Architecture Analysis and Repair of Open Source Software

John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt
Dept. of Computer Science
University of Waterloo
Waterloo, ON, Canada
{j3tran, migod, ehslee, holt}@plg.uwaterloo.ca

Abstract

As a software system evolves, its architecture will drift. System changes are often done without considering their effects on the system structure. Consequently, the *concrete* (as-built) architecture becomes convoluted over time making continuing development and maintenance activities increasingly difficult and highly error prone. This problem of architectural drift is especially pronounced in open source systems, where many developers may work in isolation on distinct features with little co-ordination. In this paper, we present our experiences with analyzing and repairing the architectures of two large open source systems: the Linux operating system kernel and the VIM text editor. For both systems, we were successful in removing many *structural anomalies* from their architectures.

1 Introduction

Useful software systems must evolve or they risk becoming obsolete [13]. A system changes because it needs to satisfy user demands and keep up with changing technology. A software system that cannot adapt quickly may lose market share to competitors. Unfortunately, as a system evolves, its *concrete* (as-built) architecture tends to drift from its *conceptual* (as-designed) architecture. This gap between the conceptual and the concrete architecture hinders program understanding and leads to development and maintenance activities that are increasingly difficult and highly error prone.

Software developed under the open source concept [4] is especially susceptible to architectural drift. Open source software (OSS) development is highly collaborative, but because it is also usually highly distributed in nature, co-ordinating the development and managing the design of an OSS system can be very difficult. Most developers are involved in OSS projects as a hobby or on a part-time basis, and they often work on only small pieces or features of the system that are of particular interest to them. These factors often lead to OSS systems that are not carefully architected, or whose architecture drifts as the system evolves.

This paper describes our attempt to analyze and repair the architecture of two large OSS systems: the Linux operating system kernel [3, 1] and the VIM text editor [5]. We analyzed their architectures for structural *anomalies* and were able to remove most of these anomalies using *forward* and *reverse* architecture repair actions [20, 19]. (We explain forward and reverse architecture repair in Section 4.)

The goal of this paper is twofold:

1. To show the effectiveness of forward and reverse architecture repair.
2. To show that architecture analysis and repair is practical to developers, especially with respect to large open source projects.

In particular, we want to demonstrate that our approach to architecture analysis and repair may be applied easily and effectively without a comprehensive understanding of the system's code. We consider that the architecture repair is effective if it narrows the gap between the system's conceptual architecture and its concrete architecture.

The organization of the paper is as follows. The next section presents the problem associated with software evolution, concentrating on the evolution of OSS. Section 3 describes how we obtain a conceptual and concrete architecture for a software system. It also gives a brief description of our architecture model and how the model can be used to help analyze structural anomalies. Section 4 describes how we repair an architecture for a software system. In Section 5, we present our experiences with the repair of the Linux and VIM architectures. A description of related work is presented in Section 6. Section 7 summarizes the contributions of this paper and describes future work.

2 Evolution and Open Source Software

A serious problem facing long lived open source systems is best illustrated with the example of the Perl system [21]. There is a current project, called Topaz [17], to rewrite the internals of Perl in C++. The primary reason for such a project is due to the difficulty in maintaining the current system (Perl 5). Salzenberg, the developer undertaking this task, has stated [17]:

> It really is hard to maintain Perl 5. Considering how many people have had their hands in it, it's not surprising that this is the situation. And you really

need indoctrination in all the mysteries and magic structures and so on-before you can really hope to make significant changes to the Perl core without breaking more things than you're adding.

A fundamental characteristic of open source development is that evolution is managed loosely, with relatively little planning. Developers are always free to contribute new features and bug fixes as they see fit.[1] Although this freedom is a factor that directly contributes to the success of OSS in general, it also contributes to architectural drift. Different developers may develop distinct, personal views of the system's architecture and their changes may interfere with one another.

Another problem with many OSS systems is that most of the active lifespan is devoted to new development and to corrective maintenance; relatively little time is spent performing preventive maintenance tasks, such as restructuring and redesign [14]. Unlike commercial software developers, who are paid to maintain a software system, most open source developers are not paid for their efforts. Preventive maintenance is often seen as a time consuming task with little intellectual reward. Furthermore, performing preventive maintenance impedes the momentum of the development process, which depends on active involvement and maintaining keen interest on the part of the developers and the users. All of these factors may contribute to the decay in an OSS system's architecture; continued evolution of the system only compounds the problems.

Being able to easily analyze and repair an architecture is an asset to most developers involved in large extant software systems, especially in the open source community. With architecture repair, a system's architecture is maintained on a *just-in-time* basis rather than through controlled evolution such as managed development. Architecture repair allows the development of the system to remain flexible without compromising its architecture.

3 Architecture Analysis

To understand how we analyze a software architecture and repair it, we describe how to recover an architecture for a software system and how it is modelled.

3.1 Recovering Architecture

When considering the structure of a large software system, it is convenient to construct a *system decomposition hierarchy* (SDH). This is a tree whose root node represents the system being analyzed, whose internal nodes are subsystems, and whose leaf nodes are modules or source files. Related modules are grouped into subsystems and related subsystems are further grouped into higher level subsystems.

[1] Of course, the project "owner" may decide not to incorporate a particular modification to the official project source. In this case, the contributor may then choose to start a parallel version of the project. Several open source projects have encountered this kind of "splitting", including the emacs text editor.

