

# Synthesizing and integrating legacy components as services using adapters

Sudhakiran V. Mudiam<sup>a,1</sup>, Gerald C. Gannod<sup>b,\*</sup>, Timothy E. Lindquist<sup>b</sup>

<sup>a</sup> Department of Computer Science and Engineering, Arizona State University - Tempe, Box 875406, Tempe, AZ 85287-5406, United States

<sup>b</sup> Division of Computing Studies, Arizona State University - Polytechnic campus, Sutton Hall Suite 140M, 7001 E. Williams Field Road, Mesa, AZ 85212, United States

Available online 19 December 2005

## Abstract

Legacy applications are prime candidates for software reuse: they have been relied upon for several years and often have a strong organizational commitment. Migrating existing legacy applications is a very natural requirement when moving to and adopting a new technology. A service-based development paradigm is one in which components are viewed as services. In this model, services interact and can be providers or consumers of data and behavior. This paper describes an architecture-based approach for the synthesis of services from legacy components and their subsequent integration with service-requesting client applications.

© 2005 Elsevier B.V. All rights reserved.

*Keywords:* Service-oriented computing; Software architecture; Re-engineering

## 1. Introduction

Software reuse is a concept that has evolved over the years. Current software reuse techniques include object orientation, component-based software development, and service-based development. Reuse of legacy applications and components is essential for leveraging large organizational investments made towards systems that have been relied upon for several years. Software organizations are committed to work with proven technologies and continually migrate these systems to newer paradigms. For instance, the .NET [11] framework is a step towards seamless reuse of older components.

Service-oriented architectures have gained much attention with the advent of web services. Services are generally characterized as software components that exhibit a number of characteristics including *modularity*, *availability*, *description*, *implementation independence*, and *publication* [4]. In the service-oriented paradigm, components are specified according to an interface needed to access a service (description) while keeping component implementations independent across services (implementation independence, modularity). Services are intended to be always on, with several alternative implementations available (availability). As a result, any or all of the services may be integrated with a client at run-time (published).

\* Corresponding address: Arizona State University - Polytechnic, Division of Computing Studies, 7001 E. Williams Field Rd., Mesa, AZ 85212, United States. Tel.: +1 480 7274475; fax: +1 480 9652751.

E-mail address: [gannod@asu.edu](mailto:gannod@asu.edu) (G.C. Gannod).

<sup>1</sup> Currently at iRadio, Connected Home, Motorola, Tempe, AZ.

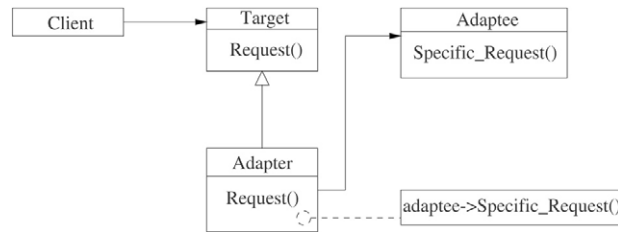


Fig. 1. Object adapter [5].

This paper describes an architecture-based approach for the creation of services from legacy components using wrapping, or adapters and the subsequent integration of these services with service-requesting client applications. The technique utilizes an architecture description language to describe components as services and achieves run-time integration using current middleware technology. The approach itself is based on a proxy model [5] and involves the automatic synthesis of “glue” code for both services and applications. The Jini interconnection technology [9] is used as a broker for facilitating service registration, lookup, and integration at run-time. Our approach utilizes ACME ADL to specify both the services and the target applications.

The remainder of this paper is organized as follows. Section 2 describes background material in the areas of design patterns, software architecture and the middleware technology that we use to enable dynamic integration (i.e., Jini). The proposed approach for constructing services and developing service-based applications is introduced in Section 3. Section 4 describes the generation of services from legacy command-line applications. The creation of the client applications using the generated services is described in Section 5. Section 6 discusses related work, and Section 7 draws conclusions and suggests further investigations.

## 2. Background

This section provides background material on design patterns (specifically the adapter and proxy patterns), software architecture, and Jini technology as they play an important role in our approach.

### 2.1. The adapter pattern

Re-engineering is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [3]. 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 re-engineering is to use the *adapter pattern* [5] whereby a legacy interface is converted into a form that a client application can utilize. 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 our approach the adapter pattern is used to re-engineer legacy command-line software to provide the software as services. Specifically, in terms of the Gamma et al. adapter pattern, we use the concept of the *object adapter* in the manner shown in Fig. 1. The legacy command-line application components are adapted to new interfaces, as required by client applications, by creating an adapter around the component. Fig. 1 uses the UML class diagram notation to describe the object adapter pattern.

