Reengineering Legacy Systems Towards Web Environments

Ying Zou
Dept. of Electrical & Computer Engineering
Queen's University
Kingston, ON, Canada, K7L 3N6
ying.zou@ece.queensu.ca

Kostas Kontogiannis
Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, Canada, N2L 3G1
kostas@swen.uwaterloo.ca

ABSTRACT

With the widespread use of the Web, distributed object technologies have been widely adopted to construct network-centric architectures, using XML, Web Services, CORBA, and DCOM. Organizations would like to take advantage of the Web in its various forms of Internet, Intranet and Extranets. This requires organizations to port and integrate their legacy assets to distributed Web-enabled environments, so that the functionality of existing legacy systems can be leveraged without having to rebuild these systems. In this chapter, we provide techniques to re-engineer standalone legacy systems into Web-enabled environments. Specifically, we aim for a framework that allows for the identification of reusable business logic entities in large legacy systems in the form of major legacy components, the migration of these procedural components to an object oriented design, the specification of interfaces of these identified components, the automatic generation of CORBA wrappers to enable remote access, and finally, the seamless interoperation with Web services via HTTP based on the SOAP messaging mechanism.

Keywords:

Software migration, software integration, software reengineering, legacy systems, multi-tier architecture, Web applications, Web services, distributed objects, software component identification, wrapping, object model identification

INTRODUCTION

With the widespread use of the Internet and pervasive computing technologies, distributed object technologies have been widely adopted to construct network-centric architectures, using Web Services, CORBA, and DCOM. Organizations would like to take advantage of the Web in its various forms of Internet, Intranet and Extranets. This requires organizations to migrate and integrate their legacy assets within distributed Web-enabled environments. In this way, the functionality of existing legacy systems can be leveraged without having to rebuild these systems.

However, legacy systems are not designed as Web applications. Generally, a Web application is a loosely coupled, component-based program, which consists of three tiers including user interface, business logic and related databases with a clear distinction. The front-end user interface is accessed through Web browsers. The business logic runs in a Web server and application server. The databases reside on back-end severs. Each component communicates with others through standard protocols such as HTTP and XML. In contrast, legacy systems are usually standalone systems with complicated structures with all three tires intermixed and tightly coupled. According to the decomposability of a legacy system, the architecture of a system can be decomposable, semi decomposable, or non decomposable [Brodie & Stonebaker, 1995]. In a decomposable system, the user interface, business logic and the related databases can be considered as distinct components with well-defined interfaces. Such systems are the best candidates for the migration. In a semi decomposable system, only interface and business logic are separate modules. The business logic and databases are not separable, due to their complex structure. Such systems are more difficult to be migrated, as the interaction of the business logic and the databases are not easy to understand. In a non-decomposable system, the system appears

as a single unstructured and monolithic module. Such systems are the most difficult to migrate, as they are treated as black boxes, and their functionality and design cannot be recovered.

Problem Definition

To modernize legacy systems with Web technologies, there are three major issues to be addressed, namely, Web accessibility, componentization, and platform transformation.

- Web accessibility is crucial requirement for a Web application. In this respect, the legacy interface should be replaced with a Web-enabled interface by placing a wrapper on top of the legacy system, or by re-generating a new interface in HTML form.
- Componentization focuses on the extraction of reusable components from the legacy systems. These components contain the evolving business logic and functionality of the legacy system. Once the components are identified, they can be easily maintained, and their interface can be described using standards, such as WSDL (Web Service Description Language) [WSDL]. Furthermore, components can be integrated with other components to produce new business functional units.
- Platform transformation involves migrating monolithic legacy systems into modern platforms such as the Web. In this context, there are two major types of strategies to be taken in order to leverage legacy assets into Web-enabled environments, namely:
    - Migration of legacy system into Web-enabled environments, and
    - Integration of the legacy system with new applications.

  The migration strategies involve the activities to transform the legacy systems into Web-enabled platforms, such as client/sever, three-tier (multi-tier) architecture, along with changes to legacy systems. The integration strategies focus on the coordination and interoperation between legacy systems and Web applications. Legacy application

integration aims to resolve the incompatibilities between legacy systems and Web applications, and allows them to seamlessly interoperate. In such a way, the lifecycle of legacy systems is greatly extended. Object wrapping is commonly used to adopt legacy systems to Web environments.

To minimize the risk involved in the reengineering process, the legacy system should be thoroughly analyzed to determine an appropriate strategy. The reengineering process should be exercised under a rigorous discipline because of the magnitude of the investment in the legacy systems, and the technical complexity inherent in such projects. As a result, software migration should be a well-planned, highly motivated and well-justified process.

Proposed Migration Framework

To address the aforementioned issues, we propose a framework that allows the migration and integration of existing stand-alone legacy services into a network centric Web-enabled environment, where they are considered as distributed Web services. The framework consists of four major steps:

- *Identification of Legacy Components:* we aim to identify software components from monolithic legacy systems. The recovered components should conform to specific user requirements (i.e. relate to specific functionality, access and manipulate specific data, are free of side effects, and comply with specific pre and post conditions).
- *Migration of Identified Components to Object Oriented Platforms:* we migrate the recovered legacy system components to collections of object classes that encapsulate specific functionality of the legacy system. As an object encapsulates data and operations, flexible systems can be easily designed and implemented using the object-oriented paradigm. Furthermore, we analyze interfaces of the recovered components and define a

specification to represent their interfaces. The service specification provides standard, enriched, and well-understood information about the interface and functionality of the offered services. The service specification language is defined in XML.

- *Migration to Distributed Environments:* we design a methodology to migrate the identified components into distributed environments by automatically generating CORBA/IDL and CORBA wrappers from the identified component interface specifications. Since the client processes only need to know the interface of their servers, the modification of the server implementation does not affect the client code as long as the server interface is kept unchanged.
- *Migration to the Web:* to enable CORBA components to be accessible by Web clients, and integrated with other Web services, we present a process based on the SOAP protocol to deploy these newly migrated components in Web service environments.

Chapter Organization

This chapter is organized as follows. Section 2 surveys existing techniques for migrating legacy systems into Web environments. Section 3 presents several approaches that recover reusable components from legacy systems. Section 4 discusses our migration techniques which focus on the transformation of identified procedural components into object oriented platforms. Section 5 describes a process that enables the migrated components to be integrated within a distributed environment using standards such as CORBA. Section 6 presents an approach to enable the migrated distributed components to interact in a Web Service environment using standards such as SOAP to extend the CORBA standard for the Web. Finally, section 7 concludes the chapter and provides some insights and directions for future trends in software reengineering and the evolution of legacy systems.

EXISTING TECHNIQUES FOR MIGRATING LEGACY SYSTEMS INTO WEB ENVIRONMENTS

Initially, the Web was presented as a global URL-based file server for publishing static information in a hypertext electronic form. With the use of Java, and scripting language, such as

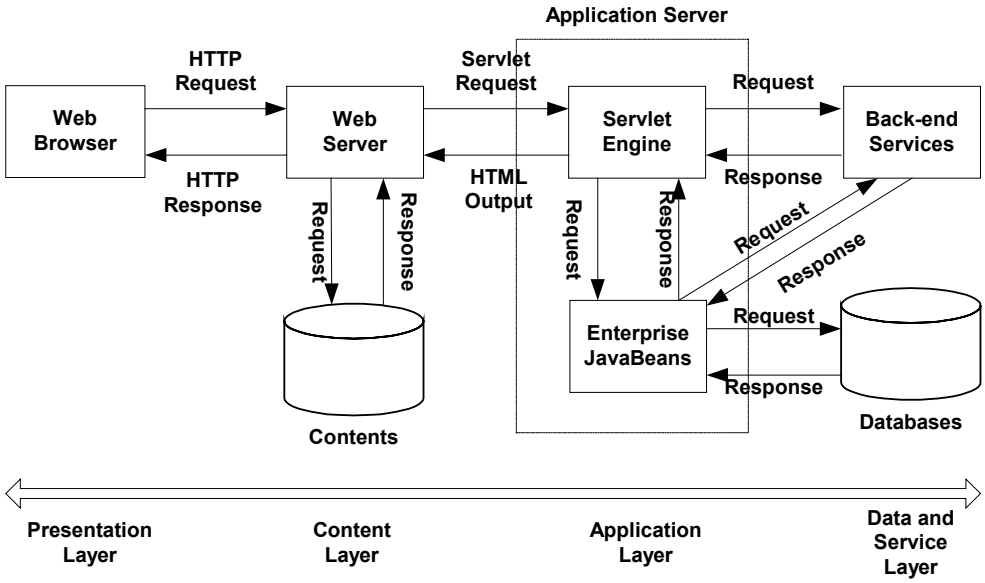

Figure 1: Control Flows in Three-Tier Architecture

CGI (Common Gateway Interface) scripts, the Web evolved into an interactive medium that provides dynamic information services in a client/server architecture. With the rapid growth of E-Commence activities, the emphasis has shifted from the client-side browser and server-side applications to multi-tier architecture based on standards such as J2EE and .NET.

