Forward and Reverse Repair of Software Architecture

John B. Tran and Richard C. Holt
Dept. of Computer Science
University of Waterloo
Waterloo, ON, Canada
{j3tran, holt}@plg.uwaterloo.ca

Abstract

As a software system evolves, it is common for the *as-built* architecture to diverge from the *as-designed* architecture. This gap between the *as-designed* (conceptual) and the *as-built* (concrete) architecture leads to a false understanding of the system, resulting in error prone maintenance decisions. We present an approach to repair an architecture of a software system. Our approach attempts to reconcile the conceptual architecture with the concrete architecture by performing a series of simple, semi-automatic repair actions. We applied our architecture repair actions to the Linux kernel and were able to repair many of the anomalies its architecture.

1 Introduction

Software systems must be continually maintained else they risk becoming obsolete [9]. As they evolve, so too do their architectures. New modules and dependencies are added to support new features, while obsolete functionalities are removed. Consequently, the *as-built* (concrete) architecture of the system will gradually diverge from its original *as-designed* (conceptual) architecture. Different architectural views [8] become inconsistent with the current implementation, making maintenance tasks difficult and error prone. In particular, the programmers' intuition about the structure of the system will no longer reflect its concrete architecture. Unless actions are taken to narrow the gap between these architectural views, system understanding will be deteriorate and maintenance will become increasingly risky.

Current research in software reverse engineering has focused on recapturing the architectural structure of large, old software systems [11, 13, 12]. Software architecture extraction tools, such as Rigi and PBS [1], extract the structure of a system from its source code. The extracted architecture, as a form of documentation, serve to help programmers understand their system [4]. For the extracted architecture to be an effective form of documentation, it must be repeatedly updated as the system evolves.

The idea of software evolution has been around since 1974 when Lehman first introduced his laws of program evolution [9]. Software evolution is seen as both a good and a bad phenomenon [9, 2]. At one extreme, a system that does not evolve becomes less useful [9]. At the other end, system evolution leads to architectural drift and increase system complexity, which reduces system maintainability and adaptability [13, 9].

We present an approach to repair the architecture of an evolving software system. We claim that our proposed repair actions are simple and safe. By simple, we mean that the repair actions are independent of the implementation details of the system (i.e., data structures and algorithms). All that is required from the maintainer is a working knowledge of the application domain and the programming language in which the system was implemented. The repair actions are also considered safe since they do not modify implementation details, and hence, they will not alter system behaviour.

We use the Linux kernel release 1.2.0 to demonstrate the effectiveness of our repair model. The kernel release is a good testbed

since it is a large system (173 KLOC) with a well understood high-level conceptual architecture [3]. Also, the gap between its conceptual and concrete architecture is sufficiently large to demonstrate any success in our proposed approach to architecture repair.

The rest of the paper is organized as follows. The next section gives a description of how we model and extract architecture. Section 3 presents two methods, forward and reverse repair, to reconcile the conceptual architecture and the concrete architecture. Section 4 describes forward architecture repair which attempts to reconcile the two architectural views by repairing the concrete architecture. Section 5 describes how reverse architecture repair attempts to reconcile the conceptual architecture to match the concrete architecture. Section 6 presents our results from repairing the Linux kernel. Section 7 summarizes our results.

2 Modelling Architecture

Murphy *et al.* suggested that we build a high-level model of the structure of the system then map the source code model so that it reflects our high-level model [12]. We call this high-level model of the system structure the *conceptual* architecture. This differs from the *as-designed* architecture because it does not contain system information, such as requirements and design specifications. In this paper, we refer to the system's structure as its architecture.

When creating a conceptual architecture for a software system, it is common to decompose the system into subsystems. Subsystems can be further decomposed into smaller subsystems, hence creating a subsystem hierarchy. At the bottom of the hierarchy are modules (source code files). Modules contain program entities which we define to be data units (e.g., variables) or computational units (e.g., procedure) that are visible in the file-level or class-level scope.