### 2.2. The proxy pattern

The *proxy pattern* [5] provides a surrogate or a placeholder for another object to control access to it. In this pattern a client accesses a realSubject only via a proxy. The proxy provides an intermediate layer between the client and the realSubject. The proxy acts as a local representative for the realSubject and typically lives in the client’s address space. It is necessary for the proxy to provide exactly the same interface as the realSubject. All the accesses to the realSubject from the client have to go through the Proxy. In most cases the client is not even aware of the proxy object and assumes that it is directly talking to the realSubject. Fig. 2 shows the interaction between a client and a realSubject via a proxy. This Fig. 2 uses UML’s object diagram notation to show the proxy pattern at run-time. In the approach described in this paper, the proxy pattern plays a central role in the definition of services as well as

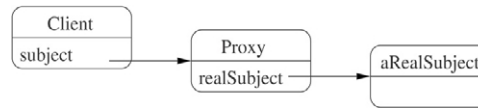


Fig. 2. Proxy pattern [5].

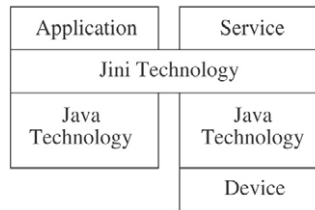


Fig. 3. Jini architecture.

in the realization of service integration. The proxy talks to the service on behalf of the client that is accessing the service. The proxy becomes part of the client, and it shields the client from the details of where and how the service is implemented. This pattern allows for the client to remain oblivious to the details of the service implementation. In the approach described in this paper the use of a proxy is central to the development and construction of wrappers and interfaces.

### 2.3. Jini

The work described in this paper is based on the use of the Jini framework [9]. In a Jini network, services are provided by devices that are connected to the network. Typically these devices consist of a variety of products ranging over cell phones, desktop devices, printers, fax machines, and Personal Digital Assistants (PDAs). Fig. 3 shows the architecture of Jini, where the Jini layer provides *discovery*, *lookup*, *remote event management*, *transaction management*, *service registration*, and *service leasing* services. When a service is registered as a member of the network, it is logged by the Jini lookup service. Once registered, a proxy [5] is stored by the lookup service. The proxy can later be transported to the clients of the service. Network members discover the availability of the service via the Jini lookup service. When a client application finds an appropriate service, the lookup service sets up a connection. We use Jini to provide a standard method for registering services, and connecting a client to the software components that are acting as services.

One of the advantages of using a Jini-based integration technique is that it facilitates construction of variable applications, where the variability is determined by the implementation of a bound service. In our approach, clients must have some prior knowledge of how to use member services.

## 3. Approach

This section describes the service-based development approach including the techniques used for defining services, specifying client applications, realizing integration, and generating glue code. The legacy components that we are interested in are command-line applications as they have well-defined interfaces. They can be specified in an architectural context based entirely on the knowledge of how the applications are used. This information is useful in generating the code that facilitates the integration at run-time.

As stated in Section 1, the service-oriented domain is characterized by modularity, availability, description, implementation independence, and publication. As a result, services and service-based approaches are more coarse-grained and more loosely coupled than components used in traditional component composition techniques. The approach described in this paper utilizes a software architecture to specify applications that operate under these characteristics. As such, a software architecture in this context defines components, their interfaces, and the mechanisms by which services (as components) can be joined in order to fulfill needed software behavior requirements. Consequently, services enable the use of a software architecture as an integration vehicle in which the architecture facilitates generation of glue code. It is the very fact that services adhere to the characteristics described above that leads to integration and code generation becoming possible at this level.

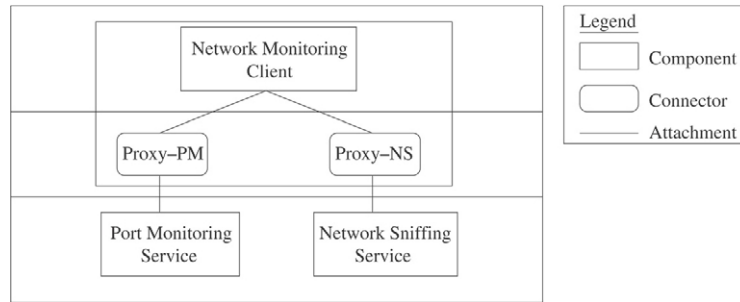


Fig. 4. Basic architecture.

### 3.1. Overview

In this paper we discuss two aspects of a software migration strategy. First, we describe an architecture-based approach for specifying and subsequently synthesizing services for command-line applications. Second, we discuss the integration of these services to form applications.

