Software Architectural Transformation

S. Jeromy Carrière, Steven Woods, Rick Kazman
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{sjc, sgw, kazman}@sei.cmu.edu

Abstract

Software architecture, as a vehicle for communication and reasoning about software systems and their quality, is becoming an area of focus in both the forward- and reverse-engineering communities. In the past, we have attempted to unify these areas via a semantic model of reengineering called CORUM II. In this paper we present a concrete example of an architecturally-motivated reengineering task. In executing this task, we perform architecture reconstruction, reason about the reconstructed architecture, motivate an architectural transformation with new architectural quality requirements, and realize this architectural transformation via an automated code transformation.

Keywords: Software Architecture, Architectural Analysis, Architecture Reconstruction, Transformation

1. Introduction

Software architecture is increasingly being viewed as a key artifact in realizing an organization's technical and business goals. Bass *et al* [1], for example, introduced the notion of the *Architecture Business Cycle* as a means of analyzing not only the technical implications of a software architecture, but also its implications with respect to requirements, constraints, and the business goals of the system or systems' various stakeholders such as cost, time to market, and opportunities to enter new markets.

The reverse engineering and reengineering research and practice communities have similarly seen the benefits to explicitly analyzing and planning a software architecture. Reverse engineering a software architecture can aid in [6]:

- analyzing the system's architecture, and hence in ensuring that the system meets its quality goals, such as performance, availability, and modifiability
- redocumenting the system and ensuring the conformance of the system to its intended design
- mining and documenting software assets in a legacy system, and
- planning for reengineering.

In prior work [7], we have presented a unified model of reengineering, called CORUM II. The model was "unified" in the sense that it explicitly addressed issues at both the level of code and of software architecture, and linked the two levels. CORUM II also discussed how the levels of representation needed to be transformed in a reengineering task. The architecture might be changed, and hence the code would need to implement this change. Or, the code might be changed, and this change should be reflected in the architecture. In either case, the models of the software need to be unified, and to remain unified through any code-based or architectural transformation, if the benefits of having a software architecture are to be realized.

This paper takes up where our previous papers left off. In this work we describe how a simple client-server system might be transformed, via the aid of code transformation tools, to meet new quality goals. Here we show how this code transformation can be *driven* from an architectural transformation, motivated by specific desired quality attribute changes. Thus, our previous work in which we discuss the necessity of examining architectural implications of code transformations is reversed: architectural change drives code change.

To aid us in reasoning about such transformations, and to aid in building tools that will automatically or semi-automatically performing such transformations, we have created a model of software architecture that we can use to reason about some specific semantics—called "features"—of architectural components and connectors. This model, first presented in [8] and further elaborated to include semantic mappings between layers of abstraction in [16], can be used to support architectural reasoning. More significantly, we can check these features when doing transformations, to ensure that the transformed version of the code conforms to the features designed into the software architecture. In this way, the quality of the architecture is much less likely to be inadvertently compromised by an incorrect implementation.

2. Background

The work presented in this paper is meant to strongly motivate an architectural perspective in the transformation of legacy software systems. In particular, this perspective on analysis of software-intensive systems derives from work at the Software Engineering Institute on the Architecture Analysis Tradeoff Method (ATAM) [9], a method for evaluating quality tradeoffs among architectural decisions.

As part of CORUM II [7] , we described the "horseshoe", a model of an architecture-based software reengineering cycle. This model rests upon feature-based semantic characterizations of software structures, from the code level to the architectural level. This model is useful in the context of ATAM as a way of representing measurable conformance between "as designed" and "as implemented" software systems. Architects wish to make broad quality statements about their systems, and these statements are only relevant for systems if the implementation is known to conform to the design. For this reason and in service of the more general goal of architecture redocumentation, architecture reconstruction is well-studied of late (see [3], [4], [6], [10] and [17]).

Also, in the horseshoe model we consider a wide range of software renovation possibilities. Software can be transformed from source to source directly, it can be manipulated in the parsed abstract syntax-tree form to give new trees and eventually new source (for example, see [13]), or among other approaches, it can be mapped from source to an architectural view, and transformed into a new architectural model and then specialized into a new system at the source level.

Krikhaar *et al* describe a "two phase" approach to software architecture transformation [11]. The first phase of their approach "extracts" an architecture description from the software, calculates quality metrics over the extracted description and analyzes the impact of potential changes to the architecture. The second phase transforms the system's implementation (source code, build descriptions, etc.) according to a set of *recipes* corresponding to the architectural changes. Examples provided focus on reallocation of functionality realized as C functions between "units" realized as C header and source files. The approach described here is similar, with the addition of a semantic framework that supports reasoning about components and connectors and the development of code transformations from architectural transformations.

