Supporting Migration to Services using Software Architecture Reconstruction

Liam O'Brien[1]
Lero-ISERC
University of Limerick
Ireland
+353 61 202434
liam.obrien@ul.ie

Dennis Smith, Grace Lewis
Software Engineering Institute
Carnegie Mellow University
4500 Fifth Avenue
Pittsburgh, PA 15213 USA
+1 412 268 6850
{dbs, glewis}@sei.cmu.edu

ABSTRACT
There are many good reasons why organizations should perform software architecture reconstructions. However, few organizations are willing to pay for the effort. Software architecture reconstruction must be viewed not as an effort on its own but as a contribution in a broader technical context, such as the streamlining of products into a product line or the modernization of systems that hit their architectural borders, that is require major restructuring. In this paper we propose the use of architecture reconstruction to support System Modernization through the identification and reuse of legacy components as services in a Service-Oriented Architecture (SOA). A case study showing how architecture reconstruction was used on a system to support an organization's decision-making process is presented.

Keywords
Architecture, Architecture Reconstruction, Service Oriented Architecture, Migration to Services, System Modernization.

1 INTRODUCTION

The area of software architecture reconstruction has made substantial progress over the past several years [3,5,7,8,9,11,14,15,18,20,21]. A number of techniques and methods have been developed along with tools to support them [4,6,12,19]. Software architecture reconstruction has been used to:

- (Re)Document the architecture of existing systems to improve the understanding of the architecture
- Check the conformance of an as-designed architecture with the as-built architecture and validate unexpected dependencies
- Trace architecture elements to the source code, for example to measure the impact of architectural changes

Stoermer, et al. [24] outlined a set of application contexts that are relevant for architecture reconstruction. In this paper we focus on how architecture reconstruction can be used to support System Modernization through the identification and reuse of legacy components as services in a Service-Oriented Architecture (SOA).

There has been much work on moving existing legacy functionality to web services or to other web environments. Sneed and Sneed [23] outline an approach for creating web services from legacy host programs. Kontogiannis and Zhou [14] outline an approach to migrating legacy applications through identification of major legacy components and migrating these procedural components to an object-oriented design, specifying the interfaces, automatically generating the wrappers and seamlessly interoperating them via HTTP based on SOAP messaging. Litoiu [17] outlines issues such as performance and scalability related to migration of legacy applications to web services.

Several obstacles need to be overcome for organizations to be able to mover more effectively toward a service-based approach:

- Usually legacy systems are not well understood or documented,
- The dependencies between the components that can be migrated to services may not be known or documented
- There is a need to determine the feasibility of whether or not to invest in doing the migration.

For a component to be migrated to a service, the component should ideally be self-contained and loosely

[1] The work outlined in the paper was carried out while the author was at the Software Engineering Institute.

Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)
0-7695-2639-X/05 $20.00 © 2005 IEEE

coupled with the rest of the system. If a component has a lot of dependencies, especially functional dependencies where the component is calling other functionality outside of it, then this would reveal that the component is not self-contained. If there are many calls or other dependencies from outside the component to the component then it may highlight that the component is tightly coupled to other parts of the system, as opposed to the loose coupling that is desirable for a service.

This paper focuses on the use of architecture reconstruction as a decision-making tool. Using architecture reconstruction can help an organization get a better understanding of legacy applications and identify and document component dependencies that may not have been documented or are unknown.

We have applied this approach with an organization that wanted to analyze the feasibility of migrating and reusing several of its legacy components as services in an SOA. To get a better understanding of the legacy system we used the ARMIN tool to analyze the legacy system and we report on the outcome of that analysis.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of architecture reconstruction and the ARMIN tool. Section 3 outlines how architecture reconstruction can be used to identify dependencies between components in a system and produce better documentation. Section 4 outlines the case study in applying architecture reconstruction for better decision making on migration to an SOA. Section 5 outlines the implications of the analysis we undertook in the case study. Section 6 presents conclusions and outlines some future work.

2 ARCHITECTURE RECONSTRCUTION

Architecture reconstruction is the process of reconstructing or recovering the architecture of an implemented system. The SEI has developed tools to support this process; it currently uses the Architecture Reconstruction and MINing (ARMIN) tool. ARMIN uses information that is extracted from the source code in the form of an RSF (Rigi Standard Format) file. An RSF file has a set of elements and relations between these elements. The process that is used to reconstruct views of the architecture is outlined in Figure 1.

To extract the information from the source code we use commercially available tools such as the Understand toolset from Scientific Toolworks Inc. [22] or the Imagix 4D tools from Imagix Corporation [10]. After the data is extracted, it is imported into ARMIN.

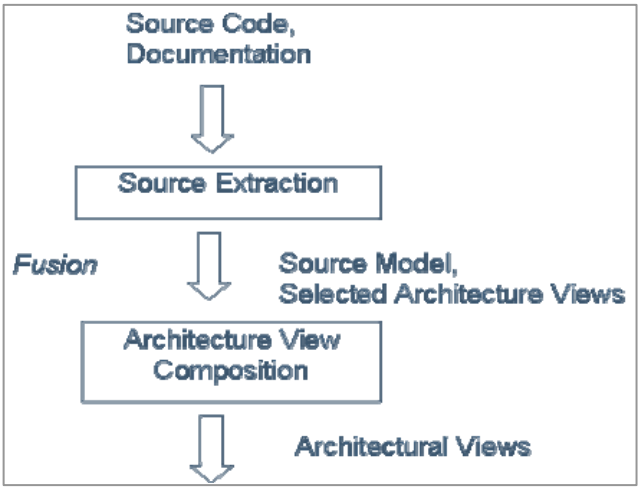

Figure 1: Architecture Reconstruction Process

The ARMIN tool has several major components:

- Navigator – for defining a project, including the elements and relations that make up the source model, and loading and browsing the source model from the RSF file.
- Aggregator – for displaying the views of the generated models.
- Interpreter – for creating the abstractions and models using ARMIN's Architecture Reconstruction Language (ARL).
- Repository – for storing the data loaded into a project.