The methodology we have developed follows closely the model suggested by Stal [13] for web services with respect to the use of an architectural perspective to integration, although the technology that we are using to realize our approach is Jini. Stal proposes that the web services model can be used as an integration vehicle, the advantage being that web services are based on standards such as XML and thus allow for interoperability between existing technologies. The approach described in this paper, while based on Jini, must deal with issues similar to those in the web services area.

The work described in this paper focuses on both *for reuse* and *with reuse* concerns. As regards *for reuse*, the approach uses adapter and proxy patterns to synthesize services from command-line applications as follows:

- (1) Specification of components as services.
- (2) Generation of services using proxies and adapters to generate glue code.

As regards *with reuse* concerns, the approach involves the construction of applications using services as follows:

- (1) Specification of a client to make use of services from a repository or network.
- (2) Generation of the client (both manual construction of client application specific code and automated generation of glue code).
- (3) Execution of the client, including integration of the specified services at run-time.

Fig. 4 shows a diagram depicting the basic architecture for a network administration example described in Section 3.2. In general, applications assume an architecture with a star topology, where proxy connectors provide the interface to services. The interactions between clients and services are heterogeneous in that the style of interaction may vary according to the nature of the service. The layers in the diagram indicate that there is implicitly a separation between different services.

In the context of the reuse concerns described above, a user (e.g., a developer) is responsible for writing the source code for the client application along with the specification of the architecture for a client. The client specification contains a description of the basic services that the client application will need in order to be a complete system. The information found in the specifications is used to generate all other source code, including code necessary to realize the connections between the client and employed services.

### 3.2. Example

Fig. 5 shows a network monitoring system that provides a network administrator with a constant update on the health of systems in a network. This application utilizes a *network sniffer* service and a *port monitoring* service. The network sniffer service gives an administrator information about traffic on the network. The port monitoring service provides information about the open ports on the various machines on a network. Together, these services facilitate determining whether certain kinds of attacks (such as ping storms) are being directed to a machine or machines. The client application supports analysis of several networks, each of which is accessed using the buttons shown on the top



Fig. 5. Running example.

portion of the GUI. From the standpoint of distribution, this application demonstrates the use of services that utilize different models of execution (strict call return and data streams). The port monitoring service is a *call return* service, whereas the network sniffer service is defined as a *communicating process* service. The port monitoring service defines a *getOpenPorts* method that returns the list of open ports on a target machine using the *nmap* tool. *nmap* is a Unix command-line application that scans all the ports and lists all the interesting open ports. The network sniffer service defines two methods, *startSniff* and *stopSniff*. Each of these methods return immediately, and the network sniffer service pumps the sniffed data to the client on a data stream using the *tcpdump* tool. The *tcpdump* is a command-line tool that dumps network traffic. As shown in Fig. 5 the administrator was interested in the open ports on the machine 10.1.1.191 as there was significant traffic on the network originating from it. The services are running on different machines, and the client is running on yet another machine in the network.

The following section refers to architectural specifications that were used in the construction of this example. This example also is the target for the adapters that are synthesized as an illustration of our approach.

#### 4. Command-line applications as services

This section describes our efforts in automating the creation of service wrappers for legacy command-line applications. We have developed an automated tool that takes as input a software architecture specification and produces glue code. Since command-line applications have a well-defined input and output interface, the service can be based entirely upon the knowledge of what the application intends to provide.

##### 4.1. Specification and synthesis

The concept of using an adapter for wrapping legacy software is not a new one [5]. As a migration strategy, component wrapping has many benefits in terms of re-engineering 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.

As regards 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.

Table 1  
Properties

Group	Attribute	Description
<i>Service properties</i>	Component-type	Architectural style this component adheres to
<i>Service port properties</i>	Signature	The port's signature
	Return	The port's return type
	Cmd	The command-line program being wrapped
	Pre	Pre-processing command
	Post	Post-processing command
	Interface	The generic interface implemented by this port
	Path	Path to the wrapped command-line program
	Port-type	The port's type based on the Component-type
	Shared-GUI	Boolean indicating shared (true) or exclusive (false) GUI
<i>Client properties</i>	Part-of-client	Identifies inclusion in client application
	GUI-CodeFile	The filename for client's GUI code
	Component-type	Architectural style this component adheres to
	Shared-GUI	Boolean indicating shared (true) or exclusive (false) GUI
<i>Client port properties</i>	Port-type	The port's type based on the Component-type
	Interface	The generic interface that this port can bind with
<i>Connector properties</i>	Connector-type	Architectural style this connector adheres to
<i>Connector role</i>	Prop-type	The connectors role based on the Connector-type

#### 4.2. Specification requirements

To aid in the development of an appropriate scheme for the wrapping activity, we defined the following requirements upon specifications. These requirements are as follows:

- (S1) 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.
- (S2) A specification of the interface of the adapted component should be as loosely coupled as possible to the target implementation language.
- (S3) The specification of the adapted component should be usable within a more general architectural context.