To help us understand the system structure, we create a conceptual model for it, which we call the *conceptual architecture*. We take the SDH and add interactions that we think exist between the subsystems. If available, we use the system documentation to help create the SDH and the conceptual architecture. However, many OSS systems have little or no documentation regarding their high level designs. To help us create an SDH and a conceptual architecture that are good models of the system structure, we may use supplementary information in the form of the system directory structure, documentation from related systems, user experiences with the system, and the source code comments.

The *concrete architecture* shows the actual interactions between the modules as implemented in the source code. These module interactions are based on dependencies[2] between program entities (*e.g.*, functions and variables). We use the PBS tools [2] to automatically parse the source code to extract the dependencies between program entities and *lift* [10] these dependencies to the module (or source file) level. For example, if function f contained in module M_1 calls function g contained in module M_2, then we consider that there is a dependency between M_1 and M_2. Using the SDH, the PBS tools lift the module dependencies to the subsystem level.

In summary, the conceptual architecture and the concrete architecture are defined as:

$$\begin{aligned} \textit{Conceptual arch.} &= \textit{SDH} + \textit{conceptual interactions} \\ \textit{Concrete arch.} &= \textit{SDH} + \textit{concrete interactions} \end{aligned}$$

3.2 Modelling Architecture

The architecture model, shown in Fig. 1, is described in two hierarchic layers (more layers may be used, depending on the size of the system and the desired level of abstraction). The top layer, the *architecture layer* (AL), describes the system structure at the subsystem and module level (both the conceptual and the concrete architecture are modelled by the AL). The bottom layer, the *entity layer* (EL), describes the system structure that exists between program entities (*e.g.*, functions and variables) and source files.

The different layers facilitate the task of architecture analysis. A new developer who is unfamiliar with the system may choose to analyze the AL to help come up to speed on the project. As the developer becomes more familiar with the system, (s)he can proceed to analyze the EL for detailed *structural descriptions*. The layers also provide developers with a way to map architectural relationships (*i.e.*, relationships between subsystems and modules) to concrete dependencies between program entities, and *vice versa*. This bridge between the source code and the architecture is useful when we are repairing an architecture. We are able to analyze the AL for structural anomalies and use the EL to map the anomalies to specific source code elements.

[2] Dependencies are static name dependencies. That is, a program entity, p, depends on another program entity, q, if p knows about q and uses it.

	Forward Repair	**Reverse Repair**
Does not change code	Kidnapping a subsystem/module Splitting a subsystem	Kidnapping a subsystem/module Splitting a subsystem Adding/removing subsystem interactions
Changes code	Splitting a module Kidnapping a program entity	N/A

Table 1: Categorization of repair actions.

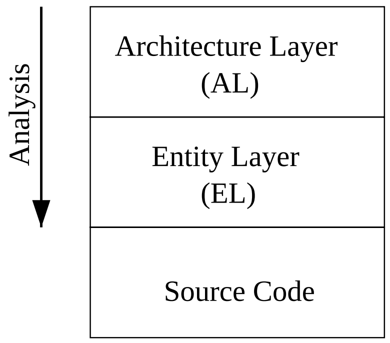


Figure 1: Architecture layers.

4 Architecture Repair

We use *forward* and *reverse* architecture repair [20, 19] to narrow the gap between a system's concrete architecture and its conceptual architecture. The gap between the architectural views are characterized by *anomalies*. Anomalies are architectural elements that are present in one view, but not the other. Since the architectural views share a common SDH, anomalies correspond to dependency edges. We distinguish two types of anomalous dependencies:

1. *Unexpected*: Architectural dependencies that are present in the concrete architecture, but are not modelled in the conceptual architecture.

2. *Gratuitous*: Architectural dependencies that are modelled in the conceptual architecture, but are not present in the concrete architecture.

Forward architecture repair changes the concrete architecture to match the conceptual architecture. This is accomplished by removing unexpected dependencies from the concrete architecture. Reverse architecture repair changes the conceptual architecture to match the concrete. Reverse repair actions can remove gratuitous dependencies from the conceptual architecture or remove unexpected dependencies from the concrete architecture.

Forward repair actions can be categorized into two categories:

1. Repair actions that change source code.

2. Repair actions that do not change source code.

Forward repair actions that change source code include *splitting* modules and *kidnapping* program entities. Those actions that do not change source code include kidnapping architectural components (*i.e.*, subsystems and modules) and splitting subsystems. The general kidnapping action involves moving an architectural component or a program entity from one parent to a new parent. For example, kidnapping a module moves the module from the current parent subsystem to a new subsystem. Kidnapping a program entity moves the entity from one module to a new module. The general splitting action involves dividing the contents of an architectural component into independent design partitions then moving one or more partitions to new subsystems. For example, splitting a module effective divides the program entities into different partitions. One or more partitions, each representing a new module, are then moved to new subsystems.

Reverse architecture repair modifies a conceptual architecture, it does not change source code. Reverse repair actions alter a conceptual architecture by:

1. Changing the SDH.

2. Adding or removing subsystem edges, which represent interactions.

Reverse repair actions that alter an SDH include kidnapping, splitting, and merging subsystems. Reverse repair can remove unexpected dependencies by adding subsystem interactions to the conceptual architecture. It can also remove gratuitous dependencies by removing subsystem interactions from the conceptual architecture.

