Architectural 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. These changes often introduce structural anomalies *between the* concrete *(as-built) and the* conceptual *(as-designed) architecture which can impede program understanding. The problem of architectural drift is especially pronounced in open source systems, where many developers work in isolation on distinct features with little co-ordination. In this paper, we present our experiences with repairing the architectures of two large open source systems (the Linux operating system kernel and the VIM text editor) to aid program understanding. For both systems, we were successful in removing many structural anomalies from their architectures.*

1. Introduction

Useful software systems must evolve or else they risk becoming obsolete [12]. A system changes because it needs to satisfy user demands and keep up with changing technology. A 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. The 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 usually 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 the OSS project 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 architectures drift as the systems evolve.

This paper describes our attempt to 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* (differences between the conceptual architecture and concrete architecture) and were able to remove most of these anomalies using *forward* and *reverse* architecture repair [19, 18]. We will 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 done easily and effectively without a deep understanding of the system's code. We consider that architecture repair is effective if it helps program understanding by narrowing the gap between the system's concrete architecture and its conceptual architecture.

The organization of the paper is as follows. The next section presents problems associated with software evolution, concentrating on the evolution of OSS. Section 3 describes the process of obtaining a conceptual and concrete architecture for a software system. We also describe our model for architecture 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. Problem Facing OSS Evolution

The problem of understanding long lived open source systems is illustrated by the Perl system [20]. There is a current project, called Topaz [16], to rewrite the internals of Perl in C++. The primary reason for such a project is due to the great difficulty in understanding the Perl software to be able to maintain it. Salzenberg, the developer spearheading Topaz stated [16]:

> 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 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 the active lifespan is devoted to new development and corrective maintenance; relatively little time is spent performing preventive maintenance [13], such as restructuring and redesign. Unlike commercial software developers, who are paid to maintain the system, most open source developers are not paid for their efforts. Preventive maintenance is often seen as an onerous 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 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.

[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, including the emacs text editor, have encountered this kind of "forking".

Being able to easily analyze and repair an architecture is highly advantageous for most developers involved in large extant software systems, especially in the open source community. The ease of repairing an architecture means that more time can be devoted to developing new features. Also, with an architecturally repaired software system, new and senior developers can more easily understand the parts of the program that need to be modified.

3. Architecture Model

Before discussing architecture repair in more detail, we will describe how the architecture for a software system is modelled. In particular, we will describe the four major concepts of our architecture model: the system hierarchy, the conceptual architecture, the concrete architecture, and the layered model.

3.1. The System Hierarchy

When considering the structure of a large software system, it is convenient to construct a *system hierarchy* (SH). 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[2]. Related modules are grouped into subsystems and related subsystems are further grouped into higher level subsystems. We often draw the system hierarchy as boxes contain smaller boxes. Figure 1a illustrates how we draw a system hierarchy with root S, internal nodes T and U, and leaf nodes M, N, P, and Q.

3.2. The Conceptual Architecture

To help us understand the system structure, we create a conceptual model for it, which we call the *conceptual architecture*. The conceptual architecture consists of a system hierarchy and conceptual interactions between subsystems and modules within the hierarchy. Conceptual interactions are based on programmers' intuition about the relationships between subsystems and modules. If available, we use the system documentation to help create the conceptual architecture. However, many OSS systems have little or no documentation regarding their high level designs. To help us create a conceptual architecture that is a good model of the system structure, we may also use supplementary information in the form of the system directory structure, documentation from related systems, user experience with the system, and comments in the source code.

[2] In this paper, we will refer to a source file as a module.

3.3 The Concrete Architecture

The concrete architecture shows the system's structure as implemented in the source code. It consists of a system hierarchy, possibly different from the SH for the conceptual architecture, and concrete interactions between subsystems and modules. Although the system hierarchy for the conceptual architecture and the system hierarchy for the concrete architecture may be different, for our case studies, they are the same.

