Toward Compositional Construction of Complex Connectors

Bridget Spitznagel
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA
sprite@cs.cmu.edu

David Garlan
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA
garlan@cs.cmu.edu

ABSTRACT
A critical issue for systems composed of independently-developed parts is the design and implementation of mechanisms that allow those parts to interact. In many situations specialized forms of interaction are needed to bridge component mismatches or to achieve extra-functional properties (e.g., security, performance, reliability). Unfortunately, system developers have few options: they must either 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, at relatively low cost. We also outline the formal underpinnings for these operators and illustrate how to reason about properties of operator composition.

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. 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.

A critical issue for such constructions is the design and implementation of the interaction mechanisms, or *connectors,*[1] that permit the components to work together. While generic forms of component interaction, such as RPC, may be sufficient in many cases, specialized forms are often needed both to make the parts work together at all, as well as to achieve desirable properties of performance, security, reliability, etc.

For example, additional mechanisms may be needed to adjust data formats (e.g., from big-endian to little endian, or for unit conversion), to compensate for different control mechanisms (e.g., synchronous versus asynchronous), or to support 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. Currently there are two alternatives: use an existing, off-the-shelf connection mechanism, or handcraft a specialized one. Neither alternative is adequate. On the one hand, it is not always possible to find an existing connector that meets the needs of the system. On the other hand, creating a new connection mechanism is 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 interaction capabilities in various combinations. 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. If each is built as a monolithic, handcrafted form of connection, the overall costs quickly become prohibitive.

In this paper we argue that 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. Basic interaction mechanisms, such as RPC, would be augmented with selected adaptations to produce more complex connectors. 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 initial steps to-

Draft of paper submitted for publication.

[1] We will use the term "connector" to refer to an interaction mechanism that supports component integration.

wards realizing such an approach. We begin by motivating the problem; next we enumerate 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 operators can be used to realize complex interactions, such as Kerberized RPC, at relatively low cost. We then outline formal underpinnings for these operators and illustrate how the formalism supports reasoning about operator properties, such as commutativity and compositionality.

2 RELATED WORK

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

The first area is software architecture, in which the treatment of connectors as first-class entities was introduced [7, 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. This treatment of connectors also enables their formal specification and analysis, independent of the components they are to 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 also is of interest due to its concern for how specific connectors can be built up in a traceable and modular way. Also related are areas of work which explore 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 then operate to resolve the mismatch [8]. Sometimes component wrappers are used. Another technique, Flexible Packaging [3], separates the component's functionality (ware) from its assumptions about the communication infrastructure (packaging); the packaging may eventually be associated with the connector.

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 [11]. Some properties of safety and (given certain restrictions) liveness can be predicted in a similar incremental way using a finite state machine model [10]. An example of recent work in the area of protocol synthesis is Ensemble [12] which enables the construction of an adaptive protocol composed of stacked micro-protocol modules.

A third area of related work is in code generation as it relates to generating connectors. UniCon [9] 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 [4]. 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 [14]. Our work differs from these in its focus on producing new connector types that may be based on a variety of existing connector types.

3 MOTIVATION

Today, when an existing connector must be altered or replaced by a hand-built mechanism, someone must take on the implicit role of connector modifier. At present this task is difficult and 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. Suppose the system's architect decides that it is necessary to improve the security of the system, and therefore that some or all of the component interactions should use authenticated communication. Further, they decide to use Kerberos [6] 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. First, changes are distributed throughout the system. If later a variant or new version of the Kerberos protocol should be used instead, the team of implementors must go through the same amount of work again. Second, it is not possible to reuse the modifications in another Java-RMI-based sys-

tem, except in the primitive sense of cutting and pasting the names of the library functions; and adding another modification to this system would (at best) entail the same amount of work. Third, there is no guarantee that the implementors will carry out the changes correctly.

The second alternative is 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. To do this 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 users need only make sure they 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. Similarly, other people who want to modify Java RMI gain nothing from the original changes (unless of course they are satisfied with exactly the same new connector type); they will have to do the same amount of work for themselves.

If connector transformation tools were available, a superior approach would be 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 before 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 a number of 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, what are the semantics of connector transformation? Ideally we would like to be able to explain a transformation precisely and reason formally about properties such as idempotence, commutativity, and compositionality. Third, 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 A SET OF TRANSFORMATIONS

There are many approaches that one might use in choosing a set of transformations. At one extreme one must 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.

We'll describe these transformations informally in this section, and then come back to ways of making them precise and automating their use. For each transformation we summarize its purpose, examples of its use, and its dimensions of variability.

