Software Architecture Recovery Using Conway's Law

Ivan T. Bowman and Richard C. Holt

University of Waterloo
Waterloo, Ontario, Canada
{itbowman,holt}@plg.uwaterloo.ca

Abstract

Architectural documentation is recognised as a mechanism for improving software quality and reducing development costs. However, many existing systems do not have any architectural documentation. To obtain the benefits of accurate architectural documentation, research suggests that we use tools to recover the architecture of a system, then continue to use these tools to keep the documentation up to date. This paper describes how the organization of system developers can be extracted and analysed to form an ownership architecture. According to Conway's law, the ownership architecture serves as a predictor of the concrete (as built) architecture, and also provides facts about the location of live design knowledge. To evaluate the usefulness of ownership architectures, we examined three large software systems: Linux[1] (800 KLOC), Mozilla (1.5 MLOC), and a commercial software development system (3.8 MLOC). Experience with these systems indicates that ownership architectures can be a powerful addition to a reverse engineering endeavour.

[1] Linux is a registered trademark of Linus Torvalds. Netscape Navigator is a registered trademark of Netscape Communications Corporation. Java is a registered trademark of Sun Microsystems, Inc. UNIX is a registered trademark, licensed exclusively through X/Open Company Ltd. Other names are properties of their respective owners.

1 Introduction

Research [4, 5, 11, 12] suggests that large systems should have a documented system architecture, which describes the overall structure of the system. Accurate documentation can reduce development costs and improve the quality of the software system. However, many existing software systems do not have any architectural documentation. Even when documentation does exist, there is often uncertainty as to whether the documentation matches the system implementation. This uncertainty reduces the effectiveness of the documentation as a definitive source for understanding the system.

Recent research in reverse engineering [6, 8, 10, 13] suggests an approach to reconstruct architectural documentation from a system implementation:

1. Begin with a high level model of the system.
2. Create a mapping between the high level model and the source code structure.
3. Extract relations from the system implementation.
4. Compare the extracted relations to the high-level model.
5. Refine the high-level model based on a visualization of the extracted relations.

Researchers have extracted several relations

from system implementations to recover the concrete architecture. Function calls, variable references, inter-process communications, header-file inclusions, and system build dependencies can be automatically extracted and used to find dependencies between subsystems [6, 7, 8, 10, 13, 14]. The automatically extracted relations may not be perfectly accurate, as Murphy et al. warn [9], but even slightly inaccurate results can be helpful in redocumenting a software system.

Another possible source of relations that might help us to deduce system structure is the organization of the development group that created the system. The relationship between the organization of a design team and the design product has long been recognized. Conway's hypothesis [2], formulated by Brooks [1] as Conway's law, states that the organization of a software system will be congruent to the organization of the group that designed the system. This is perhaps an oversimplification, but the concept is recognized as a valid forward-engineering pattern [3]. For systems developed by a large number of developers, it seems reasonable to consider how the developers of the system were organized into communicating teams. This organization might give insight into the structure of the resulting system, insight that is difficult to derive from low-level facts.

Examining the organization of developers has some attractive qualities. First, it provides the reverse engineer with a list of developers that have worked on particular subsystems. These developers have 'live' knowledge about the system structure that cannot be extracted from any source artifact. Reverse engineers can use interviews with these developers to help understand particular parts of the software system. In addition to live knowledge, there may be documentation that lists developers assigned to projects (sub-teams) that develop or maintain parts of the system. If projects can be related to subsystems, then this would provide a suggested subsystem decomposition. For example, if there is documentation stating that Stephanie worked on the linker, we can consider that the linker is a subsystem. Also, if one developer worked on two subsystems, it might be because there is a communication dependency between the two subsystems. If two subsystems are to communicate, there is likely to be communication between the developers of the two subsystems [2]. In some cases this communication might be accomplished by having a common developer.

The organization of system developers can be visualised using an *ownership architecture*[2] (for example, see Figure 2). This architecture relates developers to the code that they develop. Any two subsystems in the ownership architecture are related if there are developers that have worked on both subsystems. Unlike other relations that researchers have used, there is no obvious direction to the relations caused by shared developers. In some cases, one developer might work on subsystem A only to adapt it so that it can be used by subsystem B. In this case, the concrete architecture would show a directed dependency from B to A, but the ownership architecture would show only that A and B are related in some way. Because the ownership architecture represents such relations as bi-directional edges, we expect that ownership architectures will tend to over-estimate the dependency relations.

