigate the safety considerations associated with allowing connectors to modify themselves in unforeseen ways.

Integration of formalism. The implementation generation tool described earlier could in principle help to enforce a simple correspondence between specific wrapper processes (in the formal expression of the transformation) and equivalent added and modified code artifacts (in the implementation). Providing a higher level of guarantee of process/artifact correspondence would aid in traceability of problems, from their discovery in one, to the equivalent in the other. Another possible direction for this work is the actual semiautomated derivation of a formal description from the kind of "code fragments" used in the generation tool, and, in the other direction, deriving such code fragments from formal descriptions.

Usability of formal methods. The formal notation used in the thesis to describe connector transformations is intended to be easy to use, but is not as easy to use as the target audience would like. This research direction would require discovering what are the aspects of formal notation that people find difficult, and investigating alternative, perhaps more visual, approaches that still support an expandable collection of analyses that software engineers could find useful.

Tools for tools. Currently, the process of extending generational tool support to additional connector types is time-consuming. This research direction would involve investigating ways of generalizing or streamlining the process of extending generational tool support to additional connector types, to provide support for *tool* generation. First steps would include more rigorously defining the needs of the tool (which I have described only informally in this document), and categorizing common implementation techniques that can be used to provide these needs, such as, common techniques (including those that are operating-system specific and language specific) that may be used for interception of library calls.

5.3 Concluding Remarks

Connectors in software architecture design provide a means of embodying interactions between components, a useful abstraction for the more complex and nonlocalized nature of their equivalent in a software system implementation. Similarly, connector transformations provide a more principled and abstract way of looking at complex enhancements to connectors. The approach, the tool support, and the techniques presented in this thesis do not resolve all issues that relate to the task of incorporating additional extrafunctional properties in a software system. However, by clearing away some of the tedious problems involved in reifying such modifications, connector transformations will ultimately help to liberate designer attention to focus on those issues that, by their nature, *require* human decision-making, such as determining for a particular software system the most appropriate tradeoff between conflicting quality attributes.

References

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

[2] W. Hibbard, C. Rueden, S. Emmerson, T. Rink, D. Glowacki, T. Whittaker, D. Murray, D. Fulker, and J. Anderson. Java distributed objects for numerical visualization in VisAD. *Communications of the ACM*, 45(4ve):160–170, Apr. 2002.

[3] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *21st International Computer Software and Applications Conference (COMPSAC97)*, pages 6–13, Washington, D.C., Aug. 1997.

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

[5] R. Simmons and G. Whelan. Visualization tools for validating software of autonomous spacecraft. In *The Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS)*, Tokyo, Japan, August 1997.

[6] Sun Microsystems. Java Message Service. http://java.sun.com/products/jms.

[7] Sun Microsystems. Java Remote Method Invocation (RMI). http://java.sun.com/products/jdk/rmi.