Using a Formal Approach for Reverse Engineering and Design Recovery to Support Software Reuse

Final Report

Gerald C. Gannod, Ph.D.
Principle Investigator

Period Covered: 10/01/2001 - 09/30/2002

Arizona State University
Office of Research & Creative Studies
Box 871603
Tempe, AZ 85287-1603

Grant No. NAG-1-2241

1 Introduction

This document describes 3rd year accomplishments and summarizes overall project accomplishments. Included as attachments are all published papers from year three. Note that the budget for this project was discontinued after year two, but that a residual budget from year two allowed minimal continuance into year three.

2 Third Year Accomplishments

2.1 *Initial Investigations into Log-File Based Reverse Engineering.*

Phase 3 of the project was to involve the investigation of approaches for integrating formal and informal approaches for reverse engineering in order to support software reuse. While the third year budget was cut, we were able to use year 2 residuals to begin investigations in this area. Specifically, we began looking at an approach for combining the use of informal information such as log-file traces and structural diagrams with formal approaches such as model checking.

Software reverse engineering is defined to be a process of analyzing software components and their interrelationships in order obtain a description of the software at a high-level of abstraction. Approaches for reverse engineering have been suggested that address the reverse engineering problem from a wide variety of views ranging from structural makeup to observable behaviors using a wide variety of techniques covering both static and dynamic analysis.

A common activity that occurs within the software development community is the use of log files to generate traces of observed software behavior. Several different approaches for creating log files exist including the use of compiler directives at the time of development and post-development instrumentation.

As a resource for reverse engineering, a log file has the advantage of being an accurate account of software behavior. However, a disadvantage is that the log file potentially provides only a subset of possible behaviors of the software. As a result, to mitigate the risk of using log files as a source of information for whatever reason, approaches such as *adequate testing* are needed to ensure that a log file trace provides a reasonable or adequate amount of behavioral coverage.

Model checking approaches work by using exploration to determine whether certain temporal and safety conditions exist within the state space of some state-based model. Model checking, using tools such as Spin and SMV, has gained much attention recently due to many factors including the fact that it is relatively lightweight when compared to other formal approaches such as theorem proving. In this phase, we initiated investigations on approaches to reverse engineering that combine the use of model checking and log file analysis to reconstruct software architectures for the purpose of both understanding behavior and facilitating architecture level reuse.

2.2 *Service-Based reuse.*

During Phase 3 we also continued our investigations into the use of service-based software reuse as a means for implementing systems once individual components have been reverse engineered. Specifically, we developed approaches for achieving integration at run-time using adapted legacy components.

Service-based technologies are mechanisms whereby access to information and behavior is no longer constrained by traditional company and organizational boundaries. Web Services, for instance, employ broker-based middleware that use internet protocols such as HTTP and XML, allowing information and data to be provided by service providers and consumed by clients over the internet. A service-based development paradigm, or services model, 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. Applications built within this paradigm dynamically integrate services at runtime based on available resources which facilitates the creation of a federation of services (e.g., new "super-services") that can evolve over time.

Some of the defining characteristics of service-based technologies include modularity, availability, description, implementation-independence, and publication. 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 run-time (published).

This phase of the work involved the development of 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 whereby the "glue" code for both services and applications that make use of the services is automatically generated. The Jini interconnection technology is used as a broker for facilitating service registration, lookup, and integration at runtime.

3 Tools

Source to XML generator. As an add-on to previous tools, we developed a translator that converts an intermediate format into an XML-based representation called GXL. Further tools were planned that would have used this representation to implement formal specification generators from source code .

4 Deviations

+ Modified scope due to budget cut. The scope was significantly modified due to the elimination of the third year budget. Specifically, we were unable to complete tool construction, which would have provided a seamless environment for discovery of interfaces, adaption of components, and integration. This impacted primarily our ability to create sophisticated search capabilities.

5 Third Year Papers Published

[1] "Automated Support for Service-Based Software Development and Integration", (with Sudhakiran V. Mudiam and Timothy E. Lindquist), submitted to the Journal of Systems and Software – Special Issue on Automated Component-Based Software Engineering, Oct. 2002.

[2] "Using Log Files to Reconstruct State-Based Software Architectures", (with Shilpa Murthy), WCRE'02 Workshop on Software Architecture Reconstruction, September 2002.

[3] *"A Novel Service-Based Paradigm for Dynamic Component Integration", (with Sudhakiran V. Mudiam and Timothy E. Lindquist), in the AAAI-02 Workshop on Intelligent Service Integration, July 2002

6 Project Activities and Results

6.1 *Project Activities*

The following is a listing of all of the investigations that occurred during the funding period.

1. Service-Based Software Reuse
    a. Component Wrapping
    b. System Integration
2. Alternative Approaches to Reverse Engineering
    a. Graph-Based Approaches (Component Reuse)
    b. Log-File Based Approach (Architecture Reuse)
3. Tool Development
    a. Reverse Engineering Tools
        i. Component Wrapping Tools
        ii. Reverse Engineering Tools
            1. Dependency Graph Generator
            2. Code parser and Visual Browser
            3. C to XML Code Generator

6.2 *Findings*

The primary contribution of this research was the recognition of a move towards service-based software development approaches. Recent introduction of technologies such as .NET are an indication of the emergence of such work. The research performed under this grant directly addressed service-based issues from the standpoint of legacy system and component integration. Specifically, our work was aimed at providing infrastructures that support discovery and specification of components in order to support eventual reuse and integration of those components into new applications.

6.3 *Second Year Results*

6.3.1 Tool Development

During the second year of the project, construction of a preliminary version of the reverse engineering support tool was completed and supports the following features:

- Analysis of ANSI C programs
- Construction of graphical representations of source code
- Clustering of graph based representations of source code using a number of criteria

The tool development was partitioned into two separate projects. The first involved the development of a parser for the C program language. The second involved the development of a graphical user interface as well as the appropriate clustering routines.

6.3.1.1 Parsing

The parsing project involved the creation of a Java based parsing system that can be used to support the construction of reverse engineering tools and facilitate analysis of C source code modules. The parsing system was created using Metamata's parser generator JavaCC, JJTree (JavaCC's associated tree building tool), and the modified JavaCC provided C language grammar [1]. In addition, a second intermediate representation was developed that grouped language constructs in a manner similar to the grammar [3,4]. This was done to help provide for easier analysis, traversal, and display of the C Source by the reverse engineering tool. The parser system can be built using the tools and files given below. Once the parser is built, it can be used to parse input C source text files. Output generated by the parser will be two object files corresponding to the Abstract Syntax Tree for the source code and the Statement Class Hierarchy Representation (described further below and in API documentation for this module). These files are saved to disk with the names **<SourceFile>.ast** and **< SourceFile >.stm**, where **< SourceFile >.c** is the file being parsed.

6.3.1.2 Interface and Analysis

The analysis project involved the creation of a Java based reverse engineering system that can be used for the analysis of C source code modules. Based on a second intermediate representation generated by a JavaCC-based parser system, the reverse engineering system provides graphic multi-views for easier analysis, traversal and display of the C source codes. The system includes a source code browser, call graph viewer and global graph analyzer (with clustering capabilities). The graphical user interface was developed using Java/Swing components.

The system provides four patterns for clustering components within the global graph:

1. Clustering by file: the set of functions contained in the same source file are grouped together into one cluster.
2. Clustering by support library: software systems may use a number of functions that are accessed by the majority of its subsystems. This pattern groups such functions together into one subsystem.
3. Clustering by central dispatcher: systems commonly contain a small number of resources with a large out-degree. An example is a driver function that calls many other functions.
4. Clustering by Subgraph: This pattern looks for a particular type of subgraph in the system's graph.

6.3.2 Papers Published

The following sections describe a pair of papers that were published as a result of the sponsored project.

6.3.2.1 An Investigation into the Connectivity Properties of Source-Header Dependency Graphs

The task of understanding, modifying, and maintaining large systems can be intimidating and frustrating, especially in environments where staff turnover rates are high. The cognitive models used to tackle these tasks consist of top-down, bottom-up and hybrid techniques. Bottom-up approaches focus on the object of study with the perspective that direct source code analysis leads to formation of behavioral and structural abstractions. Top-down approaches focus on successive refinements of candidate designs with verification and validation of recovered designs against the implemented software artifacts. Hybrid approaches use a combination of both top-down and bottom up techniques in order to support a vertical analysis of a system.

Software artifacts are often organized at several different levels of discernible abstraction including files, procedures, and blocks. Each of these levels of abstraction present challenges to the maintainer. At the level of a file, the goal of the maintainer is to generate a high-level, informal model. At the level of a function and procedure, a maintainer is interested in constructing detailed structural models as well as determining the intended and actual behavior of the source code.

A *modularization* is a partitioning of a software system into components based on a variety of criteria, each depending on the approach and level of abstraction. Recent approaches have investigated the effectiveness of traditional clustering techniques and genetic algorithms for modularization at the file level of abstraction.

A *source-header dependency graph* is a bipartite graph that is formed by flattening include file dependencies and enumerating source file to header file dependencies. In this paper, we describe an approach for identifying candidate modularizations of software systems by analyzing connectivity properties of source-header dependency graphs. These modularizations provide information that can lead to coarse-grained reuse of source code and are a perfect complement to our earlier work.

6.3.2.2 A Suite of Tools for Facilitating Reverse Engineering Using Formal Methods

It is well-known that the maintenance of software is one of the most costly aspects of software development. As a program evolves, it becomes increasingly difficult to understand and reason about changes to source code. Eventually, if several changes to software are made without a corresponding modification of the software documentation, *reverse engineering* and design recovery techniques must be used in order to understand the current behavior of a system. Software reverse engineering is defined as the process of analyzing a subject system in order to identify components and component interrelationships and to create representations of that system in other forms or at other levels of abstraction. Several approaches to reverse engineering and design recovery have been suggested and include techniques for deriving structural abstractions and identifying plans embedded in code.