3. Example

3.1. Introduction

In this example we demonstrate an end-to-end instance of software transformation both motivated and controlled by specific architectural change requirements. For the purpose of clarity we have greatly simplified the example. However, the example is rooted in a real application, and the code presented, analyzed and transformed is real.

The initial system is a standard client-server application. A single server exists to provide a particular service to any number of clients. Each client is tightly coupled to the specific server by explicit knowledge of its location. Clients use a blocking "send" for a request to the service port and thus may contend for access to the server if other clients are currently using the service.

A set of architectural requirements for system change are driving a need for software modification. First, there are system performance problems that are resulting in a demand for an approach that does not result in clients blocking on service requests. Essentially, the clients are wasting time contending for service that they could be spending on useful tasks. Second, the system architect also wants to de-couple the client and server as part of an effort to increase system maintainability and to support an eventual move to multiple servers.

The changes required from the system can thus be summarized simply:

1. Move to a non-blocking client-server relationship.

2. Remove explicit knowledge of the server's location from the client implementation.

The example transformation will be undertaken using the following generic steps:

1. The architecture of the example will be reconstructed with semantic annotation based on the notion of semantic features.

2. The quality attributes of the reconstructed architecture and the new "desired" architecture will be evaluated.

3. An architectural transformation will be chosen, and a code transformation will be inferred.

4. The code will be transformed to insure conformance of the architecture to the new desired quality attributes.

5. The code transformations will be validated for correctness.

As the work presented here matures, we intend to have these steps evolve into a generalized method for architectural transformation.

3.2. Reconstruct and Transform Architecture

The first step in the transformation of this code and software architecture is to reconstruct the existing software architecture. This is done by extracting architecturally important features from its code and employing architectural reconstruction rules to aggregate the extracted (low-level) information into an architectural representation. This process is

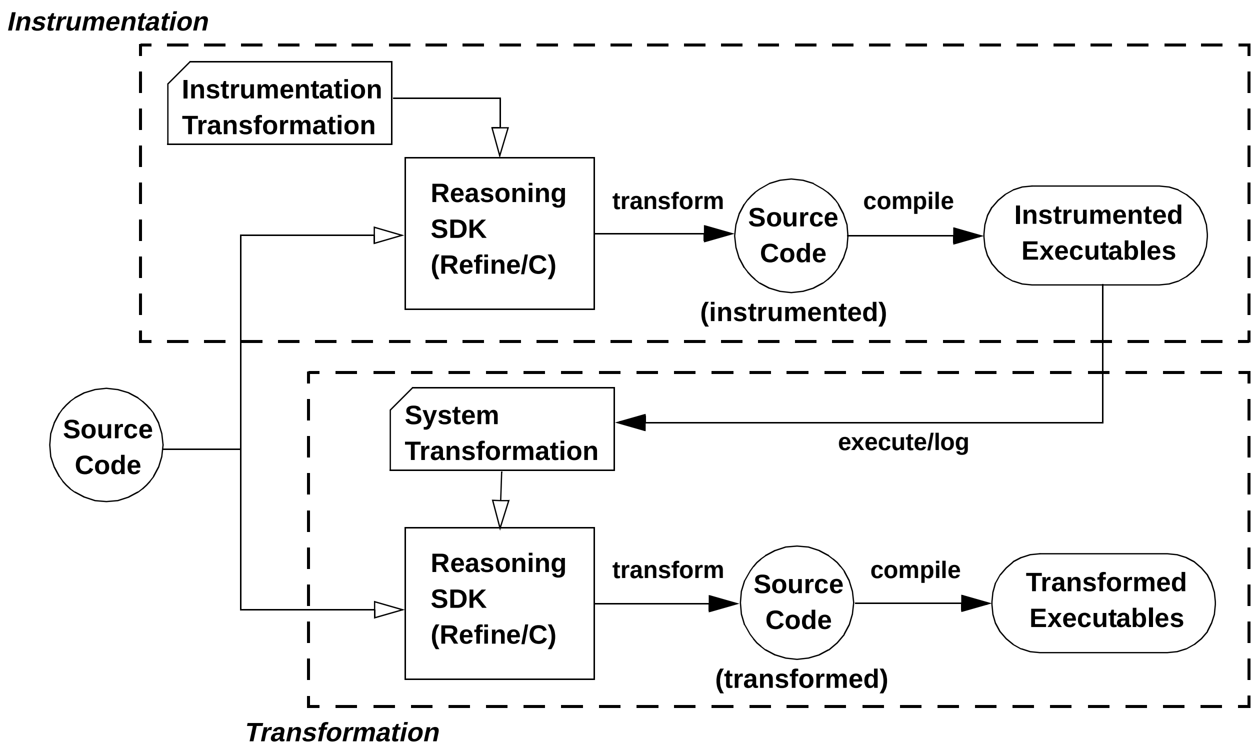

Figure 1: Summary of Instrumentation and Transformation

described extensively elsewhere (e.g. [5], [6]). We will discuss its application only briefly here.

