

An Architecture-Based Approach for Synthesizing and Integrating Adapters for Legacy Software*

Gerald C. Gannod[†], Sudhakiran V. Mudiam, and Timothy E. Lindquist

Department of Computer Science & Engineering

Arizona State University

Box 875406

Tempe, AZ 85287-5406

E-mail: {gannod, kiranmvs, tim}@asu.edu

Abstract

In software organizations there is a very real possibility that a commitment to existing assets will require migration of legacy software towards new environments that use modern technology. One technique that has been suggested for facilitating the migration of existing legacy assets to new platforms is via the use of the adapter design pattern, also known as component wrapping. In this paper, we describe an approach for facilitating the integration of legacy software into new applications using component wrapping. That is, we demonstrate the use of a software architecture description language as a means for specifying various properties that can be used to assist in the construction of wrappers. In addition, we show how these wrapped components can be used within a distributed object infrastructure as services that are dynamically integrated at run-time.

1 Introduction

A *software product line* is a collection of systems that share a managed set of properties that are derived from a common set of software assets [1]. A product line approach to software development is attractive to most organizations due to the focus on reuse of both intellectual effort and existing tangible artifacts. The systems or “derivatives” in a software product line (e.g., software based on the product line) usually share a common architecture. For a product line that leverages existing systems, an architecture may already be in place with organizational commitment to its continued use. In many cases, there is a very real possibility that the commitment to existing assets will

require migration of legacy software towards new environments that use modern technology. One technique that has been suggested for facilitating the migration of existing legacy assets to new platforms is via the use of the adapter design pattern [2], also known as *component wrapping*.

The ever increasing use of network computing has made the ability to utilize components running in heterogeneous environments paramount. Technologies such as CORBA [3] and DCOM [4] have made distributed computing realizable. However, the interaction between components in each of these schemes relies upon a centralized control broker. The Jini Connection Technology [5], in contrast, provides a mechanism whereby several components, or *services*, can be connected in a way that removes the need for centralized control, thereby expanding the ability to create *federations* of cooperating components.

In this paper, we describe an approach for facilitating the integration of legacy software into new applications using component wrapping. That is, we demonstrate the use of a software architecture description language as a means for specifying various properties that can be used to assist in the construction of wrappers. In addition, we show how these wrapped components can be used within a distributed object infrastructure as services that are dynamically integrated at run-time.

The remainder of this paper is organized as follows. Section 2 describes background material in the areas of reengineering, software architecture, and distributed components. Our proposed approach for facilitating legacy software wrapping as well as the mechanism for integrating the wrapped components into client applications is presented in Section 3. Section 4 discusses the details of an automated support tool that generates wrappers for legacy code, and Section 5 presents an example that demonstrates the approach. Related work is described in Section 6, and

*This research supported in part by NASA Langley Research Grant NAG 1-2241.

[†]Contact author.

Section 7 draws conclusions and suggests further investigations.

2 Background

This section describes background material in the areas of reengineering, software architecture, and Component-Based Software Engineering.

2.1 Reengineering via Adapters

Reverse engineering is defined as the analysis of software components and their interrelationships in order to obtain a description of the software at a high level of abstraction [6]. This term is contrasted with reengineering, which is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [6]. Since the functionality of the existing software has been achieved over a period of time, it must be preserved for many reasons, including providing continuity to current users of the software.

One approach to reengineering is to use the *adapter pattern* [2] whereby a legacy interface is converted into a form that a client application can utilize. As such, the adapter pattern allows components that otherwise could not work together because of incompatible interfaces to be combined to form a new software system. In this paper, we describe an adapter approach for automating the reengineering of legacy command-line software. Specifically, in terms of the Gamma et al. adapter pattern, we use the concept of the object adapter in the manner shown in Figure 1.

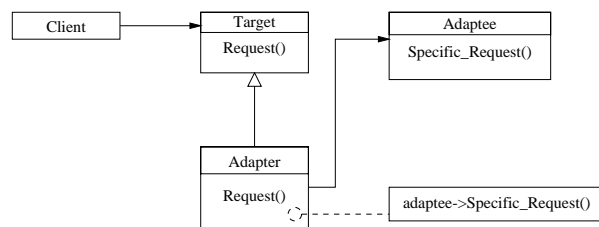


Figure 1. Object Adapter [2]

2.2 Software Architecture

A *software architecture* describes the overall organization of a software system in terms of its constituent elements, including computational units and their interrelationships [7]. In general, an architecture is defined as a *configuration of components and connectors*. A component is an encapsulation of a computational unit and has an interface that specifies the capabilities that the component can provide. In addition, the interface specifies the ways that the component delivers its capabilities. The interface of a component is specified by the *type* of the component, by one or more *ports* supported by the component,

and by the *constraints* imposed on the ports of the component, where component types are intended to capture architectural properties. Ports are the interaction points through which a component exchanges resources with its environment. Port specifications specify the signatures, and optionally, the behaviors of the resource.