A multi-tier architecture emphasizes the separation of presentation logic, business logic and back-end services. A three-tier architecture is an example of a multi-tier architecture, as illustrated in Figure 1. A three-tier architecture is instrumental for the implementation of large Web enabled systems. From an implementation point of view, the standard three-tier architecture can be seen as a composition of four layers [Dreyfus,1998] namely: *a)* presentation layer (Web

client); *b)* content layer (Web server); *c)* application layer (application server); and *d)* back-end data and services layer. The Web server is responsible for accepting HTTP requests from Web clients, and delivering them to the application server, while the application server is in charge of locating the services and returning responses back to the Web server. On the other hand, a thin client in the presentation layer has little or no application logic. It sends the request via HTTP to the Web server for an interactive view of the static and dynamic information (see Figure 1).

To provide static information, the Web server maintains a static content repository, such as a file system, where the information-based resources are stored and served as static HTML pages. Upon receiving a request from the client, the Web server retrieves the requested document from the content repository and sends it to the client. In this case, the client relies entirely on the Web server to fulfill its request. Programming languages, such as Java, and scripting standards, like CGI, can be used to enable Web clients to access databases on the server side.

To provide dynamic information generated by software services, the Web server needs to constantly interact with the application server. A servlet or Java Server Pages (JSP) provides the dynamic HTML content to clients. When the Web server receives a request for a servlet, it re-directs the client's request along with the parameters to the application server, which loads the appropriate servlet and runs it. Servlets have all the features of Java program. They also have automatic memory management, advanced networking, multithreading, and so forth. Moreover, they have enterprise connectivity through protocols such as JNI (Java Native Interface), JDBC, RMI, and CORBA. Servlets can initiate invocations to back-end services, other servlets, or to Enterprise JavaBeans (EJBs) [Picon].

Back-end software components can be considered as software services. In this context, the Web has evolved from an "*information-based*" Web into a "*services-based*" Web. Such Web services

offer specific business related functionality, reside in the application servers, can be programmatically integrated with other systems, and perform interactive tasks involving multiple steps on the user's behalf.

To allow for Web services to interoperate, common industry standards, such as SOAP (Simple Object Access Protocol) [SOAP], WSDL (Web Service Description Language) [WSDL], and UDDI (Universal Description, Discovery, Integration) [UDDI] have been proposed. In a nutshell, SOAP, encoded in XML and HTTP, is a message passing mechanism and serves as a uniform invocation mechanism between Web Services. The WSDL describes the interface for a Web service. These interfaces are indexed in searchable UDDI repositories.

Migrating Monolithic Legacy Systems to Client/Server Architecture

To achieve a client/server architecture, migration activities are mostly concerned with the separation of user interface, the business logic and the backend data of a legacy application. Consequently, the user interface is replaced with a Web based interface, written in XML or HTML. The application logic of the legacy system is wrapped as a server, and can be executed in its native environment. This approach can be applied for a transaction oriented legacy system, which involves a lot of interactions between users, such as information entry, query and retrieval. In particular, it is the best choice for semi-decomposable and non-decomposable systems. Finally, backend data can be analyzed using data reengineering techniques, which have been investigated thoroughly in [Blaha *et.al.* 1995].

In general, a legacy user interface is often character-oriented, and system pre-emptive, meaning that the application initiates an interaction, and users only response upon the applications' request. However, a Web based interface is event driven and user pre-emptive, meaning that a Web user initiates a dialogue without the system's prompts. Events include keystrokes and

cursor movements. For such an interface, a Web browser incorporates loop and call back mechanism in order to constantly listen to user triggered events and response promptly with the call back routines residing on a server. Therefore, shifting a system pre-emptive interface to a user pre-emptive interface focuses on the identification of events in the original system, and the associated call back routines.

The key step in this migration approach is to identify the user interface of the legacy system. In literature, several approaches are proposed based on the availability of the original source code. In the case where the source code is not too complicated to tackle, Moore & Moshkina present an approach to perform the user interface re-engineering [Moore & Moshkina. 2000]. This approach is composed of three steps. Firstly, the input/output logic is detected by the analysis of control flow and data flow in the original code. The result of the flow analysis is applied to identify user components that are functionally equivalent to GUI toolkits. Secondly, such components are transformed to HTML forms. In the end, the legacy system is restructured to associate functions or routines with the user components. In this case, the legacy code is examined and modified to cope with the migration. Alternately, without modifying the source code of the legacy application, the legacy user interface is detected by tracking the user interaction with the system. Stroulia, *et. al*. present another re-engineering approach [Stroulia *et. al.*1999]. Firstly, an interface-mapping task identifies all individual screens and the possible states occurring, due to screen transitions. The screen transitions are triggered by user actions, such as keystrokes and cursor movements. A graph of the system interface transitions is incrementally constructed to include screens and transitions. Such graph supports the navigation of text-based user interface, and facilities the reorganization of the user's navigation plan. Secondly, a domain model is abstracted to specify the screen transitions, information flows and exchanged information.

Finally, based on the abstract domain model, the graphical user interface is specified and generated. The new interface consists of the bridge between the original interface and Web-enabled interface.

These two approaches make an entire legacy system as a single reusable service by wrapping it as a black box. However, the larger the software component, the less flexible for reuse in different contexts it will be. In our migration framework, we aim to produce a new migrant system that is composed of several reusable components. These components can be flexibly integrated to accommodate changing business requirements over the lifetime of the software system. We migrate the system into an Object Oriented design, which is consequently migrated into the Web. Since object oriented design provides information hiding and encapsulation properties in classes, the identified software components are fine-grained classes that provides maximal reusability and flexibility.

Migrating Legacy Web Applications

A multi-tier application requires a clear separation of the user interface, business logic and databases. It is a complex and challenging process to migrate semi-decomposable and non-decomposable legacy systems into multi-tier architecture. As technology evolves, there is a practical demand to convert traditional Web applications, using CGI and Net.Data, into a multi-tier architecture using JSP, Servlets and EJBs technologies. In this case, such legacy Web applications are designed with a clear separation of client and server components. The migration task is mitigated. In this section, we present research projects in literature for this subject.

In [Lau *et. al.* 2003] an approach that deals with the separation of business logic and database in the migration of legacy e-commerce applications into EJB architecture is presented. Obsolete e-commerce applications intertwine business logic and database access code that uses SQL

statements. In contrast, the target EJB architecture separates the business logic and database queries. Specifically, EJB contains two types of enterprise beans including session beans and entity beans. The session beans are used to implement activities, while the entity beans are utilized to represent persistent data in database. For example, one entity bean corresponds to one row in a table. Instead of using SQL statements to query a database, an entity bean provides methods to manipulate tables in a database. The paper presents a migration process that follows these steps:

- Build a relational database schema mapping from the source databases to the target databases,
- Convert control blocks in the original application to session beans in EJB architecture
- Establish relations among the new entity bean objects with the conformance to the table relations in the databases.
- Generate plumbing code needed to integrate the new session beans and entity beans.

Using this approach, a prototype tool was developed to transform IBM Net.Data applications into JSP and EJB applications.

Similarly, in [Hassan & Holt 2003] a Web application migration approach is presented. The approach deals with the migration of active pages, the building blocks of a Web application. Active pages from one vendor centric development framework, such as ASP from Microsoft are converted into active pages under standardized Web development framework. Essentially, an active page consists of HTML tags for presentation, and control code that makes use of the underlying object model and programming language to provide business logic. Such object model provides comprehensive functionality needed by Web developers. For example, object model contains request and response objects to allow the interaction between client and server. It

also provides objects to maintain the states of sessions due to stateless nature of HTTP protocol. Furthermore, this paper abstracts the commonality between the object models provided by the various frameworks, into a common object model. The paper summarized the migration process in four stages [Hassan & Holt 2003]:

- The preprocessing stage removes the HTML code and comments from the active page.
- The language transformation stage translates the active code from the initial programming language to the target programming language.
- The object model mapping stage transforms access to objects in the original application framework to the corresponding framework.

The post processing stage reinserts the HTML code and comments that were removed in the preprocessing stage.

These two approaches focus on the migration of legacy Web applications into modern multi-tier architectures. Our proposed framework aims to migrate standalone procedural non-Web based systems into Web services residing in the back-end layer of a multi-tier architecture. Consequently, these back-end services can be composed to produce business logic. In the following section, we discuss our approach in more detail.