In our previous investigations, we described a formal technique for reverse engineering that involves two phases. The first phase constructs low-level, *as-built* specifications from program code. The specifications recovered from this phase are considered to be as-built since they describe a system as it was implemented rather than how the system was designed. The second phase of our investigations into reverse engineering involves the derivation of high-level abstractions from as-built specifications. By constructing high-level abstractions, several activities are facilitated, including high-level reasoning, program understanding, and software reuse.

One of the attractive properties of formal methods is that formal languages with well-defined syntax and semantics facilitate the construction and use of automated support tools. In addition, since manual application of formal techniques can be prone to errors, automated support is desirable. The size of non-trivial programs ranges from the tens of thousands to perhaps even millions of lines of code. Given the combined context of formal methods and program analysis, the construction of automated support tools is well-motivated. In this paper, we describe the development of several tools that have been designed to support a formal approach for reverse engineering. Much of this work was based on previous work that served as the precursor to the project funded under this grant.

6.4 *First Year Results*

6.4.1 Development of a Reverse Engineering Support Tool

Our primary efforts during the funding period have been directed towards the construction of a support tool for deriving formal specifications from program code. Specifically, we have defined requirements for the support tool, performed several design activities, and have commenced development. The purpose of this tool is to support two complementary reverse engineering activities. That is, the tool will support the construction of both informal and formal specifications from source code.

The informal component of the tool is intended to support techniques for generating graphical specifications of programs including the construction of data flow diagrams and call graphs using approaches such as Software Reflexion [5]. The formal component of the tool will support the approach developed by the PI for generating strongest postcondition specifications from program code [3].

The context for the reverse engineering support tool is shown in Figure 1, where an input program is analyzed by the tool along with assistance from the analyst to produce an output specification. The output specification can take many forms including a formal specification or an informal specification (e.g., diagrams). The analyst (user) can provide information at various points in the analysis via modification of source code annotations during formal analysis or by guiding the encapsulation/abstraction process during informal analysis.

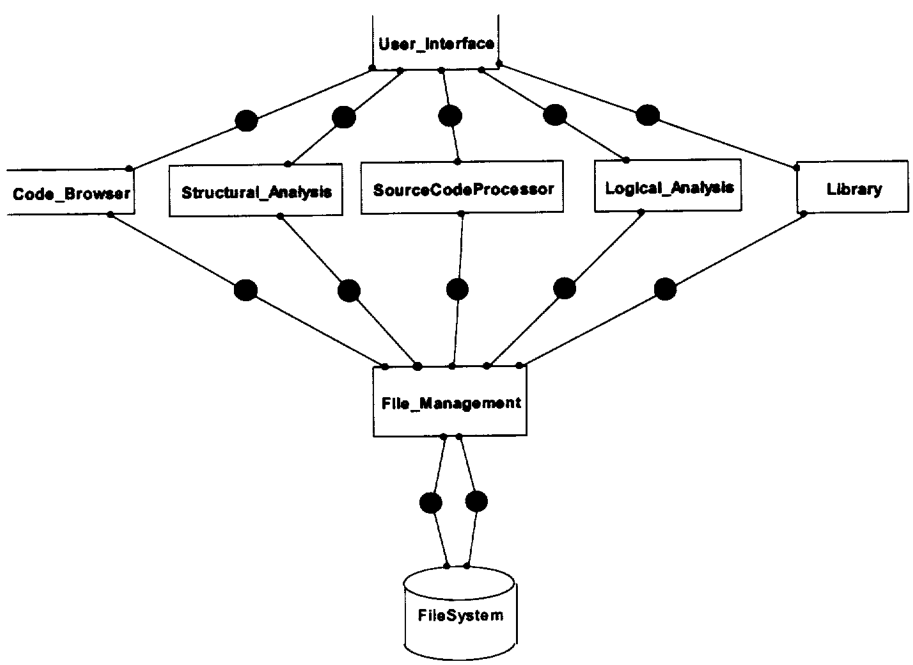


Figure 2 High-Level Architecture

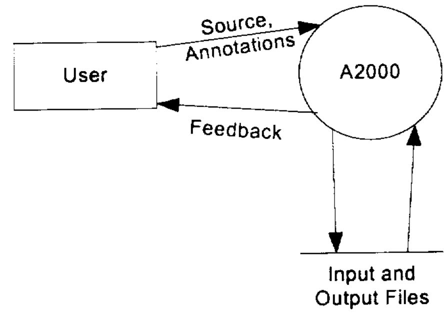


Figure 1 A2000 Context Diagram

The high-level architecture of the A2000 reverse engineering system is shown in Figure 2, where the system is decomposed using a layered architectural style. The user-interface component provides the interaction point with the user and the file management system represents an interface to a mechanism for managing input and output files. The primary challenges for constructing the support tool lies in the development of the Source Code Processor, Structural Analysis, and Logical Analysis components. Each of these components embodies the heart of the reverse engineering system as they are used to for front-end and back-end processing of source code.

6.4.2 An Architectural Based Approach for Synthesizing Wrappers

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

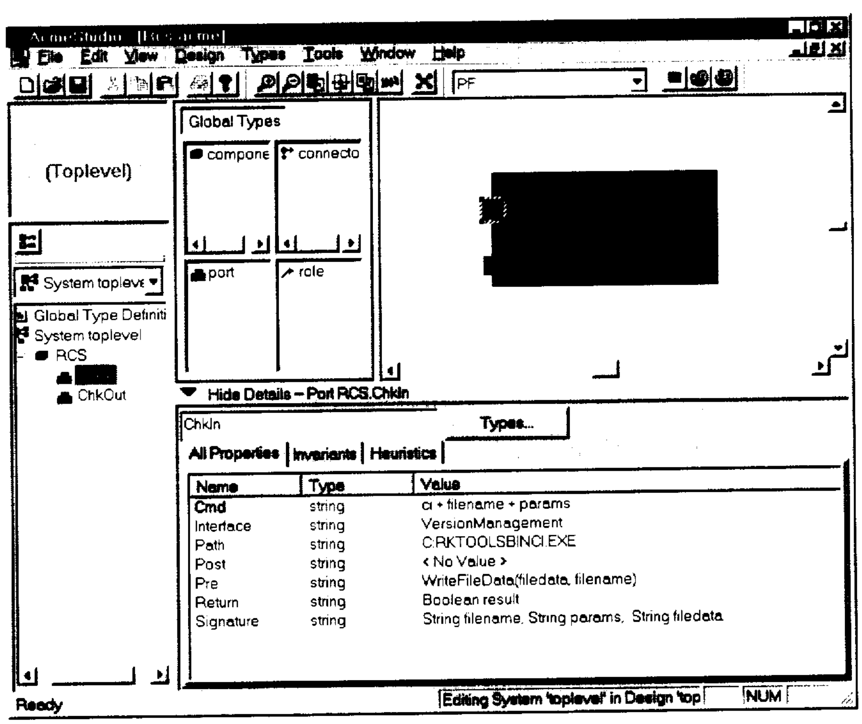


Figure 3 Specification of Legacy Component using AcmeStudio

migration of existing legacy assets to new platforms is via the use of the adapter design pattern, also known as component wrapping. We developed 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.

This work provides a complementary approach to the one to be supported by the work described in Section 1.1 for constructing software repositories that can be subsequently searched using formal and informal techniques. Specifically, the wrapping technique can be used to adapt and store existing components in libraries. The approach supports the use of the Jini interconnection technology and is supported by a tool that has been constructed using the AcmeLib toolkit [2]. Figure 3 depicts a session of the Acme Studio tool [1] that depicts the graphical specification of a legacy RCS component with two ports called ChkIn and ChkOut, which correspond to the programs "ci" and "co", respectively. By specifying these legacy programs as ports to a component, use of several programs as federated services is facilitated. In addition, by using a specification medium such as Acme, legacy components can integrated into the specification of new software architectures, thus achieving a form of software reuse.

7 Project Publications

[1] "Automated Support for Service-Based Software Development and Integration", (with Sudhakiran V. Mudiam and Timothy E. Lindquist), submitted to the Journal of Systems and Software – Special Issue on Automated Component-Based Software Engineering, Oct. 2002.

[2] "Using Log Files to Reconstruct State-Based Software Architectures", (with Shilpa Murthy), WCRE'02 Workshop on Software Architecture Reconstruction, September 2002.

[3] *"A Novel Service-Based Paradigm for Dynamic Component Integration", (with Sudhakiran V. Mudiam and Timothy E. Lindquist), in the AAAI-02 Workshop on Intelligent Service Integration, July 2002

[4] "An Investigation into the Connectivity Properties of Source-Header Dependency Graphs", Gerald C. Gannod and Barbara D. Gannod, to appear in Proceedings of the 8th Working Conference on Reverse Engineering, October 2001.

[5] "A Suite of Tools for Facilitating Reverse Engineering Using Formal Methods ," Gerald C. Gannod and Betty H.C. Cheng, in Proceedings of the 9th International Workshop on Program Comprehension, May 2001.

[6] "An Architecture-Based Approach for Synthesizing and Integrating Adapters for Legacy Software", G.C. Gannod, S.V. Mudiam, and T.E. Lindquist, in the Proceedings of the seventh IEEE Working Conference on Reverse Engineering, 2000.

ASSOCIATED PUBLICATIONS

A Novel Service-Based Paradigm for Dynamic Component Integration

Sudhakiran V. Mudiam and **Gerald C. Gannod**[*] [†]
Dept. of Computer Science & Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: {kiranmvs,gannod}@asu.edu

Timothy E. Lindquist
Dept. of Elect & Comp Engg Technology
Arizona State University - East
7001 E. Williams Field Road, Building 50
Mesa, AZ 85212
E-mail: tim@asu.edu

Abstract

We are developing a novel service-based paradigm for dynamic component integration that facilitates the creation of *Intelligent Services* from COTS (Commercial Off The Shelf) components, legacy components, and application frameworks by using techniques such as mediation and adaption or "wrapping". This framework supports the construction of applications by dynamically integrating these services at run-time based on available resources and allows for a federation of services that can evolve over time. As a part of the ongoing research effort we are utilizing an architecture based specification language that enables us to automate the process of creating these intelligent services.