We examined three large software systems to evaluate if an ownership architecture can be used as an aid in a reverse engineering endeavour. For each system we present the following architectures:

1. A conceptual architecture based on available system documentation.
2. An ownership architecture extracted from system documentation or revision control logs.
3. A concrete architecture extracted from the actual system implementation.

The rest of this paper is organized as follows: Section 2 describes our results for Linux, Section 3 describes our results for Mozilla, and Section 4 describes our results for a commercial Java development environment. Finally, Section 5 describes our conclusions about using an ownership architecture to help in redocumenting large software systems.

[2]We have coined the term *ownership architecture* because, as can be seen in Figures 1-3, this structure is similar to a conceptual or concrete architecture. Other terms that could be used instead for this structure are *ownership diagram* or *ownership view*.

2 Linux

The first system we studied is Linux, a Unix-like operating system. We analyzed the Linux kernel, which is implemented in about 800 KLOC of C. The core services of the Linux kernel (memory management and process scheduling) are implemented in a single executable image: any function in this image can call any other function. In addition to this core functionality, Linux supports loadable modules (called simply "modules" in Linux documentation). Loadable modules contain code that is optionally included in a particular configuration of the kernel. Hardware device drivers are implemented as loadable modules so that they are only included in the kernel if the hardware device is available. Loadable modules can either be statically linked into the kernel image, or loaded dynamically by a running kernel.

2.1 Conceptual Architecture

To understand the structure of the Linux kernel, we began by defining the conceptual architecture of the system. We used available Linux documentation and our experience with similar operating systems to describe what we expected to find in the system implementation. Figure 1 shows the subsystems we expected to find within Linux and the relations we expected to see at the highest level of abstraction. Each of these top-level subsystems contains sub-subsystems, but this paper only presents results for the top-level subsystems.

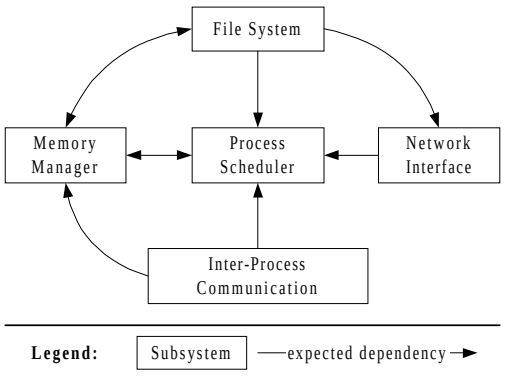

Figure 1: Linux Conceptual Architecture

Using Linux documentation, we derived the dependencies (edges) shown in Figure 1 as follows. We expected all subsystems would depend on the Process Scheduler to block and un-block user processes while waiting for hardware operations to complete, but we expected that the Process Scheduler would depend only on the Memory Manager subsystem. We expected the File System to depend on the Memory Manager for access to memory buffers, and the Memory Manager to depend on the File System for swapping memory out to secondary storage. We expected the file system to depend on the Network Interface to support the network file system (NFS), and we expected the Inter-Process Communication subsystem to depend on the Memory Manager to support shared memory between user processes.

2.2 Ownership Architecture

An important factor in the development of a large system such as Linux is the organization of the development team that implemented the system. Linux was developed by a large number of volunteers. The organization of these developers into useful contributors is one of the remarkable feats accomplished by the Linux effort. The "credits" file included with the Linux distribution gives an overview of which subsystems each of the 198 Linux developers worked on. The credits file was not directly usable because the developer associations were idiosyncratic. For example, one entry is the following: "dosfs, LILO, some fd features, various other hacks here and there." Using the documentation and source file comments we were able to translate these entries into a relation between developers and the source code they worked on.

Figure 2 shows the Linux ownership architecture based on this relation. The subsystems are clustered according to the conceptual architecture, and lines are drawn between subsystems that had developers in common.