ARMIN has been used to analyze systems written in C, C++, Fortran and Java, and in various combinations of these languages. It has been used on systems as large as 5 million lines of code.

As the person doing the reconstruction follows the reconstruction process using ARMIN, abstractions are built using ARMIN's ARL command script to aggregate information and produce higher-levels models and views of that information. Abstractions could include hiding functions inside of files or aggregating classes or files into components. This process is continued until the desired set of views is produced. For example, if a Layer view of the system is required, a typical set of abstractions may include hiding functions within classes, aggregating classes into components, and aggregating components into layers. A typical Layer view generated by ARMIN is shown in Figure 2. This view has three layers; Application, Bootloader, and Communication and shows the dependencies between the layers. These dependencies can be functional (calls) or data usage dependencies.

3 IDENTIFYING DEPENDENCIES BETWEEN COMPONENTS FOR MIGRATION TO SERVICES

In order to migrate legacy components to services in an SOA it is important to identify and understand the dependencies for each legacy component. There can be various types of dependencies between a component and the rest of the system. Dependencies include:

- Functional dependencies where the component uses other parts of the system in order to carry out its functionality or other parts of the system use the components or parts of them.
- Data dependencies where global data is shared between a component and other parts of the system.

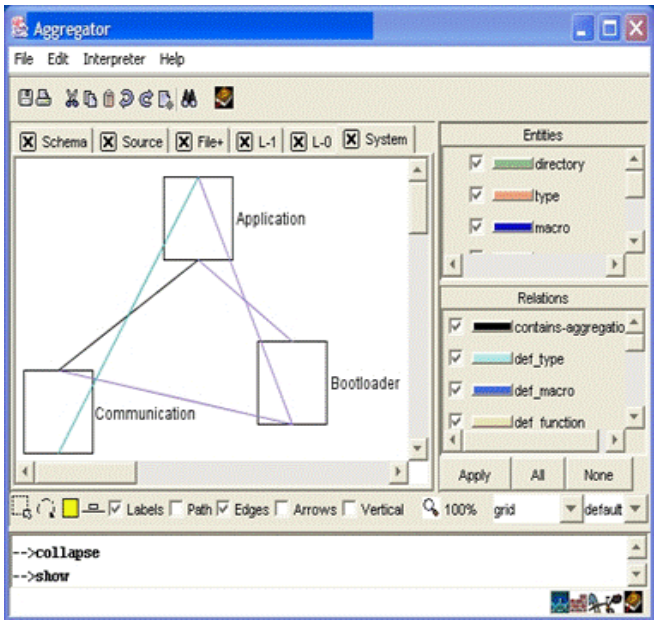

Figure 2: Layer View Generated by ARMIN

In order to identify these dependencies we can extract the detailed information from the source code that will build abstractions that highlight these dependencies. We can extract information about the elements (such as the functions, files, or classes) that are contained in the system, and additional elements such as directories if the system is decomposed on a file system.

Along with the elements we can also extract a set of relations between these elements. Specific relations between the elements that highlight dependencies include:

- Calls between functions – functions that call others – *calls function function*
- Data definitions – definition of global variables – *defines_var file variable*
- Data usage – functions that use global variables (both references and assignments) – *uses_var function variable*

We can also extract additional relations such as:

- The relationship of files and directories –files that are contained within a directory – *contains_file directory file*
- The relationship of files or directories to subsystems or potentially components – files or directories that make up a subsystem or component – *in_comp component file* or *in_comp component directory*

The latter relations are useful if the system is decomposed in a file structure and the subsystems and components can be mapped to directories. Otherwise this information may be of little use in the reconstruction process.

Identifying Components

If an organization is already investigating how to migrate parts of the legacy system to services, , they may already be familiar with the components that have been identified as potential services and the elements that make up those components (group of classes or files that constitute a component). If not, we can use various techniques to identify the components. There are different mechanisms for doing this including grouping, clustering and pattern recognition.

Once we have identified the approach for carrying out the abstractions, we can build a command script in ARMIN's ARL that will automatically build these abstractions. The ARL has built-in operators that allow for gathering all elements within a relation into a multi-dimensional list, merging parts of the list, removing elements and relations from a particular view, and generating visual representations of an underlying graph structure.

An example of a view of components for a system that is produced in ARMIN is shown in Figure 3. This view is from a reconstruction of the Duke's Bank System, which is part of the EJB tutorial from Sun Microsystems [2].