Table 1 summarizes the repair actions and the category they belong to. It should be noted that performing a repair action that changes an SDH will change both the conceptual and the concrete architecture.

Our approach to architecture repair is to contrast the conceptual architecture and the concrete architecture to identify anomalies. Then selecting an anomaly (or a set of related anomalies[3]), we apply either forward or reverse repair actions to remove the anomaly(ies) from the architectural views. This process is repeated until one of the following conditions are met:

1. All anomalies have been removed from the architectural views.

[3]A set of related anomalies may be all anomalies resulting from a single program entity or module.

2. Any attempts to remove the remaining anomalies are deemed too risky or not worthwhile.

Obviously, the first condition is ideal. However, in our experience architecture repair usually ends, after much restructuring has been done, with the realization that any remaining improvement will come only at the cost of a detailed redesign and reimplementation.[4] Forward repair actions involve only moving and regrouping program entities (and their containers); forward repair cannot remove dependencies between program entities.

It should also be noted that our approach to remove architectural anomalies is not "monotonic". That is, it is possible for a repair action to add more anomalies than it removes. Architecture analysis merely diagnoses possible problem areas in a system's architecture. The developer's skill and common sense are still required to perform the repair actions. For example, moving a variable from one module to another may add new dependencies, but this repair action may still be desirable if the modular structure of the program has been improved.

5 Case Studies

We now describe our experiences in analysing and repairing the architectures of two large open source systems: the Linux operating system kernel (release 2.0.0), and the VIM text editor (release 5.3). For each system, we assumed the role of an outsider who had not contributed to the system's development and did not have a significant understanding of its internals. The primary reason to adopt this role is because we are, indeed, not developers of either system. The secondary reason is that we want to show that architecture analysis and repair can be accomplished without having significant familiarity with the internal code structure of the system.

As non-developers, we wanted to make minimal changes to the systems' internals, and consequently we favoured repair actions that did not change the source code. Also, we prefer to remove unexpected dependencies using forward repair actions rather than reverse repair actions. The reason is that unexpected dependencies are present in the concrete architecture and that forward architecture repair is intended to repair the concrete architecture.

5.1 The Linux Kernel

Release 2.0.0 of the Linux kernel is a large system consisting of approximately 350 KLOC[5] spanning over 900 source files.

5.1.1 Linux's Initial Architecture

We used the conceptual architecture and the SDH for the Linux architecture as described by Bowman *et al.* [6, 7]. We considered this conceptual architecture to be a good candidate as it has been critically examined by others [6]. The top-level view of the conceptual and the concrete architecture for the Linux kernel is shown in Fig. 3a and Fig. 3b, respectively. Table 2 gives a brief description of each of the top-level subsystems.

In our architecture diagrams, we use a dashed box to reduce the number of edges from the diagram; it represents a "virtual" subsystem and can be interpreted as a regular subsystem. In the concrete architecture diagrams, hollow arrow heads represent unexpected dependencies. Also, the arrow head size indicates strength of the dependency between the subsystems. The number next to an arrow head gives the number of program level dependencies in the direction of the arrow.

Comparing the two architectural views, we can see that there are 487 unexpected dependencies in the concrete architecture.

5.1.2 Linux Architecture Repair

Our goal in repairing the Linux architecture was to eliminate as many of the 487 unexpected dependencies as possible. Since all the anomalies were unexpected, we started with forward architecture repair. Our strategy to repair the Linux concrete architecture was to select a subsystem and to attempt to repair all unexpected dependencies entering it. In addition, we wanted to choose the smallest subsystem to repair first. Since we were unsure of our approach to architecture repair, we wanted to start with the smallest subsystem so that any improvements to the architecture can be detected quickly. We also hoped that while repairing the small subsystems, we might *accidentally* remove other anomalies.

The scenario depicted in Fig. 2 (unexpected dependencies are represented by dashed edges) illustrates how anomalies are accidentally removed from the architecture. Suppose we are trying to remove the unexpected dependency from $S1$ to $S3$. Also, suppose that it can be justified that $S1$ can contain C. The result of $S1$ kidnapping C from $S3$ is that the unexpected dependency from $S1$ to $S3$ is removed and the unexpected dependency from $S3$ to $S2$ is accidentally removed.

We selected the Inter-Process Communication (IPC) subsystem to repair first since it was the smallest subsystem (11 modules and approximately 2900 LOC). We analyzed the unexpected dependencies entering IPC and discovered that almost all of them ended at a subsystem within IPC. This pattern suggested that this subsystem may not belong in IPC. Since we knew which modules were causing the anomalies, we quickly read their source code comments to determine whether they belong in IPC. We discovered that the modules in this subsystem were handling Linux loadable modules[6]. We concluded that it would be good to change the SDH so that Process contained these modules. Hence, a kidnapping action was performed on the subsystem which moved it from IPC to