Connectors encapsulate the ways that components interact. A connector is specified by the *type* of the connector, the *roles* defined by the connector type, and the *constraints* imposed on the roles of the connector. A connector defines a set of roles for the participants of the interaction specified by the connector. Connector types are intended to capture recurring component interaction styles.

Components are connected by configuring their ports to the roles of connectors. Each role has a domain that defines a set of port types and only the ports whose types are in the domain can be configured to the role.

Another important concept in the area of software architectures is the concept of an *architectural style*. An architectural style defines patterns and semantic constraints on a configuration of components and connectors. As such, a style can define a set or family of systems that share common architectural semantics [8]. For instance, a *pipe and filter* style refers to a pipelined set of components whereas a *layered* style refers to a set of components that communicate via hierarchies of interfaces.

2.3 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is a form of software development that is based on the construction of applications by integrating existing software components. CBSE is much more than use of object request brokers, reuse of a library of code, or modular development. It can involve activities such as building, acquiring, assembling, and evolving systems. Emerging concerns include the use of multiple suppliers of components that provide the same functionality, along with multiple versions and configurations of the components.

Currently, CBSE addresses issues and technologies such as Commercial-Off-The-Shelf (COTS) components, in-built components, and application frameworks. The emerging technologies for component integration include Enterprise Java Beans [9], CORBA [3], Jini [5], Microsoft DNA [10], and IBM San Francisco [11]. All of these technologies provide a component model where a pre-defined infrastructure acts as "plumbing" that facilitates communication between components. Tools and environments supporting all these technologies are being widely used in the industry and continue to provide many benefits.

3 Approach

In this paper we discuss two aspects of a software migration strategy. First, we introduce an architecture-

based approach for specifying and subsequently synthesizing wrappers for a certain class of legacy software. Second, we discuss the issue of integrating the wrapped components with client applications by utilizing the service mediation capabilities of Jini [5].

3.1 Specification and Synthesis

The concept of using an adapter for wrapping legacy software is not a new one [2, 12, 13, 14]. As a migration strategy, component wrapping has many benefits in terms of reengineering including a reduction in the amount of new code that must be created and a reduction in the amount of existing code that must be rewritten. One of our primary goals in this research is to develop an environment whereby a potentially large number of existing legacy software components can be adapted for use within a software development and integration framework. In particular, unlike CBSE strategies that assume the existence of reusable components, we are interested in developing an approach that facilitates the automated wrapping of existing legacy software into forms that can be easily integrated into a target application.

In regards to wrapping components, our approach uses two steps. First, a specification of the legacy software as an architectural component is created. These specifications provide vital information that is required to define the interface to the legacy software. Second, the appropriate adapter source code is synthesized based on the specification.

3.1.1 Specification Requirements

To aid in the development of an appropriate scheme for the wrapping activity, we defined the following requirements upon specifications of the interface to legacy software.

- (R1) A sufficient amount of information should be captured in the interface specification in order to minimize the amount of source code that must be manually constructed.
- (R2) A specification of the interface of the adapted component should be as loosely coupled as possible from the target implementation language.
- (R3) The specification of the adapted component should be usable within a more general architectural context.

The requirement R1 addresses the fact that we are interested in gaining a benefit from reusing legacy software. As a consequence, we must avoid modifying the source code of the legacy software. At the same time, we must provide an interface that is sufficient for use by a target application. To provide that interface, a sufficient amount of

information is needed in order to automatically construct the adapter.

As an initial investigation into the automated synthesis of the adapters, we selected command-line applications as our source of reusable legacy software. Selection of this class of legacy applications addresses the modification concern of requirement R1 since source code is not available. As such, we are required to provide an interface that is based solely on the knowledge of how the application is used rather than how it works. In addition, the selection of this class of applications has the consequence of enforcing the use of a particular architectural style, as determined by the nature of the legacy application. In this context, we defined several properties that would be needed to appropriately specify the behavior provided by a command-line application including: *Signature*, *Command*, *Pre*, *Post*, and *Path*. We identified these properties by examining the type of behaviors, inputs, and outputs generally associated with command-line applications.

