Architectural Reconstruction in the Dark

Andrew Trevors and Michael W. Godfrey
Software Architecture Group
University of Waterloo
Ontario, Canada, N2L 3G1
{adtrevors,migod}@uwaterloo.ca

Abstract

The goal of architectural reconstruction is to produce high-level models of the design of a system. The recover process is usually aided by such information as source code, design documents, and user manuals. In the absence of this type of information, the process becomes difficult. This paper details an attempt to recover the architecture of the large piece of telephony software, where very little about the system is known by us.

1 Background

"Character is what you are in the dark"

- Dr. Emelio Lizardo

The goal of architecture recovery is to produce high-level models of a software system. Generally, these models are produced by extracting a relational model of the system from source code, manipulating it, and then mapping a system structure onto it (usually in the form of grouping files or functions into subsystems or packages). There are three key sources of information that are vital to the feasibility of this process:

- The system's source code;
- development artifacts such as a detailed conceptual architecture, or user manuals; and
- access to developers or persons who understand the structure of the system.

In practice, source code is the most important source for detailed understanding of a system in architectural recovery. Not only is it the basis for the relational model of the system (*i.e.*, what the components are, and how they interact with each other) but it can serve to help formulate part of the conceptual architecture (usually the directory structure or naming conventions of files and functions can hint towards a subsystem grouping). It is also helpful in answering questions about the system (comments in code are a good source for explaining peculiarities that one may come across).

When one is available, a conceptual architecture, which represents the "as-designed" view of the system, is usually the best source for determining what system structure to map onto the relational model. Other development artifacts, such as user manuals or design documents, are also helpful.

The developers can often provide valuable information about the system which may be missing. In the absence of a conceptual architecture, the developers may be able to suggest part of one from their knowledge of the system.

While it is obvious that not all three sources are needed in order to perform a successful architectural reconstruction, usually having at least one is a minimal requirement. Being given an already extracted relational model is sufficient as long as the model's schema is known and a system structure can be obtained from the conceptual architecture or from the developers themselves. Conversely, if the system structure was not available from these sources, one could probably be derived from the source code itself.

However, if these sources are lacking, is reconstruc-

tion still possible? In other words, if we have very little knowledge of the system and no way to learn about it, and the developers have no clear idea of what the system structure looks like is the entire process still feasible?

2 Recreating the Architecture

The lack of the sources listed in the previous makes it very difficult to recover the architecture of a large piece of software. This is the case in our attempts to reconstruct the architecture of large piece of telephony software (from here on called LPOTS) from a company in Canada.

We did not have access to any source code or development artifacts, and have limited access to developers or individuals knowledgable about the system. As a result, a suitable system structure could not be generated. In order for a meaningful architectural model to be produced, this structure must be devised by other means.

The problem of devising such a structure from an existing piece of software is known as ”clustering”. There are a number of automated or semi-automated techniques for trying to generate these clusters, which are discussed in the following sections.

2.1 Automatic Clustering Algorithms

The idea of automatic clustering is to generate a system structure by trying to group files and functions together based on their interactions with each other. A number of tools, such as ACDC [3] and Bunch [1], currently exist to do this. Generally, the idea is that files/functions that use each other belong together, while ones that don't belong apart. The criteria for deciding on groupings is typically based on the ”high cohesion/low coupling” principle of software design.

Another automatic clustering approach relies on the identification of patterns. Such examples include clustering functions which use or are being used by the same functions, or by identifying well known design patterns (*e.g.*, function b, c, and d are only ever called from a, and a is being called often, then a, b, c, and d might be involved in a facade pattern).

The problem with automatic clustering techniques is that they are not entirely accurate. Any cluster generated by such a tool, especially one for a system as large as LPOTS, would have to be validated.

2.2 Use Cases & Scenarios

Use cases are often very useful for identifying related functions/files that can be grouped together. The idea behind it is to identify a list of key use cases, and then trace their execution (either with automated tools or manually). If similar use cases are using many of the same functions, then there is a good possibility that those functions should be grouped together.

The problem with using use cases as a means to identify system structure in LPOTS is twofold. Firstly, neither the source code nor an executable is available so tracing the execution of a test case is very hard (simply trying to construct a path from a list of a thousand or more functions is not possible). Secondly, even if the above was available, the knowledge needed to generate enough use cases to provide suitable coverage of the system is lacking. This being the case, it is unlikely that use cases could be used to solely produce a system structure.

3 Case Study: LPOTS

Architectural reconstruction of the LPOTS system (whose code base is several MLOC) was done with the intent of supplying high-level diagrams of the system. In order to create these diagrams, we were given a relational model of the system, which we could manipulate and visualize using SWAGKit [2]. We were also given a system structure which we could use to map onto the manipulated relational model, thus giving us an architectural model.

However, this system structure was insufficient because it only mapped functions into more than packages. When we tried to visualize this architectural model, the diagrams were too cluttered and incomprehensible to be useful. What was needed was to further group these packages until we were left with a handful (probably 15 or fewer) of top-level entities.

We attempted to use automatic clustering tools like Bunch, but were unsuccessful in obtaining a cluster. Other methods, such as tracing use cases, were not feasible because of our lack of knowledge about the system (as well as the size).

Our final attempt was to use an automatic layout tool to provide an initial layout for the system, and then cluster packages that are close together into a single package, as well as look for patterns in the layout.

This approach seemed to yield results that were promising than our previous attempts. The layout that was given had the packages clustered into several groups in a top-down fashion resembling a layered architecture ((i.e),one group at top, another group underneath, and so on). Investigation of this layout with lsedit (a visualization tool in SWAGKit) seemed to support the idea of a layered architecture as it was found that most packages interacted with other packages in the same cluster, or those in the cluster below. As well, a few of the packages located in a small cluster near the top, interacted with almost all other packages, indicating a possible library or utility package.

Overall, the results of this attempt seem promising. The next step will be to locate (or create one if not found) a clustering tool which can recognize these patterns in layouts, and cluster accordingly. Other clustering tools will also be investigated in order to generate alternative clusterings, all of which we hope to validate/invalidate by consulting with system experts.

References

[1] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proc. of ICSM 1999*, 1999.

[2] SWK Software Architecture Toolkit. Website. http://www.swag.uwaterloo.ca/swagkit.

[3] Vassilios Tzerpos and R. C. Holt. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proc. of WCRE 2000*, Brisbane, Australia, November 2000.