The requirement S1 addresses the fact that we are interested in gaining a benefit from re-using 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.

Our selection of command-line applications addresses the modification concern of requirement S1 since source code may not be available. As a result, 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.

Table 1 shows the properties used in the specification of services, clients, and connectors. A service component specification consists of two parts: *properties* and *ports* (architectural ports). In this paper, we define an architecture as a *configuration of components and connectors*. A component is an encapsulation of a computational unit and has an interface (e.g., a port) that specifies the capabilities that the component can provide. 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. Components are connected by attaching their ports to the roles of connectors.

In the specification, the properties section describes the style of the service, while the ports section describes functions provided by the service. In addition, the service specifications indicate style-based information as well as conditions or commands that need to be true or executed, respectively, in order to establish an environment necessary to use the service. Finally, a key in terms of a “service type” (e.g., an *Interface* property) is used to support a service lookup, which is later utilized during application integration.

The requirement S2 (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 S1 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 fact that our implementation would make use of Jini. We tried to keep the specification implementation independent; however, we had to add several properties that needed to specify GUI code written in Java.

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 S3, 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 [7] 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 [7]. ACME is supported by an architectural specification tool, ACMEStudio [1], that facilitates graphical construction and manipulation of software architectures.

#### 4.3. Service generation issues

As stated earlier, the class of legacy systems that we are considering is the command-line applications [6]. 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. Command-line applications have well-defined input and output interfaces. This information can be readily obtained from the usage of the application. In certain cases many pages may be available to provide the interfaces. Terminating command-line applications can be modelled as call return type, as they return after running the application. Non-terminating command-line applications can be modelled as communicating process type, as they keep sending output once the application is running.

In our approach, the specification that is needed to generate wrappers contains properties associated with the ports as shown in Fig. 6. These properties include *Signature*, *Command*, *Pre*, *Post*, *Path*, *Interface*, and *Return*. In this case, the specification describes the *NetworkSniffing* and *PortMonitor* services, which are services created by wrapping *tcpdump* and *nmap*, respectively. In the synthesis process, ACME specifications are combined with a standard template that implements the set-up routines that are required to register a service 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 outside of optional GUI code.

#### 4.4. Implementation

To support our technique for constructing wrappers for legacy software, we have created a Java support tool called *ServiceTool*. Fig. 7 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 ACMEStudio is used to create the ACME specification as shown in Fig. 6. The *ArchParser* component reads in an ACME specification similar to the one shown in Fig. 6 and builds an internal model of the architecture. The *ArchParser* uses the Java ACME API to parse the ACME specification. 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 *Interface 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.

The *ArchParser* uses the *ACMEParser* from the *ACMELib* toolkit [14] to parse ACME specifications. *ACMELib* is a library that facilitates the construction of architectural tools in Java that read, write and manipulate software

```

1  Component PortMonitor= {
2      Properties
3      {
4          Component Type: string= "Call Return";
5      };
6      Port getOpenPorts= {
7          Properties {
8              Signature: string = "String cmd";
9              Return: string = "String getOutputStream(aprocess) ";
10             Cmd: string = "nmap + cmd ";
11             Pre: string = "";
12             Post: string = "";
13             Interface: string = "PortMonitoring";
14             Path: string = "/usr/bin/nmap";
15             Port Type: string="callee";
16             Shared GUI: string="false";
17         };
18     };
19 };
20 Component NetworkSniffing= {
21     Properties
22     {
23         Component Type: string="Communicating Process";
24     };
25     Port sniff= {
26         Properties {
27             Signature: string = "";
28             Return: string = "";
29             Cmd: string = "tcpdump ";
30             Pre: string = "";
31             Post: string = "sendOutPutStream(aprocess) ";
32             Interface: string = "NetworkSniffing";
33             Path: string = "/usr/sbin/tcpdump";
34             Port Type: string="process";
35             Shared GUI: string="false";
36         };
37     };
38 };

```

Fig. 6. ACME services section.