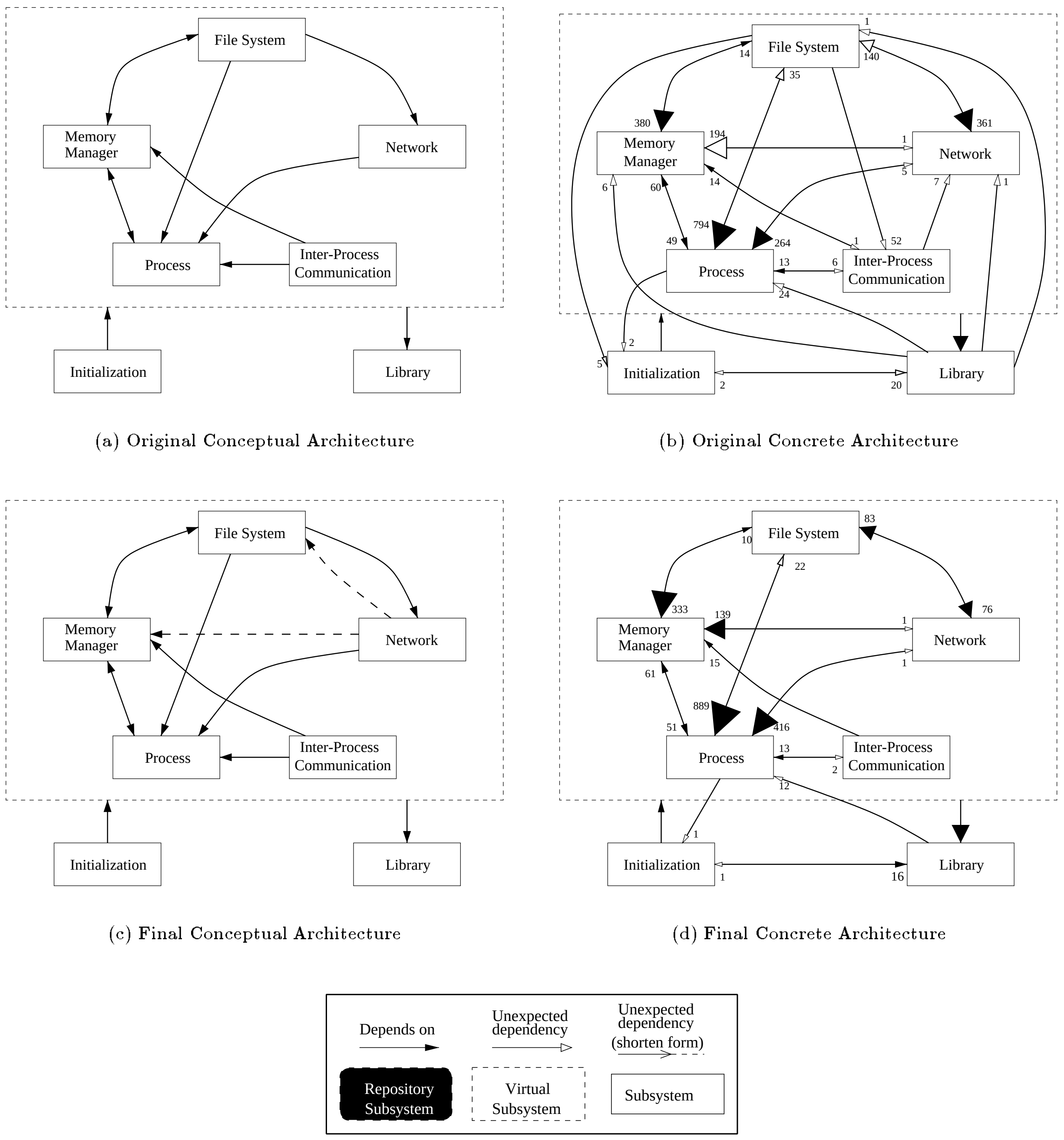
[4] Of course, the developer is free to perform such actions and construct new architectural views.

[5] Lines of source code not counting comments and empty lines.

[6] Linux loadable modules are *plug-and-play* components such as sound support, printer support, device drivers, *etc.*

Subsystem Name	Description
File System	Contains files that implement Linux file systems and file operations.
Memory Manager	Contains files that implement data structures and procedures for memory management.
Process	Contains files that implement core kernel data structures and routines.
Inter-Process Communication	Contains files that implement data structures and procedures for inter-process communication.
Network	Contains files that implement network protocols and drivers.
Library	Contains standard C library routines.
Initialization	Contains files that implement boot-up routines for initial installation of the kernel.

Table 2: Subsystem descriptions for Linux.



(a) Original Conceptual Architecture

(b) Original Concrete Architecture

(c) Final Conceptual Architecture

(d) Final Concrete Architecture

Figure 3: Repair history for the Linux architecture.

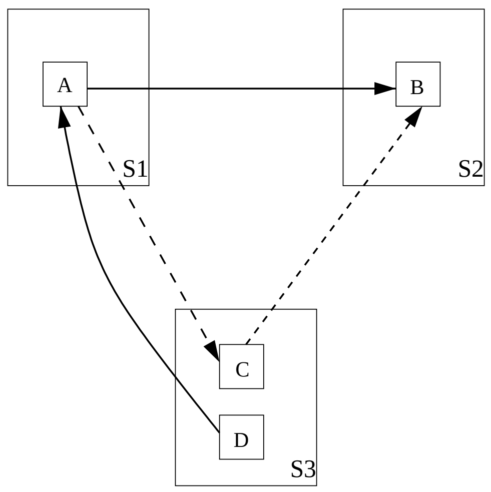


Figure 2: Accidentally removing dependency from $S3$ to $S2$ when $S1$ kidnaps C.

Process.