The *Command* property identifies the command used to invoke the application while the *Pre* and *Post* properties identify commands that are contained within the adapter code that will establish preconditions and postconditions on the execution of the legacy component, respectively. *Path* indicates the path to the given command-line application and the *Signature* property defines the types and names of the expected input and output of that application. Here, since we are dealing with command-line applications, the input types are expected to be strings. Accordingly, a certain degree of semantic information must be used in the name of the input. Fortunately, for command-line applications, these names are typically limited to filenames and command-line options.

The requirement R2 (i.e., the decoupling of a specification from a target implementation language) is based on the desire to apply the synthesis approach to a variety of target languages and implementations. In addition, this requirement facilitates enforcement of requirement R1 by ensuring that new source code is not artificially embedded in the specification. While satisfying this requirement is ideal, we found in our strategy that a certain amount of implementation dependence was necessary due to the integration strategy that we chose (see Section 3.2).

When a component has been wrapped using our technique, an interface is defined that facilitates the use of the source legacy software as part of a new application. However, as indicated by requirement R3, it is also desirable to be able to use the specification of the adapted component within a more general architectural context. That is, it is advantageous to be able to use the specification as part of the software architecture specification for new systems. In using a content-rich specification, where interfaces are defined explicitly, the added benefit of providing information

that can be integrated into an architectural specification of a target application is gained.

In order to realize the requirements placed upon desired interface specifications for legacy software wrappers, we used the ACME [15] Architecture Description Language (ADL). Specifically, we used the *properties* section of the ACME ADL to specify the interface features described earlier (e.g. Signature, Command, Pre, Post, and Path). ACME is an ADL that has been used for high-level architectural specification and interchange [15]. ACME contains constructs for embedding specifications written in a wide variety of existing ADLs, making it extensible to both existing and future specification languages. ACME is supported by an architectural specification tool, ACMEStudio [16], that facilitates graphical construction and manipulation of software architectures.

Figure 2 shows a screen capture of an ACMEStudio session in which a component has been specified as an aggregation of two wrapped applications expressed as ports. In the figure, the component labeled RCS is the aggregate component and consists of two ports, ChkIn and ChkOut. These ports are used as wrappers for the Revision Control System (RCS) programs `ci` and `co`, respectively. The bottom right portion of the figure contains a list of the properties used to derive the wrapper along with their corresponding values. The amount of knowledge that is required in order to write the wrapper specification is limited to a working knowledge of how a legacy application is used. Often this includes knowledge of the command-line parameters as well as other bits of information that can be retrieved from manual pages (if they exist) and current users.

Several benefits arise from the use of the ACME ADL to specify the legacy application wrappers. First, if the specification of an adapted component is realized as a port, then several such components can be aggregated into a single component. As a consequence, the aggregated component can offer each of the behaviors as a service through a port interface. For example, as demonstrated in Section 5, the RCS suite of applications can be aggregated into a single component that offers services such as check-in and check-out via ports of the aggregate component. Second, via the use of ACMEStudio, the specification of an adapted component can be integrated into the specification of target applications. Consequently, software architecture analysis techniques that support the use of ACME can be applied to target applications that utilize the wrapped legacy software. Finally, since library support for ACME consists of a set of standard parsers and other manipulation facilities, construction of support tools is convenient (see Section 4).

3.1.2 Synthesis

As stated earlier, the class of legacy systems that we are considering are command-line applications. Given this

constraint, we make the assumption that any client applications utilizing the wrapped components have a certain amount of knowledge regarding the interface of that wrapped component. We find this assumption to be reasonable due to the nature of legacy software migration where legacy applications have an organizational history with well-known usage profiles.

We chose the Java programming language and environment as our target migration platform for a number of reasons. First, Java continues to enjoy increases in popularity, and thus any opportunity to integrate legacy systems into Java applications has obvious benefits. Second, the object-oriented nature of Java facilitates straightforward construction of components consisting of an aggregation, or federation, of wrapped legacy components. Finally, and most importantly, we found that the services provided by the Jini Connection Technology [5] for smart devices could also be applied to software components. As a result, software can be packaged as services on a Jini network and integrated dynamically into distributed applications. Accordingly, our approach for synthesizing wrappers for legacy components is based on implementing the standard discover and join protocol that is required for Jini devices.

In our approach, the information that is needed to generate wrappers corresponds exactly to the properties associated with the ports shown in Figure 3 and previously described in Section 3.1, with some minor modifications. In addition to the Signature, Command (shown as `Cmd` in Figure 3), Pre, Post, and Path properties, we added the `Interface`, and `Return` fields. These fields define the Jini service name of a port and the value returned after interaction with the port, respectively. In the synthesis process, ACME specifications are combined with a standard template that implements the setup routines that are required to register a component on a Jini network. In addition to synthesizing the appropriate wrapper, the support tool that we have constructed to automate this process generates the appropriate source code for facilitating interaction between a potential client and the wrapped component. At present, this is an automated tool that generates fully executable code for the wrapped application and does not require the user to modify or write any new code.

