Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
The Journal of Systems and Software xxx (2004) xxx–xxx
The Journal of Systems and Software
www.elsevier.com/locate/jss

Automated support for service-based software development and integration

Gerald C. Gannod [a,*], Sudhakiran V. Mudiam [a], Timothy E. Lindquist [b]

[a] Department of Computer Science and Engineering, Arizona State University—Main, P.O. Box 875406, Tempe, AZ 85287-5406, USA
[b] Department of Electronics and Computer Engineering Technology, Arizona State University—East 7001 E, Williams Field Road, Building 50, Mesa, AZ 85212, USA

Received 16 October 2002; received in revised form 1 February 2003; accepted 2 May 2003

Abstract

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. Applications in this paradigm dynamically integrate services at runtime-based on available resources. This paper describes an architecture-based approach for the creation of services and their subsequent integration with service-requesting client applications.

1. Introduction

A service-based development paradigm, or services model (Fremantle et al., 2002) is one in which components are viewed as services. In this model, services can interact with one another and be providers or consumers of data and behavior. Some of the defining characteristics of service-based technologies include *modularity, availability, description, implementation-independence*, and *publication* (Fremantle et al., 2002). In the service-based development paradigm, a primary focus is upon the definition of the interface needed to access a service (description) while hiding the details of its implementation (implementation-independence). Since the client and service are decoupled, other concerns such as side effects become non-factors (modularity). One of the potential benefits of using a service-based approach for developing software is that at any given time, a wide variety of alternatives may be available that meet the needs of a given client (availability). As a result, any or all of the services may be integrated with a client at runtime (published).

This paper describes an architecture-based approach for the creation of services and their subsequent integration with service-requesting client applications. The technique utilizes an architecture description language to describe services and achieves run-time integration using current middleware technology. The approach itself is based on a proxy model (Gamma et al., 1995) and involves the automatic generation of "glue" code for both services and applications. The Jini interconnection technology (Edwards, 1999) is used as a broker for facilitating service registration, lookup, and integration at runtime.

The remainder of this paper is organized as follows. Section 2 describes background material in the areas of software architecture and the middleware technology we are using to enable dynamic integration (i.e. Jini). The proposed approach for constructing services and developing service-based applications is presented in Section 3. Section 4 discusses related work, and Section 5 draws conclusions and suggests further investigations.

2. Background

This section describes background material on software architecture and Jini.

* Corresponding author. Tel.: +1-480-727-4475; fax: +1-480-965-2751.
E-mail address: gannod@asu.edu (G.C. Gannod).

0164-1212/$ - see front matter © 2003 Published by Elsevier Inc.
doi:10.1016/j.jss.2003.05.002

2.1. 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 (Shaw and Garlan, 1996). 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 (e.g. port) that specifies the capabilities that the component can provide.

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. Components are connected by attaching their ports to the roles of connectors.

Another important concept is 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 (Medvidovic and Taylor, 1997).

2.2. Jini

The primary enabling feature of the work described in this paper is the existence of Jini (Edwards, 1999) for the delivery and management of services. In a typical Jini network, services are provided by devices that are connected to the network. A 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 service is plugged into a Jini network, it becomes registered as a member (e.g. service) of the network by the Jini lookup service. When a service is registered, a proxy (Gamma et al., 1995) is stored by the lookup service. The proxy can later be transported to the clients of the service. Other network members can discover the availability of the service via the lookup service. When a client application finds an appropriate device, the lookup service sets up the connection. In our approach to component integration, we use Jini 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 integration technique is that it facilitates construction of applications ''on-the-fly'' whereby components can be used on an as-needed basis. One of the disadvantages is that clients of services must have some prior knowledge about how to use each respective service.

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.

3.1. Example

Fig. 1 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 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 remainder of this section refers to architectural specifications that were used in the construction of this example.