EXTRACTING LEGACY COMPONENTS

A software component can be considered as "an independently-deliverable software package that offers services through public interfaces" [Brown & Barn 1999]. A component offers specific functionality and can be easily integrated with other COTS (Commercial Off-The-Shelf) or custom-made software components. By the use of component based technology in multi-tier architecture, the business logic can be easily composed from various back-end services as distributed software components. In such a way, a legacy system can be treated as a single back-

end service. To extend the reusability of such legacy systems, finer grained components need to be identified to serve as building blocks in new distributed systems. In particular, the identified legacy components can be gradually migrated into Java to enable the seamless integration of legacy functionality with new applications. In addition, the data, accessed by the legacy applications, may need to be migrated from its present form, such as flat files, obsolete format, and legacy databases, to distributed relational databases, or to be represented in the Enterprise Java Beans framework.

So far in the field of reverse engineering, most software analysis techniques for the identification of components in monolithic systems focus on clustering and decomposition techniques that are based on cohesion and coupling properties. It is safe to say that in most cases, legacy system decomposition has become a clustering problem. Clustering is a successful technique on decomposing a legacy system into subsystems. Nevertheless, these identified subsystems are not guaranteed to be good candidates for becoming distributed software components. The reason is that, clustering heavily depends on static code features (coupling, cohesion) as opposed to dynamic requirements that are imposed by the architecture of the new migrant target distributed application. Furthermore, it is especially difficult to recover via clustering alone, components that would have clear interfaces and minimal dependencies with the rest of the system, due to side effects and extended modifications incurred by the prolonged maintenance of the legacy code being analyzed. A possible solution would be to make the whole legacy system a reusable unit by wrapping it as a black box to provide the signature of its input/output points as an API such as presented in the previous section. However, the larger the software component wrapped, the less cohesive, flexible, and customizable it will be.

An alternative approach to address the component recovery problem is to transform the procedural code, to a new object oriented architecture, whereby components stem from collections of highly cohesive identified classes. The motivation to migrate procedural legacy systems to a new object oriented architecture and to subsequently provide an infrastructure to make the new migrant systems available in a Web-enabled environment is twofold. Firstly, object oriented languages provide mechanisms for information hiding and encapsulation and therefore it is easier to identify the loosely coupled, small-scale, reusable software components. Secondly, the objectification process provides means to define clear interfaces between the identified components, establish inheritance relations between the components, and minimize component interconnection. Collections of such classes that correspond to Abstract Data Types, can be used to associate parts of the original system to new components that are highly cohesive, and have minimal control and data flow dependencies with the rest of the system. In the following section, techniques for migrating procedural code into object oriented platforms are presented.

MIGRATING TO OBJECT ORIENTED PLATFORMS

Over the past years, it has become evident that the benefits of object orientation warrant the design and development of reengineering methods that aim to migrate legacy procedural systems to modern object oriented platforms. With properties, such as information hiding, inheritance and polymorphism inherent in object oriented designs, essential parts of such a reengineered system can be reused or integrated with other applications using network centric Web technologies, or enterprise integration solutions.

In this context, the software reengineering community has already proposed a number of different methods to migrate procedural code into object oriented platforms. In a nutshell, the

existing migration methods aim to identify Abstract Data Types (ADT) and extract candidate classes and methods from the procedural code. These methods include concept analysis [Lindig & Snelting1997, Sahraoui *et. al.*1997], cluster analysis [Mancoridis *et. al.*1998, Muller *et. al.* 1992], slicing [Lanubile & Visaggio1997], data flow and control flow analysis [Lucia *et. al.* 1997], source code features [Kontogiannis & Patil 1999, Zou & Kontogiannis 2001], and informal information analysis [Etzkorn & Davis 1997]. Moreover, it is important for the object oriented reengineering process to address specific nonfunctional requirements, such as reusability, performance, and maintainability, so that the resulting migrant system conforms to specific constraints and properties.

There is a significant number of related research for the migration of COBOL systems to object oriented COBOL[Sneed 1996], assembly to C[Ward 1999], C to C++[Kontogiannis & Patil 1999, Zou & Kontogiannis 2001], C to Java[Martin], and RPG to C++[Lucia *et. al.* 1997]. In [Lucia *et. al.* 1997], the identification of an object model from RPG programs is presented. Objects are identified from persistent data stores, while related segments of source code in the legacy system become candidate methods in the new object oriented system. In [Kontogiannis & Patil 1999, Zou & Kontogiannis 2001], an object model is discovered directly from procedural code written in C. Candidate objects are selected by analyzing global data types and function formal parameters. In [Lindig & Snelting1997, Sahraoui *et. al.*1997], a transformation system that is based on concept analysis is proposed to identify modules from C code. It is based on lattices that are used to reveal similarities among a set of objects and their related attributes. Finally, a domain based objectification method is presented in [Etzkorn & Davis 1997]. The method is based on documentation and informal information, such as user manuals, requirement

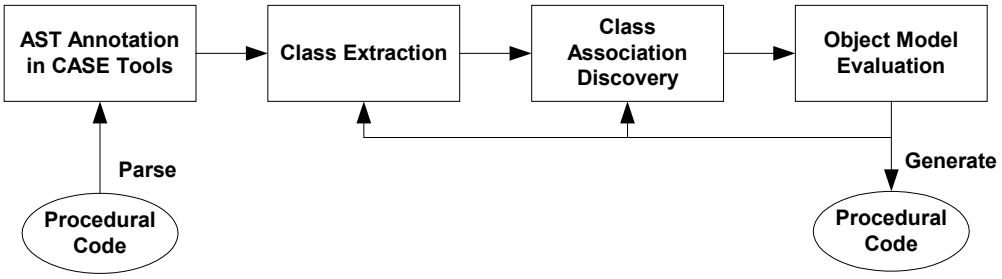

Figure 2: Object Oriented Reengineering Framework

and design specifications, and naming conventions. However, for legacy systems, the external information is not always available, and this technique may not always be applicable. In the following subsections, we propose a representative object oriented reengineering framework, where components of a procedural legacy system can be migrated to an object- oriented platform. Specifically, in addition to identifying objects based on abstract data types, the framework aims to discover an object model that incorporates the object-oriented characteristics of a well-designed system, namely the characteristics of class inheritance, polymorphism and overloading. Furthermore, we discuss an approach, which enable the migrant system to posses certain non-functional quality characteristics, such as high maintainability.

A Framework for Migrating to Object Oriented Platforms

The migration framework aims at the extraction of an object-oriented model by a series of iterative analysis steps applied at the abstract representation of the source code (i.e. the Abstract Syntax Tree level). The overview of framework is illustrated in Figure 2. By the analysis of the Abstract Syntax Tree (AST) annotation, class candidates are identified from procedural source code. Each class encapsulates a procedural global data structure and functions that use or update this data structure. In this case, the data fields in a procedural global data structure are converted into data members in the corresponding class. The functions are transformed into methods in the class. Consequently, class associations such as inheritance and polymorphism can be further

recovered from specific procedural source code features. Once object model is produced, developers can evaluate the generated object model by introducing domain knowledge to refine the identified class candidates and class associations. In the end, a target object oriented code is generated.

The key step in this migration is the extraction of classes and the identification of methods that can be attached to these classes. To facilitate this class creation phase, the framework focuses on the analysis of global aggregate data types, and formal parameter lists. Global data types and data types in formal parameter lists, found in the original legacy source code become primary candidates for classes in the new object oriented system. Similarly, functions and procedures in the original system become primary candidates for methods and are attached to the identified classes. This migration framework can be automated to a large extent using a number of different software analysis techniques. However, no matter how sophisticated the analysis techniques are, user assistance and domain knowledge play an important role in obtaining a viable, efficient object model. Therefore, the application domain of the code and the domain knowledge assist in evaluating the extracted object model. The migration process can iterate back to the class extraction phase or class association discovery phase to refine the identified object-oriented model. Finally, the target object oriented code is generated.

Object Oriented Model Discovery and Refinement

In the process of extracting an object-oriented model from procedural source code, it is important to identify the transformation steps and extraction rules that allow for the generation of a high quality target migrant system. In the following sections, we present a catalog of analysis and discovery rules that can be used for extracting a quality object model from procedural systems. The rules are divided into two categories: Class Creation and Object Model Extension. The rules

for the class creation category define the criteria for the creation of class to form an object model from the procedural code. The rules on the object model extension provide heuristics to establish association between class candidates.