Introduction

In the past, software reuse, and especially Component Based Software Engineering (CBSE) have been in the spotlight as various component technologies have matured. CBSE is much more than simply using object request brokers, setting up a library of useful code, or modular development. It involves building, acquiring, assembling and evolving systems. Traditionally, in CBSE technologies, several components are packaged together to create software systems. Emerging concerns include multiple suppliers of components that provide the same functionality, coping with multiple versions, and configuration of components. Currently, CBSE addresses issues and technologies such as Commercial-Off-The-Shelf (COTS) components, inbuilt components, and application frameworks. Emerging technologies of component integration include, Enterprise Java Beans (Thomas 1998), CORBA (Obj 1996), Jini (Richards 1999), Microsoft's DNA(Kirtland 1999), DCOM(Eddon & Eddon 1998) and IBM's San Francisco (Monday, Carey, & Dangler 1999). All these technologies provide a component model where a predefined infrastructure acts as "plumbing" that facilitates communication between components. Tools and environments supporting these technologies are being widely used in the industry and continue to provide many benefits.

We are developing a novel service-based paradigm for dynamic component integration that facilitates the creation of *Intelligent Services* from COTS (Commercial Off The Shelf) components, legacy components, and application frameworks by using techniques such as mediation and adaption or "wrapping". This framework supports the construction of applications by dynamically integrating these services at run-time based on available resources and allows for a federation of services that can evolve over time. As a part of the ongoing research effort we are utilizing an architecture based specification language that enables us to automate the process of creating these intelligent services (Gannod, Mudiam, & Lindquist 2000).

The remainder of this paper is organized as follows. The next section introduces our approach for application construction based on the use of intelligent services. An example using the approach is described in the following section, and the balance of the paper discusses related work, conclusions and future investigations.

A New Paradigm

We are developing an approach based on a new paradigm that looks at software reuse from a different perspective in which components are viewed as services available on a network. In this paradigm, components are dynamically composed into federations to make up an application (Mudiam 2000). The key features of our approach within this paradigm are:

- Components of varying granularity are bundled as "intelligent services" and made available on a network.
- The paradigm provides an integration framework, or middleware, that allows for the dynamic integration of these components (bundled as intelligent services) at run-time.

---

[*] Contact Author.

[†] This author supported in part by NSF CAREER Grant CCR-0133956.

- The paradigm facilitates the use of various patterns of interaction (architectural styles) between clients and intelligent services.
- The intelligent services provide a clear set of interfaces that are discovered dynamically at run-time and achieved using filters and adapters.

The process of component integration consists of the following steps.

1. Specification of the components as services.
2. Generation of the services along with the appropriate adapters and storing them to a repository.
3. Specification of a client to make use of services from the repository.
4. Generation of the client.
5. Execution of the client, performs the integration of the specified services at runtime.

Steps 1 and 2 are typically performed only once for each service while Steps 3 through 5 are performed as needed for each application. One of the primary goals of this work is the use of automatic synthesis to generate the source code necessary to achieve service integration. Our preliminary investigations have yielded an approach for generating wrappers of legacy applications for use as services (Gannod, Mudiam, & Lindquist 2000). In this work, a specification of a legacy software is created that defines the interface to a potential service. Second, the appropriate adapter source code is then synthesized based on the specification.

In order to specify how applications are to be integrated, we use the ACME architecture description language (ADL) (Garlan, Monroe, & Wile 1997) to describe both the available services as well as the client components that utilize those services. These specifications capture the interface of the components in such a way that the services can be regarded as black-boxes while remaining loosely coupled from implementation details.

Once the services are generated and stored in a repository, clients can make use of these services by identifying a general class of service that is required and the architectural style is necessary to properly interact with a client. This work addresses several issues that need to be considered during application specification and construction including:

- The specification of application architectures
- The specification of a *component type* in order to manage style interactions
- Graphical user interface integration including the use of shared graphical features

Again, using the ACME ADL, an application architecture is specified in order to describe the client composition and to identify the service classes that are needed by the application. Using this specification, the source code necessary to realize integration is generated (but is not bound to specific services). The remaining source code for the client, as with all applications, is provided by the user.

Integration of all services in our paradigm is performed with the execution of clients as each service becomes available. At first a client registers with a lookup service. Once services become available and join the network, the client is notified. The client integrates with the services by performing the GUI component integration as well as the service adapter integration and utilizes the service.

The research described above makes use of Jini (Richards 1999) technology to realize service integration. The components are wrapped as Jini services and we make use of the discovery and join mechanism to enable services join a Jini network.

Example

Figures 1 and 2 depict specifications of a client component and a number of services, respectively. In this example, the client consists of an editor that needs to utilize version management, printing, and compiling services. When the client runs, it joins the Jini network using Jini's discover and join protocol. As shown in the specification in Figure 1, the services that are required by the client are described as ports to which services will be bound.

```
Component Editor = {
  Properties {
    Part-of-client :
       string = "true";
    GUI-CodeFile :
       string = "ClientGUICode.java";
    Component-type :
       string = "Call Return";
  };
  Port VM_Port = {
    Properties {
       Port-type :
          string = "caller"; };
  };
  Port COMPILING_PORT = {
    Properties {
       Port-type :
          string = "caller"; };
  };
  Port PRINTING_PORT = {
    Properties {
       Port-type :
          string = "caller"; };
  }; };
```

Figure 1: Client Specification

The specifications of the services, as shown in Figure 2, describe each of the available services in the integration network. In this example we show only the specification of the RCS service and provide stubs for other services. Note that the other services would indeed be specified in the same manner as the RCS service. Each of the port specifications in a service component describe associated services. As such, the RCS service could potentially have sub-services for check in and check out.

```
Component RCS = {
Properties {
   Component-type :
   string = "Call Return";
};
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-type :
       string = "callee";
  }; }; };
Component Lzpr = {
  Properties {
   Component-type :
      string = "Call Return";
  };
  Port Print = {
    Properties {
     ......
  }; }; };
Component Javac = {
  Properties {
    Component-type :
       string = "Call Return";
  };
  Port Compile = {
    Properties {
     ......
  }; }; };
```

Figure 2: Services Specification

Initially, when no services are available, the client only has its editing GUI component as shown in Figure 3. However, since the client has access to the list of services that it needs, it utilizes the lookup services on the Jini network in order to complete service integration. As each service comes up and joins the Jini network, the client learns its existence and integrates them into the client. The Figure 4 shows the result of the integration once all the services have joined the network. In this example, we have a version management service called the *RCSService*, a compiling service called *JavacService* and a printing service called *lzprService*.

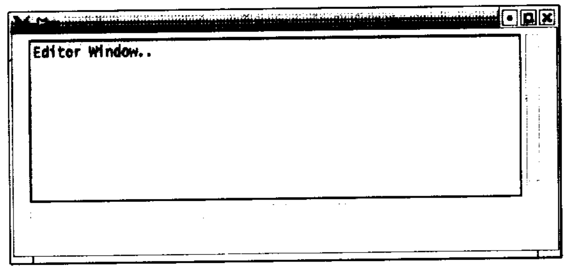


Figure 3: Initial Client

Related Work

Sullivan et al. look at systematic reuse of large-scale software components via static component integration (Sullivan *et al.* 1997). 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 (Dey *et al.* 1997) 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.

Conclusions

The availability of components on a network that provide services at run-time has many potential applications including the use of heterogeneous components executing in distributed environments. Enabling technologies such as Jini allow for dynamic component integration. We have described an architecture-based methodology of building components for reuse, and later using them with in a larger context to build applications.

We are currently building an environment for supporting the specification and synthesis of applications that utilize services through dynamic integration. The current state of this work includes tools for sythesizing both service and client integration code. We have used these tools to generate examples including the one described in this paper. Our future investigations include developing an approach for dynamically generating mediators to facilitate service interactions that bypass the need to communicate via client applications.

References

Dey, A. K.; Abowd, G.; Pinkerton, M.; and Wood, A. 1997. Cyberdesk: A framework for providing

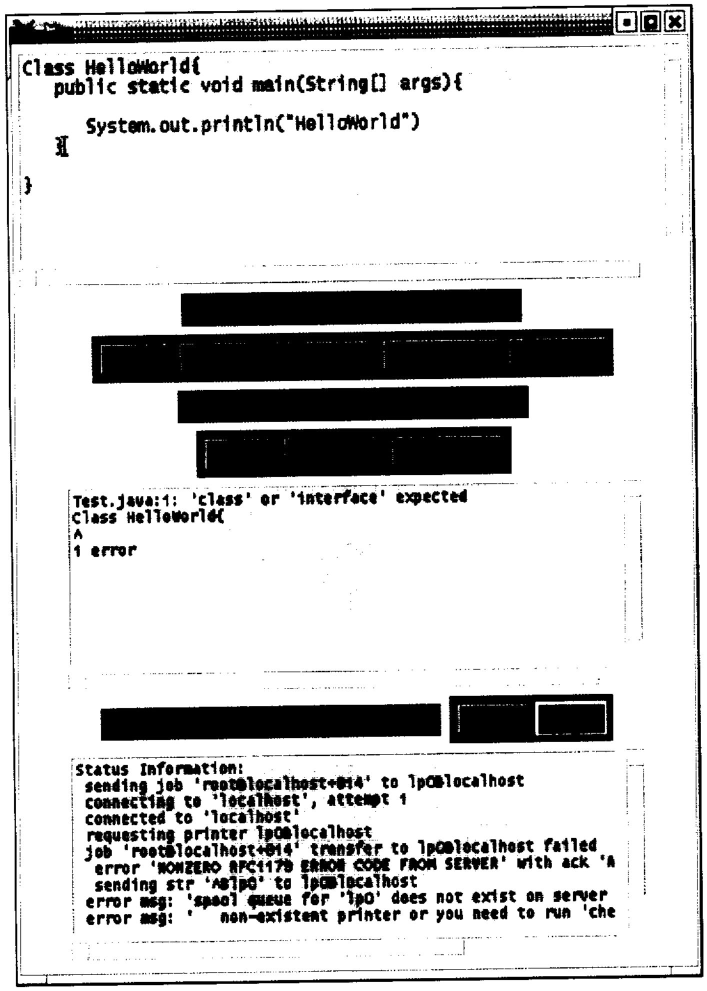


Figure 4: Client After Integration

self-integrating ubiquitous software services. Technical Report GIT-GVU-97-10, Georgia Tech.