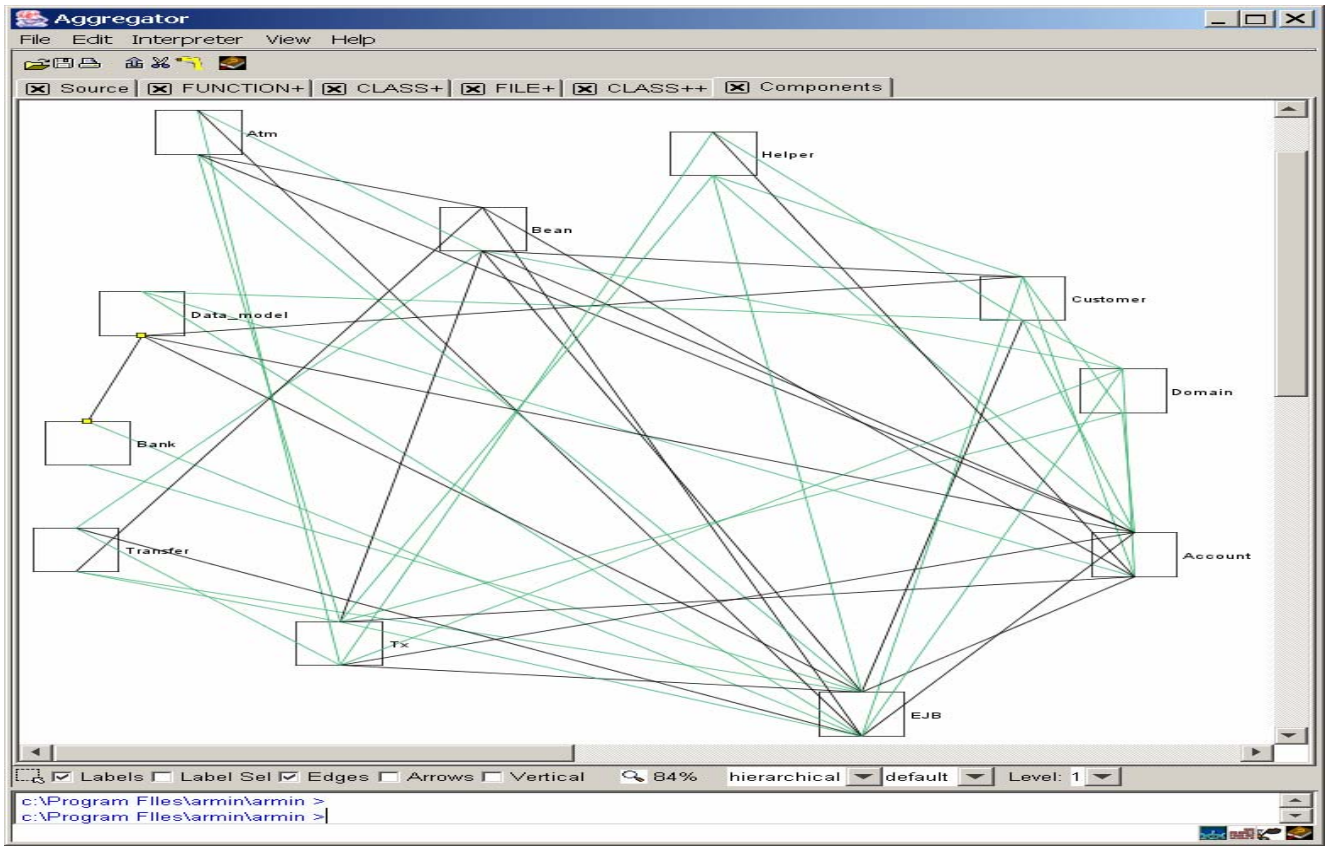

Figure 3: Component View produced by ARMIN

Identifying Dependencies between Components

During the reconstruction process, the relations between the elements are aggregated or lifted into higher level abstractions. For example, a call between functions defined in different files becomes a dependency between files when the functions are aggregated into files. If a further aggregation of files to components is done, then the call becomes a dependency between components.

Identifying all of the external dependencies of a component is important when migrating the component to a service. Of particular importance are dependencies between components that are candidate services, and between a component and the rest of the legacy system.

The dependencies that a component has, especially functional dependencies to other parts of the system, may mean that this component does not contain a self-contained piece of functionality. Depending on the number and type of dependencies from other parts of the system to the component, the component may be tightly coupled – a condition that goes against the principles for what a service should be.

From the components view that is produced in ARMIN it is possible to select a particular component and show the dependencies that component has on the other components in the system. Figure 4 shows a view of the dependencies for the Account component (from Figure 3). The arrows show the direction of the dependency. For example the Atm component is dependent on Account but there is no dependency in the other direction.

It is possible to identify the precise dependency between components represents by selecting an edge connecting two components and showing the relations that make up the connections. For example, the edge between the Account and Data_model components in Figure 4 identifies the dependencies between these components. The dependencies are shown in more detail in Figure 5.

These dependencies include functional dependencies - in this case method calls between classes in the components, and additional dependencies between the classes. By examining the code, the reason for these additional dependencies can be identified. In some cases these are caused because instance variables used in Data_model are of type AccountDetails or AccountController.

Using these techniques it is possible to identify the elements that make up the components that can potentially

be migrated to services, and the dependencies between these components and the rest of the legacy system. We have applied this approach on a large system from a DoD organization. Details of this case study are outlined in the next section.