Class Creation

In the search of an object model that can be extracted from procedural source code, we aim at achieving high encapsulation, high cohesion within a class, and low coupling between classes. In an object-oriented system, software is structured around data rather than around services of systems. Therefore, object model should be built around data that serves as seeds to gather the related data members and methods. Such data and method clusters are consequently considered as class candidates. In this respect, a class creation process is divided into three major steps: class identification, private data member identification and method attachment.

The first step is the selection of possible object classes. The object identification techniques focus on two areas: 1) the analysis of global variables and their data types, 2) the analysis of complex data types in formal parameter lists. Analysis of global variables and their corresponding data types is focusing on the identification of variables that are globally visible within a module. A module is considered as a sequence of statements, with an identifier, such as a file and a program. For each variable, its corresponding type is extracted from the Abstract Syntax Tree, and a candidate object class is generated. Data type analysis focuses on type definitions that are accessible via libraries. Examples include *typedef* C constructs. Data types that are used in formal parameter lists become also primary class candidates. The union of data types that are identified by the global variable analysis and data type analysis forms the initial pool of candidate classes. Furthermore, based on the usage of a set of data types in formal

parameter lists, groups of data types can be collected into one aggregated class to minimize the interface of the methods, which take the aggregate class as parameter instead.

In the second step, data fields in the user defined aggregated types and the global variables are converted into data attributes in the corresponding class candidates.

Finally, the procedural functions are assigned into class candidates. To achieve higher encapsulation, the data members in class candidates can only be directly used or modified by methods in the same class candidate, instead of by the methods in other class candidates. Therefore, procedural functions are attached to class candidates based on the usage of data types in the formal parameter lists, return types, and variable usage in the function body. For this task, only aggregate data types in the parameter list and return value are considered, and simple ones such as int, char, and float are ignored.

- *Private Data Member Identification*

Data type analysis: A structure is a collection of variables that can be of diverse types. Similar to C++ classes, the structure of a record groups related data items. In procedural code, for example, C struct and union are constructs that can be used to generate aggregate data structures, indicating a degree of functional and logical coupling among a group of data and functions that are using it. For migration purposes, each data member of a structure can be converted into a data field in the corresponding class.

Variable analysis: Typically, there are three main scopes to access variables in a program, namely, local scope, file scope and global scope. According to their different scopes, variables can be transformed to private data members in the identified classes. In this context, variables that can only be referenced within a function can be declared as private data members of the class that encapsulates the method that corresponds to the referencing function.

Although C++ allows for global constant definitions to be accessible from within classes, in order to obtain a better object oriented design for the migrant code, one should aim to eliminate file and global scoped variables that can be referenced from within a file or the whole program. File scope and global scope variables are similar in that they are used or updated by any function. Keeping the scope of these variables in the new object oriented system would violate the principles of encapsulation and information hiding. A possible solution is that each of these variables can become a data field of a class that relates to these variables. Or the class encapsulates the functions or part of functions that most often refer or update such variables. Finally, orphan variables that cannot be assigned to any class can be encapsulated in a container class.

- *Method Attachment*

The method identification process focuses on the discovery of functions from a procedural code that can be transformed into methods in the appropriate classes.

Parameter type analysis: A formal parameter of a function indicates that the function references a data item of a particular type. In the process of extracting an object model from procedural code, one key analysis is to consider functions of the procedural system as primary candidates for methods in the new object oriented system. These methods can be attached to a class that corresponds to the data type appearing in the formal parameter list. In this context, a function or a procedure may be able to be attached to many different classes when more than one data type exists in the formal parameter list. We then say that this method produces an *assignment conflict.* The conflicts can be resolved by evaluating the alternative designs to achieve high cohesion and low coupling of the target system. Parameter type analysis is one of the alternative ways to consider when we would like to decide whether a function should be transformed to a method, or

to which a class should be attached. The following points discuss such alternative types of analysis.

Return type analysis: The return type of a function indicates that the function updates variables of a specific type that appears in the return statement. If this return type is also in the formal parameter list, then this provides stronger evidence that the function should be a method attached to the class stemming from this data type. The reasoning behind this analysis is that classes should encapsulate methods that update the state of objects that belong to this class. Moreover, this type of analysis provides valuable insights as to which class the method should be attached to, when no other information can be extracted from the formal parameter list.

Variable usage analysis: Variable usage analysis allows for examining whether a function or a procedure references a particular data type. Based on the concept of information hiding, we are interested in attaching methods to classes in a way that a given method references as much as possible variables of the same type as the class it is attached to. Variable usage analysis provides such insights and counts as to how many times a variable of a particular class is referenced within a candidate method. This type of analysis is also of particular importance when no information from formal parameter list and return type analyses could be obtained to decide which class to attach a conflicting method.

Function splitting: One of the features in legacy procedural code is the size of functions or procedures which tend to increase with prolonged maintenance activities (i.e. when adding new functionality, or when correcting errors). This situation leads to complex code that is difficult to understand, maintain, and migrate. For the class creation process, such long function or procedure should be sliced into smaller chunks of code. The slicing criteria may vary, but the safest way is to consider slicing according to the first use of a data type that corresponds to a

class that the specific function is a candidate to be attached to as a method. Function Point analysis can also be used to confirm that a particular slice provides a highly cohesive piece of code. Finally, source code informal information such as comments and variable names provide also good cues for splitting a function or a procedure into more than one component [Fowler 2000].

Object Model Extension

Inheritance, polymorphism, and overloading, are some of the most important features of an object-oriented design. To achieve a good object-oriented design from a procedural legacy system, we have identified a number of heuristic analysis rules that can be used to establish associations between abstract data types. Below we discuss several heuristics used to identify inheritance, polymorphism, and overloading opportunities in the extracted object model of the target migrant system.

- *Inheritance Identification*

Data field cloning: A typical indicator for identifying inheritance opportunities during the object model extraction process is data field cloning. In this respect, if two or more structures differ only with respect to few fields, these can be the candidate subclasses of a more general class. The common fields from those structures are extracted and form a super class. Subclasses can inherit from it and add their own fields.

Data field mapping: Certain functions copy values from fields in one data structure to fields in another data structure. These two data structures may not be of the same type, but they share common values. The relation of these data structures can be identified as inheritance, by extracting the common fields into a super class.

Function code cloning: Code clone analysis can identify inheritance where two functions are identical with the only difference that they operate on different data types. In this case, these data types may become subclasses of a more general type and methods can be attached to the subclasses which inherit from the more general class.

Data type casting: In cast operations, the compiler will automatically change one type of data into another where appropriate. For instance, when an integral value is assigned to a floating-point variable, the compiler will automatically convert the int to a float. Casting allows to make this type conversion explicit, or to force it when it wouldn't normally happen. An implicit cast operation between two data types suggests that these data types share common data fields or are interchangeable. The inheritance between these two abstract data types can be deduced in that the casted type becomes the subclass of the type that it is casted to.

Anonymous union type: Anonymous union types denote that their data members share the same memory space. This feature provides a subtle difference from the semantics of a C struct where all members are referenced as single distinct group. By contrast, only one union data member can be referenced at a time, and different data members cannot co-exist in the same time. In this context, the common structure of the union data member can be extracted as a superclass, while each of the union data members can be transformed to a subclass. For example the *Personnel*, a C struct as illustrated in Program 1 below, contains an anonymous union type data field. The super-class, *Personnel* is created and contains the field, *age*, without the union type data member, as illustrated in Program 2. Each of the union data fields becomes a subclass.

```
typedef struct{
        union {
                int student;
                int teacher;
        }
        int age;
} Personnel
```

Program 1:*. struct definition*

```
class Personnel {
        int age;
};
class Student : public Personnel{
        int student;
};
class Teacher: public Personnel{
        int teacher;
};
```

Program 2. *Refactoring ADT into Class inheritance in C++*

- *Polymorphism Identification*

Switch statement replacement: One of the important characteristics of object-oriented design is the limited use of switch (or case) statements — Polymorphism provides an elegant way to limit the use of long, complex and cumbersome switch and case statements [Fowler 2000]. A switch statement in the procedural code that uses in its evaluation condition a type check variable, can be replaced by polymorphic methods. Specifically, each of the case statement bodies may become a polymorphic method while the data type codes that are used by the condition of the switch can be considered as candidate classes in the new migrant system.

Conditional statement replacement: The branch of a conditional statement is executed according to the return value of the evaluating condition. When the type code is used in the conditional expression, then each branch of the conditional statement can be transformed to a polymorphic method, while the type code used by the conditional expresion can be considered as a candidate class in the new design.

```
void printIt(void *itToPrint, int type)
{
        employee *thisEmp;
        market *thisMarket;

        if (type == EMPLOYEE){
                thisEmp = itemToPrint;
                // ……
        }
        if (type == MARKET){
```

```
                thisMarket = itemToPrint;
                // ……
}
```

Program 3. *Conditional Statement Replacement*

Function pointer replacement: There are two ways functions can be invoked in C: by name and by address. Invocation by name is by far the most common one, when the functions to be called are decided at the compile time. Invocation by address is used to determine at run time the concrete functions to be executed. In this context, each possible function pointer reference can become a class and their corresponding source code can become a polymorphic method.