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 Tranform 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)? Is it a pure function, or does it have memory and/or an external input?

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

Software adaptors [13], 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.

Add a Role
This tranformation 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.

An example of an *observer* is a simple eavesdropper that logs all communication. Another example is an auditor that requires additional 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 behaves as a cache, a confirmer or authorizer that determines whether a request should be allowed to proceed, and a "trusted third party" that supplies additional information such as encryption keys to be used by another transformation.

Choices that must be made during instantiation include the following: Whether the role 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?).

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 is more efficient than contacting a trusted third party to verify each message.

An instantiation 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.

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 excution. 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 they can both communicate at.

An adaptive connector's controller is able to monitor some aspect of a changing environment. The controller will dynamically change which protocol is being used, based on which one is currently "best." For example, the controller might monitor communication faults and switch to/from a more fault tolerant but less efficient protocol; or the two connectors may be connected to a primary and backup server, and the controller detects when the primary has gone up/down and uses the backup instead.

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

5 EXAMPLE

Before we give these transformations a more formal characterization there are some questions remaining in the informal description. How are connector transformations used? What steps occur in the transformation process? To answer these 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 comparing them to the tasks that appear in (or are implicit in) the protocol. In this case we end up with three transformations. Selection of transformations can be based on typical patterns of use which can be documented to make this step easier.

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 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 then used repeatedly 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[2].

[2] As an implementation note, since the example is in Java, and the Kerberos libraries are in C, they would be 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.

	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

In the alternatives described earlier, in addition to having to write those fragments, they must be inserted by hand, either in the component implementation or the stub generator, which would be a time consuming process because the fragment destinations are not localized. Table 1 illustrates some locations where code fragments must be inserted, in the case of adding Kerberos to Java RMI. The cells of the table show some transformations that must insert fragments at that location to achieve the new Kerberized connector; this insertion is done automatically by the connector transformation tool whereas in the other alternatives it would have been done manually. For example, the Sessionize transformation must add initialization code called when the client and remote objects are created, whereas the Data Transform occurs when the remote methods are called; we can see that fragments associated with these two transformations would be inserted in at least two different sections of the code.

Now that the transformations are determined, and a few code fragments have been associated with each, the transformations, fragments, and the component implementations are provided as inputs to a connector transformation tool. This tool (discussed later) will generate the implementation of the new connector, by creating wrappers, modifying component implementations, and/or producing non-code artifacts such as makefiles.

6 SEMANTICS

In order to describe the semantics of a *connector transformation*, one must first decide how to specify the semantics of an individual *connector type.* For the purposes of this research we have adopted an approach that defines the semantics of a connector as a set of protocols. Specifically, each connector is defined as a set of *roles* protocols and a *glue* protocol. Each role defines the behavior of a single participants in an interaction,

whereas the glue defines the coordination between those behaviors.

To specify the protocols themselves one could use any number of formal languages. In this case we will use a subset of CSP [5] that includes sequencing ($\rightarrow$), internal choice ($\sqcap$), external choice ($\Box$), interleaving ($|||$),and parallel combination ($\|$). Successful termination is indicated by the symbol §.[3]

To illustrate the specification of a connector type consider a simple client-server connector. As illustrated below, connector (c) has two roles *Client* and *Server.* The client role repeatedly initiates requests (indicated by the overbar) and observes the replies. At any point the client may choose (internally) to terminate successfully. The server role is similar but reversed. The server must be prepared to terminate successfully. The *glue* coordinates the requests and replies, transmitting requests of the client role to requests at the server role, and similarly for replies. For simplicity, we will use examples that are based on this two-role connector. However, in general a connector can have an arbitrary number of roles.

Connector c =
  Role Client = $\overline{\text{request!x}} \rightarrow \text{reply?y} \rightarrow \text{Client} \sqcap$ §
  Role Server = $\text{request?x} \rightarrow \overline{\text{reply!y}} \rightarrow \text{Server} \Box$ §
  Glue = $\text{Client.request?x} \rightarrow \overline{\text{Server.request!x}} \rightarrow$
    $\text{Server.reply?y} \rightarrow \overline{\text{Client.reply!y}} \rightarrow \text{Glue} \Box$ §

With this notion of connector semantics in hand, we can then define a connector transformation as a function from one or more connector types (together with some parameters specific to that transformation) to a new connector type. To specify this function we will define its signature (types of inputs and outputs), preconditions on its application, and postconditions that define the roles and glue of the new connector type, C'.