3.2. Overview

The methodology that we have developed follows closely the model suggested by Stal (2002) for web ser-

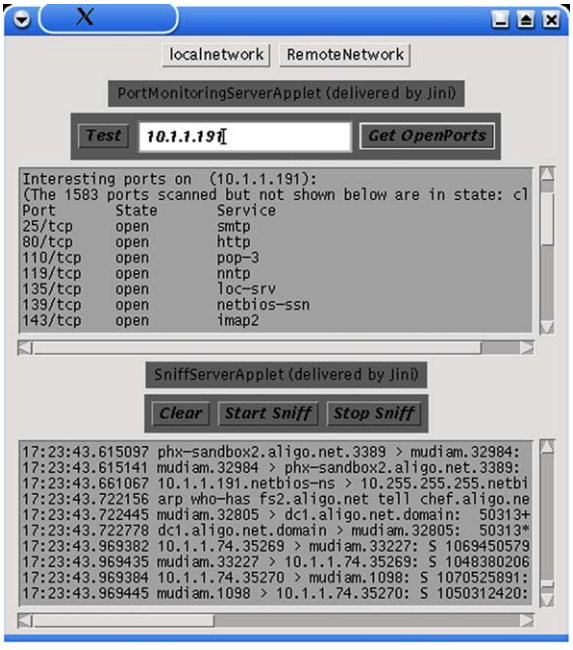

Fig. 1. Running example.

vices, although the technology that we are using to realize our approach is Jini. The approach itself focuses on two concerns with respect to software reuse. That is, it addresses both *for reuse* and *with reuse* concerns. With respect to *for reuse*, the approach involves the construction of services via the use of adapter and proxy synthesis. Specifically, the methodology involves two steps for creating services as follows: (1) specification of components as services, and (2) generation of services using proxies via the construction of appropriate adapters and glue code. These services are consequently registered and made available on a network.

With respect to *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), and (3) execution of the client, including integration of the specified services at runtime.

Within our approach, a user (e.g. developer) is responsible for writing the source code for the client application along with the specification of the architecture for a client. Among other things, the client specification contains a description of the basic services that the client application will need in order to be a complete system. All other source code, including code necessary to realize the connections between the client and employed services, is generated based on the specifications describing clients, services, and connectors.

3.3. Service generation

In this section we describe some of the issues related to automating the creation of service wrappers. To support these activities, we have developed an automated tool that takes as input a software architecture and produces glue code. A primary source of reusable components that we employ in our approach are legacy command-line applications (Gannod et al., 2000). In order to generate services from legacy components, we take the approach of wrapping the components by utilizing the interface provided by the component. Since command-line applications have a well-defined input and output interface, the interface of the application as a service can be based entirely upon the knowledge of what the application intends to provide.

3.3.1. Specification and synthesis

The concept of using an adapter for wrapping legacy software is not a new one (Gamma et al., 1995). 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.

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.

3.3.2. 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 from the target implementation language, and (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 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.

Our selection of command-line applications addresses the modification concern of requirement S1 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.

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*. The properties section describes 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. *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

Table 1
Properties

Group	Attribute	Description
Service properties	Component-Type	Architectural style this component adheres to
Service port properties	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
Client properties	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
Client port properties	Port-Type	The port's type based on the Component-Type
	Interface	The generic interface that this port can bind with
Connector properties	Connector-Type	Architectural style this connector adheres to
Connector role	Prop-type	The connectors role based on the Connector-Type

necessary due to the fact that our implementation would make use of Jini.

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 (Garlan et al., 1997) 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 (Garlan et al., 1997).

3.3.3. Synthesis

As stated earlier, the class of legacy systems that we are considering are command-line applications (Gannod et al., 2000). 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.