2.1 Architecture Schema

We represent software architecture using box and arrow diagrams. Boxes (*components*) represent subsystems and modules, and arrows (*connectors*) correspond to dependencies between system components. Our architecture diagrams do not show program entities and the relationships between them. Relationships among program entities are *lifted* [6] to the module level. For example, if two program entities contained in separate modules depend on one another, then their parent modules will also depend on each other. Omitting program entities from the architecture diagram simplifies the diagram by reducing the number of boxes and potential edges. Consequently, the system structure modelled by the architecture diagram becomes clearer to a developer.

The schema for our architecture diagrams is given in Figure 1. There are two types of relations, a dependency relation (solid edge) and a containment relation (dashed edge). The schema permits dependencies between subsystems and modules. It permits subsystems to contain modules as well as other subsystems. However, modules cannot contain other components.

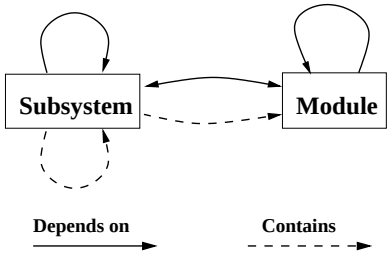


Figure 1: Architecture schema.

2.2 Extracting Architecture

We create the conceptual architecture of system using existing system documentation. If the documentation is insufficient to describe the whole system, we may combine it with descriptions of other related systems, plus our own intuition about the application domain.

We build the concrete architecture by mapping the source model onto the conceptual architecture. We use the PBS tools to extract the source model from system implementation. We begin by extracting system artifacts from the source code using a source code extractor called `cfx` [1]. The output of `cfx` (a set of relations between functions and variables) is too

detailed for our architectural model. We use `grok` [1, 5], a relational calculator, to abstract file-level relations from the output of `cfx`. This provides us with the source model needed to construct the concrete architecture.

3 Forward and Reverse Repair

As software systems evolve, their concrete architecture will gradually *drift* from their conceptual architecture. Maintenance tasks add dependencies where none exists in the conceptual architecture, or remove dependencies from the concrete architecture that are present in the conceptual architecture. Unless preventive maintenance [10] is regularly performed, the gap between the conceptual and the concrete architecture will continue to widen.

The gap between the conceptual architecture and the concrete architecture is characterized by *anomalies* between the two architectural views. Anomalies are architectural elements (i.e., modules, subsystems, or dependencies) that are present in one view, but not the other. Architecture elements that are in the concrete architecture, but not in the conceptual architecture are said to be *unexpected*. Conversely, architecture elements that appear in the conceptual architecture and not in the concrete architecture are said to be *gratuitous*[1].

To reconcile the conceptual architecture and the concrete architecture, we minimize their anomalies. This can be achieved by minimizing gratuitous elements from the conceptual architecture and unexpected elements from the concrete architecture. We call the former *reverse* architecture repair, and the latter, *forward* architecture repair. More precisely,

- **Forward architecture repair** means repairing the concrete architecture to match the conceptual architecture.

- **Reverse architecture repair** means repairing the conceptual architecture to match the concrete architecture.

[1] Murphy *et al.* [12] use the terms *absent* and *divergent* instead of *unexpected* and *gratuitous*.

4 Forward Architecture Repair

The goal of forward architecture repair is to make the concrete architecture match the conceptual architecture by removing unexpected dependencies from the concrete architecture. It is accomplished by moving architectural components (i.e., subsystems and modules) between subsystems or by moving program entities between modules. Forward architecture repair consists of three phases:

1. identify the program entity (i.e., variable, function, or module) generating an unexpected dependency,

2. perform architecture impact analysis [7] to determine all modules and subsystems that would be affected if the program entity was moved to a new subsystem, and

3. determine whether repairing the unexpected dependency will be a simple and low risk task. If the repair can be done easily, without affecting system functionality, then a repair action (described below) is applied to remove the unexpected dependency from the concrete architecture.