While the investigations that are described here are limited to our efforts to adapt command-line applications for use within a Jini-based software integration environment, we are pursuing investigations into broadening the context of this approach to other legacy software components including GUI-based applications. In addition, due to the ability of Java and consequently Jini to run anywhere, we are investigating approaches for automatically wrapping applications that exist within heterogeneous operating system environments.

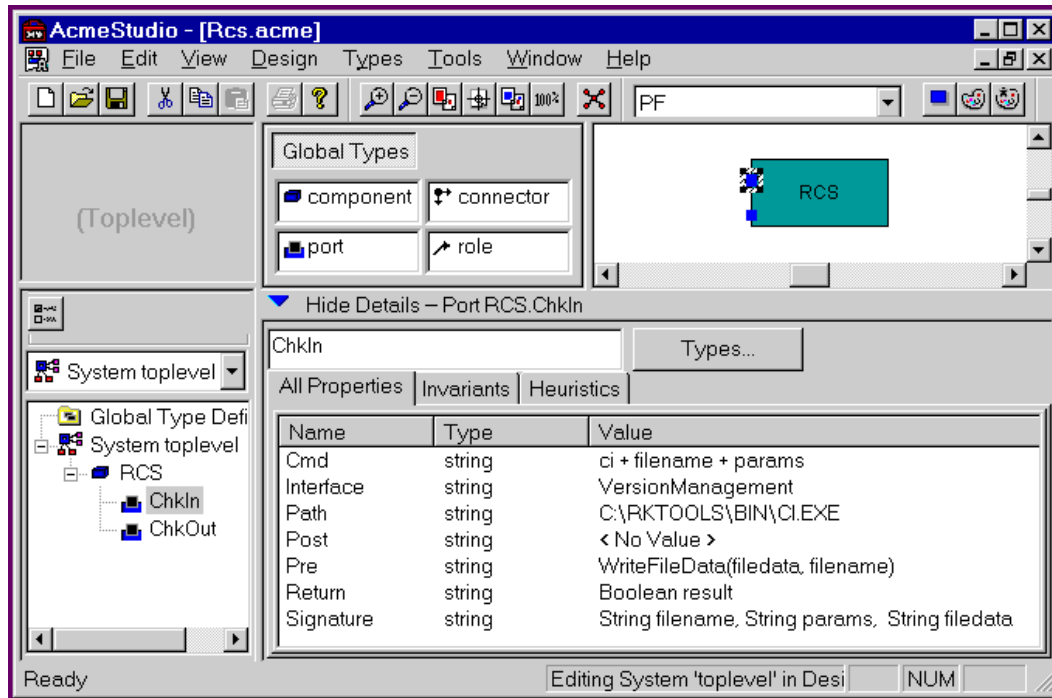


Figure 2. ACMEStudio Session

3.2 Integration

The primary enabling feature of the work described in this paper is the existence of the Jini technology for the delivery and management of services. In a typical Jini network, services are provided by devices that are connected to the network. Typically these devices consist of a variety of products ranging from cell phones, desktop devices, printers, fax machines, and Personal Digital Assistants (PDAs).

Figure 4 shows the layered architecture of the Jini Connection Technology where the Jini Technology layer provides distributed system services for activities such as discovery, lookup, remote event management, transaction management, service registration, and service leasing. When a “device” is plugged into a Jini network, it becomes registered as a member (e.g., service) of the network by the Jini lookup service. Once registered, other network members can discover the availability of the device through the lookup service. When a client application finds an appropriate device, the lookup service facilitates the connection but then is no longer involved in subsequent interactions between the client and device. In our approach to component integration, we use the Jini technology to provide a standard method for registering and connecting a client to corresponding software components that are acting as services.

One of the advantages of using this Jini-based integra-

tion technique is that it facilitates construction of applications “on-the-fly” whereby components can be used on an as-needed basis. Another advantage arises from the fact that services must implement a fairly general set of routines in order to participate in a Jini network. As a result, the synthesis of wrappers is fairly straightforward. One of the disadvantages of this approach stems from the fact that the interface to the services must be well-defined or must be negotiated between the client and the device upon connection. That is, the client must have some knowledge about how to use the service. As stated earlier, this assumption is mitigated by the fact that the class of software applications that we are wrapping typically have a history within an organization.