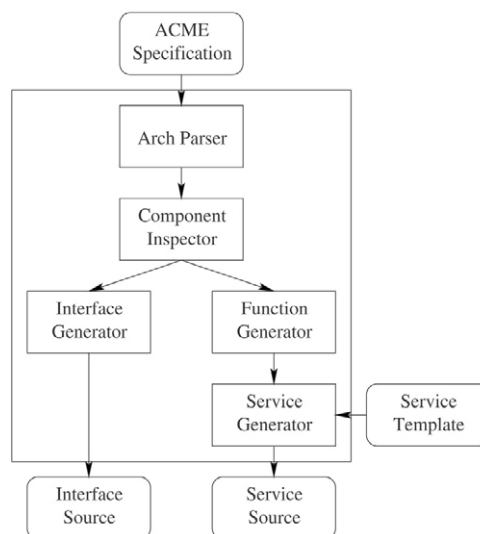


Fig. 7. Service tool architecture.

---

```

1 public class <put-ServerName> extends UnicastRemoteObject
2 implements <put-InterfaceName>, ServiceIDListener,
3 Serializable {
4     public <put-ServerName> () throws RemoteException
5     {
6         super ();
7     }
8     <put Functions>
9     ...
10    ...
11    ...
12    ...
13    public static void main (String[] args)
14    {
15        ...
16        ...
17        aeAttributes = new Entry[2];
18        aeAttributes[0] = new Name ("<put-ServerName>");
19        aeAttributes[1] = new <put-ServerName>Applet ();
20        inServer = new <put-ServerName> ();
21        joinmanager = new JoinManager
22        (
23        inServer,
24        aeAttributes,
25        inServer,
26        new LeaseRenewalManager ()
27        );
28        ...
29        ...
30    }
31    ...
32    ...
33 }

```

Fig. 8. Excerpt of the ServiceTemplate.

---

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 [10] and Wright [2]) 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.

Fig. 8 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 (shown in bold) that are placeholders for the generated code. Fig. 9 shows a portion of the generated PortMonitor Service using the *ServiceTool* for the PortMonitor ACME specification shown in Fig. 6.

The component name (on lines 1 and 20 of Fig. 6) is used in the placeholders <put-ServerName> (on lines 1, 4, and 18–20 of Fig. 8) for the final name of the adapter component which becomes the name of the service (lines 1, 4, and 38–40 of Fig. 9). The port and its properties for the components (on lines 6–18 of Fig. 6) are used to generate the functions that go in the placeholder <put-Functions> (on line 8 of Fig. 8). The generated functions are shown on lines 8–32 of Fig. 9 for the port *getOpenPorts*. The Interface property (on line 13 of Fig. 6) is used in the placeholder <put-InterfaceName> (on line 2 of Fig. 8). This interface is implemented by the PortMonitor service (on line 2 of Fig. 9).

The *getOpenPorts()* function (on lines 8–32 of Fig. 9) is generated by the *Function Generator* (Fig. 7) from the specification using the Signature, Path, Cmd, Pre, and Post properties (on lines 8–16 of Fig. 6) of the *getOpenPorts* Port. The Signature property goes in to the signature (line 8 of Fig. 9) of the *getOpenPorts()* method. The Path property goes into the Java process (line 11 of Fig. 9) that needs to be executed.

```

1 public class PortMonitor extends UnicastRemoteObject
2   implements PortMonitoring, ServiceIDListener,
3   Serializable {
4   public PortMonitor () throws RemoteException {
5     super ();
6   }
7
8   public String getOpenPorts(String cmd)  throws RemoteException {
9
10    Process aprocess = null;
11    String app = new String("/usr/bin/nmap " + cmd);
12    try{
13      Runtime runtime = Runtime.getRuntime();
14      aprocess = runtime.exec(app );
15      BufferedReader in = new BufferedReader(
16        new InputStreamReader(aprocess.getInputStream()));
17      aprocess.waitFor();
18      StringBuffer sb = new StringBuffer ();
19      while( (String s =in.readLine())!= null )
20      {
21        sb.append (s);
22        sb.append ("\n");
23      }
24      in.close();
25      return (sb.toString());
26    }catch (Exception e){
27      e.printStackTrace();
28      System.out.println ("PortMonitoringServer:
29        Error invoking nmap" + e);
30    }
31    return ("PortMonitorServer: Error while executing nmap.");
32  }
33  ...
34  public static void main (String[] args)
35  {
36    ...
37    aeAttributes = new Entry[2];
38    aeAttributes[0] = new Name ("PortMonitor");
39    aeAttributes[1] = new PortMonitorApplet ();
40    inServer = new PortMonitor ();
41    joinmanager = new JoinManager
42    (
43    inServer,
44    aeAttributes,
45    inServer,
46    new LeaseRenewalManager ()
47    );
48    ...
49  }
50  ...
51 }

```

Fig. 9. Generated PortMonitor service.

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 `getOutputStream()` routines (shown in Fig. 10) are available as functions for use within the Java code to provide standard stream input support.