Phase 1 is done automatically using a `grok`. `grok` reads in a set of relations describing the structure of the system. The relations are of the form, 'file $f1$ depends on file $f2$' and 'subsystem S contains file $f1$'. Using binary relational algebra, `grok` is able to query the architecture of the system about its structure. We write `grok` scripts to isolate each program entity that causes an unexpected dependency. In phase 2, we, again, use `grok` to determine the impact of moving the program entity (identified in phase 1) to a new subsystem. The relevant questions here are, *"What modules depends on this program entity"* and *"What modules does this program entity depend on?"*. We want to be certain that any repair actions taken will not have any adverse affect on the system and will not add new unexpected dependencies to the concrete architecture. In phase three, we determine whether applying the repair action to the program entity will violate the *semantics* of the architecture. Architecture semantics is

the architects' intuition about the composition of a subsystem, and which features should or should not be contained in it. For example, a graphical user interface subsystem should not contain modules responsible for file I/O. If the repair action is deemed safe (i.e., no adverse affect on the system, no addition of new *unexpected* dependencies, and no semantic violations to the architecture), then a repair action is performed on the program entity.

We will now present three repair actions, namely, kidnapping an architectural component, splitting a component, and relocating a program entity (see Figure 2).

4.1 Kidnapping

The simplest and safest repair action is to *kidnap* [16] an architectural component (i.e., to move the component from its original subsystem to a new one). The kidnapping action is simple and safe because it does not modify any source code. Figure 2a shows component **B** being kidnapped from subsystem **S1** by subsystem **S2**. The dashed edge in the figure represents an unexpected dependency and the solid edge corresponds to a permissible dependency.

Kidnapping is recommended if all program entities within the module (in the case of a subsystem, all modules in the subsystem) need to be moved to the destination subsystem to minimize unexpected edges.

4.2 Splitting

Another repair action is to *split* the architectural component in two and have a subsystem adopt one of the new component. Figure 2b illustrates the splitting action. Component **B** is split into two new sub-components, **B1** and **B2**, where **B2** contains the program entities generating the unexpected dependency (shown by the dashed edge). Component **B2** is then *adopted* into subsystem **S2** to remove the unexpected dependency from **S2** to **S1**.

Splitting a module involves modifying source code. This action is not as safe as splitting a subsystem where it is the modules that are being moved to the new subsystem. Care must be taken when splitting a module to avoid any adverse affect on the functionality of the system.

In general, we perform splitting when kidnapping fails to remove unexpected dependencies in the concrete architecture, but moving a subset of the component will do the job. With the splitting repair action, we are identifying and moving only those program entities that are causing the unexpected dependency.

4.3 Relocating

It is common in large software systems to have a module contain program entities that semantically belong in another module. This problem can arise when the owner of the module does not understand the whole system. When a new service is required, the owner simply implements the service within his or her module.

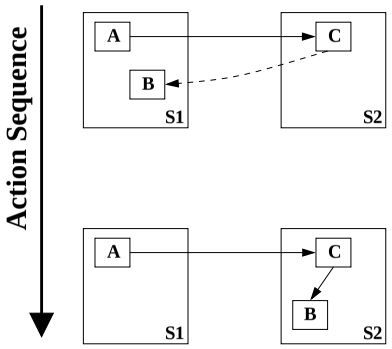
To correct this problem, we *relocate* the program entity to the appropriate module. This repair action is depicted in Figure 2c. Module **C** depends on one or more program entities in module **B**. We remove the program entities from **B** and add them to **C**. As a result, the *unexpected* dependency from **C** to **B** disappears.

Relocating a program entity only applies to modules. Since the action modifies source code, it must be done with care to preserve system functionality. The primary concern is that the relocated program entity must be *visible* during compilation or linking after the repair action. For example, suppose that variable v is declared in module **B**. Suppose that another module, **D**, uses v via a source inclusion of **B**. If v is moved to **C**, then **C** must be included in **D** to avoid errors during compilation.