Once the connection between a client application and a service has been established, the interaction between the components will appear to be similar to the architecture shown in Figure 5. In this figure, the Service component provides two services to the Client_Application component via the interface provided by the Adapter connector. While the diagram in Figure 5 shows only one service being utilized by the client application, it is possible for several services to be connected to the client. In addition, the interaction between the services is not limited to a call-return style where services return values to the client but can also include interaction between services using a wide variety of architectural styles.

```

Component RCS = {
  Properties {
  };
  Port ChkIn = {
    Properties {
      Signature : string =
        "String filename, \
         String params, \
         String filedata ";
      Return : string = "Boolean result";
      Cmd : string = "ci + filename + params";
      Pre : string =
        "WriteFileData(filedata, filename)";
      Post : string = "";
      Interface : string = "VersionManagement";
      Path : string = "C:\\RKTOOLS\\BIN\\CI.EXE ";
    };
  };
  Port ChkOut = {
    Properties {
      Signature : string =
        "String filename, String params";
      Pre : string = "";
      Return : string = "String filedata";
      Cmd : string = "co + filename + params";
      Interface : string = "VersionManagement";
      Post : string =
        "filedata = ReadFileData( filename )";
      Path : string = "C:\\RKTOOLS\\BIN\\CO.EXE ";
    };
  };
};

```

Figure 3. ACME Specification of RCS Services as ports

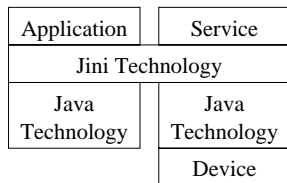


Figure 4. Jini Architecture

4 Implementation

To support our technique for constructing wrappers for legacy software, we have created a Java support tool called *ServiceTool*. Figure 6 shows the detailed architecture of *ServiceTool* which takes an ACME specification and produces a wrapper configured for a Jini network. In the diagram, the rectangles with the square corners represent software components while the rectangles with the rounded corners represent files. The *ArchParser* component reads in an ACME specification similar to the one shown in Figure 3 and builds an internal model of the architecture. The *Component Inspector* component uses the output of the *ArchParser* to access the interface specification of the wrapper component and produces a set of ports. The *Inter-*

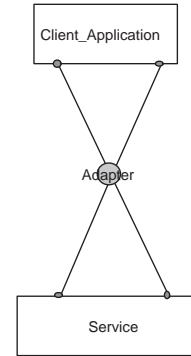


Figure 5. Conceptual Architecture

face Generator component uses the set of ports to generate the interface or connector to the service. The *Function Generator* component uses the same port information to generate functions that implement the service. The *Service Generator* component uses these functions along with the *ServiceTemplate* to generate the final Java source code for the service.

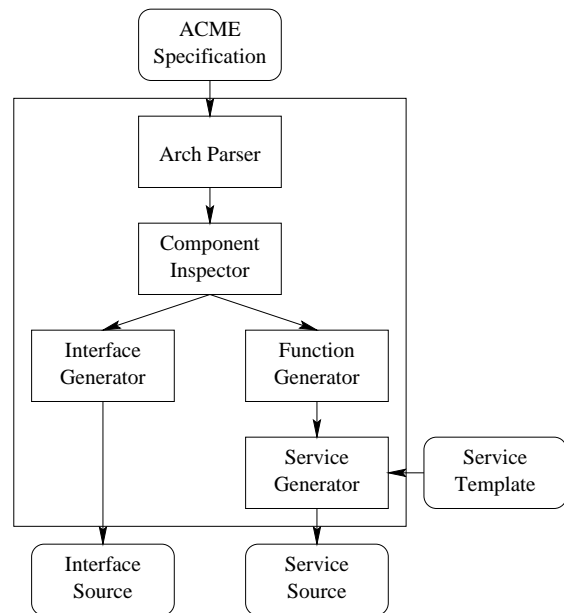


Figure 6. Service Tool Architecture

The *ArchParser* uses the *ACMEParser* from the *ACMELib* toolkit [17] to parse ACME specifications. *ACMELib* is a library that facilitates the construction of architectural tools in Java that read, write and manipulate software architectures specified in the ACME ADL. The *ACMELib* framework is designed to support the rapid development of two classes of applications (1) tools that

translate between "native" ADLs (such as Rapide [18] and Wright [19]) and (2) native ACME-based architectural design and analysis tools.

The *Service Generator* component is implemented as an awk script that replaces tags in the *ServiceTemplate* file with functions generated by the *Function Generator* component and the names of services.