Our simple client-server system is implemented in C using Unix processes and socket communication. To extract the system's architecture, we performed a static extraction of the source code and the system's makefiles to identify function-call and built-from relations. The former identifies function calls within the system as well as calls to the socket primitives used for communication between client and server (these are discussed in detail in Section 3.3.1). The latter identifies how the executables in the system (one each for the client and server) are built from the source files. In addition, we performed dynamic extraction to determine the run-time connectivity of the system as a whole. This was necessary because clients and servers using socket communication are bound via use of "port numbers"; these are passed as parameters to the socket communication primitives. Rather than attempting to perform the sophisticated data flow analysis necessary to identify the port numbers statically, we instrumented the system via a code transformation, and executed it. See the top portion of Figure 1. Based on the static analysis and the results collected from running the instrumented system, we can reconstruct the run-time connectivity of the architecture, as described in [5].

Once the architecture has been extracted, we can view the components and connectors of the system as possessing architecturally significant properties, such as those listed in Table 1 and Table 2. These tables list the possible features of architectural elements (both components and connectors), and are divided into information that can be derived from a temporal perspective and information that can be derived from a static perspective of an architectural element. These features are described in greater detail in [8].

Table 1 summarizes the features derived from the static view, and the possible values for each. The static view's features are enumerated below as a set of categories and an explanation of the reasoning behind each category:

- *Data scope:* what is the largest scope across which data can be passed by the element? Possible answers are that the element passes data within a virtual address space, across virtual address spaces but within a single physical address space, or across a network.
- *Control scope*: what is the largest scope across which control can be passed by this element?
- *Transforms data:* are the element's outputs a transformation of its inputs?
- *Binding Time*: when are the sources and sinks of an element bound? Possible answers are at specification time, invocation time, or execution time.
- *Blocks:* does this element suspend when it transfers control to another element?
- *Relinquish:* if this element has a thread of control, does it

Table 1: Static View Features

Architectural Element	STATIC FEATURES								
	Data scope	Control scope	Transforms data?	Binding time	Blocks?	Relin-quish?	Ports	Associations	
								Per conn	Life-time
Possible answers	V P D n/a	V P D n/a	Y N n/a	S I E	Y N NC	Y N n/a	I O IO	*n*	*n*

ever voluntarily relinquish it?

- *Ports*: what are the directions of connections with other elements (i.e. is data or control coming in, going out, or some combination)?
- *Associations*: what number of elements can bind to this port simultaneously?

Table 2 summarizes the features derived from the temporal perspective, and the possible values for each. Temporal features include:

- *Times of control acceptance:* this feature describes the times when an element can receive control.
- *Times of data acceptance:* this feature describes the times when an element can receive data
- *Times of control and data transmission:* these features describe the times when an element can transmit data and control
- *Forks:* this feature is true if the element can spawn a new thread of control
- *State retention:* this feature describes the conditions under which an element retains state (i.e., where its behavior is a function of previous invocations). Possibilities are that it cannot do so, that it can do so but only within a single thread of control, or that it can do so across multiple threads of control.

For the purposes of this paper, we will only discuss those features that are germane to the presentation of the case study. For example, Table 3 shows the set of static features for a socket: it has a distributed data scope and no control scope (that is, it can transfer data anywhere on the network but it never transfers control), it doesn't transform data, its ports are bound at specification or invocation time (i.e. in the source code or via a command line parameter), and so forth.

Similarly, Table 4 shows the set of temporal features for a socket: it never accepts or transmits control, it accepts data at any time, and so forth. For the purposes of this case study, the crucial features are that a blocking socket call ("Blocks=Y") is used, that its ports are bound at invocation time ("Binding Time=I"), and that the associations are 1 ("Associations=1", that is, each port on a connector is bound to one and only one association with a component).