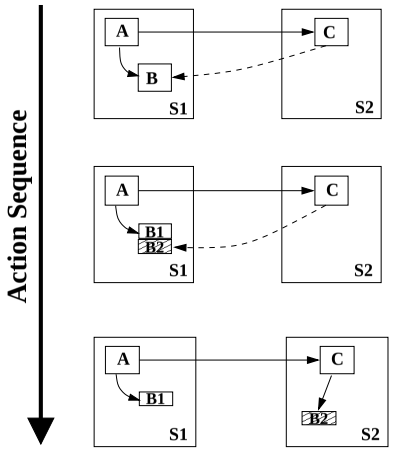
We have introduced forward architecture repair and now we will discuss reverse architecture repair.

5 Reverse Architecture Repair

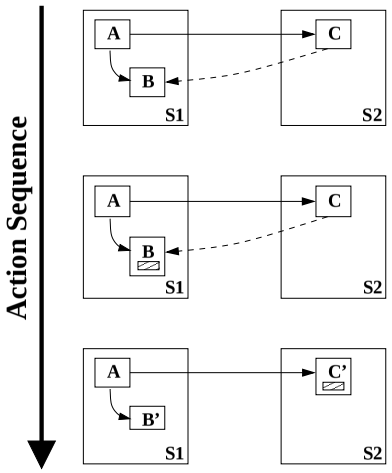
The goal of reverse architecture repair is to make the conceptual architecture match the concrete architecture. A conceptual architecture is highly flexible with respect to changes. Modifying it does not alter source code, and as a result, the repairer is less restricted in his or



(a) Kidnapping



(b) Spitting



(c) Relocating

Figure 2: Forward architecture repair actions.

her repair actions to remove gratuity. Repair actions that are available include kidnapping, splitting, and merging subsystems (similar to Figure 2). Any changes to the conceptual architecture, however, should be consistent with the our schema.

Our repair of the Linux architecture (presented in Section 6) did not include any reverse architecture repair action. The reason behind this is that the Linux conceptual architecture, created by Bowman *et al.* [3], had all the gratuities removed from it.

In our architecture diagrams, the system structure is represented by a subsystem hierarchy. Dependencies that exists between nested modules result in dependencies between their ancestors. We say that the dependencies between the ancestors are *induced* from the dependencies between their descendants. For example, in Figure 3, the dependency from **A** to **B** induces the dependency between their parents, **S1** and **S2**. When removing a *gratuitous* element from the conceptual architecture, care must be taken to remove all dependencies that are induced up the system decomposition hierarchy. Referring back to Figure 3, if we want to remove the dependency from **A** to **B**, the induced dependency from **S1** to **S2** must be removed as well.

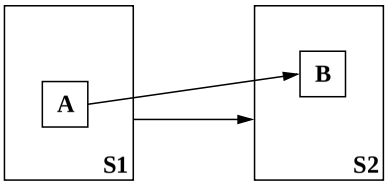


Figure 3: Induced dependency.

It should be noted that reverse architecture repair may require the architecture schema to be altered. This occurs when a software system is reengineered to utilize a new programming language or a new programming paradigm.

6 Repairing the Linux Kernel Architecture

The Linux kernel is a Unix-like operating system originally based on the Minix micro-

kernel [15]. It was initially released on the Internet in 1991 and since then, has grown from 8 KLOC[2] in release 0.01 to over 1.2 MLOC in its latest major release (2.2.0).