We were able to make some predictions about the Linux system by examining Figure 2. First, we noticed that the File System has had 80 developers. Therefore, it seems likely that there is significant structure within the File System. Regardless of other considerations such as maintainability and reusability, the number of developers requires that there

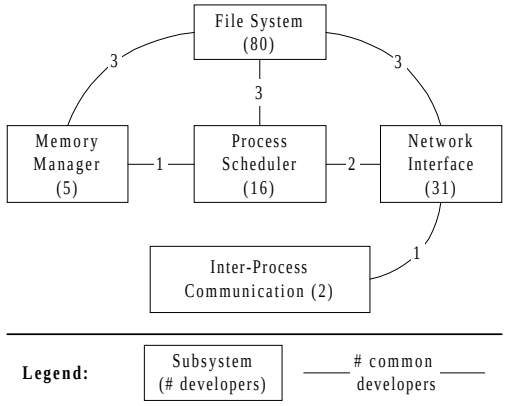

Figure 2: Linux Ownership Architecture

be a well-defined interface that allows developers to operate independently. In contrast to the File System, the Inter-Process Communication subsystem has only two developers listed in the credits file. With this small number of developers, these strict implementation-hiding principles are not required for organizational reasons. After examining the system implementation, we found that the predictions based on the ownership architecture were correct. The File System does have a strictly defined interface with many independent sub-subsystems, but the Inter-Process Communication is implemented without the same amount of implementation hiding.

2.3 Concrete Architecture

To determine what the actual relations are between subsystems, we need to look at the definitive artifact—the system implementation. We extracted a concrete architecture description from the implementation of the Linux Slackware 2.0.27 kernel configured for the Intel 80386 processor. We did this using the Portable Bookshelf [4] to extract two relations from the source code implementation: a function call relation and a variable reference relation. We included uses of data types as references to variables. If one source file `A.c` called a function or referenced a variable in another file `B.c`, we considered that `A.c` was *dependent* on `B.c`. After extracting these dependencies, we clustered source files into subsystems using the source directory structure, file naming conventions, and source code examination. After clustering, we induced dependency relations between subsystems based on the relations between their contained source files. Figure 3 shows the results of this extraction process.

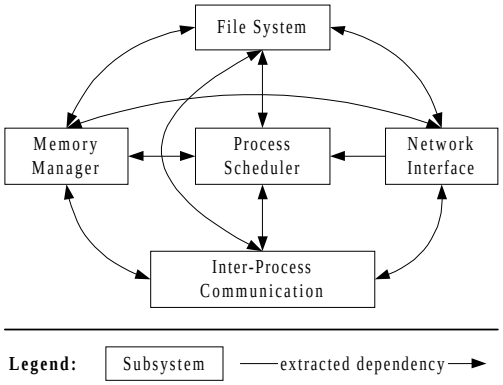

Figure 3: Linux Concrete Architecture

We expected there to be some discrepancy between our conceptualization of the Linux kernel architecture and its implementation; however, we found that there were substantial differences. Although the conceptual architecture had low connectivity (9 out of a possible 20 directed edges), the concrete architecture was almost fully connected, missing only one edge, from the process scheduler to the network interface. From our studies of the differences we found two primary reasons for the discrepancies between the conceptual and concrete architectures:

1. Linux developers did not follow strict implementation hiding principles–because of efficiency, expediency, or possibly because they were unaware of existing interfaces.
2. Our mapping from source code to the conceptual architecture contained some inaccuracies. For example, the extractor we used considered that taking the address of a function was an actual call to the function.

The concrete architecture (Figure 3) is closer to the ownership architecture (Figure 2) than our conceptual architecture (Figure 1). The conceptual architecture did not predict 10 of the dependencies that we found in the concrete architecture. If we assume that common de-

velopers introduce a bi-directional dependency, then the ownership architecture missed predicting only 8 of the dependencies in the concrete architecture, although it incorrectly predicted one dependency which was not found.

Our results with Linux suggest that there can be significant discrepancies between a conceptual architecture based on system documentation and a concrete architecture extracted from the system implementation. In the case of Linux, the ownership architecture was a better predictor of the system structure than the conceptual architecture, and also provided some insight for understanding the system structure.