Generic pointer parameters replacement: The generic C pointer is denoted by "void *". A "void *" variable can contain the address of any data type. Often this technique is used to write highly generic functions that need to deliver some small piece of non-generic functionality. An example is illustrated in Program 3 below, where the address of struct data type in C, along with a type code is passed into the generic function as a parameter at the printIt() function. At run time, the appropriate struct is accessed by address. In this case, the generic function can be converted into a polymorphic method whereas its behavior is determined according to the type of the object that is applied upon. Moreover, type codes that can be referenced by the generic pointer parameter can be transformed to classes in the new migrant system.

Source code cloning: When two or more functions are identified as clones with minor differences in their structure and the data types they use, these functions can be overloaded on the data types they differ. The constraint is that these functions should return the same data type.

Functions with common prefix or suffix name: Similar prefix and suffix names in functions or procedures provide important cues for overloading. For example, execl(), execv(), execlp(),

execp() are functions to execute unix processes in various ways and all can be overloaded according to the type they operate upon.

Functions with union type parameter: Functions with a union type parameter may become candidates for overloaded methods. The reason is that these functions usually have different behavior according to the type of the union parameter they are applied upon. These functions can be transformed into several overloaded methods with different parameter types that are obtained from the original union structure definition. Each overloaded method can operate on the specific case of the original union structure.

Object Oriented Code Generation

Once the classes have been discovered and associations between them where established using operations and heuristics on the AST, the code of the final object oriented system must be generated. The main objective of this task is to automatically generate Java or C++ source code from the identified object model. The new code should encapsulate functionality from the original legacy system being reengineered, and should be compiled with the fewest possible corrections by the user. The code generation process traverses the ASTs for each class and generate syntactically correct C++ (or Java) code placed in .h and .cpp files. In some cases human intervention could be required for obtaining a better quality of even fully compilable code. However, this intervention should involve only minor changes and should not increase the complexity of the migration process.

The source code generation process is based on three steps. In the first step, from the obtained class list and method resolution results, the source code for the header files can be created. Once the header files are created, the second step allows the users to make manual modifications by utilizing a UML visualization editing and authoring tools. In this respect, additional classes can

be added in order to increase the design quality of the migrant system. The results from the method resolution process are examined and necessary changes can be made here. Finally, in the third step, any changes and relationships are incorporated in the source code and errors are removed and new skeleton code that conforms to a highly cohesive object model that is endorsed by the user is generated. The following two sections discuss this process in more detail.

- *Generating Class Header Files*

For each class in the identified object model, C++ or Java source code can be automatically generated. Fields in the records or structures of the original code can become private member variables for newly generated classes. The bodies of functions in the original system can be transformed into bodies of public methods of the class they are assigned to. Moreover, if a class type appears in a function's formal parameter list, and if that function is attached as a method to that class, then the corresponding parameter is removed from the formal parameters. The functions with class type as a parameter can be kept as non-static. Static keyword from C functions is removed. Conversely, the functions returning assigned class type become static members. Also, the assigned class type is removed from the formal parameter list of mapped function. In order to access member variables, accessors and mutators are created.
As an example, consider the following data type and function prototype.

```
typedef struct ubi_btNodeStruct {
        struct ubi_btNodeStruct *Link[ 3 ];
        char                    gender;
        char                    balance;
           } ubi_btNode;
typedef ubi_btNode *ubi_btNodePtr;
ubi_btNodePtr ubi_RemoveTree (ubi_btRoot *RootPtr, ubi_btNode *Node)
```

These can be transformed into a class and method respectively as shown below.

```
class  ubi_btNode {
private:
   ubi_btNode *Link[ 3 ];
   char   gender;
   char   balance;
public:
   void putLink(int index, ubi_btNode *value) { };
              ubi_btNode  * getLink(int index) {}
              void putGender(char value){};
              char getgender(){};
              void putBalance(char value){};
              char getBalance(){};
}
ubi_btNode* ubi_btNode:: ubi_btRemoveTree(ubi_btRoot *RootPtr);
```

- *Generation of Method Code*

In this step, object-oriented source code is generated, by utilizing the source code from the procedures or functions of the original legacy system. A source code generation module applies source code transformation rules as it traverses the Abstract Syntax Tree (AST) nodes that correspond to the original procedures or function bodies. For each identified class, public methods are traced from their corresponding AST function nodes and transformation routines are called to generate the corresponding method body in C++ syntax or Java. As an example, consider the following code fragment.

```
ubi_btNodePtr  ubi_avlRemove ( ubi_btRootPtr  RootPtr, ubi_btNodePtr   DeadNode)
{
     if ( ubi_btRemove(RootPtr,DeadNode))
     RootPtr->root = Debalance(RootPtr->root,
                               DeadNode->Link[01],
                               DeadNode->gender);
     return (DeadNode);
}
```

If ubi_avlRemove is attached to ubi_btNode in the object model identified in the earlier steps, then following piece of C++ code is generated.

```
ubi_btNodePtr ubi_btNode::ubi_avlRemove (ubi_btRootPtr  RootPtr )
{
     if ( ubi_btRemove(RootPtr))
     RootPtr->putRoot(Debalance(getLink(0x01),getGender()));
     return  this;
}
```

In the method body, to access member variables of other classes, accessors and mutators are used. Accessors and mutators for a class, are automatically generated by the object model specification. However, no matter how complete the analysis is, it may be the case that the generated object oriented code does not meet the requirement set by the user. For example, some methods cannot be assigned automatically to just one class. Such methods have to be split into smaller methods and assigned to the most appropriate classes. This process has to be done by applying slicing techniques and manual user intervention. In order to assist the user in evaluating the new generated code, the header files are further exported to object modeling and authoring tools (such as *Together/C++* [Together]*, Rational Rose*[Rational]) as well as, to metrics tools (such as *Datrix* [Datrix]).

Quality Evaluation

As mentioned above, object-oriented methodologies aim to facilitate the reuse and maintenance of a system. Reusability and maintainability can be quantified by internal quality attributes, such as, abstraction and encapsulation. Specifically, the encapsulation is judged by the high cohesion in a class and low coupling between classes. The abstraction is reflected by object identification, inheritance and polymorphism. The satisfaction of such quality attributes is a major indicator of the success of the project. Therefore, it is critical to incorporate these target qualities into the migration process.

We propose a quality driven migration framework [Zou & Kontogiannis 2002]. Such framework aims to monitor and evaluate software qualities at each step of the re-engineering process. The re-engineering process is considered as a sequence of transformations that alter the system's state. The initial state corresponds to the original system and the final state corresponds to the

target migrant system. To incorporate quality control in the migration process, each state can be specified by a set of properties related to one or more target qualities (i.e. performance, maintainability, reusability) and denoted by feature vectors. The association of source code features with software quality requirements is achieved by the use of soft-goal interdependency graphs [Mylopoulos, 1992]. In this context, each transformation is denoted by a likelihood that contributes towards the desired quality goals. Thus, a consecutive transformation is selected based on the likelihood that it will improve the final quality of the system. A full transformation path can be formed by the composition of transformations from original state to the final state. The research addresses three major issues namely, the identification of a comprehensive set of possible transformation rules, the association of source code features with target qualities, and the design of an algorithmic process that can identify an optimal sequence of transformations to yield a target system with the desired qualities. To achieve the first research objective, the use of soft-goal graphs is proposed within the context of Non-Functional Requirements. Such graphs allow for modeling the impact of specific design decisions towards a specific target requirement. The leaves of such graphs correspond to concrete system attributes that impact all the other nodes to which they are connected. In such graphs, nodes represent design decisions, and edges denote positive or negative dependencies towards a specific requirement. The second research objective is addressed by devising a comprehensive, yet extensible list of transformation rules that can be used to migrate a procedural system to an object oriented one. A list of possible transformation rules is discussed in the previous subsections. Each transformation rule is formally specified in terms of pre/post conditions by the use of UML and OCL. Finally, the third research objective can be achieved by the utilization of Markov models and the Viterbi algorithm [Zou & Kontogiannis 2002], whereby the optimal sequence of transformations towards achieving

the desired qualities is identified. The selection of the transformations is based on quantitative methods and a probabilistic model that aims to quantify the magnitude by which deltas in specific source attributes after applying a transformation may contribute towards achieving a desired system property.