Eddon, G., and Eddon, H. 1998. *Inside Distributed COM*. Microsoft Press.

Gannod, G. C.; Mudiam, S. V.; and Lindquist, T. E. 2000. An architecture-based approach for synthesizing and integrating adapters for legacy software. In *Seventh Working Conference in Reverse Engineering*, 128–137. IEEE Computer Society.

Garlan, D.; Monroe, R. T.; and Wile, D. 1997. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, 169–183.

Kirtland, M. 1999. *Designing Component-Based Applications: Build Enterprise Solutions with Microsoft Windows DNA*. Microsoft Press.

Monday, P.; Carey, J.; and Dangler, M. 1999. *San-Francisco Component Framework: An Introduction*. Addison-Wesley.

Mudiam, S. V. 2000. A novel service based paradigm for dynamic component integration. Ph.D. Proposal, Arizona State University.

Object Manangement Group. 1996. *CORBA: Architecture and Specification V2.0*, formal/97-02-25 edition.

Richards, W. K. 1999. *Core Jini*. Prentice-Hall.

Sullivan, K. J.; Cockrell, J.; Zhang, S.; and Coppit, D. 1997. Package oriented programming of engineering tools. In *Proceedings of the International Conference on Software Engineering*, 616–617.

Thomas, A. 1998. Enterprise java beans technology. Technical report, Patricia Seybold Group.

Using Log Files to Reconstruct State-Based Software Architectures *

Gerald C. Gannod[†‡]and Shilpa Murthy
Dept. of Computer Science & Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: {gannod, smurthy}@asu.edu

Abstract

A common activity that occurs within the software development community is the use of log files to generate traces of observed software behavior. As a resource for reverse engineering, a log file has the advantage of being an accurate account of software behavior. Model checking approaches work by using exploration to determine whether certain temporal and safety conditions exist within the state space of some state-based model. In this paper we describe an approach to reverse engineering that combines the use of model checking and log file analysis to reconstruct software architectures.

1 Introduction

Software reverse engineering is defined to be a process of analyzing software components and their interrelationships in order obtain a description of the software at a high-level of abstraction [1]. Approaches for reverse engineering have been suggested that address the reverse engineering problem from a wide variety of views ranging from structural makeup to observable behaviors using a wide variety of techniques covering both static and dynamic analysis.

A common activity that occurs within the software development community is the use of log files to generate traces of observed software behavior. Several different approaches for creating log files exist including the use of compiler directives at the time of development and post-development instrumentation [2].

As a resource for reverse engineering, a log file has the advantage of being an accurate account of software behavior. However, a disadvantage is that the log file potentially provides only a subset of possible behaviors of the software. As a result, to mitigate the risk of using log files as a source of information for whatever reason, approaches such as adequate testing [3] are needed to ensure that a log file trace provides a reasonable or adequate amount of behavioral coverage.

Model checking approaches work by using exploration to determine whether certain temporal and safety conditions exist within the state space of some state-based model. Model checking, using tools such as Spin and SMV, has gained much attention recently due to many factors including the fact that it is relatively lightweight when compared to other formal approaches such as theorem proving.

In this paper we describe an approach to reverse engineering that combines the use of model checking and log file analysis to reconstruct software architectures. The remainder of this paper is organized as follows. Our approach to reverse engineering is presented in Section 2. Section 3 presents related investigations and Section 4 draws conclusions and outlines future investigations.

2 Approach

This section describes the underlying conceptual basis as well as the reverse engineering approach being used to reconstruct software architectures.

2.1 Underlying Conceptual Basis

Figure 1 depicts the general context for a software development approach that involves the use of model development, model checking, and log file generation (or other debugging approaches). In the general context, the relationship between models and programs is labeled as *implements-models*. The semantics of this relationship emphasizes the fact that the model is implemented by a program and that a program is modeled by a model.

Model checking tools such as Spin [4] facilitate verification of the ordering of events within a model. In Spin the verification is achieved using linear temporal logic (LTL) while SMV [5] achieves this using computational tree logic (CTL). Here we assume the use of Spin and LTL. The *maps-to* relationship between log files and LTL specifications emphasizes the analogy between log files and LTL specifications. As with the relationship between models

*This research supported by NASA Langley Grant NAG-1-2241.
†Contact Author.
‡This author supported in part by NSF CAREER Grant CCR-0133956.

and programs, there is a difference in the level of abstraction between log files and LTL specifications.

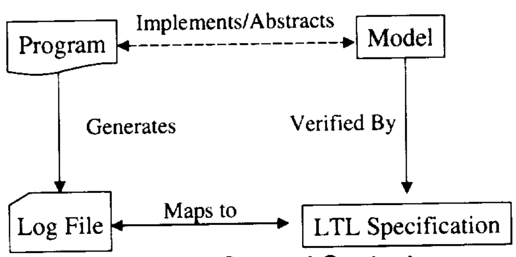
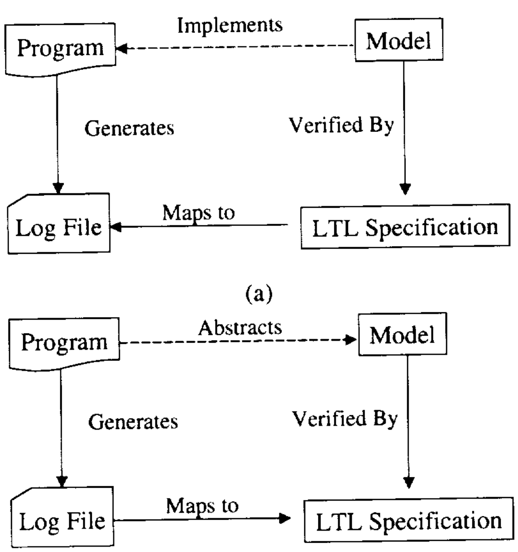


Figure 1. General Context

Andrews' developed an approach to software testing based on the concept that given a state-based model such as a finite automata or statechart, a program implementing the model must generate behaviors that correspond directly to events in the model [6]. While the approach does not use a specific model checking environment like Spin or SMV, the employed verification technique used does achieve similar goals, e.g., verification of the model by using log file events to drive model execution. As a result, a situation arises as shown in Figure 2(a), where a model is used to aid in the development of a program, and a log file is generated during program execution. In this case, the model is assumed to be correct and thus the verification against an LTL specification ignored. If the log file events do not correspond to model events, then the program is considered faulty and must be modified.



(a)

(b)

Figure 2. (a) Testing Context (b) Reverse Engineering Context

An interesting result can be derived if the relationships between program/model and logfile/LTL specification are reversed as shown in Figure 2(b). Namely, it provides a means for verifying correspondence between existing systems and models reconstructed using reverse engineering. That is, assuming that the program is correct (e.g., we are interested in learning what the program does rather than verifying it against requirements) then any log file produced is an accurate representation of what happens during execution. Therefore, any model constructed to represent the program behavior must at some level generate the events found in a log file with the same temporal orderings, assuming a well-defined logging policy.

2.2 Process

We are currently developing a reverse engineering approach based on the context provided in Figure 2(b). The approach has four primary steps that are necessary to be successful. In Step 1, log files must be generated from the program. This step, in some cases, is already completed since a common activity for some developers is to generate traces as an aid to debugging. However, there is a risk in using such traces since the logs can be inconsistently updated (e.g., not all events are captured). As a result, it is necessary to develop logging tools that follow specific logging policies on *what to log* and *when to log*.

In Step 2, a model must be reconstructed from source code and other sources of information. In our technique, we assume that the system of study utilizes a state-based architectural style and that we are interested in deriving high-level state charts as well as some intermediate models. Several approaches have been suggested for deriving state models from code and other artifacts including the work by Systa [7]. In our approach, model construction can be free form in the sense that it does not have to be strictly bottom-up. In fact, the model can be constructed top-down with little or no direct code analysis. As long as the verification step ends in a valid model, the approach used for deriving the model is unimportant.

Step 3 involves the mapping of log file events to LTL or CTL specifications, with the choice depending on the tool used to perform model checking. The mapping of log file events to LTL propositions requires the use of event abstraction techniques [9] especially since many events in a log file may be decompositions of some higher level event that is captured in a model. We are developing an approach that will work by partitioning log files into disjoint equivalence classes of events and mapping a representative of a partition to specific events contained in a model. The result preserves ordering while relaxing a need for exact event mapping.

Finally, in Step 4, model checking is used to determine whether the sequence of events captured in a log file and encoded with a LTL specification are consistent with the candidate models developed in Step 2. When an invalid verification occurs the conclusion that can be drawn is that either the model is incorrect, the encoding of the LTL specification is incorrect, or both. In the case of an incorrect model, modification and refinement of the model becomes necessary. In regards to the correctness of an LTL specification, as long as the generated specification preserves ordering located in the log file *and* the propositions correspond to events in the model, the verification will provide accurate results.

Upon completing the four steps, what an analyst would gain is a state-based architectural and design view of a system that has been verified against observable behavior exhibited by the system of study. Using the evaluation dimensions defined by Gannod and Cheng [8], the abstraction *distance* and *traceability* of the model from the original source code is influenced entirely by the approach used to perform Step 2 rather than by the underlying reverse engineering framework described here. In terms of *accuracy*, the verification step ensures that the model is accurate with respect to generated log files. However, the completeness of the verification depends on adequacy criteria used to determine how many log files to generate. The *precision* of the approach is considered to be high since the representation used to encode the models must be formal enough to employ model checking techniques.

3 Related Work

Andrews [6] provides a framework for automatically analyzing log files using state machines in the context of testing. The Log Files are created using logging policies and a standard format which specifies certain keywords. A log file analyzer is specified formally using the *Log File Analysis Language* (LFAL) and is built as a set of parallel state machines with each log file machine checking one thread of event.

Basten investigated the use of event abstraction to reduce the apparent complexity of distributed computations [9]. The work is based on the study of four aspects of event abstraction: a model describing primitive behavior, a formalism for specifying abstract behavior in terms of activity and causality, abstract descriptions of behavior, and verification of primitive or abstract descriptions against specified behavior.