3 Mozilla

The second system we studied is Mozilla, the web browser previously available as the popular Netscape Navigator. Mozilla is an interesting example for a reverse-engineering case study because it is a large commercial software system that can be examined by any researcher under the Netscape Public Licence (NPL).

Mozilla is implemented in about 1.5 MLOC of `C` and `C++`. The source files are compiled into object files, and these are combined into either libraries (`.lib` files), dynamic-link libraries (`.dll`'s), or executables (`.exe`'s). The dynamic-link libraries can export particular functions, but all other functions within a dynamic-link library are inaccessible outside the library.

3.1 Conceptual Architecture

To understand the structure of Mozilla, we began by forming a conceptual architecture based on available documentation. A "module ownership" document proved useful in determining a meaningful subsystem decomposition. This document lists all of the 44 *modules* (defined as groups of related source files) within the Mozilla system. Each module lists related naming conventions, source directory locations, additional documentation, and associated developers. This document provided a useful starting point; however, with the large number of modules (44), it seems that there should be a higher level grouping of these modules. We clustered the listed modules into subsystems using available documentation. Next, we used available documentation to add relations that we expected to find in the architecture. Figure 4 shows the resulting conceptual architecture for the Mozilla system.

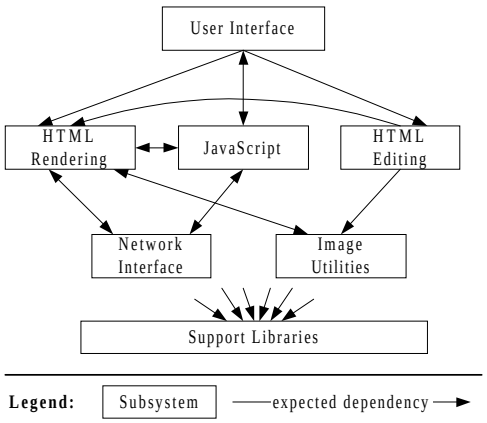

Figure 4: Mozilla Conceptual Architecture

The conceptual architecture is reasonably simple: see Figure 4. There is a User Interface subsystem which depends on an HTML Rendering subsystem and an HTML Editing subsystem. The HTML Rendering subsystem depends on a Network Interface subsystem which controls all access to the network. In addition, both the HTML Rendering subsystem and HTML Editing subsystems depend on an Image Utilities subsystem which provides image rendering capabilities. The HTML Editing subsystem also depends on the HTML Rendering subsystem for HTML parsing capabilities and to provide a preview of the HTML. There is a JavaScript subsystem. Since JavaScript can be used to control the user interface or create HTML dynamically, JavaScript is interdependent with both the User Interface subsystem and the HTML Rendering subsystem. The Network Interface subsystem initiates an HTML viewing session, and thus depends on the HTML Rendering subsystem. Also, the Network Interface may need to resolve URL's that contain JavaScript—this is accomplished through a dependency on the JavaScript subsystem. Finally, there are several support libraries that are used pervasively throughout the Mozilla system. These are grouped into a Support Libraries subsystem; dependency lines

to this subsystem are omitted from Figure 4 for clarity.

3.2 Ownership Architecture

Although the module ownership document does not provide dependency relations directly, it does list developers and the subsystems they have worked on. In contrast with the Linux credits file, it was relatively easy to extract relations between developers and modules because the document was structured by module not by developer; we did not need to interpret idiosyncratic entries. We used this information to develop the ownership architecture of the Mozilla system in Figure 5. In this architecture, we show how many developers have worked on each subsystem, and we also show which subsystems had developers in common.