The next subsystem which we selected to repair was the Network subsystem. We began by trying to remove unexpected dependencies from IPC to Network. We discovered that all seven unexpected dependencies ended up at the module `ipc.h`. This module was used only by modules in IPC. Furthermore, it did not depend on any other modules other than those in the Library subsystem. Since we expect modules in the same subsystem to be tightly coupled [16], this pattern strongly suggested that the module may have been placed in the wrong subsystem. It also suggested that a kidnapping action could be used as a repair action. After viewing the source code of `ipc.h`, we concluded that a better SDH would be to have IPC contain the module. With the new SDH, all seven unexpected dependencies were eliminated from the concrete architecture.

A similar pattern was found when we analyzed the unexpected dependencies from Process to Network. As before, we performed a kidnapping action and were able to remove more unexpected dependencies.

For the other subsystems, such as Memory Manager, File System (FS), and Process, forward architecture repair did not make any significant improvements to the concrete architecture. Architecture analysis revealed that several program entities were declared within the "wrong" subsystems, and hence, they generated unexpected dependencies. Most of these unexpected dependencies were removed by kidnapping program entities and splitting modules. During one particular analysis of an unexpected dependency, we discovered that the dependency was not necessary. The dependency existed because a source file declared a variable that was defined elsewhere. However, this file did not use the variable, and hence, the dependency was not necessary. We removed this unexpected dependency by deleting the variable declaration. In essence, this action is equivalent to kidnapping the variable to a "trash bin."

	Action Type	**Count**
Forward Repair	Kidnapping subsystems/modules	19
	Splitting modules	4
	Kidnapping program entities	9
Reverse Repair	Add Dependency	2

Table 3: Frequency of repair actions applied to the Linux architecture.

At this point in our repair process, we were faced with two anomalies between the architectural views. The concrete architecture indicated dependencies from Network to FS and from Network to Memory Manager. However, the conceptual architecture did not have these subsystem interactions. Since forward architecture repair was no longer effective, we resorted to reverse architecture repair. Analyzing the unexpected dependencies from Network to FS, we discovered that sockets were extensions of the inode data structure. and hence, the Network subsystem used file operations to manipulate sockets. With this fact, we added the interaction from Network to FS in the conceptual architecture.

Examining the unexpected dependencies from Network to Memory Manager, we discovered that all the dependencies were the result of Network modules using Memory Manager entities that handle swap spaces and standard memory referencing routines such as `malloc`. With this information in mind, we referred to documentation on computer networks and discovered that network modules use memory management routines to buffer network packets/frames. This fact justified the reverse repair action of adding a subsystem interaction from Network to Memory Manager.

5.1.3 Repair Summary

The repaired conceptual architecture and concrete architecture for the Linux kernel are shown in Fig. 3c and Fig. 3d, respectively. The dashed edges in the conceptual architecture indicate newly added subsystem interactions. The repair actions narrowed the gap between the conceptual architecture and the concrete architecture from 487 anomalies down to 40. This is a reduction of more than 90% of the anomalies between the the conceptual architecture and the concrete architecture.

In our repair of the top-level architecture of the Linux kernel, we performed both forward and reverse architecture repair. Table 3 summarizes the repair actions that were performed. The forward repair actions included 19 kidnappings of subsystems and modules, 4 module splittings, and 9 kidnappings of program entities. For the reverse repair actions, we added the conceptual interaction from Network to FS and from Network to Memory Manager. We also documented that the Math-Emulator subsystem within the Library subsystem is permitted to depend on Memory Manager.

5.2 The VIM Text Editor

The VIM text editor, release 5.3, is another large open source system, consisting of 100 KLOC across 115 source files. VIM has all the basic features of Vi [9], plus additional features such as a graphical user interface, multiple windows and buffers, multi-level undo, and wildcard expansion.

5.2.1 VIM's Initial Architecture

We used Lee's SDH and conceptual architecture for VIM which were created using system documentation, domain knowledge about text editors, and experiences using VIM [12]. The breakdown of VIM into subsystems is summarized in Table 4.

Lee's conceptual architecture for VIM, shown in Fig. 4a, is modelled as a *repository style* [18] architecture. The central data structures are found in the Global subsystem which all other subsystems are allowed to use. To simplify the diagram, the Global subsystem is drawn in black to denote that all other subsystems depend (or are permitted to depend) on the Global subsystem.

Using the PBS tools and the SDH described by Lee, we automatically extracted the concrete architecture (see Fig. 4b.). As Fig. 4b shows, the concrete architecture is almost fully connected. There are 313 anomalies between the architectural views (Fig. 4a and Fig. 4b). More interestingly, the structure did not reflect a true repository style architecture (*i.e.*, there were dependencies leaving the Global subsystem to other subsystems).

5.2.2 VIM Architecture Repair

We use the same strategy to repair the VIM concrete architecture as we did to repair the Linux concrete architecture. That is, we selected a subsystem and attempted to repair all unexpected dependencies entering it. However, unlike the Linux repair in which we chose the smallest subsystem to repair first, for the VIM architecture we selected the subsystem that we believed would have the greatest impact on the architecture. The primary reason for the difference in strategy is that we were now more experienced with architecture repair. Another reason is that we were not confident with our SDH and conceptual architecture. We wanted to start with the major subsystems so that we could quickly determine whether our models were reasonable representations of the system structure.