Systa [7] discusses an experimental environment for reverse engineering Java software with respect to the concepts of static and dynamic views. These views contain overlapping information about software artifacts and their relations, which form a connection for information exchange between the views. Static information extracted from Java class files is viewed using the Rigi environment [10]. The dynamic information generated by running the software under a debugger is viewed as scenario diagrams using the SCED prototype tool. SCED state diagrams can be synthesized from these scenario diagrams. The SCED scenario diagrams are further used for slicing the Rigi view and the Rigi view in turn is used to guide the generation of SCED scenario diagrams and for raising their level of abstraction.

4 Conclusions and Future Investigations

Our initial experience with the approach has shown a great deal of promise with some moderate risks in each of the process steps. An ability to capture appropriate event information (Step 1) and map those events (Step 3) are especially crucial to the success of the approach since inaccuracies in these steps reduce the impact of the verification step.

To date, we have completed the construction of a logging tool that is based on the use of the Java Debbugging Platform [11] in order to log events in Java programs without resorting to behavior modification as is possible with code instrumentation techniques. We are currently developing tools to support the derivation of LTL specifications from log files using event abstraction techniques [9].

References

[1] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[2] Kevin Templer and Clinton L. Jeffery. A Configurable Automatic Instrumentation Tool for ANSI C. In *Proc. of the Automated Software Engineering Conference*, 1998.

[3] Elaine J. Weyuker. The Evaluation of Program-Based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[4] Gerard Holzmann. The Spin Model Checker. *Transactions on Software Engineering*, 23(5):279–295, May 1997.

[5] K. L. McMillan. Getting started with smv. Cadence Berkeley Labs, March 1999.

[6] James H. Andrews. Testing using Log File Analysis: Tools, Methods and Issues. In *Proc. of 13th Annual International Conference on Automated Software Engineering (ASE'98)*, pages 157–166, Honolulu, Hawaii, October 1998.

[7] Tarja Systa. On the Relationship between Static and Dynamic Models in Reverse Engineering Java Software. In *Proc. of the 6th Working Conf. on Reverse Engineering (WCRE99)*, pages 304–313, 1999.

[8] Gerald C. Gannod and Betty H. C. Cheng. A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques. In *Proc. of the 6th Working Conf. on Reverse Engineering*. IEEE, October 1999.

[9] Twan Basten. Event Abstraction in Modeling Distributed Computations. In *K. Ecker and M. Krmer, editors, Workshop on Parallel Processing, Proc.*, pages 46–65, Lessach, Austria, September 1993.

[10] Scott R. Tilley, Kenney Wong, Margaret-Anne Storey, and Hausi A. Müller. Programmable Reverse Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[11] The Java Platform Debugger Architecture(jpda). [Online Available] http://java.sun.com/products/jpda.

Automated Support for Service-Based Software Development and Integration

Sudhakiran V. Mudiam and Gerald C. Gannod[1 2]
Dept. of Computer Science & Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: {kiranmvs,gannod}@asu.edu

Timothy E. Lindquist
Dept. of Electronics and Computer
Engineering Technology
Arizona State University - East
7001 E. Williams Field Road, Building 50
Mesa, AZ 85212
E-mail: tim@asu.edu

[1]Contact Author.
[2]This author supported in part by NSF CAREER Grant CCR-0133956.

Abstract

A service-based development paradigm 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. Applications built within 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. The technique utilizes an architecture description language to describe services and achieves run-time integration using current middleware technology.

1 Introduction

Service-based technologies are mechanisms whereby access to information and behavior is no longer constrained by traditional company and organizational boundaries. *Web Services*, for instance, employ broker-based middleware that use internet protocols such as HTTP and XML [Stal, 2002], allowing information and data to be provided by service providers and consumed by clients over the internet. 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. Applications built within this paradigm dynamically integrate services at runtime based on available resources which facilitates the creation of a federation of services (e.g., new "super-services") that can evolve over time.

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 run-time (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] whereby the "glue" code for both services and applications that make use of the services is automatically generated. The Jini interconnection technology [Richards, 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 reengineering, software architecture and the middleware technology we are using to enable dynamic integration (i.e., Jini [Richards, 1999]). The proposed approach for constructing services and developing service-based applications is presented in Section 3. An example demonstrating the proposed approach is described in Section 4, Section 5 discusses related work, and Section 6 draws conclusions and suggests further investigations.

2 Background

This section describes background material on specific patterns supported, software architecture and the Jini Interconnection Technology.

2.1 The Proxy Pattern

Proxy Pattern [Gamma et al., 1995] 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 access to the realSubject from the client have to go thro the Proxy. In most cases the client is not even aware of the proxy object and it assumes that it is directly talking to the realSubject. Figure 1 shows the interaction between a client and a realSubject via a proxy. In the approach described in this paper, the proxy pattern plays a central role in the definition of services as well as the realization of service integration.

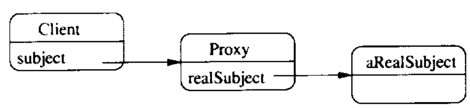


Figure 1: Proxy Pattern [Gamma et al., 1995]

2.2 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 [Chikofsky and Cross, 1990]. 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* [Gamma et al., 1995] 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 Figure 2.

2.3 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

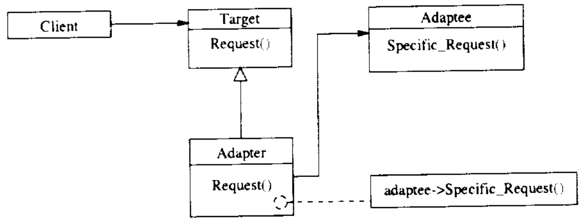


Figure 2: Object Adapter [Gamma et al., 1995]

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 ways 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 [Medvidovic and Taylor, 1997]. 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. There are several architecture description languages that allow the user to describe the architecture of a system.

2.4 Jini

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 3 shows the layered architecture of the Jini Interconnection 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 or "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 is stored by the lookup service. The proxy (often implemented with an adapter) can later be transported to the clients of the service. Once registered, other network members can discover the availability of the device (e.g., service) through the lookup service. When a client application finds an appropriate device, the lookup service sets up 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. We are using Jini technology outside the scope of its original intention of being able to allow devices to interact with one another by using the discover and join protocols supported by this technology to implement service-based software development.

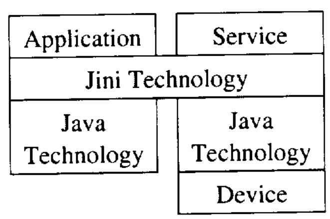


Figure 3: Jini Architecture

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. Another advantage arises from the fact that services must implement a fairly general set of routines in order to participate in a Jini network. 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. We have implemented our framework on Jini for a number of reasons including the fact that it allows for the creation of federations of services. More importantly, an advantage of using Jini over the emerging Web services technologies is that many of the current and future capabilities of the web services infrastructure are already supported by Jini, thus providing a view into forthcoming capabilities of web service-based software development without needing to wait for the completion of development of web service support technology.

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 Overview

The approach described in this paper follows a services-based model for software development. The methodology that we have developed follows closely the model suggested by Stal [Stal, 2002] for web services, although the technology that we are using to realize our approach is Jini [Richards, 1999]. It is our belief that the technique is applicable to the web services domain and that translation into that technology would require little change at the conceptual level and a modest amount of change at the technology level.

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 the components as services
2. Generation of the services using proxies via the construction of appropriate adapters and glue code. These services are consequently registered and made available on a network.

In the context of this paper, we address the creation of command-line applications [Gannod et al., 2000] as services. As part of our ongoing research work we are investigating approaches for integrating a wide-variety of services including web services and .NET components.

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 the 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 runtime

Figure 4 shows a diagram depicting the basic architecture for applications created within the infrastructure described in this paper. Structurally the architecture uses a layered style with a layer of services accessed via a layer of proxies. 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 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. At run-time, the services are bound to the client component using the Jini lookup service.

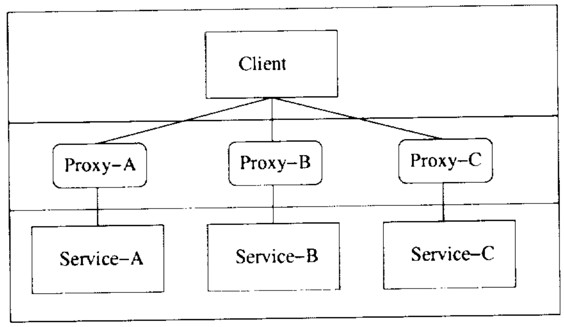


Figure 4: Basic Architecture

3.2 Service Generation

In order to generate services from legacy components, we take the approach of wrapping the components by utilizing the interface provided by the component. For example, command-line applications have a well defined interface. The interface is based entirely on the knowledge of how these applications are used rather than how they work.

3.2.1 Specification and Synthesis

The concept of using an adapter for wrapping legacy software is not a new one [Gamma et al., 1995, Sneed, 1996, Cimitile et al., 1998, Jacobsen and Kramer, 1998]. 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.

3.2.2 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.

(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.

(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.

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 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. 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 post-conditions 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 file names and command-line options.

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" (encoded as an *Interface* property) is used to support a service lookup, which is later utilized during application integration.

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	Does this specify the 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

Table 1: Properties

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 Interconnection Technology [Richards, 1999].

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]. 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 [Acme Studio, 2002], that facilitates graphical construction and manipulation of software architectures.

Figure 5 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.