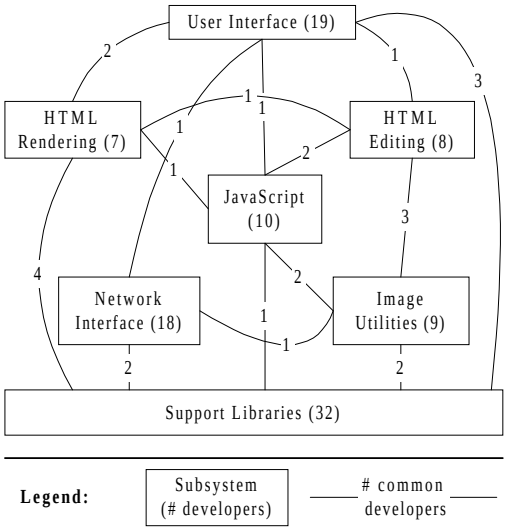

Figure 5: Mozilla Ownership Architecture

Figure 5 provides insight about the structure of the Mozilla system. First of all, only a few developers are associated with more than one subsystem (61 developers are identified with a single subsystem, and only 20 are identified with more than one subsystem). Secondly, there were common developers between subsystems that we did not expect to be related (for example, one developer is identified with both the Network Interface subsystem and the Image Utilities subsystems). This might be a result of developers moving between subsystems, but it also suggests that our conceptual architecture did not capture all of the relations between subsystems.

3.3 Concrete Architecture

For Mozilla, instead of doing an extraction based on source code files, we examined the symbols defined or referenced by object files in the compiled system. This approach identified function calls and variable references between object files. Examining object files did not find all function calls: `C++` virtual function calls and calls through function pointers were not extracted. In addition, we did not extract any relations based on data type usage. We did not extract all possible relations, but the extractor ran quite quickly. It took 48 minutes to compile the source code for Mozilla, but only 82 seconds to extract the relations between the compiled object files.

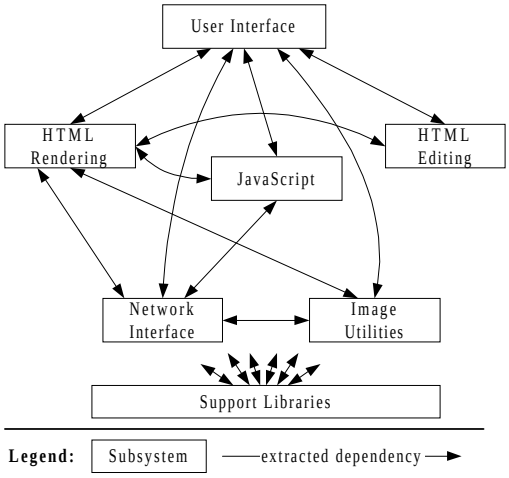

Figure 6: Mozilla Concrete Architecture

After clustering object files into subsystems based on the conceptual architecture and the directory structure described in the documentation, we formed relations between the subsystems based on relations between their contained object files. Figure 6 shows the results of visualising these subsystem relations. The results of the visualisation are similar to the results for Linux: there is a substantial difference between the conceptual architecture and the concrete architecture. The conceptual architec-

ture has only 20 out of a possible 42 directed edges. The concrete architecture has 34 of these edges. One of the connections that we expected in the conceptual architecture (HTML Editing depends on Image Utilities) was not found in the concrete architecture.

The conceptual architecture predicted one edge which was not found, and did not predict 15 edges which were found. In contrast, the ownership architecture predicted 6 edges which were not found, and did not predict 10 that were found.

The ownership architecture of Mozilla was better than the conceptual architecture at predicting possible relations, but both were substantially different from the concrete architecture.

4 Aleph

The last system that we studied is a commercial Java development environment. This environment, which we will call Aleph, is a fairly large system implemented in 3.8 MLOC of `C` and `C++`. It includes a Java compiler, a Java debugger, and a user interface that integrates these features. The user interface also supports features to reduce the effort required for common development tasks. Aleph is based on a previous product for `C++` development.

4.1 Conceptual Architecture

As with the other systems, we began by forming a conceptual architecture for Aleph. In the case of Aleph, we had access not only to system documentation, but also to the developers responsible for designing and implementing the system. Figure 7 shows the conceptual architecture of the Aleph system based on system documentation and informal interviews with six of the system developers.

The user-interface (UI) of Aleph integrates several subsystems into one integrated development environment that users interact with. First, the UI controls a Java Debugger which allows users to interactively debug their applications. In addition to the debugger, the UI allows users to graphically design user-interface elements in their applications. This is accomplished by the Design-Time Components subsystem. The source code for the user's application is edited with a Code Editor subsystem, and compiled with a Java Compiler. The Build Engine subsystem determines which files need to be compiled and invokes the Java compiler as needed. All of the information for a user's application is stored by the Project Persistence subsystem. There is a `C++` Compiler subsystem which was developed for the previous `C++` product, but this was not expected to be used by the rest of the Aleph system since Aleph only targets Java.