These features are extracted via a blend of techniques: they can be extracted from the source code, by recognizing a keyword such as "socket" or "bind" or "accept" and then doing a simple table lookup (i.e. once the primitives of socket communication are recognized by a static extraction, the features of the mechanisms can be looked up in a table); and some features can be extracted through mechanism-specific extraction. For example, to determine that the binding of ports in a socket connection is done at invocation time, it is necessary to recognize the socket connection, know where the port number appears in the Unix "socket" call and to know the source of the value that is provided to that call (i.e. whether it comes from a constant, a command-line parameter, or a value generated at run-time). This is a mechanism-specific extraction: one needs to know the specifics

Table 2: Temporal View Features

Architectural Element	TEMPORAL FEATURES							
	Accepts control at other than t_s?	Transmits control at other than t_e?	Accepts data		Transmits data		Forks?	Retains state?
			At t_s?	At other than t_s?	At t_e?	At other than t_e?		
Possible answers:	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	S D SD N

Table 3: Unix Socket Static Features

Socket	D	n/a	N	I	Y	n/a	I; O	1	n

Table 4: Unix Socket Temporal Features

Socket	N	N	A	NC	A	NC	N	N

Table 5: Publish-Subscribe Static Features

Publish-Subscribe	D	n/a	N	I, E	N	n/a	I; O	n	n

of how Unix sockets are implemented to perform it.

Note that the transformations that we are making are solely to the connectors of this architecture. As a design goal we wanted to ensure that the architectural semantics of the client and server remained unchanged (i.e. they remain Unix processes that consume and produce data respectively, they need not be co-located, etc.).

Recall that the transformations that we want to make to the system involve making the client and server more loosely connected to each other in several ways. The client and server need not work in lock step—the client does not need to block waiting for the server to do something and, in fact, the server does not need to receive control from the client; they can each operate in their own process or thread and simply communicate via data. For this reason during our architectural reengineering, we want to change the value of the “Blocks” feature to “Blocks=N”. Also, we would like to alter the “Binding time” and “Associations” features to support execution-time binding and multiple client-server associations, respectively (“Binding Time=E” and “Associations=n”).

We may now consider which alternative connectors will meet our requirements. Consider the characterization of the publish-subscribe connector shown in Table 5 (only the static features are shown, because the temporal features are unchanged from those of the socket). The key differences between the connectors are as follows: publish-subscribe connects *n* input and *n* output ports together per association, whereas a socket only connects 1 input port to 1 output port; the ports are bound at invocation time for the socket, compared with execution time for the publish-subscribe connector; and this publish-subscribe mechanism does not block, whereas in this implementation the sockets *did* block. Now one may argue here that a socket is not *inherently* bound at connection time—one might be able to construct software where it is bound at specification or execution time. But the potential feature values of a connector are not of interest to us for re-engineering an application. We are interested in what *is* implemented, not what might be implementable. We thus characterize, as the features of the socket connection, nothing more or less than what we find when we reverse engineer the existing systemNow, what are the architectural implications of these transformations with respect to the achievement of the system’s quality attribute goals? In particular, these transformations will affect the performance and reliability of the system and should thus be understood as part of any design decision in a disciplined design process. This is one of the main reasons for doing architectural analysis when reengineering—we need to understand the implications of code-level mechanisms on the architecture and thus on the quality attributes of the resulting reengineered system.

The performance of the transformed system, with the new connector, is largely unchanged, except that clients can now do useful work while they are “blocked”. That is, servers will still serve requests as quickly as they can, and clients will still experience the same end-to-end latency, but this latency will, in the transformed system, not be experienced as blocking time, but rather as time during which other work can be accomplished. In this way the potential resource utilization of the system is increased.

The reliability of the transformed system is worse in that there is a new component added (the publish-subscribe “broker”, described in Section 3.3.2) which can be a new, independent source of failures. However, mitigating this added reliability risk is the fact that we have decoupled the client and the server and so can easily add additional redundant servers which will increase the reliability of the system as a whole.

The point to note here is that the choice of architectural mechanisms, as implied by the code transformations, have substantial consequences on system qualities that we want to engineer. We cannot simply treat code transformations as architecture-neutral. They can have profound implications for the quality of the system.

3.3. Code Transformation

Based on the mechanism identified as an appropriate response to the new architectural requirements, we must identify the corresponding realization of that mechanism in the implementation. Effecting this realization will involve a manipulation of the existing implementation to transform the old mechanism to the new mechanism.

In Section 3.2 we proposed that the current use of Unix sockets should be replaced by use of publish-subscribe. This is a change of an architectural connector; to effect the corresponding code transformation, we must identify:

- the primitives that are used to perform socket communication;
- the primitives that are used to perform publish-subscribe communication; and
- a mapping between these two sets of primitives.

This process would, in general, be supported by appropriate reference material accompanying the element feature tables. Because no such reference currently exists, the process is now manually performed.

Once a mapping between the current and new primitives has been identified, we perform an automated transformation of the source code.