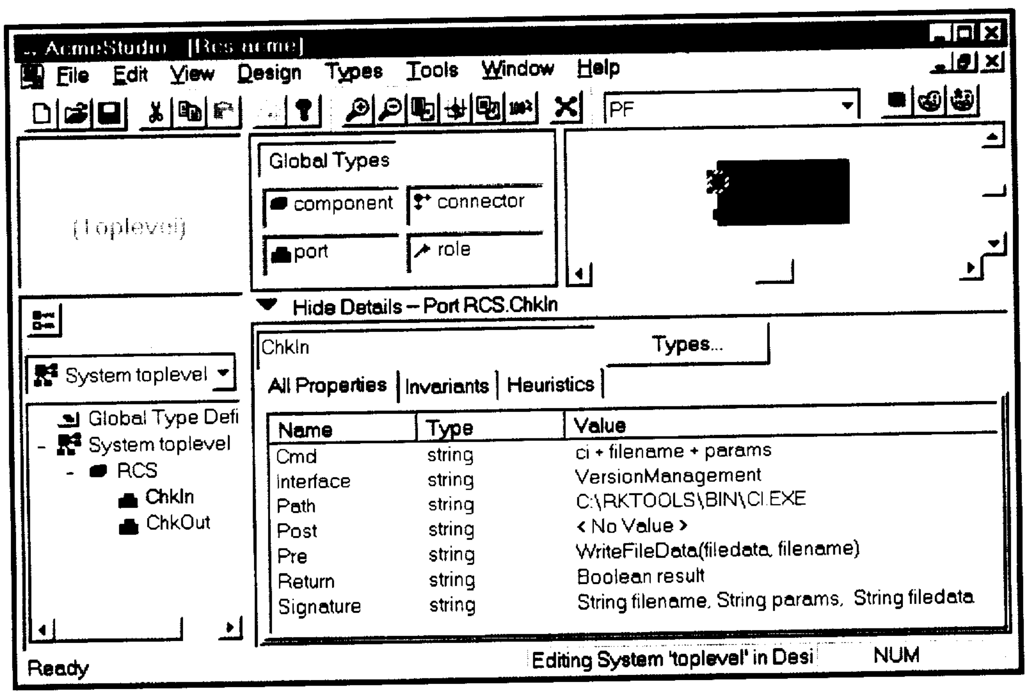


Figure 5: ACMEStudio Session

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 4, 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 3.5.1).

3.2.3 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 [Richards, 1999] 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 defined by Jini Interconnection Technology.

In our approach, the information that is needed to generate wrappers corresponds exactly to the properties associated with the ports shown in Figure 6. In addition to the Signature, Command (shown as `Cmd` in Figure 6), 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 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.

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 appli-

```
Component RCS = {
  Properties {
    Component-type : string = "Call Return";
  };
  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-type : string = "callee";
      Shared-GUI: string = "true";
  }; }; };
Component Lzpr = {
  Properties {
    Component-type : string = "Call Return";
  };
  Port Print = {
    Properties {
     ......
    }; }; };
Component Javac = {
  Properties {
    Component-type : string = "Call Return";
  };
  Port Compile = {
    Properties {
     ......
    }; }; };
```

Figure 6: ACME Spec: Services Section

cations. 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.

3.3 Client Generation

Once the services are generated and stored in a repository, a client application can be architect-ed. 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.

3.3.1 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.

(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, architectural mismatches can be determined by stating the assumption that a component is making regarding style.

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.

3.3.2 Synthesis

The second stage of our approach involves the synthesis of application code. Figure 7 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 Figure 8 are used to determine the relationships between client applications and desired services. Second, information regarding GUI's provided by services is used to determine how to realize the GUI in a client application.

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. Figure 9 shows a piece of code (e.g., POC) for the client template and depicts the replacement points in with in "⟨⟩" tags. Given the nature of the code being generated (e.g., code used to realize integration between services and clients that utilize those services), we have found that this approach allows the developer to focus primarily upon client operation issues while delegating connector and integration concerns to automated tools.

While a template-based approach lacks generality and freedom necessary to synthesize complex connector and integration code, the very nature of service-based components dictates a level of interaction that promotes loose coupling between components. In addition, at the core of service-based applications is the

```
Component Editor = {
  Properties {
     Part-of-client : string = "true";
     GUI-CodeFile : string = "ClientGUICode.java";
     Component-type : string = "Call Return";
     Shared-GUI: string = "true";
  };
  Port VM_Port = {
     Properties {
        Port-type : string = "caller";
        Interface : string = "VersionManagement" ;
  }; };
  Port COMPILING_PORT = {
     Properties { Port-type : string = "caller"; ..};
  };
  Port PRINTING_PORT = {
     Properties { Port-type : string = "caller"; ..};
   }; };
```

Figure 7: ACME Spec: Client Section

notion of common interfaces between clients and services. As a result, code that implements connection and integration operations can be easily parameterized, further leading towards a template-based software synthesis approach.

3.4 Integration and Runtime Considerations

3.4.1 Architectural Style Verification

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 criteria whereby components can only be connected to components of the same type. For example, a *Call Return* component can only be connected to other *Call Return* components through a *Call Return* connector. Our future investigations include expanding the verifier to allow for mappings of other types, where appropriate.

3.4.2 Client Generation

A client can be generated once the *Arch Verifier* validates the architecture. A synthesis tool called *Generator* makes use of a client template similar to the one shown in Figure 9 and fills the appropriate sections with the list of services (indicated by the ⟨`List of Services`⟩ tag) that are to be integrated with the client and the user-defined GUI components (indicated by the ⟨`GUI-CodeFile`⟩ tag) of the client. Typically, the developer of these client GUI components by the can assume that certain service classes provide common GUI facilities. However, the actual names of shared and exclusive GUI components need not be known by a client.

```
Connector checking_out = {
  Properties { Connector-type : string = "Call Return"; };
  Role caller = {
     Properties { Prop-type : string = "output"; };
  };
  Role callee = {
     Properties { Prop-type : string = "input"; };
  }; };
Connector checking_in = {
  Properties { Connector-type : string = "Call Return"; };
  Role caller = {
     Properties { Prop-type : string = "output"; };
  };
  Role callee = {
     Properties { Prop-type : string = "input"; };
  }; };
Connector compiling = {
   Properties { connector-type : string = "Call Return"; };
   Role callee = {
      Properties { Prop-type : string = "input"; };
   };
   Role caller = {
      Properties { Prop-type : string = "output"; };
   }; };
Connector printing = {
  Properties { Connector-type : string = "Call Return"; };
  Role callee = {
     Properties { Prop-type : string = "input"; };
  };
  Role caller = {
    Properties { Prop-type : string = "output"; };
  }; };
Attachments {
   Editor.VM_Port to checking_in.caller;
   Editor.VM_Port to checking_out.caller;
   Editor.COMPILING_PORT to compiling.caller;
   Javac.Compile to compiling.callee;
   Editor.PRINTING_PORT to printing.caller;
   Lzpr.Print to printing.callee;
   Rcs.ChkOut to checking_out.callee;
   Rcs.ChkIn to checking_in.callee;
};
```

Figure 8: ACME Spec: Connector Section

3.4.3 GUI Integration

Since many services can be GUI-bound, it is necessary to provide a facility for integrating the GUI components. In our framework, the adapters for the various services (as shown in Figure 14) 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 "CodeWindow" or "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 adapters are used as is but may interact with one or more of the shared components. For both shared and exclusive

```
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 on the network
   //and get the adapter for the service.
   .....
  }
  private static boolean alreadyAdded(String key)
  {
   //check if a shared component
   // with key has been added.
   .....
  }
  private static boolean inMyList(String serviceName)
  {
   // verify if the serviceName is in the
   // list of services.
   .....
  }
}
```

Figure 9: Client Template POC

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

3.5 Implementation

In this section we discuss the implementation aspects of the service generation and the client generation using tools that we have built.

3.5.1 Service Generation

To support our technique for constructing wrappers for legacy software, we have created a Java support tool called *ServiceTool*. Figure 10 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 6 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 *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.

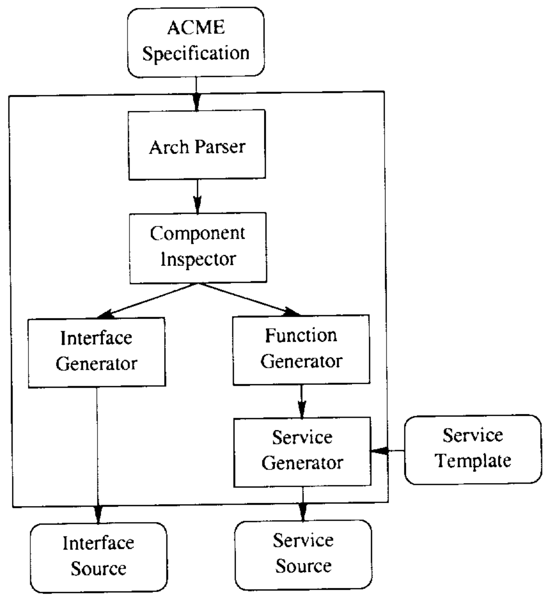


Figure 10: Service Tool Architecture

The *ArchParser* uses the ACMEParser from the ACMELib toolkit [Acme Lib, 1997] 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 [Luckham and Vera, 1995] and Wright [Allen and Garlan, 1997]) 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 11 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 11 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 `ReadFileData()` and `WriteFileData()` routines (shown in Figure 12), are available as functions for use within the Java code to provide standard read-from and write-to file support, respectively.