REPRESENTATION OF MIGRATED COMPONENTS

Within the context of re-engineering legacy systems into a new platform, such as distributed environments, it is important to analyze the legacy software to recover its major functional components that relate to an application domain and specific business logic. In the previous sections, we have discussed the techniques to identify reusable components and transform them into object oriented design. Once specific legacy components have been migrated into objects, their behaviors can be specified in terms of well-defined object oriented interfaces and provide the required description for the middleware wrapping.

In general, the interface of a software component may contain information related to data types, references to external specifications that point to related components, descriptions of public attributes and methods, and return types and parameters that specify input and output. The representation of a component interface can be independent of a specific programming language, making thus possible for wrapping technology to integrate software components into heterogeneous operating platforms and environments.

OMG IDL provides such a language-independent interface description for distributed CORBA components. It supports the basic specification for distributed components, such as the operations and attributes provided by the component. However, IDL cannot describe all of the information, such as, the pre/post-conditions, and semantic descriptions of functionality, of the distributed component. For example, the server component may match perfectly to the client

request by the signatures in the IDL interface, but the functionality of the offered service may not be the best one sought by the client process. Moreover, IDLs do not provide any additional information about the server's external dependencies such as, the call-back invocation of a client's method. Although IDL is human-readable in terms of its syntax, it is a programmable specification language which can be compiled into executable code. The description for the functionality of the interface can be indicated in comments, but it is not easy for an algorithmic process to derive such information from the IDL syntax or free-style IDL comments alone.

XML is another key technology that is widely used as distributed standard format for data representation, data integration, data storage, and message exchanging. XML files can be transferred via HTTP on the Web, and provide a machine understandable and human readable syntax, which allows developers to define their own tags in different domains [Elenko & Reinertsen 1999]. In this context, XML tags can be used to define the configuration information and provide the semantic specification for distributed software components. As a result, the combination of XML and CORBA IDL can facilitate the specification and localization of distributed services in Web-based environments.

The XML interface representation of a software component consists of several aggregated elements, such as data type definitions, interface operation definitions, and interface data member definitions. The data type definition publishes the types other than the defined interface for the purpose of invoking interface methods. The interface data member definition declares the accessor and mutator methods associated with a data member. Such an example specification for a component extracted from the AVL GNU tree libraries, is illustrated in Figure 3. The AVL tree library was written in C. By the use of the object identification techniques discussed in previous sections, C++ classes are extracted from the procedural code. The new migrant AVL tree

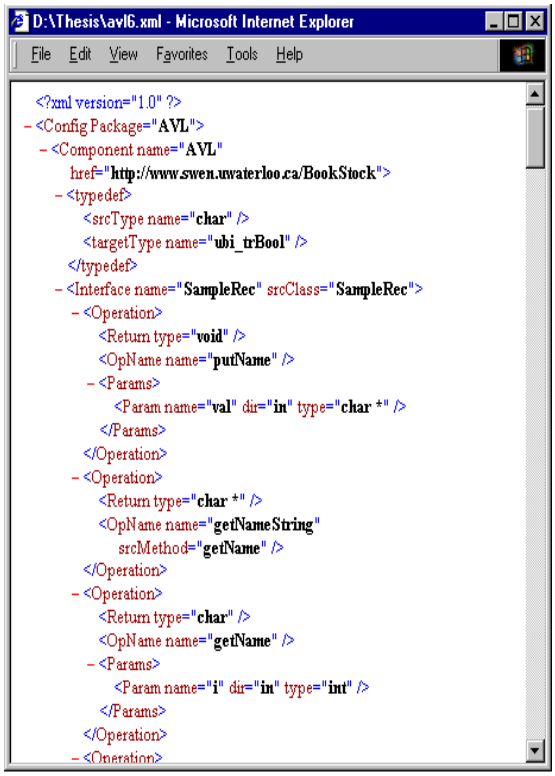

Figure 3: XML Representation of Component Interface

libraries [Kontogiannis & Patil 1999, Zou & Kontogiannis 2001] can be considered as a collection of distributed components that consist of several classes. Therefore, the interface for the AVL tree component consists of several sub interfaces that correspond to wrapper classes. Furthermore, with the XML specification, additional information can be easily encoded, such as the self-description information, URL address, pre- and post-conditions, and performance characteristics for the component. Such information introduces new functionality to wrappers, and can be used at the run-time by a binder to efficiently locate a service in a transparent fashion to the client.

GENERATION OF OMG IDL AND CORBA WRAPER

Essentially, the wrapper acts as a façade: it offers clients a single, simple interface to the underlying objects. It glues together the CORBA distributed capabilities and the standalone

```
module AVL{

interface corba_ubi_btRoot;
interface corba_ubi_btNode;
interface corba_SampleRec;

typedef char corba_ubi_trBool;

interface corba_SampleRec{
        void putName(in string val);
        string getName();
        void putNode(in corba_ubi_btNode val);
        corba_ubi_btNode getNode();
        long getDataCount();
        void putDataCount(in long aval);
};

interface corba_ubi_btNode {
        void putBalance(in char val);
        char getBalance();
        long Validate();
        //......
};

interface corba_ubi_btRoot{
        corba_ubi_trBool ubi_avlInsert(
                in corba_ubi_btNode  NewNode,
                in corba_SampleRec  ItemPtr,
                in corba_ubi_btNode OldNode );
        // ......
};
};
```

Figure 4: AVL Component Interface Definition components.

The wrappers implement message passing between the calling and the called objects, and redirect method invocations to the actual component services. The operations in the interface are the public operations in the CORBA wrapper. The concrete process to accomplish wrapping is accomplished in terms of three major steps.

The first step focuses on the specification of components in CORBA IDL as shown in Figure 4. In the second step, a CORBA IDL compiler translates the given IDL specification into a language specific (e.g. C++), client-side stub classes and server-side skeleton classes. Client stub classes and server skeleton classes are generated automatically from the corresponding IDL specification. The client stub classes are proxies that allow a request invocation to be made via a normal local method call. Server-side skeleton classes allow a request invocation received by the server to be dispatched to the appropriate server-side object. The operations registered in the interface become pure virtual functions in the skeleton class.

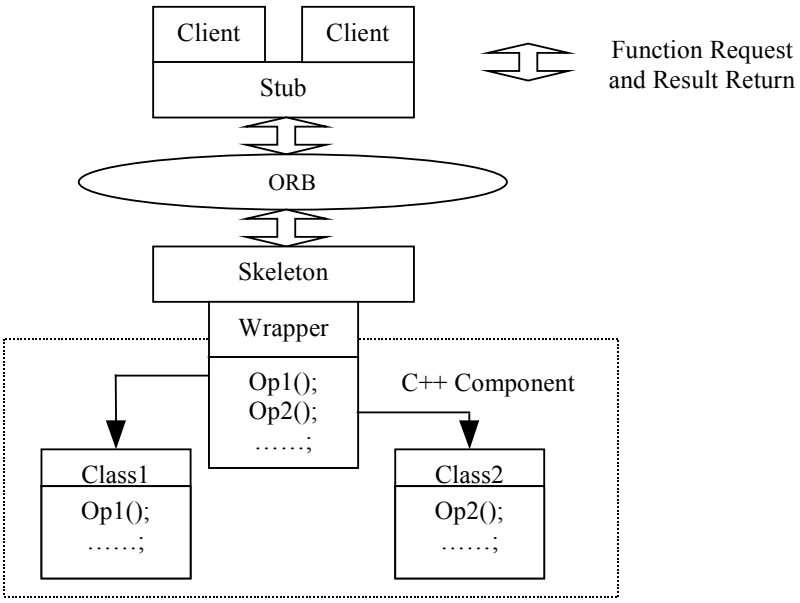

Figure 5: Wrapping a component

The third step focuses on wrapper classes that are generated and implemented as CORBA objects, directly inheriting from the skeleton classes. The wrapper classes encapsulate the standalone C++ object by reference, and incarnate the virtual functions by redirecting them to the encapsulated C++ class methods. The runtime structure of the wrapping process is illustrated in Figure 5. The new functionality of the legacy object can be added to the wrapper class as long as the method name is registered in the interface. The wrapper class can be generated automatically, using a high- level XML-based interface specification, such as the one presented in Figure 3 and Figure 4. For example, the ubi_btRoot class is one of the classes identified within the AVL tree component. The wrapper_ubi_btRoot inherits from the skeleton class sk_AVL::_sk_corba_ubi_btRoot, which is generated from the CORBA IDL to C++ compiler (Figure 6). The wrapper class, wrapper_ubi_btRoot, encapsulates a reference of ubi_btRoot class as shown in Figure 7. When a client invokes a method through CORBA, it passes the parameters with data types registered in the IDL interface to the server side object. The wrapper classes translate the CORBA IDL specific data types from the client calls to the data types used by the

```
class wrapper_ubi_btRoot :
        public _sk_AVL :: _sk_corba_ubi_btRoot
{
private:
        ubi_btRoot& _ref;
        CORBA::Boolean _rel_flag;
        char *_obj_name;
public:
wrapper_ubi_btRoot(
        ubi_btRoot& _t,
        const char *obj_name=(char*)NULL,
        CORBA::Boolean _r_f=0):_ref(_t);
~wrapper_ubi_btRoot();
ubi_btRoot* transIDLToObj(AVL::corba_ubi_btRoot_ptr obj);
void putRoot(AVL::corba_ubi_btNode_ptr val);
AVL::corba_ubi_btNode_ptr getRoot();
AVL::corba_ubi_trBool ubi_avlInsert(
        AVL::corba_ubi_btNode_ptr NewNode,
        AVL::corba_SampleRec_ptr ItemPtr,
        AVL::corba_ubi_btNode_ptr OldNode);
void Prune();
//Other methods are eliminated.
};
```