3.3.1. Socket Communication Primitives

The use of sockets for client/server communication is well known to any Unix programmer; it is described extensively in the systems programming literature [14]. The generic sequence of steps taken by a server is:[1]

1. *create* a new socket
2. *bind* the socket to a well-known service indicator, specified using a *port number*
3. *listen* for connections from clients
4. *accept* a client connection; this operation blocks until a client attempts to make a connection
5. *send* and *receive* messages to/from the connected client

The typical sequence of steps taken by a client is:

1. *create* a new socket
2. *connect* to a server by hostname and well-known service indicator
3. *send* and *receive* messages to/from the server

Figure 2 summarizes the use of the socket communication primitives.

3.3.2. Publish-Subscribe Primitives

For the purposes of this paper, a small C library was implemented to support a simple publish-subscribe mechanism. This mechanism is used as follows. A server *subscribes* to a *channel* specified by a well-known service indicator; this is a non-blocking operation. The server then *waits* for messages from clients; this operation blocks until a message arrives. A client *publishes* messages on a channel by specifying a service indicator; this is a non-blocking operation.

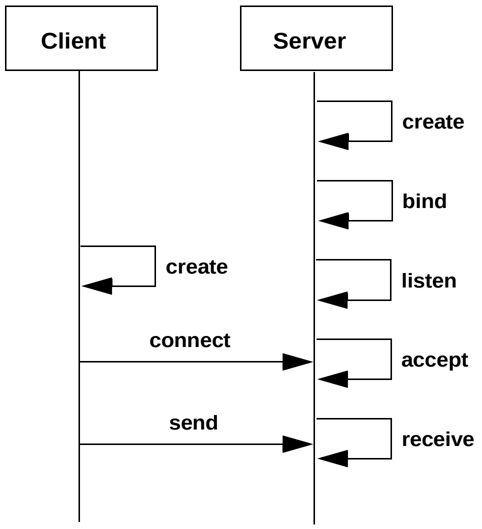

Figure 2: Socket Communication Primitives

The primitives are summarized in Table 6.

Table 6: Publish-Subscribe Primitives

Publish-Subscribe Primitive	Arguments	Return
`subscribe`	channel: integer (in)	0 on success < 0 on failure
`publish`	message: string (in) channel: integer (in)	0 on success < 0 on failure
`wait_msg`	message: string (out) max_size: integer (in)	number of bytes received

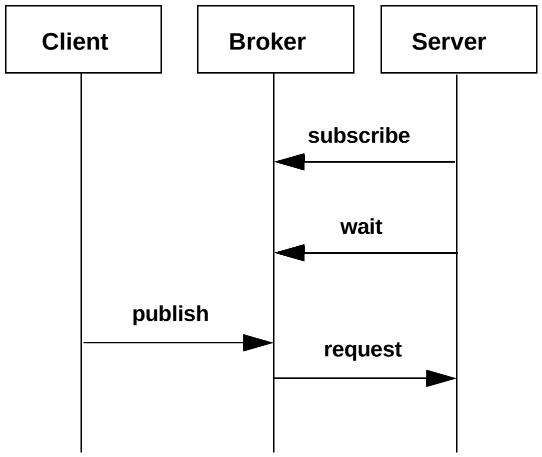

Figure 3: Publish-Subscribe Communication Primitives

1.Note that these steps describe the process for using *connection-oriented* sockets, also known as *stream* sockets.

The use of these primitives is summarized in Figure 3. Note the presence of an additional component, called “Broker”. This component is responsible for managing the channels by recording server subscriptions, accepting published data from clients and passing published data to subscribed servers. In our simple implementation, when data is published to a channel, the broker selects one server subscribed to that channel to which the published data should be passed. We recognize that this is unlike the “standard” operation of a publish-subscribe mechanism described in the literature, in which data is passed to all servers subscribed to a given channel. This difference is inconsequential to the concepts being discussed in this paper.

3.3.3. Primitive-to-Primitive Mapping

We now must establish how the current socket primitives should be mapped onto the new publish-subscribe primitives. Of course, there is no unique or universally correct choice as any mapping must be validated with respect to the particular source code context in which the mapping will be performed.

```
01  fdSocket = socket( AF_INET, SOCK_STREAM, 0 );
02  if( fdSocket < 0 ) {
03      perror( "socket failed" );
04      return( -1 );
05  }
...
06  sain.sin_port = htons( 8181 );
...
07  if( bind(
08          fdSocket,
09          (struct sockaddr *)&sain,
10          sizeof( sain ) ) < 0 ) {
11      perror( "bind failed" );
12      close( fdSocket );
13      return( -1 );
14  }

15  if( listen( fdSocket, 1 ) < 0 ) {
16      perror( "listen failed" );
17      close( fdSocket );
18      return( -1 );
19  }

20  for( ;; ) {
21      fdNew =
22          accept(
23                  fdSocket,
24                  (struct sockaddr *)&sainClient,
25                  &iAddr_len );
26      while( (iBytes =
27                      recv(
28                          fdNew,
29                          rgcMsg,
30                          256,
31                          0 )) > 0 ) {
32          rgcMsg[iBytes] = 0;
33          printf( "%s\n", rgcMsg );
34      }
35  }
```