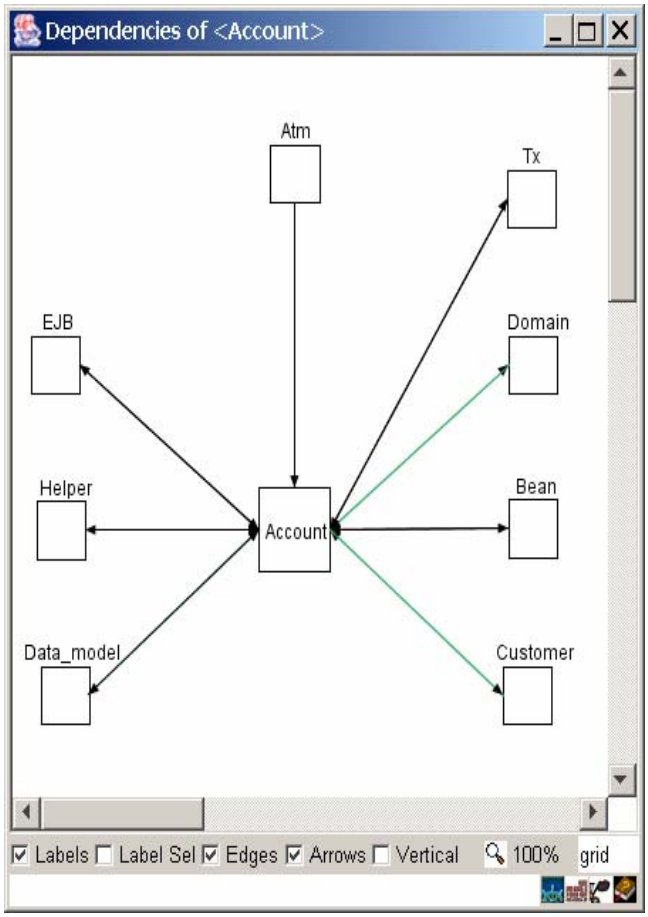

Figure 4: Dependencies for the Account Component

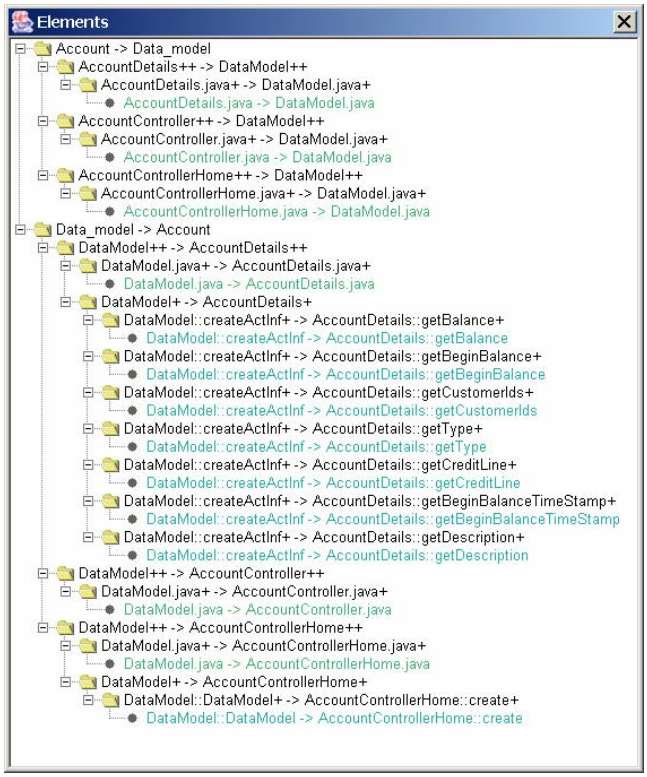

Figure 5: Dependencies between Account and Data_model

4 CASE STUDY IN SERVICE MIGRATION

In the case study, existing components of a Command and Control (C2) system were being evaluated for their potential to become services in an SOA. The owners of the C2 system recognized that if a selected set of components from their system are converted to services, they may have applicability for a broad variety of purposes. Our role was to perform a preliminary evaluation of the feasibility of converting a set of their components to services within a DoD SOA.

We initially met with the government owners of the system and the contractors who had developed the system to get an overview of the legacy systems, the history of the systems, the migration plans, and the drivers for the migration. We were also provided with a brief orientation to the target SOA. The DoD SOA was being developed so that C2 applications can be built as a set of interactions between infrastructure services (e.g., communication, discovery) and services that are specific to a domain (application domain services).

The system owner had done a preliminary identification of potential application domain services (ADS) that could be built from components of the legacy system. This analysis was derived from high level requirements for potential applications that were being targeted as users of services to

be provided by the SOA. The system owner had matched legacy functionality to these high level requirements and provided some initial estimates of the contents of the potential services.

Target SOA

The target SOA is currently under development. We investigated the target SOA by analyzing available documentation and by meeting with the developers. It is being built using a variety of commercial products and standards, along with significant custom code. The SOA effort is focused on satisfying a number of specific quality attributes important to the DoD, such as performance, security, and availability. In order to meet these needs, the SOA imposes a number of constraints on potential services. Because the SOA is still under development, the specifications for how to deploy and write services are still unclear. The target SOA is illustrated in Figure 6.