Figure 7 contains a portion of the *ServiceTemplate* file which contains all of the application and service independent source code and provides the routines necessary to integrate the legacy code into a Jini network. Specifically, the *ServiceTemplate* contains functions that implement the discover and join protocol for registering a service with the lookup service. The *ServiceTemplate* also contains tags that are place-holders for the automatically generated functions. For instance, in Figure 7 the tag `<put-ServerName>` is a place-holder for the final name of the adapter component.

```
public class <put-ServerName>
  extends UnicastRemoteObject
  implements
    <put-InterfaceName>,
    ServiceIDListener,
    Serializable
{
  public <put-ServerName> () throws RemoteException
  {
    super ();
  }
  ...
  <put-Functions>
  ...
}
```

Figure 7. Excerpt of the ServiceTemplate

In addition to the *ServiceTemplate*, there is also a reusable set of functions that can be utilized in an interface specification and consequently in the generated wrappers. For instance, the `ReadFileData()` and `WriteFileData()` routines (shown in Figure 8), are available as functions for use within the Java code to provide standard read-from and write-to file support, respectively.

5 Example

To demonstrate the use of the component wrapping and integration technique described in this paper, we developed a simple editing application. The intent of this application is to support text file editing with version control. For the version control portion of the application, a version management service was utilized while the remainder of the application consists of source code written specifically for the editor.

```
void WriteFileData(String filedata, String filename) {
  // the generic WriteFileData..

  try{
    File f = new File(filename);

    System.out.println("IsFile(): " + f.isFile());

    PrintWriter out =
      new PrintWriter( new FileOutputStream-
        Stream(f), true);
    out.print(filedata);
    out.close();
  }
  catch (Exception e)
  {
    System.out.println
      ("Server: Error writing file: " + e);
  }
}

String ReadFileData(String filename){
  try{
    BufferedReader in =
      new BufferedReader(new FileReader(filename));

    String s;
    StringBuffer sb = new StringBuffer ();

    while( (s =in.readLine()) != null )
    {
      sb.append (s);
      sb.append ("\n");
    }
    in.close();
    // returning the whole file as a string..
    return (sb.toString());
  }
  catch (Exception e)
  {
    System.out.println
      ("Server: Error writing file: " + e);
    return("Server: Error writing file: "+ e);
  }
}
```

Figure 8. Sample Library Routines

5.1 Specification

In order to provide version management services, we constructed an adapter specification using the ACMEStudio tool, as shown in Figure 2. The final result of the ACMEStudio session yielded the ACME specification shown in Figure 3, where the wrapped components are specified as ports that offer interfaces to the `ci` (check in) and `co` (check out) RCS programs. The properties for `ChkIn` state that the port has an input signature consisting of three Strings: `filename`, `params`, and `filedata`. This signature corresponds to the formal parameter list that will be used as a call to a `ChkIn` method. Similarly, the `Return` property within the ACME specification states that for the `ChkIn` port, the returned output will be a Boolean variable called `result`.

One of the important properties contained within the ACME specification is the `Interface` property which facilitates the registration of the wrapped component as a

service on a Jini network. In this case, the interface is specified as `VersionManagement`.

Two other fields of interest are the `Cmd` and `Path` which specify how the adapter interface will invoke the wrapped component and the path to the actual command-line application, respectively.

The last part of the `ChkIn` port specification is the `Pre` field, which identifies the command that is needed to setup the interaction between the client and service component. In this example, the `Pre` field states that the routine `WriteFileData` should be invoked with parameters `filedata` and `filename`. This routine will write to the file named `filename` the data contained in the `String filedata`.

A similar specification of the `ChkOut` port is also given in Figure 3 and differs primarily in the post-processing property. Once the ACME specification is completed, the `ServiceTool` application can be used to generate a pair of files named `RCSServer.java` and `VersionManagement.java`. These files correspond to the component that implements the wrapping of the legacy software and the connector that provides an interface to the wrapper.

5.2 Client Application

Client applications in the Jini framework are implemented as Jini services. In the case of our example, the Editor component requires the use of a Version Management Service. In the source code for the Editor component, the appropriate functions must be written to implement both the graphical user interface for the application as well as the protocols for discovering and joining a Jini network. Client applications can also provide services to others components, if they have any to offer. In our example, however, the Editor does not provide any other services and just makes use of a Version Management service.

5.3 Integration and Execution

In our example scenario, the Editor component joins the Jini network by discovering the lookup service and registering with lookup. As shown in Figure 9, the Editor uses the lookup service to find a version management service that is registered on the Jini network. Plausible version management services that could be registered include RCS, SCCS and CVS components.