## 5. Client generation

Once the services are generated and stored in a repository, a client application can be architected. First we need to specify the client application taking into account the architectural style of each of the services. Once a client is specified, it can be verified and generated. In this section we look at the requirements for specifying the client and then describe synthesis of the client.

```

String getOutputStream(Process process ) {
    try{
        BufferedReader in = new BufferedReader(
            new InputStreamReader(process) );
        process.waitFor();
        StringBuffer sb = new StringBuffer ();
        while( (String s = in.readLine()) != null ) {
            sb.append (s);
            sb.append ("\n");
        } in.close();
        return (sb.toString());
    } catch (Exception e) {
        return("Error getting Stream:"+e.getMessage());
    }
}

```

Fig. 10. Sample library routines.

### 5.1. Specification

Refer again to [Table 1](#) which, in addition to the properties for service specifications, contains the properties of client application components and connectors. When dealing with integration at the component level, two issues arise (among others) that are of interest. First, the problem of architectural style mismatch [12] occurs when the underlying assumptions made by components conflict. Second, most modern applications provide a Graphical User Interface (GUI). As a result, integration of off-the-shelf components can leverage these user interfaces in order to take advantage of previously built technology. To cope with these issues we impose two requirements on the specification of client applications as follows:

- (C1) The specification of the components should capture the notion of architectural style so that the high-level interaction between clients and services can be verified.
- (C2) The specification must facilitate the use of shared and exclusive GUI components.

The requirement C1 addresses the fact that a component must provide a notion of architectural style. A component's style plays a very important role when it interacts with other components by imposing interaction constraints. Using a basic style attribute (by name), architectural mismatches can be determined by simple keyword matching.

Requirement C2 addresses the fact that a service may provide a GUI that allows a user to access and control the service. In this context, there may be GUI components provided by services that are either *sharable* by other services or *exclusive* to the service. A sharable GUI component can be used by both the client and other integrated services, while an exclusive GUI component can only be used by the service that provides the interface.

### 5.2. Client generation issues

The second stage of our approach involves the synthesis of application code. [Fig. 11](#) shows a sample specification of a client. The information contained within client specifications is used to support the synthesis of client code. This synthesis step utilizes two features. First, the information regarding connectors and attachments, such as those shown in [Fig. 11](#), is used to determine the relationships between client applications and desired services. Second, information regarding GUIs provided by services is used to determine how to realize the GUI in a client application.

In our framework, the wrappers for the various services can implement a common interface that allows the client to get a handle on the shared and exclusive components of a GUI. Shared components are potentially used across multiple services and are identified using a name taken from a standard GUI vocabulary (for example "ResultsWindow"). The name is then used to identify which GUI components can be shared across services. Such shared components facilitate the integration of the GUI components by allowing re-use of widgets that provide the same functionality. An exclusive component is independent and cannot be shared between services. The exclusive GUI components of the wrappers are used as is, but may interact with one or more of the shared components. For both shared and exclusive components,

---

```

Component NetworkMonitor = {
  Properties {
    Part-of-client : string = "true";
    GUI-CodeFile : string = "ClientGUICode.java";
    Component-type : string = "Call Return";
    Shared-GUI: string = "false";
  };
  Port PortMonitoring_PORT = {
    Properties {
      Port-type : string = "caller";
      Interface : string = "PortMonitoring" ;
    };
  };
  Port Sniffing_PORT = {
    Properties {
      Port-type : string = "caller";
      Interface : string = "NetworkSniffing";
    };
  };
};
Connector open_ports = {
  Properties {
    Connector-type : string = "Call Return";
  };
  Role caller =
  {
    Properties {
      Prop-type : string = "output";
    }; };
  Role callee =
  {
    Properties {
      Prop-type : string = "input";
    }; };
};
Connector sniffing = {
  Properties { Connector-type : string = "Data Stream"; };
  Role input =
  { Properties { Prop-type : string = "input"; }; };
  Role output =
  { Properties { Prop-type : string = "output"; }; }; };
Attachments {
  NetworkMonitor.PortMonitoring_PORT to open_ports.caller;
  NetworkMonitor.Sniffing_PORT to sniffing.output;
  NetworkSniffing.sniff to sniffing.input;
  PortMonitor.getOpenPorts to open_ports.callee;
};

```

Fig. 11. Portion of ACME client specification.