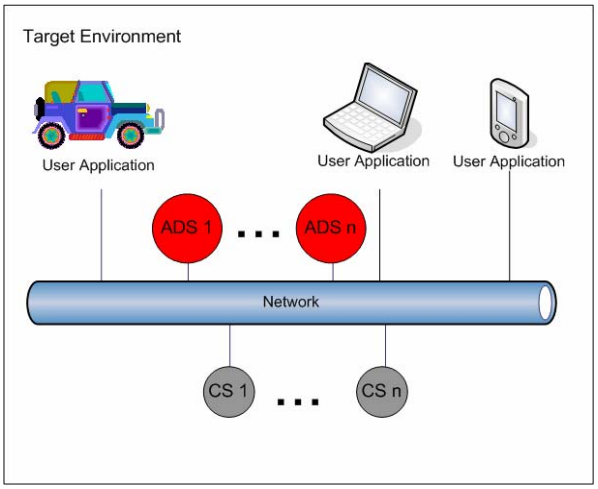

Figure 6: Physical View of the Target SOA

Figure 6 shows that the SOA includes common services (CS) that are to be used by user applications and application domain services (ADS). The SOA owns the interfaces for the common services. The environment then allows for a set of ADSs which will derive their requirements from user applications. Groups within the DoD are invited to submit proposals for services to meet these requirements, either by building them from scratch or by migrating them from legacy components. These requirements then need to be analyzed in detail and matched to existing functionality to determine what can be used as-is, what has to be modified, and what has to be new development.

Even though the full details of compliant services for the SOA have not yet been worked out, the SOA imposes a number of constraints on organizations that are developing ADSs from existing legacy components. Some of the constraints/requirements for developers of ADSs include:

1. An ADS needs to be self-contained, that is, it should be able to be deployed as a single unit.

   As expressed earlier, the reason why SOAs have been a success, especially in industry, is because they provide standard interfaces to legacy systems, while these systems remain largely unchanged.

   This is not the case in the target SOA. Services need to be stand-alone so that they can be deployed as needed on standardized platforms. In a legacy component, functionality that has been identified as part of a service needs to be fully extracted from the system, including code that corresponds to shared libraries or the core of a product line.

2. In the target SOA, an ADS has to be able to be deployed on a Linux operating system.

   For Windows-based legacy components this could be a problem, especially if there are dependencies on the operating system through direct system calls or if there is a dependency on commercial products that are only available for Windows systems. Ideally, system calls should be eliminated. If it is not possible, they should be evaluated to see if there are equivalents in the Linux operating system or if this functionality is part of one of the common services.

3. All services will share a common data model and all data will be accessed through a Data Store common service.

   The need for a common data model is driven by a desire for information to be shared and understood by all user applications. As a result, services will no longer define internal data. All data will be defined as part of the common data model. Legacy components need to replace all dependencies on databases and file systems with calls to the data store service and make sure that all the data they need is part of the common data model.

4. An ADS will use the Discovery common service to find and connect to other services.

   If the ADS will rely on other services, code to discover and connect to these services will have to be written. Once the service is developed it needs to be advertised. This is done by registering the service with the naming service. Once this advertised service has been registered, other applications that wish to use this service will perform a discovery on the available services and choose which service(s) they desire to use.

5. An ADS will use the Communications common service for communicating with other services.

The target SOA provides tools for generating data readers and data writers that will take incoming and outgoing data and format it accordingly.

Preliminary Analysis

The current system, written in C++ on a Windows operating system, has a total of about 800,000 lines of code and 2500 classes. In addition, the system has dependencies on a commercial database and a second product for visualizing, creating, and managing maps. Both commercial products have only Windows versions.

We met with the contractor and representatives of the government to focus on a limited number of legacy components and to select criteria for further screening. We focused on seven potential services that the government team had previously identified as part of its initial analysis of ADS requirements. These seven potential services contained 29 classes. The 29 classes that we selected enabled us to focus on potentials for high payoff. In conjunction with the team, we developed criteria for evaluating the potential reusable components.

Given the known and projected constraints of the target SOA, we performed a preliminary analysis of the legacy components using OAR [1] to provide a set of initial reuse estimates.

From this preliminary analysis we found that there was not adequate high-level documentation. Most of the documentation was in the form of code comments and from a tool *DOxygen* that extracts after-the-fact data from the C++ code, such as classes, attributes, dependencies, and comments. However, during the analysis we found that the DOxygen tool only picked up first-level dependencies. This indicated that the coupling and the amount of code that was used by each class was higher than could be estimated from the existing documentation. There was also no consistent programming standard, leading to idiosyncrasies between different programmers. This increased the difficulty of our analysis, and it would also increase the difficulty of any reuse. As might be expected from a relatively recent object-oriented system, we found the overall cohesion to be high. The contractor provided estimates for converting the components into services, based on a set of simplifying assumptions on the actual make-up of the target SOA and the final set of user requirements.

Because of the inadequacie in the architecture documentation, and the underestimation of the amount of code used by the potential services, there remained a number of gaps in our understanding of the system. For example, it was mentioned that one of the services made extensive use of the data model. This data model had over 1000 classes and was used by every class included in the potential services. Even though our analysis did not initially focus on the data model, because of its size it now represented the largest potential source of reuse in our study. However, the constraints of the target SOA may not enable the reuse of the data model.

As a result, it was not possible to accurately know how many other classes are used by a specific service. In addition the estimates for rehabilitation of the legacy components would have been understated. For example, the calls to user interface code would need to be removed, and it would be necessary to know where these are located.

To get a better understanding of these issues we performed a code analysis and an architecture reconstruction.

Code Analysis and Architecture Reconstruction

To address these issues, we first analyzed the code through a code analyzer "Understand for C++". This analysis provided:

- Data dictionary
- Metrics at the project, file, class, and function level
- Invocation tree
- Cross reference for include files, functions, classes/types, macros and objects
- Unused functions and objects

The code analysis enabled us to produce input for the architecture reconstruction tool that would identify dependencies.