Figure 6: An Example Wrapper Class

```
ubi_btRoot* wrapper_ubi_btRoot :: transIDLToObj(
        AVL::corba_ubi_btRoot_ptr obj)
{
 if (CORBA::is_nil(obj)) return NULL;

// set up the data members of _ref object.
// these data members are primary data types.
_ref.putCount(obj->getCount());
_ref.putFlags(obj->getFlags());

//translate the ubi_btNode to corba_ubi_btNode_ptr
//by wrapper class rootWrap
ubi_btNode *rootImpl= new ubi_btNode();
if (rootImpl==NULL)return NULL;
wrapper_ubi_btNode rootWrap(*rootImpl,
                        _obj_name);

//translate corba_ubi_btNode_ptr type returned from
//obj->getNode() to ubi_btNode * by transIDLToObj()
// in wrapper object rootWrap.
_ref.putRoot(rootWrap.transIDLToObj(
                        obj->getRoot()));

//.....
  return &_ref;
}
```

Figure 7: Example for Object Type Translation

encapsulated C++ classes. Figure 7 illustrates the transformation from the CORBA specific type such as corba_ubi_btRoot_ptr to the ubi_btRoot used in the corresponding C++ method. In the same way, the wrapper classes convert the returned values from the C++ server class to the CORBA IDL specific data type, which is the wrapper class. In such a way, the wrapper object

not only allows clients send requests to the legacy object, but also allows the legacy object to return its result back to the clients. Since IDL does not support overloading and polymorphism, each method and data field within the interface should have a unique identifier.

If the polymorphic and overloaded methods occur in one class, it is necessary to rename these methods by adding a prefix or suffix to the original name when they are registered in the interface, avoiding changing the identified objects. This "naming" technique allows unique naming conventions throughout the system, without violating code style standards. The wrapper classes are responsible to direct the renamed overloaded and polymorphic methods to the corresponding client code.

If polymorphic and overloaded methods occur in the inheritance relationship, one can take advantage of C++ upcast feature, that is, to only register the sub-class in the component interface, and upcast the sub-class to its super class when the polymorphic or overloading methods in a super class are invoked.

Automatic Generation of IDL and CORBA Wrapping

Once the interface is defined, the process of generating CORBA wrappers is similar to each identified components. Therefore, automatic generation of wrappers and IDL specifications is feasible by providing the information about each interface, such as, signatures, and renamed operation name for overloaded methods. A tool can be built to facilitate the automatic generation. Specially, such a tool implements three tasks, namely, the mapping between C++ to IDL Data Type, the mapping between CORBA C++ to Native C++ Data Mapping, and the generation of IDL and wrapper code. We discuss the details for each task in the following subsections.

- **C++ to IDL Data Type Mapping**

In the XML interface specification, the signatures are indicated in terms of C++ types. Specifically, an IDL generator tool reads the XML interface description and converts the interface descriptions to IDL types, conforming to the mapping rules of the OMG IDL specification, and writes an IDL style interface to a new file.

The complete mapping of basic data types is given in [Henning & Vinoski1999]. For example, C++ “long” is mapped into IDL “long”, C++ “char *” to IDL “string”. Due to the platform independence of OMG IDL, some basic C++ data types are not specified, such as “int”, which is 16 bits in MS-DOS and Win3.1, but is 32 bits in Win32. For the identified component, we assume it works under a 32 bit operating system.

- **CORBA C++ to Native C++ Data Type Mapping:**

CORBA C++ stubs are generated from IDL compilers. Usually the CORBA C++ stubs are prefixed with the CORBA namespace. To invoke methods in the C++ code, parameters are passed from CORBA C++ wrapper objects to native C++ classes. Since the primary data types are passed by value, the parameters can be directly passed into native C++ methods, according to the same mapping rules from C++ to IDL. For complex data types, such as class, string, array, parameters are passed by reference. Signatures of such types in wrapper classes are defined by pointers. In this case, the IDLToObj() method is added to CORBA wrapper object. The IDLToObj() method translates from a CORBA C++ wrapper pointer to a native C++ class pointer.

- **IDL and Wrapper Code Generation:**

According to the XML component interface representation, the wrapper IDL interface is generated. The <Interface> tag in the XML specification (Figure 3) denotes the interface to be generated. Externally visible operation identifiers are extracted from the children elements under

the <Operation> elements. Operation name, return type and parameters' names and their corresponding types are some of the children elements. Meanwhile, C++ to IDL type conversion is automatically performed. In addition, the type definition information indicated under the <typedef> element is added to the top of the generated IDL file. Other global variables are defined as well. The generated IDL interface is shown in Figure 4. The CORBA wrapper header file, which declares a wrapper class in CORBA C++, is created using the same technique.

The wrapper function bodies are generated in a separated file, which provides the implementation of each methods. According to the "srcClass" attribute (Figure 3) value in the <Operation> tag and "srcMethod" attribute value in <OpName> tag, the wrapper re-directs the invocation of its public operations to the corresponding source methods in the encapsulated source C++ object. At the same time, the wrapper generator provides the "glue" code, which is responsible for translating the CORBA IDL specific data typed to the native C++ data typed, and vice versa.

Although most of CORBA IDL compilers provide the generation of a "tie" class from the IDL specification, such tie class is not sufficient for legacy code wrapping. Tie classes simply call the corresponding methods in the encapsulated class. However, they do not provide any functionality to translate CORBA IDL specific data types to native C++ data types. The code only works, when the parameters are basic data types. For a complex data type, they cannot convert it to a native C++ pointer that can be recognized by the legacy code. In this context, the tie class is too simple to address the requirements for legacy code wrapping. The XML-based interface description of the software components, and the CORBA IDL wrapper generation process overcome this limitation.

DEPLOYING MIGRATED COMPONENTS IN WEB SERVICE ENVIRONMENTS

Web service environments are Java based platforms. However, legacy systems are mostly written in procedural code. As aforementioned, CORBA provides a language neutral solution to integrate legacy code in a distributed environment. Although CORBA provides the IIOP protocol to enable Web clients to access CORBA objects, IIOP messages cannot pass through firewall. Therefore, to enable the universal access to legacy components in the Web, the best way is to combine the CORBA and Web services, and implement CORBA objects as Web services. In previous section, we provided techniques to automatically generate CORBA wrappers that encapsulate the identified software components as distributed objects. The wrappers implement message passing between the calling and the called objects, and redirect method invocations to the actual component services. Furthermore, CORBA objects can be integrated into Web services environments. In this context, the SOAP protocol is adapted as a uniform messenger between CORBA objects and the rest of Web services. The layered structure is illustrated in Figure 8. A Web client sends a request, in conformance to SOAP, through HTTP to a Web server. The Web server redirects the request to the servlet engine, which is responsible for processing the HTTP/POST and HTTP/GET requests. The SOAP run time engine handles the interpretation and dispatching of SOAP message. At the heart of the structure, a SOAP/CORBA IDL translator, is implemented as a collection of Java Beans to process inbound SOAP requests, which represent invocations to back-end CORBA objects. Internally it encapsulates all the services like SOAP serialization and deserialization, bridges the gap between the SOAP RPC and CORBA IDL and finally, generates the stub to dispatch the requests to the back-end services on the client's behalf. A fully functional two-way SOAP to CORBA converter is provided by

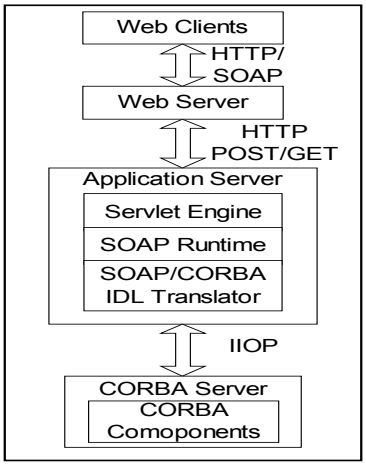

Figure 8: SOAP Wrapping Structure

[Sopa2Corba]. Meanwhile, the WSDL (Web Service Description Language) is used to describe the services of back-end CORBA objects and to register them in a UDDI central repository. Thus the services can be located easily during runtime by querying the UDDI repository.