We have taken an earlier release, release 1.2.0 of the Linux kernel, and attempted to reconcile its conceptual and concrete architecture. The high-level conceptual architecture, shown in Figure 4a, was described by Bowman *et al.* [3]. It should be noted that Bowman *et al.* did their case study using release 2.0.27a of the Linux kernel. We were able to use their conceptual architecture for release 1.2.0 because, at this level of abstraction, the conceptual architecture did not change between the two kernel releases. Figure 4b shows the concrete architecture of the built system for kernel release 1.2.0. Edges show dependencies and edges with empty arrow heads depict unexpected dependencies. The actual count of the number of module-to-module dependencies causing the edge is shown next to the arrow head. In our architecture diagrams, arrow head size corresponds to the number of module-to-module dependencies causing an edge. A large arrow head means a large number of module-to-module dependencies, while a small arrow head indicates a small number of module-to-module dependencies.

The dashed box is added to simplify the diagram. Rather than adding edges from the Initialization subsystem to the the major subsystems (File System, Memory Manager, Process Scheduler, Inter-Process Communication, and Network Interface), we use the notation of a double-headed arrow to the dashed box to show that the Initialization subsystem depends on all subsystems within the box. Likewise for the Library subsystem, except that the double-headed arrow is coming out of the box, which means that all subsystems within the dashed box depends on Library.

The post-repair concrete architecture is shown in Figure 4c. It can be seen that the *post-repair* architecture has significantly fewer unexpected edges than the original concrete architecture. The rest of the section will describe the repair actions taken to obtain Figure 4c. For the purpose of brevity, we do not describe all our repair work on the architecture of the Linux kernel. Instead, we give an account of

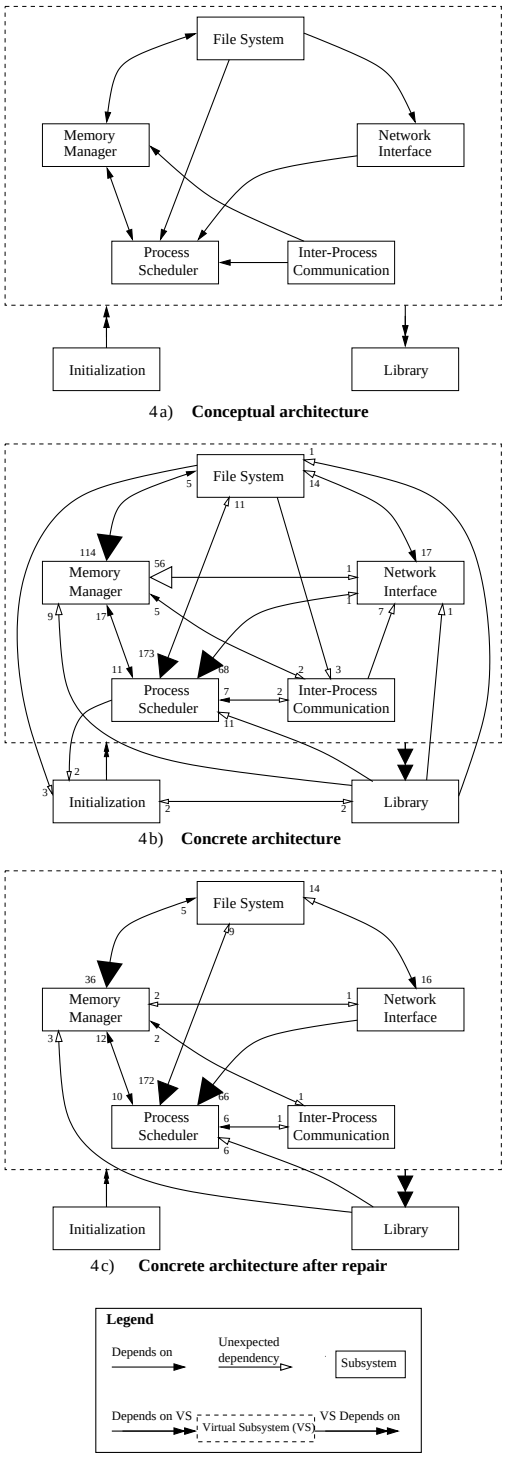


Figure 4: Architecture of the Linux kernel.