Consider the server source code fragment above. Line 01 calls the socket creation primitive, `socket`; lines 02-05 perform error checking on the result of this operation. Line 06 configures part of the `sain` structure (other initialization has been elided) using a constant, 8181, as the port number. This represents the well-known service indicator. Lines 07-14 perform the `bind` operation, in effect registering the server against the service indicator on the particular host executing the server process. Lines 15-19 perform the `listen` operation, preparing the socket to receive connections from clients. The remainder of the program executes within an infinite for-loop; lines 21-25 accept new client connections and lines 26-33 receive a message from the currently connected client.

Socket communication is inherently bidirectional, allowing both clients and servers to send and receive data once a connection has been established. We have simplified this example substantially by having data flow only from client to server. This simplification makes it straightforward to establish that the server will be responsible for calling the `subscribe` and `wait_msg` primitives. In the more general case, other primitive-to-primitive mappings would be more appropriate.

Now, based on the server code fragment, we must determine which of socket, `bind`, `listen`, `accept` and `recv` should be mapped to each `subscribe` and `wait_msg`. Generally, the server need only subscribe to a channel once, so one of `socket`, `bind` or `listen` is the appropriate choice for a mapping to `subscribe`. Because `wait_msg` is the primitive used to receive data, `recv` is the obvious candidate for mapping to `wait_msg`. We will discuss the treatment of those primitives that we choose not to map below.

Let us briefly consider the implications of different choices for the mapping to the `subscribe` primitive in the server code. If we were to choose to map the `socket` primitive to `subscribe`, we would have the following:

```
fdSocket = subscribe(...);

if( fdSocket < 0 ) {
    perror( "socket failed" );
    return( -1 );
}
```

(We will discuss the arguments to `subscribe` below.) Note that the semantics of the `fdSocket` variable have been altered between the old code and the new code: previously `fdSocket`, as its name indicates, stored a handle to the socket; now `fdSocket` simply records the return code (indicating success or failure) of the call to `subscribe`. The alternative is to map one of `listen` or `bind` to `subscribe`; we show the situation for `bind`:

```
if( subscribe(...) < 0 ) {
    perror( "bind failed" );
    close( fdSocket );
    return( -1 );
}
```

(We will discuss what happens to the call to `close` below.) This case is much more consistent, as the semantics of the return from `subscribe` match the semantics of the return from `bind`. This is, of course, not a coincidence; the publish-subscribe primitives were developed with this task in mind, so we were careful to make their substitution straightforward. In general, replacement primitives must be chosen (or developed, perhaps by wrapping other implementations) to fit within the semantic context of the primitives they are replacing.

Now consider the fragment of client code below. Line 06 determines the address of a particular host, specified by the user as a command line argument. This is the host on which the server process is expected to be running. Lines 11-12 initialize the `sain` structure with the host name and port number. Lines 13-20 perform the server connection and lines 21-23 send data to the server.

```
01  fdSocket = socket( AF_INET, SOCK_STREAM, 0 );
02  if( fdSocket < 0 ) {
03      perror( "socket failed" );
04      return( -1 );
05  }

06  phe = gethostbyname( argv[1] );
07  if( phe == NULL ) {
08      perror( "gethostbyname failed" );
09      exit( -1 );
10  }
...
11  sain.sin_addr.s_addr = inet_addr( rgcAddr );
12  sain.sin_port = htons( 8181 );

13  if( connect(
14          fdSocket,
15          (struct sockaddr *)&sain,
16          sizeof( sain ) ) < 0 ) {
17      perror( "connect failed" );
18      close( fdSocket );
19      return( -1 );
20  }

21  send( fdSocket,
22           argv[2],
23           strlen( argv[2] ), 0 );
```

In the client case, we need only to map one of the socket primitives to the `publish` primitive.[1] `send` is obviously the appropriate choice.

1.In the general bidirectional communication case, we would also need to map primitives to `subscribe` and `listen_msg`.

3.3.4. The Transformation

Now that we have chosen the primitive-to-primitive mapping, we may effect the actual transformation of the source code. However, we have one more question to answer: where do we acquire the channel number to be used as the argument to the `subscribe` and `publish` primitives? The obvious choice is the port number used by the existing socket primitives, but where does this information come from? One alternative is to use the information collected during the architecture reconstruction activity; another is to transform the source code to continue using the port number as currently computed. We will opt for the former here.

Table 7 summarizes the selections for primitive mappings.