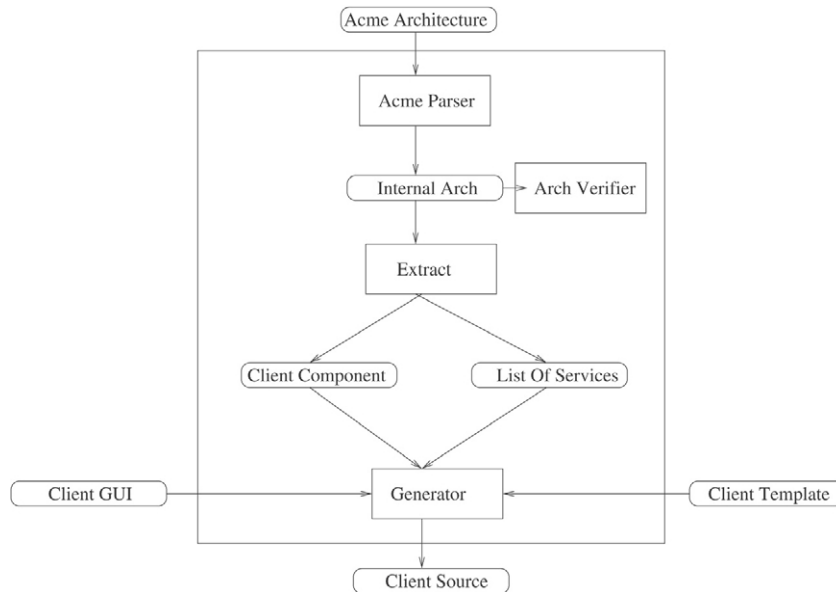


Fig. 12. ClientGenTool architecture.

the interaction with the client GUI and application is seamless since the wrappers handle direct interaction with the services while the client need only interact with the wrappers.

As mentioned earlier, specifications in our framework capture the style characteristics of components. To facilitate this verification a tool called *Arch Verifier* is used to verify that the styles of components are consistent. It does so by verifying that all the attachments between client and service components match. Our current implementation imposes an exact match criterion whereby components can only be connected to components of the same type. For example, a *Call Return* component can only be connected to either a *Call Return* component through a *Call Return* connector or to a *Communicating process* component through a *Data Stream* connector. Our future investigations include expanding the verifier to allow for mappings of other types, where appropriate.

### 5.3. Implementation

To support the construction of client applications, we have created a support tool written in Java called *ClientGenTool*. Fig. 12 shows the detailed architecture of the *ClientGenTool* which takes an ACME architecture (specification) and produces the Client source using a template-based approach. In this figure, the rectangles with the square corners represent software components while the rectangles with the rounded corners represent data stores or repositories.

The *ArchParser* reads in ACME architecture specifications similar to the ones shown in Figs. 6 and 11 and builds an internal model of the architecture (*Internal Arch*). The *ArchParser* uses the *ACMEParser* from the *ACMELib* toolkit [14] to parse ACME specifications. The *Arch Verifier* uses the *Internal Arch* model to verify the style of the architecture based on the *Component-type* and the *Port-type* properties. Once it is verified, the *Extract* component identifies the *Client Component*, the *Client GUI*, and the *List of Services* from the *Internal Arch* model. All these are used by the *Generator* component along with the *Client Template* (shown in Fig. 13) to produce the *Client Source*. The *Client Template* has placeholders for the list of services that will be integrated, shown in bold as **(ListOfServices)**.

The client specification for our example describes a *NetMonitor* component having two ports, namely *PortMonitoring\_PORT* and *Sniffing\_PORT*. The *PortMonitoring\_PORT* is connected using an *open\_ports* connector to a *PortMonitoring* component. The *Sniffing\_PORT* is connected using a *sniffing* connector to the *NetworkSniffing* component. The connector types match the component types that they connect. The *open\_ports* is a *Call Return* type connector which connects two *Call Return* type components. The *sniffing* is a *Data Stream* type connector which connects a *Call Return* type component and a *Communicating Process* type component. The connectors are verified

---

```

public class MyClient {
    static String[] services = <ListOfServices> ;
    static Hashtable state = new Hashtable();

    <addClientComponentFuncs>

    public static void main (String[] args) {
        //discover and join the jini network.
        .....
        //discover all the services and get their proxies
        .....
    }
    private static boolean alreadyAdded(String key) {
        //check if a shared component
        // with key has been added.
        .....
    }
    ....
}

```

Fig. 13. Client template POC.

---

```

public class MyClient {
    static String[] services = {"PortMonitoring", "NetworkSniffing"} ;
    static Hashtable state = new Hashtable();

    public static void main (String[] args) {
        //discover and join the jini network.
        .....
        //discover all the services and get their proxies
        .....
    }
    private static boolean alreadyAdded(String key) {
        //check if a shared component
        // with key has been added.
        .....
    }
    ....
}

```

Fig. 14. Generated client code fragment.