[2] Number of lines of code not counting comments and empty lines.

our repair actions on the Memory Manager subsystem, the Inter-Process Communication subsystem, and the Network Interface subsystem.

6.1 Repairing the Memory Manager Subsystem

Figure 4b shows that the Memory Management subsystem is used by all other subsystems. This pattern suggests that there may be one or more omnipresent [11] modules in the subsystem. An *omnipresent* module is one that is used by many other modules or vice versa. An example of an omnipresent module would be one that prints debugging information. Upon further inspection, we discovered that four modules contained in the Memory Manager subsystem were omnipresent. One module in particular, `kmalloc.c`, which defines procedures and variables for memory allocation, was used by many of the modules in the kernel code. Since any routine that performs a memory reference will depend on `kmalloc.c`, it was more appropriate for it to be contained in the Library subsystem. We analyzed the impact on the architecture of having Library kidnap `kmalloc.c` from Memory Manager. After determining that this repair action would not generate new unexpected dependencies in the architecture, the decision was made to have Library kidnap `kmalloc.c`.

Along with the kidnapping action, we performed several splitting actions, the reason being that the program entities defined in `kmalloc.c` were declared in other modules within Memory Manager. We split away those declarations and moved them into a new module in the Library subsystem. As a result, the cardinality of the dependency from Network Interface to Memory Manager dropped from 52 dependencies down to 2. This repair action also reduced the number of dependencies from the Library subsystem to Memory Manager from 9 down to 3.

We were unable to completely eliminate the unexpected dependencies from Network Interface and Library. There were a small set of procedures and variables in the Memory Management subsystem that were used by the Network Interface subsystem and by the Library subsystem that were tightly coupled with other modules. Removing those dependencies requires modifications to many other modules, and in some cases, would require re-implementing several functions. These repair actions are deemed too risky or too difficult so they were not done.

6.2 Repairing the Inter-Process Communication Subsystem

We performed one repair action for the Inter-Process Communication subsystem. We kidnapped the subsystem LOADABLE-MODULE into the Memory Manager subsystem. This action was justified after reading the comments and source code of the modules LOADABLE-MODULE and noting that these modules contain memory management routines for Linux loadable modules[3]. As a result of this kidnapping action, the unexpected dependencies from the File System subsystem to the Inter-Process Communication subsystem were removed from the concrete architecture.

6.3 Repairing the Network Interface Subsystem

We began by trying to remove unexpected dependencies from the Inter-Process Communication subsystem (IPC) to Network Interface subsystem. We discovered that all seven violations ended up at the module `ipc.h` contained in the Network Interface subsystem. Furthermore, this module was used only by modules in IPC and did not depend on any other modules other than those in the Library subsystem. This was a strong indication that the module may have placed in the wrong subsystem. It also signified that a kidnapping action could be done safely without adding extra unexpected dependencies. After viewing the source code of `ipc.h`, we concluded that Bowman *et al.* had mistakenly placed the module in the Network Interface subsystem. After having the Inter-Process Communication subsystem kidnap `ipc.h` from the Network Interface subsystem, all seven dependencies from IPC to Network Interface were eliminated (see Figure 4b and Figure 4c).

[3]Linux loadable modules are *plug-and-play* components, such as sound support or printer support

Next, we analyzed the dependencies from the Process Scheduler subsystem to the Network Interface subsystem. We observed that the dependencies were caused by exactly one module in the Process Scheduler. Using `grok`, we determined that a kidnapping action, done on this module, by Network Interface would not result in new unexpected dependencies. Furthermore, the kidnapping action seemed reasonable since the name of the module, `inet.h`, suggested that it is involved with the Network Interface. Upon further inspection, we discovered that `inet.h` is used only be modules within the Network Interface subsystem. This also suggested that it belongs in the Network Interface subsystem. The result of having Network Interface kidnap `inet.h` from Process Scheduler was the removal of the unexpected dependency from Process Scheduler to Network Interface.