To illustrate this approach we now look at the specification for variants of two of the five transformations from the previous section. (Space does not permit us to detail all five, although the others are similar to the examples shown here.) We then illustrate the application of that transformation on the client-server connector type shown above.

Data transform
This transformation modifies a connector C, with role processes $r_0 \ldots r_n$, so that the data carried by some events is transformed. This is a data transformation in which events are neither created nor destroyed, only renamed. For each role r_i, a function f_i is provided which transforms the alphabet of that process; f_{glue} transforms the alphabet of the glue's process. The result of this transformation is a new connector type C'.

Inputs
C: connector type with roles $r_0 \ldots r_n$ and glue g = Glue(C)
f_i: Alphabet(r_i) $\rightarrow\Sigma$
f_{glue}: Alphabet(g) $\rightarrow\Sigma$
Result
C': new connector type

Preconditions
The functions f_i and f_{glue} are injective.
Postconditions
Roles(C') = $f_i(r_i) | i \in 1 \ldots n$
Glue(C) = $f_{glue}(\ g\ )$

Suppose that the server in the example above uses a big-endian format but the component connected to the client role will use little-endian. Let f_{client} be the function that maps $\overline{\text{request!L(x)}}$ to $\overline{\text{request!x}}$ and maps reply?y to reply?L(y), where L(n) is the little-endian representation of n. Similarly f_{glue} is the function that maps Client.request?L(x) to Client.request?x and maps $\overline{\text{Client.reply!y}}$ to $\overline{\text{Client.reply!L(y)}}$. In this case f_{server} is the identity function. Here is the result:

Connector c1 =
  Role Client = $\overline{\text{request!L(x)}} \rightarrow \text{reply?L(y)} \rightarrow \text{Client} \sqcap$ §
  Role Server = $\text{request?x} \rightarrow \overline{\text{reply!y}} \rightarrow \text{Server} \Box$ §
  Glue = $\text{Client.request?L(x)} \rightarrow \overline{\text{Server.request!x}} \rightarrow$
    $\text{Server.reply?y} \rightarrow \overline{\text{Client.reply!L(y)}} \rightarrow \text{Glue} \Box$ §

In this new connector, the client will send and receive the request and reply data in the little-endian format, while the server sends and receives the data in the original big-endian format, with translation occurring between them in the connector.

Add an eavesdropper role
This transformation modifies a connector C by adding a role r and enabling it to eavesdrop on events in the set E. Events in E are drawn from the set of all events that the glue of C might use (this set is the "alphabet" of C's glue). A function f is used to rename the events of E, which are qualified with the names of other roles, so that they are instead qualified with the name of the new role r, enabling r to participate in them. We require that r does not have the same name as any role in C.

Inputs
C: a connector type
r: a role specification, named name(r)
$E \subseteq$ Alphabet(Glue)

[3] This is essentially the subset of CSP used by the Wright architecture description language. See [5] for details on its definition and principal uses [1].

f: E $\rightarrow\Sigma$
Result
C': new connector type

Preconditions
$\forall$ r0 $\in$ Roles(C_0) $\bullet$ name(r) != name(r0)
Postconditions
Roles(C') = Roles(C_0) $\cup$ $\{r\}$
Glue(C') = (Glue(C_0) ||| $RUN_{f[E]}$) $\underset{E \cup f(E)}{\|}$ P
where P = ($\Box$ e : E $\bullet$ e $\rightarrow$ f[e] $\rightarrow$P) $\Box$ §

In the first postcondition, we can see that the roles of the original connector are unchanged by this transformation, so that we would expect the original components would not need to be modified.

The second postcondition defines the new glue. First, the original glue is interleaved with a RUN process whose alphabet is $f(E)$[4]. The process that results from this interleaving is like the original glue, but is also willing to participate, at any point, in any event of $f(E)$.

Next, the interleaved process is placed in parallel with a new process P. P's purpose is to cause each event $e \in E$ to be followed by the corresponding event $f(e)$; we will see in the example below that this causes the information in e to be sent to the new role r. The subscript of the parallel operator indicates that the two processes it joins will synchronize on events in the set $(E \cup f(E))$; they don't have to synchronize on any other events and can perform them independently of one another. This ensures that, first, when an event in E occurs the corresponding $f(e)$ follows without other events intervening, and second, P's unwillingness to participate in events outside $E \cup f(E)$ will not affect the rest of the glue, which does expect such events to occur.