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

Figure 11: Excerpt of the ServiceTemplate

```
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(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 12: Sample Library Routines

3.5.2 Client Generation

To support the construction of client applications, we have created a support tool written in Java called *ClientGenTool*. Figure 13 shows the detailed architecture of the *ClientGenTool* which takes an Acme architecture (specification) and produces a Client source. In this figure, the rectangles with the square corners represent software components while the rectangles with the rounded corners represent data stores or repositories.

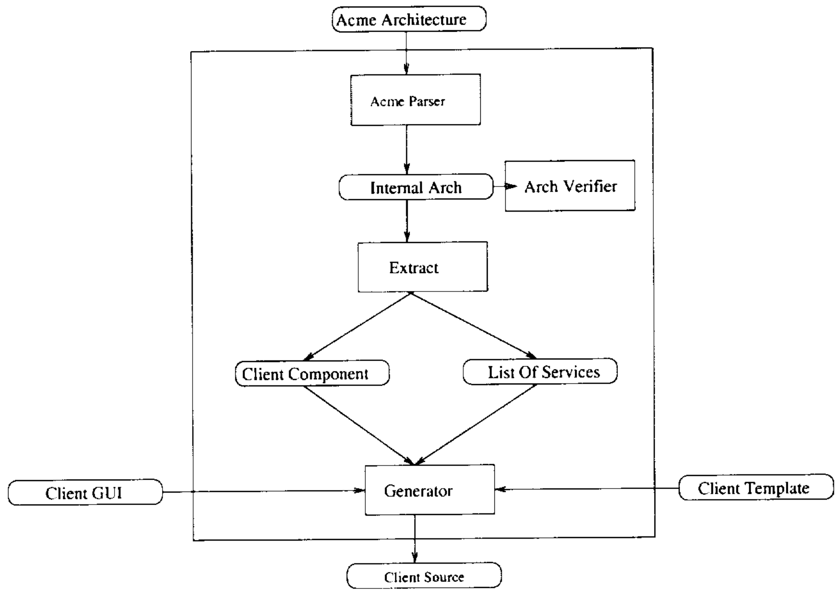


Figure 13: ClientGen Tool Architecture

The *ArchParser* reads in ACME architecture specifications similar to the ones shown in Figures 6, 7 and 8 and builds an internal model of the architecture *(Internal Arch)*. The *ArchParser* uses the ACMEParser from the ACMELib toolkit [Acme Lib, 1997] 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*, *Client GUI*, and the *List of Services* from the *Internal Arch* model. All these are used by the *Generator* component along with *Client Template* to produce the *Client Source*.

4 Examples

To demonstrate the use of the integration technique described in this paper, we have created two examples, a simple editing application and a simple network monitoring example.

The intent of the simple editing application is to support text file editing with version control, compilation and printing facilities. The intent of the simple network monitoring example is to monitor various servers on a network by creating a consolidated console using simple administration tools running on the servers. All the services were generated using *ServiceGenTool* [Gannod et al., 2000] while the client applications were generated using *ClientGenTool*.

4.1 Editing Application

Figure 14 shows the conceptual architecture of the editing application. The client application interacts with services via each of their respective adapters. The adapters provide GUI components that are integrated by the client to render the client application. These GUI components may interact with components from other services (e.g., they may be shared or exclusive). If a GUI component is exclusive, it must be integrated directly by the client as required by the service. If the GUI component is shared, it may be used only if a similar component does not already exist. For example, the RCS service adapter provides a shared editing panel component named "CodeWindow". If the client already has a "CodeWindow" component, the shared component from the RCS service adapter is not used.

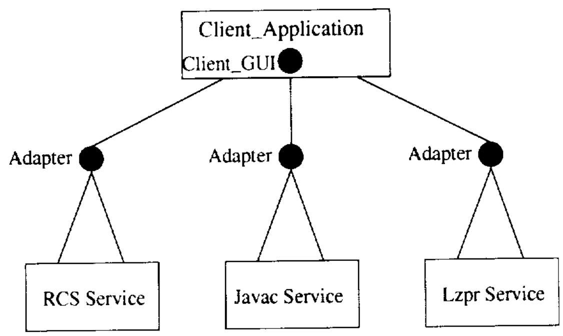


Figure 14: Conceptual Architecture: Editing Application

4.1.1 Specification

Figures 6-8 contain portions of the specifications used to describe and generate the editor application. As shown in Figure 7, the Editor component is a *Call Return* component, as indicated by the *Component-Type* property. The Editor component has three ports, namely `VM_Port`, `COMPILING_port`, and `PRINTING_port`. These ports correspond to the services that the editor seeks to bind with at run-time.

The connectors `checking_out`, `checking_in`, `printing`, `compiling`, shown in Figure 8 connect the editor to the appropriate version management, printing, and Java compilation components, respectively. The *Connector-Type* for these connectors must match the *Component-Type* property of the two components it connects. The attachments section in the specification (shown in Figure 8) describes the attachments between roles and ports. While the specification shown in this examples identifies specific services, attachments can actually be made to generic components that identify a class of services.

4.1.2 Client Application

The *ClientGenTool* generates the client using specifications similar to the ones described above. Once generated, the client becomes the focal point for the integration of the various services. Specifically, at run-time the client joins a Jini network and looks for services with which to integrate. Once found, the services are bound via an adapter and the GUIs are integrated along with the GUI provided by the client. Figure 15 depicts the editing application as a result of executing the client side code. In this case, the client has been executed on a Jini network that does not have any of the desired services available.

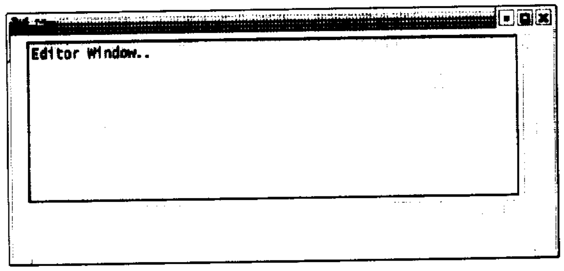


Figure 15: Client by itself

4.1.3 Integration and Execution

As each service is executed it joins a Jini network, the client learns of its existence and integrates with it by interacting with the corresponding adapter. The service adapter provides GUI components that can be either exclusive or shared. In the case of exclusive components, they are used as is. In the case of shared components, they may be used in lieu of another similar component. Figure 16 shows the result of the integration once all the services have joined the Jini network.

Once all the services have been integrated, the application appears as shown in Figure 16. In the diagram, the top text pane is the original editing window provided (and shared) by the client. The buttons directly below the top text pane are provided by the RCS service and the Javac service, respectively. The middle and bottom text panes are exclusive GUI components needed by the Javac and Lzpr services.

Getting the client application to integrate with the set of services requires that the Jini network must be up and running along with an associated lookup service. Since Jini is build on top of Java RMI, an RMI daemon must be running prior to execution of the lookup service. Finally a web server must be running in order to enable code delivery across the Jini network.

4.2 Network Monitoring Example

Figure 17 shows the conceptual architecture of the network monitoring application. Similar to the simple editing application, the client application here interacts with services via each of their respective adapters, each of which provide GUI components that are integrated to render the client application. This application integrates the following services: TopService, UpTimeService and TcpDumpService. The TopService provides the top CPU processes information using the *top* command. The UpTimeService provides information about how long a system has been up along with the load average using the *uptime* command. The TcpDumpService provides a dump of the traffic on network using the *tcpdump* command. Each of these services executes on different machines on a network. The network monitoring client integrates these services and provides a unified client that then enables a user to monitor the machines.

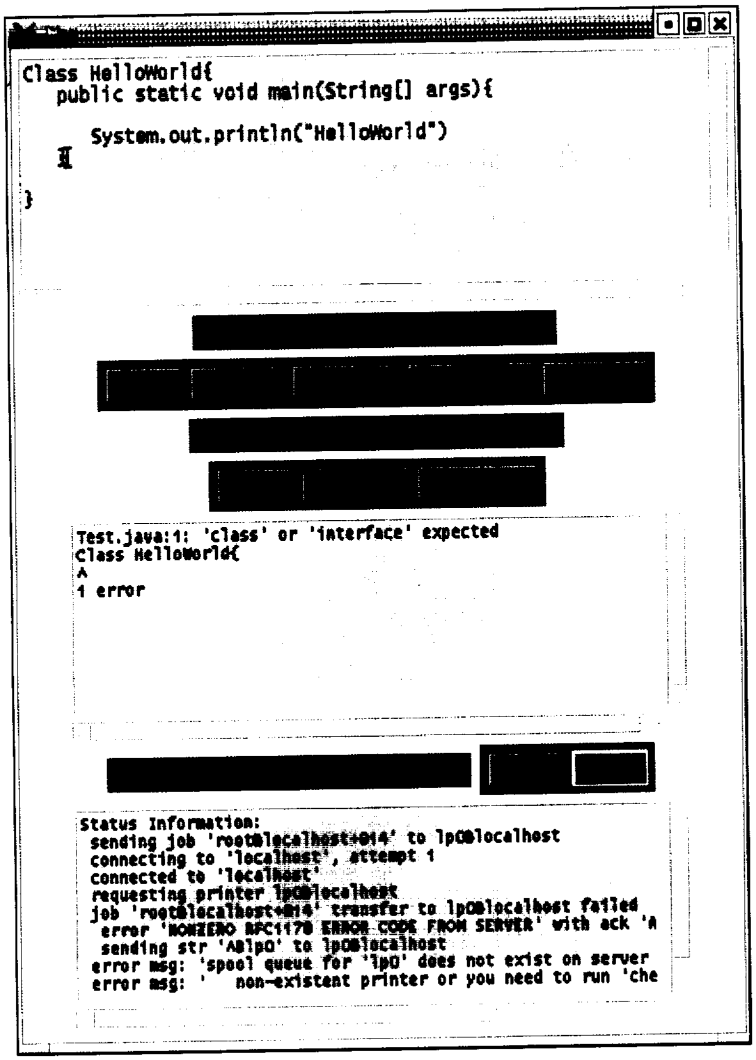


Figure 16: Client after Integration

4.2.1 Specification

To build this application, we constructed specifications similar to those found in Figures 6- 8. In this example, a monitoring client is connected to the TopService, UpTimeService and a TcpDumpService via connectors. The monitoring client is a CallReturn component, as are the services TopService, UpTimeService and TcpDumpService.

4.2.2 Client Application

The Client Application is connected with TopService via a `get_top` connector to the `TOP_Port`. Similarly the client application is connected to the UpTimeService via a `get_uptime` connector to the `UPTIME_Port` and to the TcpDumpService via a `get_tcpdump` connector to the `TCPDUMP_Port`. The

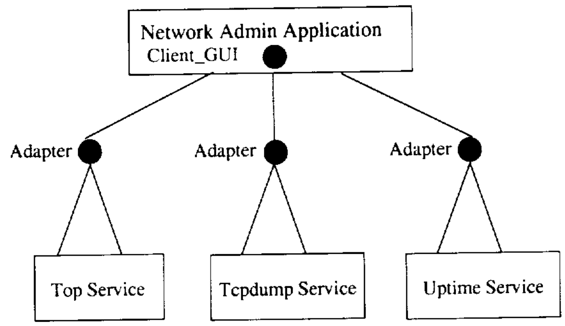


Figure 17: Conceptual Architecture: Network Monitoring Application

produced specification is used by the *ClientGenTool* to generate a client. Each of the services provide exclusive GUI components namely "TopResults", "UpTimeResults" and "TcpDumpResults".

4.2.3 Integration and Execution

The client when executed becomes the focal point for the integration. When the client comes up on the Jini network, it looks for the TopService, UpTimeService, TcpDumpService. As each of these services become available on the network, the client is notified and is bound with the adapter provided by the service. The adapter is used by the client to reach and interact with the service. The GUI components are also integrated using the shared and exclusive GUI components that are provided by the services. Figure 18 shows how the network monitoring client would look once all the services are available and integrated by the client.