We examined the Global subsystem first, because we believed that repairing the VIM architecture so that it exhibited a strict repository style would clarify the architecture. Our analysis of the architecture revealed that several data structures within the Global subsystem depended on data structures declared outside it. The files containing these data structures were not initially placed in the Global subsystem because they were grouped with source files with similar names (*i.e.*, Lee grouped `foo.c` together with `foo.h`). To remove the unexpected dependencies leaving Global, we kidnapped these files from Command, GUI, and Terminal to Global. In the process of removing many unexpected dependencies leaving the Global subsystem, many other unexpected dependencies were accidentally removed from the concrete architecture.

Next, we attempted to repair the Utility subsystem. Analyzing the anomalous dependencies entering Utility, we noticed that many of the anomalies ended at the files `misc1.c` and `misc2.c`. These were two monolithic files consisting of over 3400 LOC and 1500 LOC, respectively. Their names suggested that they contain a variety of unrelated program entities. This fact was verified by the comments within the files — *"functions that didn't seem to fit elsewhere"* and *"various functions"*. Analyzing the anomalies entering these two files, we discovered that they implemented functions and variables to:

a. Handle indentation for C and Lisp programs.
b. Format files.
c. Manipulate the editing screen.
d. Map keyboard and mouse input.
e. Handle regular expressions and wildcard expansions.
f. Manipulate strings and manage memory[7].

It was clear that `misc1.c` and `misc2.c` implemented features for many different subsystems, the obvious subsystems being Command (a), File (b), GUI (c), and Terminal (d). To repair many of the anomalies entering Utility, we partitioned the program entities within these miscellaneous files into five sets:

1. Program entities that belong in the Command subsystem.
2. Program entities that belong in the File subsystem.
3. Program entities that belong in the GUI subsystem.
4. Program entities that belong in the Terminal subsystem.
5. Program entities that do not belong to either Command, File, GUI, or Terminal.

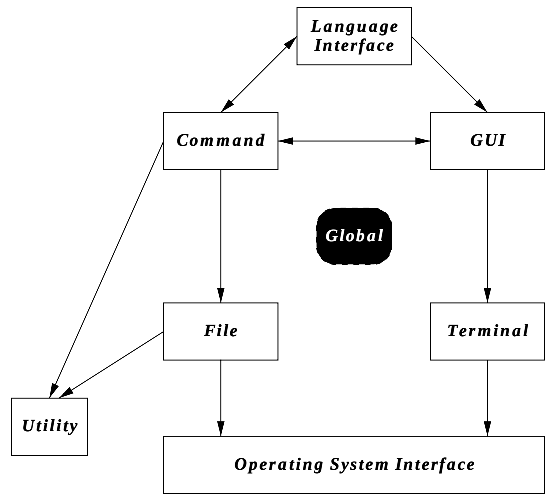
We split the files along these partitions and moved them to their proper subsystems. After this splitting action, `misc1.c` contained program entities to handle regular expressions and `misc2.c` contained program entities for memory management and string manipulations. We relocated the remaining program entities from `misc1.c` to the file `regexp.c`, which implemented most of the regular expression engine. For `misc2.c`, we split the string routines into a new file called `vim_stdlib.c` and the memory routines into a new file called `memory.c`. After all these actions, both the "miscellaneous" files were empty so they were removed from the system.

The two new files, `vim_stdlib.c` and `memory.c`, were used by many of the files in the VIM system. If we kept these files in Utility, we would have to change our conceptual architecture to allow dependencies from the GUI, Terminal, and OS Interface to Utility. We decided to split Utility into two subsystems: Utility and VIM Library. The Utility subsystem still

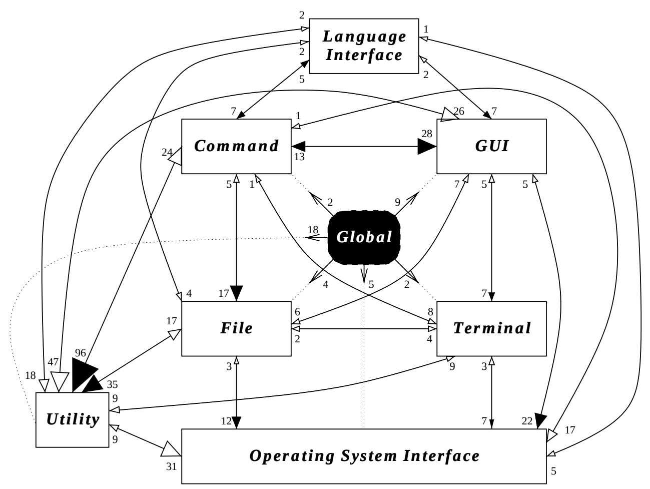
[7]Vim implemented its own memory management and string manipulation procedures to achieve portability.

Subsystem Name	Description
Command	Processes user commands.
File	Contains file read/write operations and program entities to manipulate buffers.
Global	Contains global data structures, variables, and macros.
GUI	Handles the graphical user interface.
OS Interface	Supports various operating systems such as DOS, MacOS, UNIX, MS Windows 95/NT.
Terminal	Declares data for keyboard and mouse button mappings.
Utility	Implements the regular expression engine, multiple undo routine, and character sets.