---

by the *Arch Verifier* in the client generation process. Fig. 14 shows the generated code fragment for the client, where the `<ListOfServices>` is replaced with `PortMonitoring` and `NetworkSniffing`, each of which correspond to the *type* of the component connected to the NetworkMonitor client.

## 6. Related work

The Jini [9] approach to service integration goes beyond what the basic web services [13] paradigm provides by defining how services can be used within a larger application context and providing support for code transportation. In this way, Jini is a precursor to the multitude of facilities now available in the way of UDDI, semantic web services, and so forth.

Wohlstadter et al. [17] have described an approach to generating wrappers for command-line programs. The Cal-Aggie Wrap-o-matic project is a tool that generates wrappers that can be accessed using CORBA. Their approach is similar to ours in that they propose an extension to CORBA IDL to capture the specification.

Jaeger et al. [8] have proposed a methodology for developing semantic descriptions of web services using OWL-S. They recognize the lack of tool support for the development of semantic descriptions. A three-step process is introduced in which a tool will: create a template using existing software artifacts (e.g. software models, WSDL), automate the identification of relevant ontologies, and perform a classification based on those ontologies. Their methodology is focused around the use of a matchmaking algorithm to identify relevant ontologies and classify elements in the semantic description with those ontologies. Our work differs in that it relies on Jini technology to perform matchmaking as well as to achieve other service maintenance activities. Our approach uses ACME instead of OWL-S and is thus limited in the nature of the interactions that can be expressed.

## 7. Conclusions

The use of services and service-oriented architectures [13] continues to gain much attention. Semantic web services and the use of languages such as OWL-S [15] have been developed to facilitate architecture-based description and composition of services into applications. In this paper, we have described a service-oriented approach based on the use of Jini [9]. The approach relies on the use of command-line applications to provide services and the ACME ADL as the primary specification medium.

Our current investigations include the use of more semantically rich languages such as OWL-S coupled with standard software description languages such as UML [16]. In this context, we are developing an approach based on concepts of model-driven development that uses the work described in this paper as a foundation for the investigations.

## Acknowledgement

The second author was supported in part by NSF CAREER Grant CCR-0133956.

## References

- [1] AcmeStudio: A graphical design environment for acme, <http://www.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html>, 2002.
- [2] R. Allen, D. Garlan, A formal basis for architectural connection, *ACM Transactions on Software Engineering and Methodology* (1997).
- [3] E.J. Chikofsky, J.H. Cross, Reverse engineering and design recovery: A taxonomy, *IEEE Software* 7 (1) (1990) 13–17.
- [4] P. Fremantle, S. Weerawarana, R. Khalaf, Enterprise services, *Communications of the ACM* 45 (10) (2002) 77–80.
- [5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, 1995.
- [6] G.C. Gannod, S.V. Mudiam, T.E. Lindquist, An architecture-based approach for synthesizing and integrating adapters for legacy software, in: *Proc. of the 7th Working Conf. on Reverse Engineering*, November 2000, IEEE, 2000, pp. 128–137.
- [7] D. Garlan, R.T. Monroe, D. Wile, Acme: An architecture description interchange language, in: *Proc. of CASCON'97*, November 1997, pp. 169–183.
- [8] M.C. Jaeger, L. Engel, K. Geihs, A methodology for developing owl-s descriptions, in: *First International Conference on Interoperability of Enterprise Software and Applications Workshop on Web Services and Interoperability*, February 2005.
- [9] W.K. Edwards, *Core Jini*, Prentice-Hall, 1999.
- [10] D. Luckham, J. Vera, An event-based architecture definition language, *IEEE Transactions on Software Engineering* 21 (9) (1995) 717–734.
- [11] B. Meyer, .NET is Coming, *IEEE Computer* 34 (8) (2001) 92–97.
- [12] M. Shaw, D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [13] M. Stal, Web services: beyond component-based computing, *Communications of the ACM* 45 (10) (2002) 71–76.
- [14] The acme tool developer's library (acmelib), [http://www.cs.cmu.edu/~acme/acme\\_downloads.html](http://www.cs.cmu.edu/~acme/acme_downloads.html), 1997.
- [15] The OWL Services Coalition, Owl-s: Semantic markup for web services. Available at <http://www.daml.org/services/owls/1.0/owl-s.html>, 2003.
- [16] J.T.E. Timm, G.C. Gannod, A model-driven approach for specifying semantic web services, in: *IEEE International Conference on Web Services*, July 2005, pp. 313–320.
- [17] E. Wohlstadt, S. Jackson, P.T. Devanbu, Generating wrappers for command line programs: The Cal-Aggie Wrap-o-matic project, in: *International Conference on Software Engineering*, 2001, pp. 243–252.