As mentioned earlier, there were inconsistencies in the quality and documentation between different parts of the code that complicated the analysis:

1. Since there was not a consistent coding standard, we could identify individual differences between programmers.
2. Some parts of the code were much more difficult to navigate, with less cohesion and a more awkward file organization. Naming standards were different for files, classes, attributes, and method names. Code organization styles were different.
3. The organization of files was not standardized either. For example, it is not clear why some files that do not perform user interface (UI) functions are located in UI folders. Another example is that some *include* files are with the code files and others in a separate folder. Some files contained more than one class and there are no clear criteria for when this is allowed.

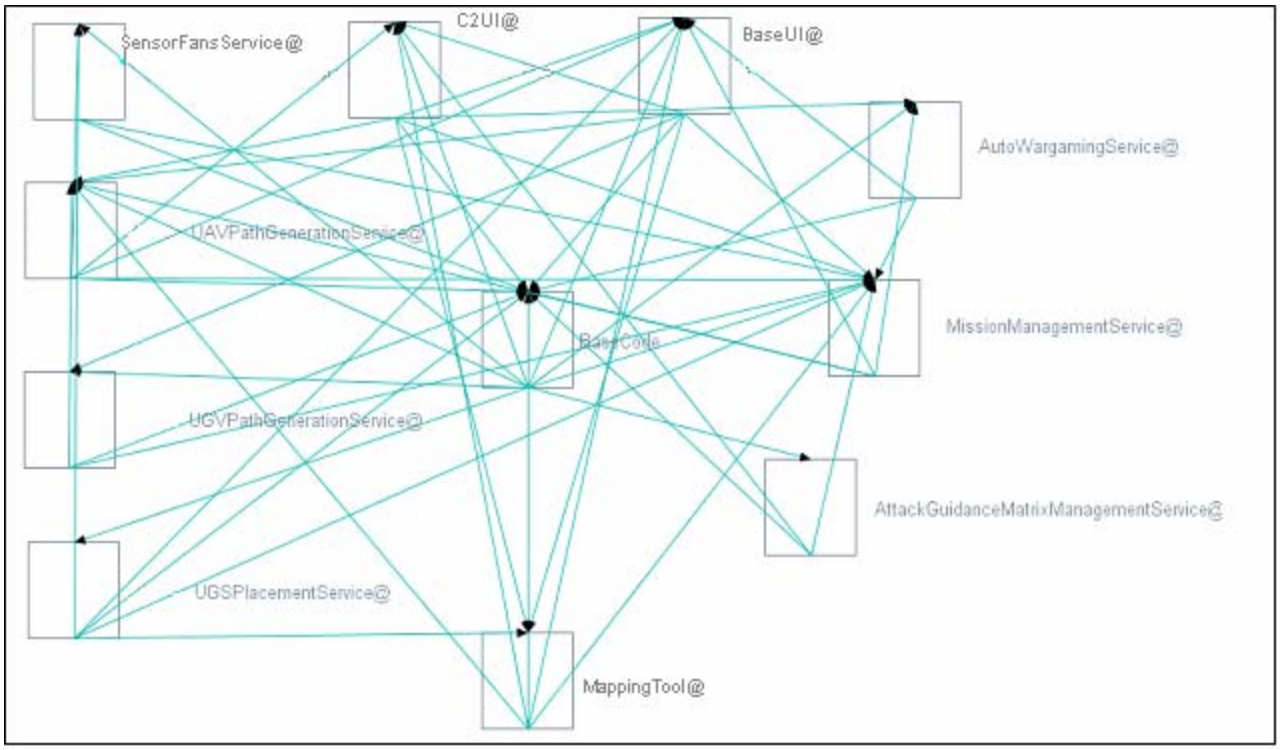

Figure 7: Component View of the Command and Control System produced by ARMIN

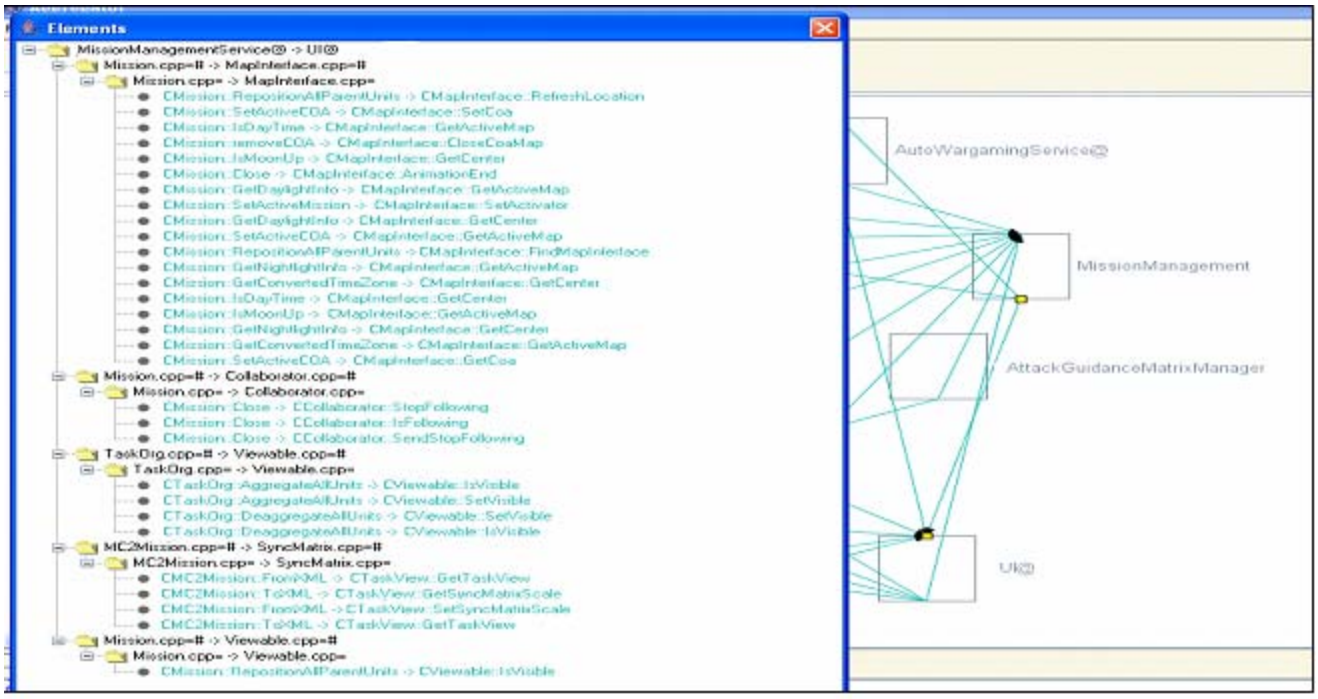

Figure 8: Dependencies between Potential Services

Despite these difficulties, that required us to gain quick high-level familiarity with the code, we were able to produce the input for the architecture reconstruction tool.

We next conducted the architecture reconstruction using ARMIN. To begin the architecture reconstruction, we took the output from the code analysis, and performed a focused analysis of the as-built architecture structure.

We aggregated the code into several groups

- One for each service analyzed
- One for code directly dependent on the commercial mapping software
- One for user interface code
- One for the rest of the code—data model, base classes, utilities, and code that did not belong to any of the above groups

Figure 7 shows the initial component view obtained using ARMIN.

In our analysis, we were interested in

- Dependencies between services and user interface classes
- Dependencies between services and the commercial mapping software
- Dependencies between services
- Dependencies between the services and the rest of the code that mainly represented the data model

Figure 8 shows an example of dependencies uncovered by the analysis. We found a substantial number of undocumented dependencies between classes indicating a higher level of risk and difficulty for the migration effort than had been apparent from the preliminary analysis. For example the preliminary estimate for one of the potential services was 4859 LOC. After the architecture reconstruction analysis that uncovered undocumented dependencies, we found that this service would actually consist of 66,178 LOC.

The architecture reconstruction also enabled us to document the central role of the data model, and to identify it as a potentially valuable reusable component, even though it had not been identified during the initial analysis. However, this finding was tempered by the fact that in the target SOA environment, potentially all services will have to use a common data model. If this is the case, all elements of the data model will have to be mapped to existing elements of the common data model. Negotiations would have to take place to make sure that data elements that are needed by the services become part of the common data model.