7 Conclusion

We have introduced forward and reverse software architecture repair. In forward architecture repair, we modify the concrete architecture to make it match the conceptual architecture. In reverse architecture repair, we modify the conceptual architecture to match the concrete architecture. Here, the repairer is less constrained in his or her repair actions, but the end results are similar (i.e., gratuitous elements are removed from the conceptual architecture).

We demonstrated our approach to architecture repair using the architecture of the Linux kernel (release 1.2.0). The results of our repair work are summarized in Table 1. We performed 19 kidnappings, 3 splittings, and 8 relocating actions (shown in Table 2). From our repair actions, we were able to reduce the number of unexpected edges in Figure 4b by 63%, from 16 to 6 edges. This also correspond to a reduction of module-to-module dependencies from 156 to 56.

The forward repair actions proposed in this paper were not able to repair all anomalies in the concrete architecture of the Linux kernel. Subsystems that are tightly coupled with other subsystems could not be repaired using our proposed actions. This suggests that costlier, riskier maintenance procedures be used to repair such anomalies.

	Before	**After**
Unexpected dependencies (subsystem level)	16	6
Unexpected dependencies (module level)	156	58

Table 1: Result from repairing Linux kernel version 1.2.0

Action Type	**Count**
Kidnapping	19
Splitting	3
Relocating	8

Table 2: Frequency of repair actions.

We believe that our repair actions, which have removed many anomalies from the Linux architecture, are useful to the Linux community. The Linux kernel is continually growing and understanding the system as a whole is becoming extremely difficult. Repairing the concrete architecture to conform to a conceptual model will reduce the learning curve for new developers. Furthermore, we believe that the simplicity of our repair actions should appeal to the Linux community where it is said that, *"the best written code is a working code"* [14].

References

[1] Portable Bookshelf (PBS) tools. Available at http://www.turing.cs.toronto.edu/pbs.

[2] K. Bennett. Software Evolution: Past, Present and Future. *Information and Software Technology*, 38:673–680, 1996.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. *Proceedings in the 21st International Conference on Software Engineering, Los Angeles*, May 1999.

[4] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontongiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf.

IBM Systems Journal, 36(4):564–593, October 1997.

[5] R. C. Holt. Binary Relational Algebra Applied to Software Architecture. CSRI Tech Report 345, University of Toronto, March 1996.

[6] R. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.

[7] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A Two-phase Process for Software Architecture Improvement. Available at http://adam.wins.uva.nl/-x/sai/sai.html.

[8] P. B. Kruchten. 4+1 View Model of Architecture. *IEEE Software*, pages 42–50, November 1995.

[9] M. M. Lehman. Programs, Cities, Students, Limits to Growth? *Imperial College of Science and Technology Inaugural Lecture Series*, 9:211–229, 1974. Also in *Programming Methodology*. D Gries ed., Springer, Verlag (1978). 42–62.

[10] B. Lientz, E.B. Swanson, and G.E. Tompkins. Characteristics of Applications Software Maintenance. *Communications in the ACM*, 21:466–471, 1978.

[11] H. A. Müller, O. A. Mehment, S. R. Tilley, and J. S. Uhl. A Reverse Engineering Approach to Subsystem Identification. *Software Maintenance and Practice*, 5:181–204, 1993.

[12] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. *Proceedings of the Third ACM Symposium on the Foundations of Software Engineering*, October 1995.

[13] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[14] E.S. Raymond. The Cathedral and the Bazaar. Available at Available at http://www.tuxedo.org/-esr/writings/cathedral-bazaar/.

[15] A.S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall, 2/e edition, 1997.

[16] V. Tzerpos and R. C. Holt. The Orphan Adoption Problem in Architecture Maintenance. *Proceedings of the Working Conference on Reverse Engineering 1997, Amsterdam*, October 1997.