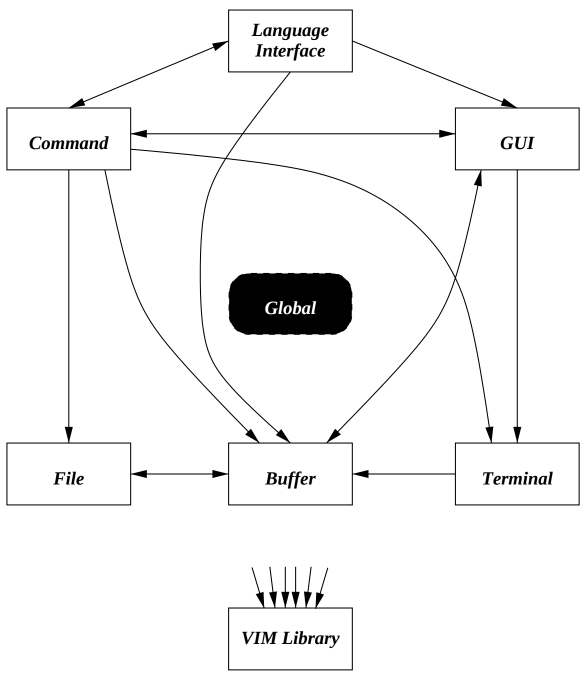
Table 4: Subsystem descriptions for VIM.



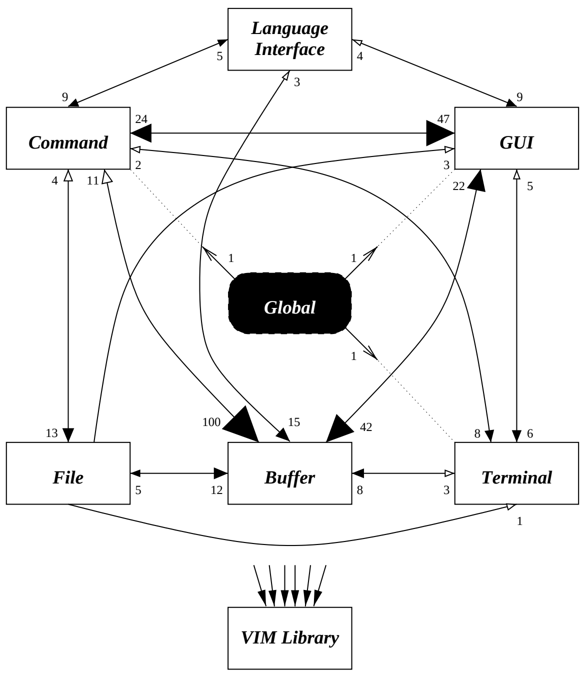
(a) Original Conceptual Architecture



(b) Original Concrete



(c) Final Conceptual Architecture



(d) Final Concrete Architecture

Figure 4: Repair history for the VIM architecture.

contained the regular expression engine, the multi-undo routine, and the implementation of the character sets. The VIM Library subsystem contained the implementations to manage memory and strings. We also changed the conceptual architecture to allow all other subsystems to depend on VIM Library.

The next subsystem which we repaired was the File subsystem. When we analyzed dependencies from GUI to File, we discovered that we had misunderstood the purpose of the file `buffer.c`. Instead of managing buffers for memory allocation, `buffer.c` was managing a different type of buffer. In a VIM session, when you have a split screen, each window in the session is considered to be a buffer. One instance of the GUI subsystem depending on the Buffer subsystem is that the contents of the buffer is parsed to determine the colouring of the buffer content. With this discovery, we decided that a buffer is a separate concept from a file or a screen, and we decided to split off a new subsystem from the File subsystem called Buffer. We also added the subsystem interactions from GUI, File, Command, and Terminal to Buffer in our conceptual architecture. In addition, we added the subsystem interaction from Buffer to GUI since modifying a buffer requires the screen to be updated. These reverse architecture repair actions removed all unexpected dependencies between File and GUI. It also removed all but one dependency between File and Terminal.

Now that there was a new subsystem (Buffer), we returned to the Utility subsystem and attempted to continue where we had previously left off. With the new Buffer subsystem, we were able to move the multi-undo module from Utility to Buffer. The multi-undo feature was maintaining a list of buffers, each buffer having its own change history, and as such, it was reasonable for the module to be contained in Buffer. Another repair action which we performed was to rename Utility to Expressions and to move the subsystem to Buffer. The reason to move Expressions to Buffer was that, after analyzing the unexpected dependencies caused by the regular expression engine, we determined that a regular expression engine is used to match pattern within a buffer. To keep the conceptual architecture simple, we modified the SDH so that Expression was a subsystem within Buffer.

After the kidnapping actions done by Global (described above), the OS Interface contained only one module, `os_unix.c`. This module contained machine dependent routines to process files, handle the graphical interface (*i.e., X Window*), and handle UNIX terminal settings (*e.g.*, terminal I/O and stty settings). We decided to split this module along these partitions and moved the new modules to the proper subsystem. Effectively, we removed the OS Interface subsystem and added an OS interface that is orthogonal to the subsystems.

5.2.3 Repair Summary

From our repair work for VIM, we were able to determine a better conceptual architecture for VIM (*e.g.*, the concept of buffers are now shown in the architecture). Furthermore, we were able to remodularize three source files, each averaging over 3000 LOC. The result of our repair work is summarized in Table 5. We were able to remove approximately 88% anomalies between the architectural views, from 313 to 39. The repaired conceptual architecture and the repaired concrete architecture are shown in Fig. 4.