Once the Editor finds that an `RCSServer` service (or other version management service) is available, it requests that the lookup service return the proxy object that was registered by the `RCSServer`. The proxy object acts as a connector to the `VersionManagement` service, e.g., an interface that knows how to interact with the `RCSServer` and acts as an intermediary between the client application and `RCSServer` components. At this point, as shown in Figure 10, the `RCSServerApplet` becomes an integral part of the Editor. In this example, the Editor provides the user

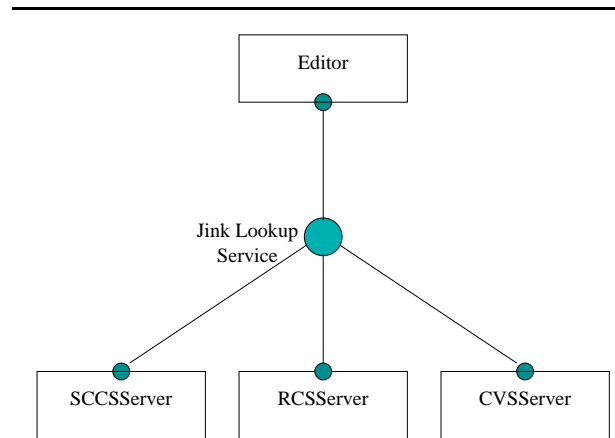


Figure 9. Editor and RCSServer Interaction

interface required to edit files. It also contains an editing panel that allows the user of the Editor to access the CheckIn and CheckOut Functions available from the `RCSServer` component. Figure 11 shows the resulting editor applet.

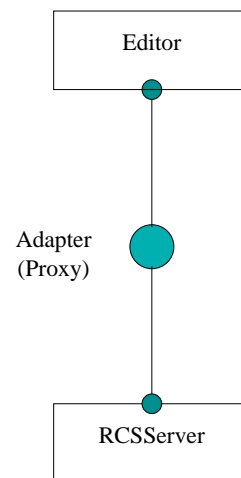


Figure 10. Editor and RCSServer Interaction

The power of Jini comes from the fact that proxy objects contain code that become an integral part of a client application. Another key to Jini Technology is also its spontaneous networking that allows clients and services to become aware of one another and integrate seamlessly.

Getting the Editor to talk to `RCSServer` through the `RCSServer` proxy adapter requires that a number of Jini services exist (e.g., are running) prior to integration. First, the Jini network must be up and running along with an associated lookup service. Since Jini is built on top of the Java RMI, an RMI daemon must be running prior to execut-

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class RCSServerApplet extends Applet
    implements ActionListener, Entry
{
    Button b1; // 1 passed to callServer..
    Button b2; // 2 passed to callServer..
    Button b3; // 3 passed to callServer..
    Button b4; // 4 passed to callServer..
    Button b5; // for clearing text area.. to create

    Label l1;
    Label l2;
    TextArea ta;
    String s1, s2;
    Panel panel1, panel2;