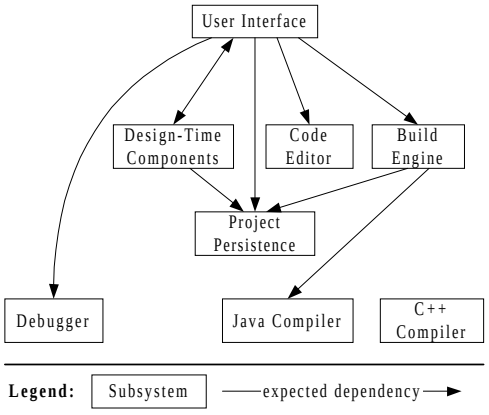

Figure 7: Aleph Conceptual Architecture

4.2 Ownership Architecture

The concept of code ownership is well established among the developers of the Aleph system. Aleph is composed of 37 *projects*. Each project is compiled into either a library, a dynamic-link library, or an executable. Projects are stored separately in a revision-control system, and each project can be built and tested independently of other projects. Projects are often *owned* by a single developer. Other developers might make changes to a project, but only after discussing the changes with the project owner. Some large projects have several active developers. For these projects, each active developer owns a group of files within the project.

Project and source file ownership is not documented in any formal fashion, but is part of the culture of the development group. Since

there is no formal documentation, it was not possible to extract the ownership relation automatically. Instead, we examined the source code revision logs. These logs list all modifications to source files. Because the revision control system is structured by projects, we were able to automatically extract developer ownership of projects based on modifications to files contained within the projects. After extracting the ownership, we grouped projects according to the subsystem decomposition of the conceptual architecture. Figure 8 shows the results of the extraction at the highest level of abstraction.

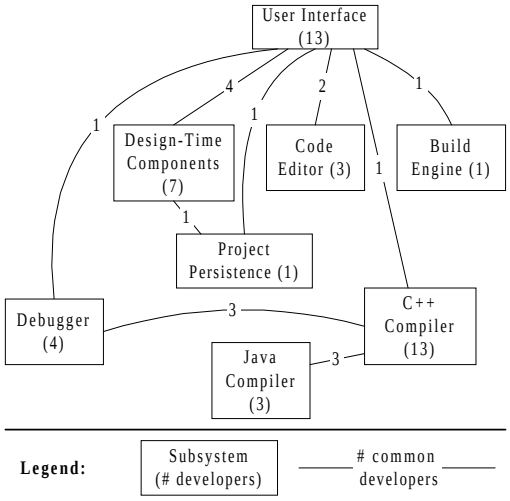

Figure 8: Aleph Ownership Architecture

Figure 8 provides some insight about the Aleph system. First, we notice that although Aleph is larger than the other systems, it has far fewer developers than Mozilla or Linux: Aleph has only 30 developers, whereas Linux has 198 and Mozilla 81. According to Brooks [1], this indicates that Aleph will have better conceptual integrity compared to the other subsystems. Also, the number of developers within particular subsystems can be interpreted in the following way. The UI subsystem has 13 developers whereas the Project Persistence subsystem has only one developer. This suggests that the UI subsystem has significant internal structure (it contains several subsystems), and that the Project Persistence subsystem has a simple internal structure (it contains no subsystems).

4.3 Concrete Architecture

Since the project ownership concept is well ingrained in Aleph developers, it may not be surprising that there is similarity between the conceptual architecture described by Aleph developers and the ownership architecture. To determine if these diagrams correspond to the concrete system structure, we need to examine the system implementation.

We used a source-code extractor for Linux and an object-file extractor for Mozilla. For Aleph, we did two extractions. First, we automatically extracted static linkage relations from map files produced by the linker. These map files list all of the libraries and dynamic-link libraries that are used to build each project. In addition to these static linkages, Aleph also loads dynamic-link libraries at runtime by file name. We manually inspected source code, performing any necessary data-flow analysis, to extract these dynamic linking relationships. In combination, the static and dynamic linking relationships show the linkages between projects in Aleph. These linkages include function calls and variable access. In this case, we extracted virtual function calls, but not data type usage. Figure 9 shows the inter-subsystem relations that we determined based on inter-project linkages.