	Action Type	**Count**
Forward Repair	Kidnapping subsystems/modules	36
	Splitting modules	17
	Kidnapping program entities	25
Reverse Repair	Kidnapping subsystems	1
	Splitting subsystems	2
	Adding Dependency	7

Table 5: Frequency of repair actions applied to the VIM architecture.

6 Related Work

Krikhaar *et al.* propose a two-phase approach to architecture improvement [11]. In the *architecture impact analysis* phase (Phase I), ideas on how an architecture may be improved are performed on the architecture model. Changing the architecture model allows the architect to determine the architectural impact resulting from the the modifications without changing the source code. If the effects are desirable, the changes are submitted to a recipe. In phase two, the *transformation* phase, the recipe is used to implement the changes required to improve the system's architecture. In both the two-phase architecture improvement approach and the forward and reverse architecture repair approach, an architecture model is used to analyze the impact of a particular repair action. A difference with forward and reverse architecture repair is that the repair actions can change a conceptual model as well as a concrete architecture. Also, rather than idealizing changes (as in the two-phase approach), the forward and reverse architecture approach determine the required changes based on the difference between a conceptual architecture and a concrete architecture.

When recovering and analyzing the architecture of a software system, we have adopted the reflexion model approach suggested by Murphy *et al.* [15]. The reflexion model allows a developer to derive an system's architecture that reflects his/her mental model. Since the architecture is a *reflexion* of the developer's intuition, understanding the the system structure is much easier. Murphy *et al.* used the terms *absent* and *divergent* instead of *gratuitous* and *unexpected.*

7 Conclusions

We have presented two open source systems which we repaired using forward and reverse repair actions. For both systems, we were able to improve the concrete architecture by removing many of the unexpected dependencies. For the VIM repair work, we were also able to repair the conceptual architecture. We believe that the new conceptual architecture for VIM accurately reflects the system structure.

The entire repair process can provide insight into the current state of the system structure. If most of the repair actions fall under the category of "does not change code", this indicates that our original models (the SDH and the conceptual architecture) are poor models of the current system structure. If most of the repair actions are those that change source code, this indicates that the system lacks modularity. For example, from our experience with Linux leads us to believe that it is a well structured system with a few modularity problems; the majority of the anomalies in the Linux architecture were the result of our SDH being inaccurate. From our experience with repairing the VIM text editor, we determined that the original conceptual architecture and SDH were not accurate models. Furthermore, the high count in the number of kidnapping program entity and splitting module action indicates that the system lacked modularity, as evident by the files `misc1.c`, `misc2.c`, and `os_unix.c`. For both case studies, we have shown that our approach to architecture analysis can easily identify problem areas in a system's architecture. Furthermore, we have also described how our approach to architecture repair can be used to improve the conceptual model and the SDH, as well as improve the global modularity of the system.

8 Future Research

One area of future research is to seek external validation of our work from the Linux and VIM communities. We have recently begun to interact with the VIM community, including the "owner" of the VIM project, Bram Moolenaar, who has expressed interest in our project. We hope to demonstrate that the repaired VIM architecture, especially our repair work with the files `misc1.c` and `misc2.c`, is an improvement to the existing system structure.

Also, while our tool for architectural analysis (the PBS system) is fairly mature [2], most of our architectural repair actions were carried out using a text editor and various shell scripts. We intend to extend the functionality of PBS to support simple repair actions, thus making architectural repair faster and less prone to routine error.

References

[1] Linux Online! Available at {`http://www.linux.org`}.

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

[3] The Linux Kernel Archives. Available at {`http://www.kernel.org`}.

[4] The Open Source Page. Available at {`http://www.opensource.org`}.

[5] The VIM (Vi IMproved) Home Page. Available at {`http://www.vim.org`}.

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

[7] N. Brewster. The Linux Bookshelf, September 1998. Available at {`http://www.cs.toronto.edu/~brewste/bkshelf/linux/V2.0.27a/index.html`}.

[8] F. P.-Jr. Brooks. *The Mythical Man-Month.* Addison Wesley, 1995.

[9] W. Joy. An Introduction to Display Editing. Electrical Engineering Computer Science, University of California, Berkeley, CA, USA.

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

[11] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A Two-phase Process for Software Architecture Improvement. *Proc. of the Intl. Conference on Software Maintenance*, September 1999.

[12] E. H. S. Lee. Architecture of VIM, 1998. Available at {`http://plg.uwaterloo.ca/~ehslee/vim/vim_arch.html`}.

[13] 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.

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

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

[16] G. J. Myers. *Reliable Software Through Composite Design.* Mason/Charter, London, England, first edition, 1975.

[17] C. Salzenberg. Topaz: Perl for the 22nd Century, September 1999. Available at {`http://www.perl.com/pub/1999/09/topaz.html`}.

[18] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[19] J. B. Tran. *Software Architecture Repair as a Form of Preventive Maintenance.* Master's thesis, University of Waterloo, Waterloo, ON, Canada, 1999.

[20] J. B. Tran and R. C. Holt. Forward and Reverse Architecture Repair. To be published in *Proc. of CASCON '99*, Toronto, Nov. 1999.

[21] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl.* O'Reilly, second edition, 1996.