To better understand the effects of this transformation, consider an example. Suppose that we were to add an eavesdropper role to the connector given in the example. For simplicity, we will use a role that is willing to accept (and record) any event:

Role Eaves = RUN_Σ

Let E be {Client.request?x , Server.reply?y}. That is, we want the eavesdropper to listen to requests as they are sent by the client, and replies as they are sent by the server. Our function f will map Client.request?x to $\overline{\text{Eaves.request!x}}$ (for any x $\in$ T, where T represents the type of data allowed on that channel), and will map Server.reply?y to $\overline{\text{Eaves.request!y}}$.

Connector c2 =
Role Client = $\overline{\text{request!x}} \rightarrow$ reply?y $\rightarrow$ Client $\sqcap$ §
Role Server = request?x $\rightarrow$ $\overline{\text{reply!y}} \rightarrow$ Server $\Box$ §
Role Eaves = RUN_Σ
Glue = ((Client.request?x $\rightarrow$ $\overline{\text{Server.request!x}} \rightarrow$
Server.reply?y $\rightarrow$ $\overline{\text{Client.reply!y}} \rightarrow$ Glue $\Box$ §)
||| $RUN_{f[E]}$) $\underset{E \cup f(E)}{\|}$ P
where P = (Client.request?x $\rightarrow$ $\overline{\text{Eaves.request!x}}$
$\Box$ Server.reply?y $\rightarrow$ $\overline{\text{Eaves.reply!x}} \rightarrow$ P) $\Box$ §

Example
Providing a formal definition of each transformation has the obvious value of precision in documenting what the transformation does and the what conditions under which it is applicable. However, we can also use the formalism to do much more.

A number of questions arise when defining such transformations. Which transformations can be combined with others? Are they communtative? Do certain combinations lead to deadlock? Does a transformation require a change to components that used the initial connector? The formal specifications provide partial answers to these questions.

To illustrate we will show how the formalism can help us show that the transformations defined above are not commutative. First, we apply the add-an-eavesdropper transformation to c1:

Connector c3 =
Role Client = $\overline{\text{request!L(x)}} \rightarrow$ reply?L(y) $\rightarrow$ Client $\sqcap$ §
Role Server = request?x $\rightarrow$ $\overline{\text{reply!y}} \rightarrow$ Server $\Box$ §
Role Eaves = RUN_Σ
Glue = Client.request?L(x) $\rightarrow$ $\overline{\text{Server.request!x}} \rightarrow$
Server.reply?y $\rightarrow$ $\overline{\text{Client.reply!L(y)}} \rightarrow$ Glue $\Box$ §)
||| $RUN_{f[E]}$) $\underset{E \cup f(E)}{\|}$ P
where P = (Client.request?L(x) $\rightarrow$ $\overline{\text{Eaves.request!L(x)}}$
$\Box$ Server.reply?y $\rightarrow$ $\overline{\text{Eaves.reply!x}} \rightarrow$ P) $\Box$ §

In the final two lines we see that the eavesdropper receives the requests in the client's format, and the replies in the server's format. Now, we apply the little-endian data transform to c2:

Connector c4 =
Role Client = $\overline{\text{request!L(x)}} \rightarrow$ reply?L(y) $\rightarrow$ Client $\sqcap$ §
Role Server = request?x $\rightarrow$ $\overline{\text{reply!y}} \rightarrow$ Server $\Box$ §
Role Eaves = RUN_Σ
Glue = Client.request?L(x) $\rightarrow$ $\overline{\text{Server.request!x}} \rightarrow$
Server.reply?y $\rightarrow$ $\overline{\text{Client.reply!L(y)}} \rightarrow$ Glue $\Box$ §)
||| $RUN_{f[E]}$) $\underset{E \cup f(E)}{\|}$ P
where P = (Client.request?L(x) $\rightarrow$ $\overline{\text{Eaves.request!x}}$
$\Box$ Server.reply?y $\rightarrow$ $\overline{\text{Eaves.reply!x}} \rightarrow$ P) $\Box$ §

[4] We can expect $f(E)$ and E to be disjoint: all events in E are qualified with names of roles of C, all events in $f(E)$ are qualified with the name of r, and no role in C has the same name as r.

Above, we see that the eavesdropper receives requests in the original format, not in the client's format.

7 IMPLEMENTATION

A Simple Prototype Tool

As an initial exploration toward the goal of rapid, easy development of useful "good-enough" connectors, we have written a prototype development tool that applies a restricted set of transformations to a specific connector type, producing a Java implementation of a more complex connector. Although the transformations were not fully general they were diverse enough to begin investigating the issue of compositionality of transformations in implementation and to explore the Kerberos example seen in the motivation section. This work is a starting point and illustrates what can be done when source code for components is available.