In our approach, the specification that is needed to generate wrappers contains properties associated with the ports as shown in Fig. 2. 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 specifica-

```
Component PortMonitor= {
   Properties
   { Component-Type: string= "Call Return";  };
   Port getOpenPorts= {
      Properties {
       Signature: string = "String  cmd";
       Return: string = "String getOutputStream(aprocess)";
       Cmd: string = "nmap + cmd ";
       Pre: string = "";
       Post: string = "";
       Interface: string = "PortMonitoring";
       Path: string = "/usr/bin/nmap";
       Port-Type: string="callee";
       Shared-GUI: string="false";
      };  };  };
Component NetworkSniffing= {
   Properties
   { Component-Type: string="Communicating Process"; };
   Port sniff= {
      Properties {
       Signature: string = "";
       Return: string = "";
       Cmd: string = "tcpdump ";
       Pre: string = "";
       Post: string = "sendOutPutStream(aprocess)";
       Interface: string = "NetworkSniffing";
       Path: string = "/usr/sbin/tcpdump";
       Port-Type: string="process";
       Shared-GUI: string="false";
      };  };  };
```

Fig. 2. ACME services section.

tions are combined with a standard template that implements the setup 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 option GUI code.

Both the service and client synthesis steps utilize a template-based approach to synthesize code. That is, a standard file has been created that has stubs containing place holders that must be instantiated with either service or client specific parameters. Fig. 3 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 Fig. 3 the tag `<put-ServerName>` is a place-holder for the final name of the adapter component.

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()` routine (shown in Fig. 4) is available as a function for use within the Java code to provide standard stream input support.

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

Fig. 3. Excerpt of the service template.

```
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. 4. Sample library routines.

The amount of automation that has been achieved through the approach described above is dependent on the degree of graphical user interface (GUI) support that is desired. For a service, the code synthesis step can be fully automated if no GUI support is desired. Otherwise, the amount of manual code construction is limited to GUI support.

3.4. 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 subsection we look at the requirements for specifying the client and then describe synthesis of the client.

3.4.1. 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 (Shaw and Garlan, 1996) 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, and (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 as well as other integrated services while an exclusive GUI component can only be used by the service that provides the interface.

3.4.2. Synthesis

The second stage of our approach involves the synthesis of application code. Fig. 5 shows a sample specification of a client. The information contained within client specifications are 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. 5 are 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 reuse 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, 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.

```
Component NetMonitor = {
  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. 5. Portion of ACME client specification.

3.5. Discussion

As stated in Section 1, the service-oriented domain are 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. 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 the integration and code generation become possible at this level. However, the approach does lack in its ability to address needs that are more specific than what individual services provide. To cope with this, we are developing an approach that allows for the creation of federated services, where services are combined to meet some higher-level objective.

4. Related work

Recently, the use of web services has gained attention with vendors releasing webservices toolkits that allow for building and using webservices. Webservices and .NET (Meyer, 2001) are based on the SOAP and XML (Seely and Sharkey, 2001) protocols. The Jini approach to service integration goes beyond what the webservices paradigm provides by defining how services can be used within a larger application context and providing support for code transportation.

FIELD (Reiss, 1990) is one of the classical approaches to tool integration built using a central server that distributed messages to other tools that were interested in them. It is a message-based broadcast system that sends message strings between the tools selectively (selective broadcasting). In this sense, this approach is a precursor to service-based development.

Urnes and Graham (1999) describe an approach to facilitate the use of groupware in a distributed environment by using architectural annotations. In this approach, they achieve distribution by partitioning the component space across a network. In our approach, services are potentially developed by different organizations and thus the choice of what to distribute is not

available. The component model being addressed by Urnes and Graham, as such, is finer-grained and violates implementation-independence, a tenet of service-based development.

Grundy et al. (2000) discuss issues and experiences in constructing component-based software engineering environments. They created a variety of useful software engineering tools using their tool set (JViews, JComposer, etc.). They use ''plug and play'' and an event-based composition approach to achieve component integration. In this framework, components are more tightly coupled and their granularity is fine-grained. In contrast, our approach is based on dynamic integration of coarse-grained services that are loosely coupled.

Mezini et al. (2000) proposed pluggable composite adapters for expressing component integration and component gluing. This creates a clean separation of customization code from application and framework implementations and thus results in better modularity, extensibility and maintainability. This work provides a potential strategy for dealing with component mismatches, which is currently ignored in our approach.