4.3 Discussion

In the examples described above, the effort needed to generate the services was centered primarily upon the creation of the appropriate specifications as well as the code needed to provide a GUI interface to the service. In the client applications, the effort was likewise focused. However, as demonstrated by Figures 15 and 16 no effort (e.g., no extra code) was necessary to realize final GUI and application integration outside of the client application specific GUI, which for the editor application amounts to just a text pane and for the network adminstration application amounts to a frame panel.

In comparison to a "one-time" application development, the amount of code development in the service-based approach required less effort since the adapter and glue code for service and client components is automatically generated. In comparison to development of applications that utilize existing services, the amount of effort required is greatly reduced since each subsequent application (assuming existence of services) is limited to client specification and code development.

With regards to other component and middleware technologies, the approach used here has the advantage of supporting seamless integration based on availability of services. However, along with other service-

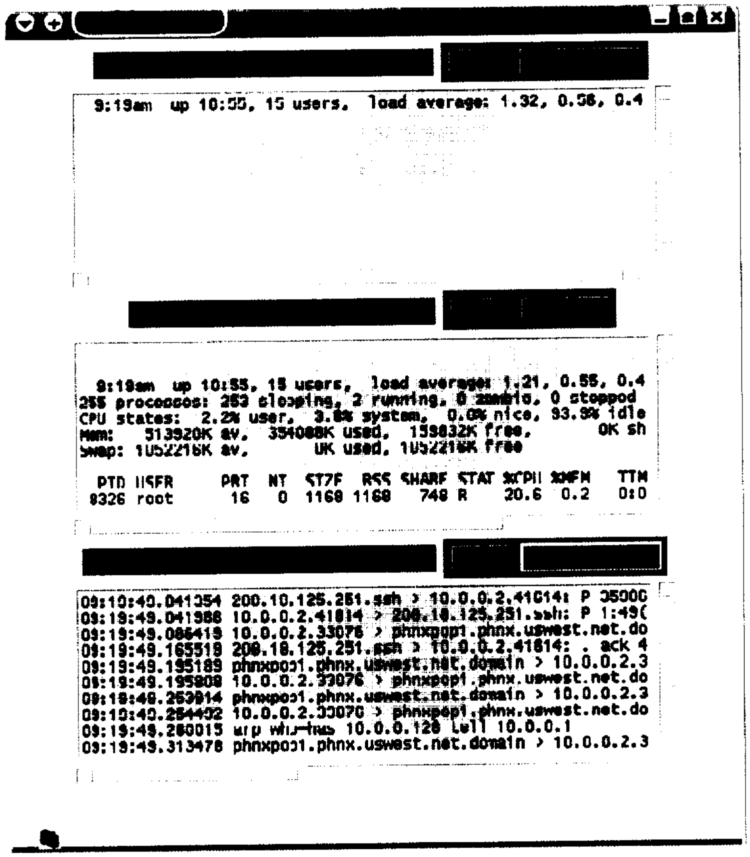


Figure 18: Network Monitoring Application

based technologies including web services, has the disadvantage of assuming loose coupling, thus relying heavily upon a priori knowledge regarding service behavior and data integration concerns.

5 Related Work

Recently, the use of WebServices [Oellermann, Jr., 2001] has gained attention with vendors releasing web-services toolkits that allow for building and using webservices. Webservices are based on the SOAP and XML [Seely and Sharkey, 2001] protocols over transport layers such as HTTP and HTTPS. Our approach to service integration goes beyond what the webservices paradigm provides. It does not talk about how to use the applications but rather what these services provide and where they reside. We take a step further in that we also try to define how these services need to be used with a larger context of several kinds of services. We make use of the ability to transport code provided by the Jini technology.

An interesting area of research work namely Semantic Web [Semantic Web, 2002] has also started to gain a lot of momentum. The Sematic Web is an extenstion to the existing web as it exists today, in that it allows the information to be given a well-defined meaning, allowing computers and people to work together.

The DAML [DAML, 2002] is an agent markup language that is a first step in trying to specify information that can be interpreted by computers as opposed to humans.

Jacobsen and Kramer [Jacobsen and Kramer, 1998] 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 component, we must address similar concerns.

Thakkar et al. describe the Ariadne service-specification language and process and the Theseus service-execution platform [Thakkara et al., 2002]. Ariadne adopts a relational-database view of existing resources and supports the construction of wrappers treating these resources as relational databases. Resources specified in Ariadne can be combined in terms of data-flow specification and these compositions are executed at run-time by their Theseus environment. They look at data integration as opposed to behavioral integration.

Sullivan et al. look at systematic reuse of large-scale software components via static component integration [Sullivan et al., 1997]. 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 [Dey et al., 1998] 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.

Finally, Grechanik et al. describe ways to integrate and reuse GUI driven applications [Grechanik et al., 2002] by wrapping applications as objects and accessing them programatically via an API. They achieve this using interceptors that capture GUI events which can be later replayed. In comparison, our approach achieves GUI integration by requiring that services on a Jini network provide a GUI via an adapter and proxy. The GUI integration is realized by using Jini's built-in facility for transporting direct handles to the proxies and integrating them directly into client applications at run-time.

6 Conclusions and Future Investigations

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 as well as GUI 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 [Mudiam et al., 2002]. 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.

References

[Acme Lib, 1997] Acme Lib (1997). The acme tool developer's library (acmelib). http://www.cs.cmu.edu/~acme/acme_downloads.html.

[Acme Studio, 2002] Acme Studio (2002). Acmestudio: A graphical design environment for acme. http://www.cs.cmu.edu/~acme/ AcmeStudio/ AcmeStudio.html.

[Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*.

[Chikofsky and Cross, 1990] Chikofsky, E. J. and Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17.

[Cimitile et al., 1998] Cimitile, A., DeCarlini, U., and DeLucia, A. (1998). Incremental migration strategies: Data flow analysis for wrapping. In *Working Conference on Reverse Engineering*, pages 59–68. IEEE, IEEE CS Press.

[DAML, 2002] DAML (2002). Daml-based web-services description language. [Online] Available http://www.daml.org/services.

[Dey et al., 1998] Dey, A. K., Abowd, G., , and Wood, A. (1998). CyberDesk: A Framework for Providing Self-Integrating Context-aware Services. *Knowledge-Based Systems*, 11(1):3–13.

[Fremantle et al., 2002] Fremantle, P., Weerawarana, S., and Khalaf, R. (2002). Enterprise services. *Communications of the ACM*, 45(10):77–80.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman.

[Gannod et al., 2000] Gannod, G. C., Mudiam, S. V., and Lindquist, T. E. (2000). An Architecture-Based Approach for Synthesizing and Integrating Adapters for Legacy Software. In *Proc. of the 7th Working Conference on Reverse Engineering*, pages 128–137. IEEE.

[Garlan et al., 1997] Garlan, D., Monroe, R. T., and Wile, D. (1997). Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183.

[Grechanik et al., 2002] Grechanik, M., Batory, D., and Perry, D. E. (2002). Integrating and reusing gui-driven applications. In *Proc. of the Intl Conf. on Software Reuse (LNCS 2319)*, pages 1–16. Springer-Verlag.

[Jacobsen and Kramer, 1998] Jacobsen, H.-A. and Kramer, B. J. (1998). A design pattern based approach to generating synchronization adaptors from annotated idl. In *Proceedings of the Automated Software Engineering Conference*, pages 63–72. IEEE, IEEE CS Press.

[Luckham and Vera, 1995] Luckham, D. and Vera, J. (1995). An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734.

[Medvidovic and Taylor, 1997] Medvidovic, N. and Taylor, R. N. (1997). Exploiting architectural style to develop a family of applications. *IEE Proc. in Software Engineering*, 144(5-6):237–248.

[Mudiam et al., 2002] Mudiam, S. V., Gannod, G. C., and Lindquist, T. E. (2002). A Novel Service-Based Paradigm for Dynamic Component integration. In *Proc. of the AAAI-02 Workshop on Intelligent Service Integration, Edmonton, Alberta, Canada*. American Association for Artificial Intelligence.

[Oellermann, Jr., 2001] Oellermann, Jr., W. L. (October 2001). *Architecting Web Services*. a! Press.

[Richards, 1999] Richards, W. K. (1999). *Core Jini*. Prentice-Hall.

[Seely and Sharkey, 2001] Seely, S. and Sharkey, K. (August 2001). *SOAP: Cross Platform Web Services Development Using XML*. Prentice Hall.

[Semantic Web, 2002] Semantic Web (2002). The semantic web. [Online] Available http://www.w3.org/2001/sw/.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall.

[Sneed, 1996] Sneed, H. M. (1996). Encapsulating legacy software for use in client/server systems. In *Working Conference on Reverse Engineering*, pages 104–119. IEEE, IEEE CS Press.

[Stal, 2002] Stal, M. (2002). Web services: beyond component-based computing. *Communications of the ACM*, 45(10):71–76.

[Sullivan et al., 1997] Sullivan, K. J., Cockrell, J., Zhang, S., and Coppit, D. (1997). Package Oriented Programming of Engineering Tools. In *Proceedings of the International Conference on Software Engineering*, pages 616–617.

[Thakkara et al., 2002] Thakkara, S., Knoblock, C. A., Ambite, J.-L., and Shahabi, C. (2002). Dynamically composing web services from on-line sources. In *Proc. of the AAAI-02 Workshop on Intelligent Service Integration, Edmonton, Alberta, Canada*, pages 1–7. American Association for Artificial Intelligence.