When no transformations are used, the tool naively generates a Java-RMI connector; this is the base connector. The tool has knowledge of what initialization steps to add, etc., to prepare for and perform a remote method invocation.

Four kinds of connector transformations are implemented, with some restrictions: data transform, add a role, aggregate, and sessionize. Each transformation inserts code fragments at places that are specific to that transformation; these fragments are supplied by the person using the tool.

The data transform is a translation of the arguments of a remote method and/or the value returned (if any). This can occur at sender, receiver, or both ends. We have used this transformation to change the arguments sent without changing their number or type, as well as to change the type of the return value and the number and type of the arguments, for example to convert an arbitrary number of arguments into a single byte array (the data format expected by JGSS).

The role addition can be an observer that receives information on what method was called, the arguments and the return value, without having an effect itself on the communication, or can be a participant that is called on to confirm the validity of communicated data, for example obtaining information from a trusted third party and using it to check a received message for authorization.

The aggregation is restricted to two similar connectors. Similar in this case means that each is either the Java-RMI base connector or is a connector derived from it using this transformation tool. The controller which decides whether to switch the active connector is restricted in where it is allowed to operate also; it can be consulted only at specific points, such as when a remote call is about to be made or when an exceptional value is returned; in this implementation it cannot, for example, cause the client to stop waiting or try another server, if the call is taking too long to return.

The sessionize transformation can introduce instance variables that will be used to store computed or received results to save recomputation or retransmission time; it also deals with their initialization which will occur at object creation.

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; this can include complex connectors that are the result of applying a series of transformations to the base connector. To do this, the prototype tool produces 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 essentially 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 more advanced version or a tool which operated on a different base connector would likely also generate non-code artifacts such as system configuration and makefiles.

The Kerberos example

We investigated a "real world" modification, described earlier in section 3: adding Kerberos authentication to a Java RMI connector.

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 the example programs distributed with JGSS. The code fragments and sequence of transformations to apply can be be given as inputs to the general tool, or to create a specific Kerberizing-Java-RMI tool they could be hard-wired in. In either case the tool will operate on source code for components and produce 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

	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

to actually 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. Brief 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 that is not easily exploited by naive cut and paste.

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 Kerberos. Table 2 illustrates some places where code fragments are inserted, in the example of adding Kerberos to Java RMI; the "reused" code shows what portion of the Kerberos-specific code was borrowed from an existing demo program.

In summary we have built a prototype tool which performs a subset of the connector transformations described in this paper to produce implementations of a variety of complex connectors derived from a single basic connector. These transformations are composable and we have used the tool to create and apply a complex change, Kerberization, using three of the transformations, with less effort than would have been required using another approach.

8 DISCUSSION

The work described in this paper is a 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. While more work will be needed to decide this question, our experience in the domains of security and reliability suggest that the transformations are easily applied to a wide variety of connectors. On the negative side, however, the formalism is a little messy. The fact that there are only a small number of connectors helps ameliorate this situation, however.

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 semantic notation of transformations should be independent of any particular connector, the implementation of a transformation tool is specific to the base connector type. 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, and such extension of the implementation will be addressed in future work.

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 shows whether the 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 to change it. 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, sometimes source is available and that additional leverage may be desirable. Second, sometimes transformations that change roles are themselves desirable to resolve mismatch, when the components do not match the original interface and the connector is being transformed to match the components' expectations.

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

five basic transformations and illustrated how they can be used to create complex forms of interaction, such as Kerberized RMI. We also outlined an approach to defining their semantics formally as protocol transformation, and illustrated how such formalism can be used to reason about the transformations. Finally, we briefly described a prototype tool that can be used to apply these transformations in the special case of Java RMI-based interactions.

As indicated by the title of this paper, 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 or modify 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 develoment platforms (beyond Java). Further, there are considerable opportunities for exploiting the formal theory to carry out detailed analyses about transformation composition and compatibility. This line of research would also benefit from looking at other forms of protocol specification – for example incorporating some notion of timine. Finally, more case studies are needed, particularly in other domains beyond security.

ACKNOWLEDGEMENTS

This research was supported by the Defense Advanced Research Projects Agency and Rome Laboratory, USAF, under Cooperative Agreement F30602-97-2-0031, 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. We would like to thank Mary Shaw, Jeannette Wing, and the members of the ABLE group Drew Kompanek, Zhenyu Wang, Jianing Hu, and Joao Sousa.

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

[4] 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.

[5] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

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

[7] 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.

[8] 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.

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

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

[11] 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.

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

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

[14] 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.