5. Conclusions

The web-based services paradigm has gained attention recently with the development of technologies such as SOAP (Seely and Sharkey, 2001). The benefits of such technologies has obvious advantages such as application sharing, reuse, and inter-operability between organizations. Services extend these benefits by providing facilities for on-the-fly integration and component introspection. In this paper, we described an approach for addressing component integration via the use of services in the context of Jini interconnection technology. Specifically, the approach utilizes synthesis to generate code necessary to realize component integration. To facilitate integration, the ACME ADL is used to specify both services and target applications, and is used a medium for performing service compatibility checking.

We are currently developing an environment that will assist in the creation of applications within the service-based paradigm and will support service browsing to facilitate application design. In addition, we are investigating approaches for allowing services to collaborate beyond the scope of a client application in order to create federated groups of services. Furthermore, we are developing technologies similar to the ones described in this paper in order to support service-based application within the .NET and web service frameworks.

Acknowledgements

G. Gannod is supported in part by NSF CAREER grant CCR-0133956.

References

Edwards, W.K., 1999. Core Jini. Prentice-Hall.

Fremantle, P., Weerawarana, S., Khalaf, R., 2002. Enterprise services. Commun. ACM 45 (10), 77–80.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Longman.

Gannod, G.C., Mudiam, S.V., Lindquist, T.E., 2000. An architecture-based approach for synthesizing and integrating adapters for legacy software. In: Proc. 7th Working Conf. Reverse Eng., IEEE, pp. 128–137.

Garlan, D., Monroe, R.T., Wile, D., 1997. Acme: an architecture description interchange language. In: Proc. CASCON'97, pp. 69–183.

Grundy, J., Mugridge, W., Hosking, J., 2000. Constructing component-based software engineering environments: issues and experiences. Inform. Software Tech. 42 (2).

Medvidovic, N., Taylor, R.N., 1997. Exploiting architectural style to develop a family of applications. IEE Proc. Software Eng. 144 (5–6), 237–248.

Meyer, B., 2001. .NET is coming. IEEE Comput. 34 (8), 92–97.

Mezini, M., Seiter, L., Lieberherr, K., 2000. Component integration with pluggable composite adapters. Software Archit. Comp. Technol..

Reiss, S.P., 1990. Connecting tools using message passing in field environment. IEEE Software 7 (7), 57–66.

Seely, S., Sharkey, K., 2001. SOAP: Cross Platform Web Services Development Using XML. Prentice-Hall.

Shaw, M., Garlan, D., 1996. Software Architectures: Perspectives on an Emerging Discipline. Prentice-Hall.

Stal, M., 2002. Web services: beyond component-based computing. Commun. ACM 45 (10), 71–76.

Urnes, T., Graham, T., 1999. Flexibly mapping synchronous groupware architectures to distributed implementations. In Proc. of Design, Specification and Verification of Interactive Systems.

Gerald C. Gannod is an Assistant Professor in the Department of Computer Science and Engineering at Arizona State University and is a recipient of a 2002 NSF CAREER Award. He received the M.S. (1994) and Ph.D. (1998) degrees in Computer Science from Michigan State University. His research interests include software product lines, software reverse engineering, formal methods for software development, software architecture, and software for embedded systems.

Sudhakiran V. Mudiam received the Ph.D. degree (2003) from Arizona State University and is a software architect with Aligo, Inc. He received an M.S. (1997) from the Indian Institute of Technology, Madras (Chennai), India. His research interests include software engineering, distributed and object-oriented systems, software design, software architecture, service-oriented software engineering, and Wireless Application platforms.

Timothy E. Lindquist is Professor and Chair in the Department of Electronics and Computer Engineering Technology at Arizona State University East Campus in Mesa, Arizona. He received the Ph.D. (1979) degree from Iowa State University. His research interests include software engineering, automated support for processes, distributed web-based applications, and distributed object computing.