CONCLUSION AND FUTURE WORK

Ongoing technological advances in object oriented programming, multi-tier architectures, and Web services pose great challenges in the modernization and evolution of legacy systems. In this chapter, we presented a comprehensive migration framework that leverages legacy software assets into Web environments. Instead of wrapping a legacy system as a black box, we focuses on the techniques to decompose legacy systems into finer grain components, transform the identified components into object oriented designs, wrap migrated components as distributed objects, and facilitate the universal Web enabled service invocation and integration of these components. In this context, these components can be easily reused and programmably composed to produce new business logic within Web service oriented architecture. As a core of this framework, we adopt several software engineering design heuristics and rules as a guideline to identify object models from procedural code and establish inheritance and polymorphism

relations among identified class candidates. We aim to achieve good qualities, such as high encapsulation, high cohesion, low coupling and high abstraction in the identified components. In this respect, these components have minimum dependencies on external resources and offer clear interfaces for later wrapping and integration. Moreover, it is important to specify the properties of distributed components in terms of functional and non-functional requirements, such as reliability, security, and performance. Therefore, the components can be selected to meet exact users' requirements. In this context, we utilized XML and CORBA IDL to specify component interfaces. Such an XML based interface specification allows the automatic CORBA wrapper generation. Moreover, we can incorporate QoS requirements in the component identification and wrapping process. The capabilities of the identified components can be enhanced to conform to specific quality criteria and requirements, which the user considers to be essential for the success of the migration into a distributed environment.

We believe that future research on the evolution of legacy systems in Web environments is very critical and needed due to the pervasive adoption of distributed architecture. Extensive research on the migration, description, composition, and localization of existing legacy systems to the Web service infrastructure is needed to adapt valuable legacy assets to continuously evolving requirements and environments. Furthermore, existing Web applications are the legacy systems of tomorrow. Reengineering techniques would be needed to restructure such applications in order to improve the qualities of today's Web applications.

REFERENCE

Aversano, L., Canfora, G., Cimitile, A., & Lucia, A. D. (2001). "Migrating Legacy Systems to the Web: an Experience Reprot", Proceedings of the Fifth Conference on Software Maintenance and Reengineering (CSMR).

Blaha, M. R. & William J. Premerlani, W. J. (1995). “Observed Idiosyncracies of Relational Database Designs”, Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Ontario; July 14-16.

Bodhuin, T., Guardabascio, E. & Tortorella, M. (2002).“Migrating COBOL Systems to the WEB by using the MVC design pattern”, in the proceedings of the Ninth Working Conference on Reverse Engineering.

Brodie, M. J. & Stonebaker, M. (1995). “Migrating Legacy Systems – Gateways, Interfaces & Incremental Approach”, Morgan Kaufmann Publishers, Inc., San Francisco, CA.

Brown, A. & Barn, B. (1999). “Enterprise-Scale CBD: Building Complex Computer Systems From Components”, in the proceedings of Software Technology Education & Practice, September.

Datrix. http://www.iro.umontreal.ca/labs/gelo/datrix/.

Dreyfus, P. (1998). “The Second Wave: Netscape on Usability in the Services-Based Internet”, IEEE Internet Computing, March/April.

Elenko, M. & Reinertsen, M. (1999). “XML & CORBA”, http://www.omg.org.

Etzkorn, L. H. & Davis, C. G. (1997).“Automatically Identifying Reusable OO Legacy Code”, Computer, IEEE, October.

Fowler, M. (2000). “Refactoring: Improving the Design of Existing Code”, Addison-Wesley.

Henning, M. & Vinoski, S. (1999). “Advanced CORBA Programming with C++”, Addison-Wesley.

Kontogiannis, K. (1998). “Code Migration Through Transformations: An Experience Report”, IBM Centers for Advanced Studies Conference.

Kontogiannis, K. & Patil, P. (1999). "Evidence Driven Object Identification in Procedural Code", Software Technology Education & Practice, Pittsburgh, Pennsylvania.

Lanubile, F. & Visaggio, G. (1997). "Extracting Reusable Functions by Flow Graph-Based Program Slicing", IEEE Transactions on Software Engineering, Vol. 23, No. 4, April.

Lau, T., Lu, J., Mylopoulos, J. & Kontogiannis, K. (2003). "The Migration of Multi-tier E-commerce Applications to an Enterprise Java Environment", Information Systems Frontier 5:2, 149-160.

Lindig, C. & Snelting, G. (1997). "Assessing Modular Structure of Legacy Code based mathematical Concept Analysis", in the Proceedings of the 19th International Conference on Software Engineering.

Lucia, D., Di Lucca, G. A., Fasolino, A. R., Guerra, P. & Petruzzelli, S. (1997). "Migrating Legacy Systems toward Object Oriented Platforms", in the Proceedings of International Conference of Software Maintenance (ICSM97).

Mancoridis, S., Mitchell, B. S., Chen, Y. & Gansner, E. R. (1998). "Using Automatic Clustering to Produce High-Level System Organizations of Source Code", IEEE Proceedings of the 1998 Int. Workshop on Program Understanding (IWPC'98).

Martin, J. "Ephedra – A C to Java Migration Environment", http://ovid.tigris.org/Ephedra/.

Moore, M. & Moshkina, L. (2000). "Migrating Legacy User Interfaces to the Internet: Shifting Dialogue Initiative", Proceedings of Working Conference on Reverse Engineering, Brisbane, Australia, December.

Muller, H., Orgun, M., Tilley, S. & Uhl, J. (1992). "Discovering and Reconsstructing Subsystem Strctures Through Reverse Engineering", technical Report DCS-201-IR, Department of computer Science, University of Victoria, August.

Mylopoulos, J., Chung, L. & Nixon, B. (1992).“Representing and Using Nonfunctional Requirement: A Process-Oriented Approach”, IEEE Transactions on Software Engineering, Vol. 18, No. 6, June.

Picon, J., *et al*, “Enterprise JavaBeans Development Using VisualAge for Java”, http://www.redbooks.ibm.com.

Rational Rose. http://www.rational.com/products/rose/index.jsp?SMSESSION=NO

Sahraoui, H. A., Melo, W., Lounis, H. & Dumont, F. (1997). “Applying Concept Formation Methods To Object Identification In Procedural Code”, In Proceedings of the Twelveth Conference on Auotmated Software Engineering.

Sneed, H. (1996).“Object Oriented COBOL Recycling”, in the Proceedings of the Third Working Conference of Reverse Engineering.

SOAP. “SOAP: Simple Object Access Protocol”, http://www-4.ibm.com/software/developer/library/soapv11.html

Soap2Corba and Corba2Soap, http://soap2corba.sourceforge.net/.

Stroulia, E. *et. al*, (1999). “Reverse Engineering Legacy Interface: An Interaction-Driven Approach”, In the Proceedings of the 6th Working Conference on Reverse Engineering, 6-8 October, Atlanta, Georgia USA.

Together. http://www.borland.com/together/controlcenter/index.html.

UDDI. “Universal Description, Discovery and Integration: UDDI Technical White Paper”, http://www.uddi.org.

Umar, A. (1997). “Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies”, Prentice Hall PTR.

Ward, M. P. (1999). “Assembler to C Migration using the FermaT Transformation System”, International Conference on Software Maintenance.

WSDL. Web Services Description Language (WSDL) 1.0", http://www-4.ibm.com/software/devleoper/library/w-wsdl.html.

Zou, Y. & Kontogiannis, K. (2000). “Migration and Web-Based Integration of Legacy Services”, the 10th Centre for Advanced Studies Conference (CASCON), Toronto, Ontario, Canada, November, pp. 262-277.

Zou, Y. & Kontogiannis, K. (2001). “A Framework for Migrating Procedural Code to Object-Oriented Platforms”, in the Proceedings of the eighth IEEE Asia-Pacific Software Engineering Conference (APSEC), Macau SAR, China, December, pp. 408-418.

Zou, Y. & Kontogiannis, K. (2002). “Migration to Object Oriented Platforms: A State Transformation Approach”, Proceedings of IEEE International Conference on Software Maintenance (ICSM), Montreal, Quebec, Canada, October, pp.530-539.