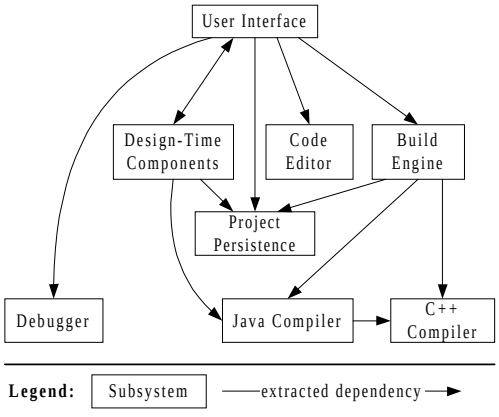

Figure 9: Aleph Concrete Architecture

In contrast to Linux and Mozilla, Aleph's concrete architecture is very similar to its conceptual architecture (see Figures 7 and 9). There are the following differences: the Design-

Time Components subsystem depends on the Java Compiler, the Java Compiler depends on the C++ Compiler, and the Build Engine depends on the C++ Compiler. After investigating, we found explanations for these unexpected dependencies.

The Design-Time Components subsystem has a capability to add new Java components to the development environment. This capability uses utilities from the Java Compiler to generate a C++ adaptor. The Build Engine then invokes the C++ Compiler to compile this C++ adaptor. The Java Compiler uses one library which was developed in the C++ Compiler, and thus there was an unexpected dependency between the Java Compiler and the C++ Compiler. The developer organization showed that some of the Java Compiler developers had also worked on the C++ Compiler. Perhaps it is not surprising that they decided to re-use some of the utilities that they had developed earlier.

In the case of Aleph, the conceptual architecture was closer to the concrete architecture than the ownership architecture was. The ownership architecture missed predicting 4 edges which we found in the concrete architecture, and predicted 10 that we did not find in the concrete architecture. The conceptual architecture missed only 3 edges, and did not predict any that we did not find in the concrete architecture.

If we had not had a good conceptual architecture for Aleph, then the ownership architecture would have been quite useful in predicting the concrete architecture.

5 Conclusions

This paper has introduced the idea of an *ownership architecture* for a software system, and has shown how such a structure is useful in reverse engineering. Based on case studies of three large software systems—Linux, Mozilla, and Aleph—we have shown that the ownership architecture is a good predictor for the concrete architecture and is closely correlated with the conceptual architecture.

See Table 1 for the sizes of these three systems and the interconnectivity counts for their top-level conceptual, ownership, and concrete architectures.

	Linux	Mozilla	Aleph
MLOC	0.8	1.5	3.8
# Developers	198	81	30
# Subsystems	5	7	8
Possible Edges	20	42	56
Concrete Edges	19	34	12
Conceptual Edges	9	20	9
Ownership Edges	12	30	18

Table 1: Summary of Systems

If Conway's conjecture [2] is correct, then the ownership architecture will be a good predictor of the extracted concrete architecture. Table 2 examines the effectiveness of the conceptual and ownership architectures as predictors of the concrete architecture by showing the following:

1. predicted edges (E)—the number of edges that were predicted
2. correct edges (K)—concrete edges that were correctly predicted
3. false negatives (M)—edges that were not predicted but were found in the concrete architecture
4. false positives (V)—edges that were predicted but not in the concrete architecture

	Linux	Mozilla	Aleph
Conceptual			
Edges (E)	9	20	9
Correct (K)	9	19	9
False Negative (M)	10	15	3
False Positive (V)	0	1	0
Ownership			
Edges (E)	12	30	18
Correct (K)	11	24	9
False Negative (M)	8	9	2
False Positive (V)	1	6	9

Table 2: Conceptual and Ownership Architectures as Predictors of Concrete Architecture

We found that the ownership architecture is at least as good as the conceptual architecture in correctly predicting edges in the concrete architecture: the ownership predicted more correct edges (K) for both Linux and Mozilla,

and the same number for Aleph. However, the ownership architectures consistently overestimated the relations that we found in the extracted architecture. These overestimates seem to be caused partly by developer movement that is not related to code-reuse (such as developers changing projects). This 'noise' could be reduced somewhat by only including relations where a developer worked on two subsystems within a given time frame. Another cause of overestimation is our assumption that shared developers introduce a bi-directional dependency. This overestimation seems difficult to resolve because the ownership architecture does not predict the direction of the dependency introduced.