The architecture of the system is an example of the application of the *Model View Controller (MVC)* pattern. The architecture reconstruction found undocumented violations of the MVC architecture which would need to be addressed in any migration effort—specifically calls from the model to the view.

An example of the unanticipated impacts of moving from a standard systems development effort to an SOA can be seen from the product line approach that was taken in the initial development of the system. The product line approach was an excellent choice for the current application; however, it might increase the difficulty of the migration effort due to the potential requirement for services to be stand-alone for ease of deployment in the target SOA. The large dependency on base code and multiple levels of inheritance makes it difficult to isolate services. A potential solution to this problem would be to consider each service in itself as part of a product line, but this of course would require the set of core components to be potentially redefined.

5. IMPLICATIONS OF THE ANALYSIS

In looking at the potential for the service migration, the preliminary analysis suggested that the current legacy code represents a set of components with significant reuse potential. However, because the current legacy system does not have sufficient architecture or other high-level documentation, it was difficult to understand the "big picture" as well as dependencies between different classes and potential services. The architecture reconstruction provided an "as-built" representation of the structure of the system and its dependencies. It suggested that the significant dependencies between classes and potential services will make reuse and deployment of services more difficult. If the migration to service effort were to move forward, the results of the architecture reconstruction could enable a starting point for understanding how to disentangle these dependencies

In addition there is a risk in making migration decisions now because the target SOA has not been fully defined. While its overall structure has been defined, many of the specific mechanisms for interacting with it are still pending. Thus, it is not yet clear what the requirements for being a service in this environment will be in 12 or 18 months.

We also recommended that the government organization require the following changes from its contractors to make reuse of its legacy components more viable:

- Suitable set of architectural views
- Consistent use of programming standards
- Documentation of code to enable comments to be extracted using an automated tool
- Documentation of dependencies, especially when they violate architecture paradigms

Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)
0-7695-2639-X/05 $20.00 © 2005 IEEE

In addition, we recommended that the government organization take a proactive approach in working with the developers of the target SOA to understand implications of the current and evolving SOA plans.. The government organization should also work closely with the developers of the applications that will be using these services. Even though the technical part of the communication will be handled by a common service, the data that is transferred during the communication has to be negotiated—the contents of both the request and the response message that is communicated between the application and the service need to be defined. The government organization is continuing to work on its migration effort taking the points that we raised in our analysis into consideration.