Table 7: Selected Primitive-to-Primitive Mappings

Socket Primitive	Publish-Subscribe Primitive
`socket`	-
`bind`	`subscribe`
`connect`	-
`listen`	-
`accept`	-
`send`	`publish`
`recv`	`wait_msg`

As shown in Figure 1, we apply the Reasoning SDK (formerly Refine/C) [12] to implement our code-level transformations. The Reasoning SDK provides an environment for language definition, parsing and syntax tree querying and transformation. Syntax tree queries and transformations are expressed in the Refine language. Figure 4 shows the Refine code used to transform the `bind` socket primitive into the `subscribe` publish-subscribe primitive.

Lines 02-11 define the precondition for the transformation, while lines 12-19 describe the postcondition. Of specific interest are:

- line 02: apply the transformation only to AST nodes that are of type `function-call`
- line 04: apply the transformation only for functions with surface syntax `bind`
- line 05: search upward in the AST to find a node of type `function-def`; this is the definition of the function containing the call to `bind`
- line 07: search upward in the AST to find a node of type `file`; this is the file containing the function performing the call
- line 09: consult the data structure `*bind-connect-map*` to identify the channel number for the particular file and function; `*bind-connect-map*` is a mapping between

```
01  rule transform-bind-rule(a)
02    cls::function-call(a)
03    & a ~in *seen*
04    & surface-syntax-seq(cls::function-called(a)) = ["bind"]
05    & parent in ancestors-of-class(a, 'cls::function-def)
06    & parent-func = symbol-to-string(cls::id-name(cls::function-identifier(parent)))
07    & parent-file in ancestors-of-class(a, 'cls::file)
08    & parent-file-name = cls::file-pathname(parent-file)
09    & channel = *bind-connect-map*(concat(parent-file-name,"//",parent-func))
10    & not (channel = undefined)
11    & args = cls::function-call-args(a)
12    -->
12    a in *seen*
13    & cls::function-called(a) = cls::make-identifier-ref(*replace-funcs-map*("subscribe"))
14    & cls::int-literal(arg)
15    & cls::int-value(arg) = channel
16    & cls::function-call-args(a) = [arg]
17    & surface-syntax(a) = undefined
```

Figure 4: Refine code for bind-to-subscribe transformation

(file, function) pairs and channel numbers populated by the information collected during the architecture reconstruction activity

- line 13 replaces the `function-called` attribute of the `function-call` node with a new identifier retrieved from `*replace-funcs-map*`; `*replace-funcs-map*` is a mapping between strings and references to `function-def` nodes
- lines 14-15 construct the argument to the function call; the argument is an integer literal containing the channel number
- line 16 replaces the `function-call-args` attribute of the `function-call` node with a sequence (delimited by '[', ']') containing the integer channel argument

Transformations similar to these were defined for each of the primitives to be transformed. Each maps the parameters passed to the socket primitive onto the appropriate parameters for the corresponding publish-subscribe primitive. After the transformations are applied, we have:

Server lines 7-10:
```
if(subscribe(8181) < 0) {
```

Server lines 26-31:
```
while((iBytes = wait_msg(rgcMsg, 256)) > 0){
```

Client lines 21-23:
```
publish(argv[2], 8181);
```

All other lines in the source code remain unchanged.

Unfortunately, our task is not complete, as there is a great deal of "dead code" remaining in the source code. An appropriate technique to apply in the the removal of this dead code is slicing. A backward slice maps a source code statement, *S*, onto all source code statements that might affect the value of variables used at *S*. A forward slice maps a statement onto all statements that use variables affected by the statement. [15] It is important to note that we must consider control flow in addition to data flow: our new primitives may be executed within control structures whose conditions are dependent on variables other than those used directly by the primitives. We may thus build the union of the forward and backward slices (including control flow) from our new primitives and remove all statements not included in the union. We have not explored this aspect of our transformation in depth, but we have experimented with CodeSurfer [2] as an environment to support dead code removal.

Server:

```
#include "publish_subscribe.h"

main( int argc, char **argv ) {
  if( subscribe ( 8181) < 0 ) {
    return( -1 );
  }

  for( ;; ) {
    while( (iBytes = listen_msg ( rgcMsg, 256) ) > 0
      rgcMsg[iBytes] = 0;
      printf( "%s\n", rgcMsg );
    }
  }
}
```

Client:

```
#include "publish_subscribe.h"

main( int argc, char **argv ) {
  publish ( argv[2], 8181) ;
}
```

Above we see the final result of our transformation, with dead code removed (manually, in this case, guided by CodeSurfer). Of course, there are additional issues that we have not discussed here:

- there is an additional `#include` statement in both the client and server to provide visibility of the publish-subscribe primitives
- the first command-line argument to the client (previ-

ously the host name; the second argument is the message to send to the server) is no longer used

As mentioned above, it is necessary to validate the semantic correctness of the transformations performed. In this simple example, we performed this validation by hand. As this approach matures, other techniques, such as regression testing, should be explored.