    public void actionPerformed (ActionEvent evt) // A
    {
```

Figure 11. Editor Session

ing the Jini Lookup service. Finally, a web server must be running in order to enable code delivery across the Jini network.

The following is a summary of the order of execution for the aforementioned services along with the example application components:

1. Web server (httpd)
2. An RMI daemon (rmid)
3. The Jini Lookup service (reggie)
4. RCSServer
5. Editor

The Editor can also be up and running on the network before any of the Version Management services become available. In that case, the Editor can request that the Lookup service notify it once any service that matches its required specification becomes available. Once a matching service registers with the Lookup service, the Editor is notified and subsequently connects with the service.

6 Related Work

Several approaches for component wrapping have been suggested in the reengineering literature. From the practical experience perspective, Sneed [12] described efforts to encapsulate legacy systems at several different levels. These levels, corresponding to a COBOL environment, include encapsulation at the *job*, *transaction*, *program*, *module*, and *procedure* level. In addition, Sneed demonstrated the use of wrapping techniques to wrap running assembler

programs in order to integrate them into more current systems via a CORBA interface.

Cimitile et al. [13] describe an approach to wrapping that involves the use of data flow analysis in order to discover various properties of source code. The approach focuses on the static analysis of code in order to determine appropriate decompositions of different program components as well as discovery of the formal parameters for the interfaces to these programs. At this time our approach does not involve analysis of source code but rather takes the perspective of a continuous engineering effort where reuse of existing host applications with minimal code rewriting is desired [20].

Jacobsen and Kramer [14] describe an approach for synthesizing wrappers based on the specification of a modified CORBA IDL description. In their approach, they address the problem of object synchronization within the context of the CORBA standard and define a technique based on the application of the adapter design pattern. In many ways, Jacobsen and Kramer's approach is similar to the one described in this paper. We are currently interested in software at a higher level of granularity than the ones often provided via ORB-based interfaces, thus our approach has some potential for full automation. However, as we increase our scope to include a more general class of components, we must address similar concerns.

Sullivan et al. look at systematic reuse of large-scale software components via static component integration [21]. That is, they use an OLE-based approach for component integration. To demonstrate the use of their scheme, they developed a safety analysis tool that integrates application components such as Visio and Word. In our approach we use a dynamic approach for component integration and thus, can utilize a wide variety of components whose interfaces are discovered at run-time.

CyberDesk [22] is a component-based framework written in Java that supports automatic integration of desktop and network services. This framework is flexible and can be easily customized and extended. The components in this framework treat all data uniformly regardless of whether the data came from a locally running service or the World Wide Web. The goal of this framework is to provide ubiquitous access to services. This approach is similar to our proposed approach in that they use a dynamic mapping facility to support run-time discovery of interfaces.

7 Conclusions and Future Investigations

In the context of software product lines, where the reuse of existing assets is desired, reengineering plays an important role in the realization of the recovery of those assets from previously existing products. The ability to quickly adapt those existing assets for use in new products becomes paramount, especially in an industry dominated by time-to-

market issues.

One of the primary issues in reuse or component-based software engineering techniques is the notion of architectural mismatches between integrated components. The approach described in this paper makes the assumption that interactions between clients and services registered on the Jini network interact in a well-defined manner. Currently, we are investigating techniques for embedding high-level interaction semantics (e.g., architectural style) within the ACME specifications of both wrapped components and target client application code.

Our future investigations will focus upon the development of a framework for supporting the construction and integration of software, where the reusable components are utilized as services in a dynamic integration environment. In addition, we are investigating techniques for the construction of the interface ACME specifications via reverse engineering whereby the properties needed to specify a component wrapper are automatically derived from source code.

References

- [1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, 1995.
- [3] Object Management Group. *CORBA: Architecture and Specification V2.0*, formal/97-02-25 edition, July 1996.
- [4] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.
- [5] W. Keith Richards. *Core Jini*. Prentice-Hall, 1999.
- [6] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [7] M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [8] N. Medvidovic and R. N. Taylor. Exploiting architectural style to develop a family of applications. *IEE Proc. in Software Engineering*, 144(5-6):237–248, Oct-Dec 1997.
- [9] Anne Thomas. Enterprise java beans technology. Technical report, Patricia Seybold Group, 1998.
- [10] Mary Kirtland. *Designing Component-Based Applications: Build Enterprise Solutions with Microsoft Windows DNA*. Microsoft Press, 1999.
- [11] Paul Monday, James Carey, and Mary Dangler. *San Francisco Component Framework: An Introduction*. Addison-Wesley, 1999.
- [12] Harry M. Sneed. Encapsulating legacy software for use in client/server systems. In *Working Conference on Reverse Engineering*, pages 104–119, Monterey, USA, Oct 1996. IEEE Computer Society, IEEE Computer Society Press.
- [13] A Cimitile, U DeCarlini, and A DeLucia. Incremental migration strategies: Data flow analysis for wrapping. In *Working Conference on Reverse Engineering*, pages 59–68, Hawaii, USA, Oct 1998. IEEE Computer Society, IEEE Computer Society Press.
- [14] H.-Arno Jacobsen and Bernd J. Kramer. A design pattern based approach to generating synchronization adaptors from annotated idl. In *Proceedings of the Automated Software Engineering Conference*, pages 63–72, Hawaii, USA, Oct 1998. IEEE Computer Society, IEEE Computer Society Press.
- [15] David Garlan, Robert T. Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.
- [16] AcmeStudio: A graphical design environment for acme. <http://www.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html>.
- [17] The acme tool developer's library (acmelib). http://www.cs.cmu.edu/~acme/acme_downloads.html.
- [18] D. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.
- [19] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
- [20] Kostas Kontogiannis. Distributed objects and software application wrappers: A vehicle for software re-engineering. In *Working Conference on Reverse Engineering*, page 254, Hawaii, USA, Oct 1998. IEEE Computer Society, IEEE Computer Society Press.
- [21] K. J. Sullivan, J. Cockrell, S. Zhang, and D. Coppit. Package oriented programming of engineering tools. In *Proceedings of the International Conference on Software Engineering*, pages 616–617, 1997.
- [22] Anind K. Dey, Gregory Abowd, Mike Pinkerton, and Andrew Wood. Cyberdesk: A framework for providing self-integrating ubiquitous software services. Technical Report GIT-GVU-97-10, Georgia Institute of Technology, 1997.