6. CONCLUSIONS AND NEXT STEPS

We found that the use of architecture reconstruction to understand the as-built system provided an essential step in making decisions on the migration of the legacy components to services. The initial task of determining how to expose functionality as services, while seemingly straightforward, can have substantial complexity. Our conclusions to the client, while not definitive, did point out a number of issues that they had not previously considered. The use of architecture reconstruction techniques, in conjunction with other analytical methods, provides an essential set of analytical methods for decision-making.

We are currently developing a method that integrates architecture reconstruction with other analytical methods for reuse decision making, such as OAR. This method, Service-Oriented Migration and Reuse Technique (SMART) focuses on the specific issue of migration of legacy components to an SOA [16]. The next steps in the development of the method will be to:

- Include a greater number of specific tools that directly address the SOA concerns that need to be addressed when exposing functionality as services. We are developing the Service Migration Inventory (SMI) as the first of such tools.
- Incorporate decision rules on when it is most useful to include the code analysis and architecture reconstruction steps as part of the process.
- Make the process repeatable so that it can be used by the wider community. The tools and decision rules being developed are a first step in developing a repeatable process.

REFERENCES

1. Bergey, J.; O'Brien, L.; and Smith, D. "Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line," 316-327. Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2). San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002.
2. Bodoff, S.; Green, D.; Jendrock, E.; & Pawlan, M. http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html
3. Bowman, T.; Holt, R. C. and N. V. Brewster, Linux as a Case Study: Its Extracted Software Architecture. *Proceedings of the International Conference on Software Engineering*, Los Angeles, May 1999.
4. The Dali Architecture Reconstruction Workbench: http://www.sei.cmu.edu/ata/products_services/dali.html
5. Eixelsberger, W.; Ogris, M.; Gall, H. and Bellay, B., Software architecture recovery of a program family, *Proceedings of the International Conference on Software Engineering*, pp 508 –511, Kyoto Japan, April 1998.
6. Finnigan, P. J.; Holt, R.; Kalas, I.; Kerr, S.; Kontogiannis, K.; Mueller, H.; Mylopoulos, J.; Perelgut, S.; Stanley, M. and Wong, K., The Portable Bookshelf, *IBM Systems Journal*, Vol. 36, No. 4, pp. 564-593, November 1997.
7. Guo, G.; Atlee, J. and Kazman, R., A Software Architecture Reconstruction Method, *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, Texas, February 22-24, 1999 pp 225-243.
8. Harris, D.R.; Reubenstein, H. B. and Yeh, A. S., Recognizers for Extracting Architectural Features from Source Code, *Proceedings of the 2nd Working Conference on Reverse Engineering*, 1995.
9. Harris, R.; Reubenstein, H. B. and Yeh, A. S., Reverse Engineering to the Architectural Level, *Proceedings of the International Conference on Software Engineering (ICSE)*, pp 186-195, April 1995.
10. Imagix Corporation's Imagix 4D: http://www.imagix.com/
11. Kazman, R. and Carrière, S. J., Playing Detective: Reconstructing Software Architecture from Available Evidence, *Journal of Automated Software Engineering,* pp 107-138, April 1999.
12. KLOCwork inSight: http://www.klocwork.com/Accelerator.htm
13. Kontogiannis, K. and Zou, Y. Reengineering Legacy Systems Towards Web Environments, in "Managing Corporate Information Systems Evolution and Maintenance", Idea Group Publishing, Hershey, PA, USA. pp. 138-146, 2004.
14. Krikhaar, R. L., Software Architecture Reconstruction,

Ph.D. Thesis, University of Amsterdam, 1999.

15. Laine, P. K., The Role of Software Architecture in Solving Fundamental Problems in Object-Oriented Development of Large Embedded Systems, *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. Amsterdam, The Netherlands, pp 14-23, August 28-31, 2001.

16. Lewis, G, Morris, E., O'Brien, L., Smith, D., Wrage, L., SMART: The Service-Oriented Migration and Reuse Technique, (CMU/SEI-2005-TN-029, http://www.sei.cmu.edu/publications/documents/05.reports/05tn029.html.

17. Litoiu, M. Migrating to Web Services: Latency and Scalability. *Proceedings of the Fourth IEEE Workshop on Web Site Evolution (WSE'02), 2002.*

18. Mendonça, N. C. and Kramer, J., Architecture Recovery for Distributed Systems, *SWARM Forum at the Eight Working Conference on Reverse Engineering,* Stuttgart, Germany, October 2001.

19. The Portable Bookshelf: http://swag.uwaterloo.ca/pbs/

20. Riva, C., Reverse Architecting: An Industrial Experience Report, *Proceedings of the Seventh Working Conference on Reverse Engineering*, Brisbane, Australia, pp 42-50, November 23-25, 2000.

21. Sartipi, K. and Kontogiannis, K., A Graph Pattern Matching Approach to Software Architecture Recovery, *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001),* Florence, Italy, pp 408-419, November 7-9, 2001.

22. Scientific Toolworks Inc's Understand for C/C++/ Java /Fortran/Ada: http://www.scitools.com/

23. Sneed, H. and Sneed, S. Creating Web Services from Legacy Host Programs. *Proceedings of the Fifth IEEE Workshop on Web Site Evolution (WSE'03),* Amsterdam, Netherlands, 2003.

24. Stoermer, C., O'Brien, L. and Verhoef, C. Moving Towards Attribute Driven Software Architecture Reconstruction. *Proceedings of 10th Working Conference on Reverse Engineering, 2003,* WCRE 2003, Victoria, British Columbia, 2003.