4. Conclusions

Now that the example has been shown in detail, it is instructive to take a step back and think about what this paper is trying to argue for. We have taken a common example of a legacy application and a common set of reengineering goals (decouple client and server knowledge of each other to aid in the future modifiability of the system, increase performance, and make it easy to increase the reliability) and we have shown how we can reason about each of these goals at the architectural level. We do this by understanding and focussing on the architectural features that we need to manipulate. We can then instantiate the architectural changes via automated code transformation.

This paper is a reaction to the relatively undisciplined way in which reengineering is done. In particular, some efforts at reengineering are code-based, and some are aimed at re-architecting a system. But few efforts try to do both, and we are aware of little work that explicitly attempts to tie these two approaches together. But tying the approaches together is crucial for ensuring: the conceptual integrity of the architecture being reengineered, the quality attributes of the resulting system, the preservation of component level qualities.

In short, what we are presenting here is a disciplined process for reengineering a system that takes into account all levels of representation and reasoning about software. However, it must be stressed that tool support for this type of reasoning is only just beginning to reach maturity. The approach presented here is far from a fully automated solution.

5. Future Work

There are several directions in which the work presented in this paper should be extended. The most obvious of these addresses the issue of scale: we would like to explore the applicability of the techniques presented here in larger, real-world software systems. Also, we hope to validate the use of semantic features for driving architectural transformations; evaluation of the adequacy of the feature set as applied to our example will guide evolution of the features. Finally, we are interested in initiating a process of cataloging additional architectural elements and their features; this will lead to the development of a catalog of architectural transformations and their potential mappings to code transformations.

6. REFERENCES

[1] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1997.

[2] GrammaTech Inc., “CodeSurfer”, http://www.grammatech.com/products/codesurfer/codesurfer_index.html.

[3] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H.A. Mueller, J. Mylopoulos, S.G. Perelgut, M. Stanley, and K. Wong, “The Software Bookshelf”, *IBM Systems Journal*, 36(4), 1997.

[4] D. Jerding and S. Rugaber, “Using Visualization for Architecture Localization and Extraction”, *Proceedings of WCRE’97*, (Amsterdam, The Netherlands), October 1997, pp. 56-65.

[5] R. Kazman, S. J. Carrière, “View Extraction and View Fusion in Architectural Understanding”, *Fifth International Conference on Software Reuse*, (Victoria, B.C.), June 1998, pp. 290-299.

[6] R. Kazman, S. J. Carrière, “Playing Detective: Reconstructing Software Architecture from Available Evidence”, *Journal of Automated Software Engineering*, 6:2, pp. 107-138 (April 1999).

[7] R. Kazman, S. Woods, S. J. Carrière, “Requirements for Integrating Software Architecture and Reengineering Models: CORUM II”, *Proceedings of WCRE’98* (Honolulu, HI), October 1998, pp. 154-163.

[8] R. Kazman, P. Clements, L. Bass, G. Abowd, “Classifying Architectural Elements as a Foundation for Mechanism Matching”, *Proceedings of COMPSAC 1997*, (Washington, D.C.), August 1997, pp. 14-17.

[9] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and S.J. Carrière. The Architecture Tradeoff Analysis Method, *Proceedings of ICECCS’98*, 1998.

[10] R. Krikhaar, *Software Architecture Reconstruction*, Ph.D Thesis, University of Amsterdam, Amsterdam, The Netherlands.

[11] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, C. Verhoef, “A Two-phase Process for Software Architecture Improvement”, *Proceedings of ICSM’99*, (Oxford, UK), September 1999.

[12] Reasoning Inc., http://www.reasoning.com.

[13] A. Sellink, C. Verhoef, “An Architecture for Automated Software Maintenance”, *Proceedings of IWPC’99*, (Pittsburgh, PA), May 1998.

[14] R. Stevens, *Unix Network Programming*, Prentice-Hall, 1990.

[15] M. Weiser, “Program Slicing”, *IEEE Transactions on Software Engineering,* SE-10(4), pp. 352-357 (July 1984).

[16] S. Woods, S. J. Carrière, R. Kazman, “A Semantic Foundation for Architectural Reengineering and Interchange”, *Proceedings of ICSM’99*, to appear.

[17] A. Yeh, D. Harris, M. Chase, “Manipulating Recovered Software Architecture Views”, *Proceedings of ICSE 19*, (Boston, MA), May 1997, pp. 184-194.