Although the ownership architecture tends to overestimate relations, it proves to be a useful mechanism for predicting system structure. In the systems we studied, it was easier to extract ownership architectures than concrete architectures. It took us less then 12 hours to extract the ownership architectures for the three systems, but it took several weeks to completely extract the concrete architectures of these large systems (of which only the highest level of abstraction is shown in this paper). Because it is easier to extract the ownership architecture and it seems to provide reasonable predictive powers, developers trying to understand large systems might choose to begin by forming an ownership architecture. This architecture might be based on revision control logs, source code comments, documentation, or interviews with experienced developers. Because of the tendency to overestimate, ownership architectures can not be used as the only tool for system understanding, but they provide a useful starting point for program understanding. Ownership architectures seem to be useful in conjunction with a conceptual architecture formed from system documentation (which we found tended to underestimate relations in the concrete architecture).

Our results can be interpreted as an empirical validation of Conway's 1968 conjecture that the structure of a software system is a direct reflection of the structure of the development team. We note that an ownership architecture provides a further benefit; namely, it shows which developers have "live" knowledge about the system in their heads that can be extracted with appropriate interviews.

Acknowledgements

We would like to thank Saheem Siddiqi and Meyer Tanuan for their contributions toward extracting the Linux architecture. Gary Farmaner assisted us with the source-code extraction tools. The developers of the Aleph system assisted by discussing the conceptual and concrete system architectures, explaining the reasons for some of the discrepancies. Susan Sim provided helpful comments on an earlier draft of this paper. We would also like to thank the anonymous reviewers for their constructive comments and suggestions.

About the Authors

Ivan T. Bowman is an MMath student in the Department of Computer Science at the University of Waterloo. Bowman received his BMath degree from the Computer Science Department of the University of Waterloo in 1995. His research interests include software architecture, reverse engineering, and program visualization. (See WWW `http://plg.uwaterloo.ca/~itbowman`).

Richard C. Holt is a professor at the University of Waterloo in the Department of Computer Science. Holt received his Ph.D. degree in Computer Science from Cornell University in 1971. His research has included work in operating systems, compiler development, and software construction methods. He is an author of the Turing programming language. His recent work has concentrated on Software Landscapes, which provide a visual formalism for software architectures. (See WWW `http://plg.uwaterloo.ca/~holt`).

References

[1] F. P. Brooks, Jr. *The Mythical Man-Month*. Addison Wesley, anniversary ed. edition, 1995.

[2] M. E. Conway. How do comittees invent? *Datamation*, 14(4):28–31, 1968.

[3] J. O. Coplien. A development process generative pattern language. In *Proceedings of PLoP 1994*, Monticello, Illinois, Aug. 1994.

[4] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Oct. 1997.

[5] D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[6] R. Kazman and J. Carrière. Playing detective: Reconstructing software architecture from available evidence. Technical Report CMU/SEI-97-TR-010, Carnegie Mellon University, 1997.

[7] R. Kazman and J. Carrière. View extraction and view fusion in architectural understanding. In *5th International Conference on Software Reuse*, Victoria, BC, Canada, June 1998.

[8] S. Mancoridis and R. C. Holt. Recovering the structure of software systems using tube graph interconnection clustering. In *IEEE Proceedings of the 1996 International Conference on Software Maintenance*, Monterey, California, USA, Nov. 1996.

[9] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *18th International Conference on Software Engineering*, Berlin,Germany, Mar. 1996.

[10] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, Washington, DC, Oct. 1995. IEEE Computer Society Press.

[11] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[12] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, and D. M. Young. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, Apr. 1995.

[13] V. Tzerpos and R. Holt. A hybrid process for recovering software architecture. In *Proceedings of CASCON 1996*, Toronto, Canada, Nov. 1996.

[14] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 11(6):501–520, Jan. 1995.