The concrete interactions are based on dependencies between program entities[3] (*e.g.*, functions and variables). We use the PBS tools [2] to automatically parse the source code and extract the dependencies between program entities. The tools then *lift* [9] these dependencies to the module and subsystem level. For example, in Fig.1b the dependency from module M to module N is lifted from the program entity dependency between M and N, and the dependency from T to U is lifted from the dependency from N to P.

3.4 The Layered Model

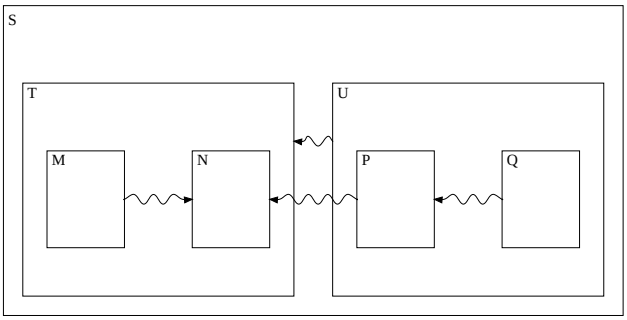
The architecture model we use consists of two layers (more layers may be used, depending on the level of abstraction which the user wants to represent the system structure):

- *Architectural layer* (AL): Describes interactions between subsystems and modules, as well as the system hierarchy (both the conceptual and the concrete architecture are modelled by the AL).

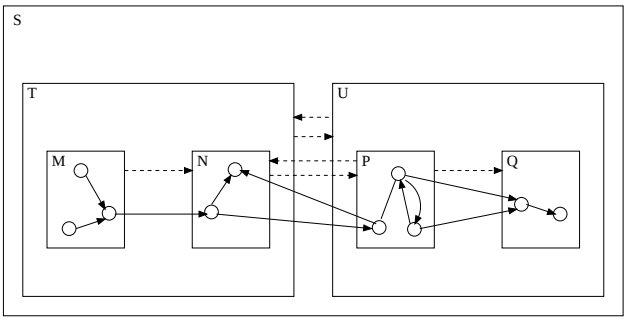
- *Entity Layer* (EL): Describes the system structure that exists between program entities (*e.g.*, functions and variables) and modules.

The different layers facilitate the task of architecture analysis and program comprehension. 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 he becomes more familiar with the system, he can proceed to analyze the EL for detailed *structural descriptions*. The layers also facilitate the architecture repair process. Structural anomalies are diagnosed at the architectural layer by contrasting the conceptual and concrete architecture. These anomalies, in the case of unexpected dependencies, can be mapped to the entity layer to determine the program entities causing them.

[3] Dependencies are static name dependencies. That is, a program entity, p, depends on another program entity, q, if p uses the name of q. Examples of static name dependencies are function calls and variable references.



(a) Conceptual architecture



(b) Concrete architecture with program entity dependencies

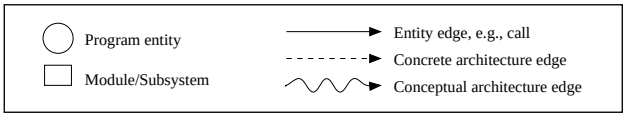


Figure 1. Modelling a conceptual and concrete architecture.

4. Architecture Repair

We use *forward* and *reverse* architecture repair [19, 18] to narrow the gap between a system's concrete architecture and its conceptual architecture. The gap between the architectural views is characterized by *anomalies*, which are architectural elements that are present in one view, but not the other. Since the architectural views often have the same system hierarchy, anomalies commonly correspond to subsystem and module dependencies. We distinguish two types of anomalous dependencies:

1. *Unexpected dependencies*: These architectural dependencies are present in the concrete architecture, but not

	Reverse Repair	Forward Repair
Modifies the AL	Kidnapping subsystems or modules Splitting subsystems Merging subsystems Adding/removing subsystem interactions	Kidnapping subsystems or modules Splitting subsystems or modules
Modifies the EL	NA	Kidnapping program entities Splitting modules

Table 1. Categorization of repair actions.

in the conceptual architecture. Referring to Fig.1b, the dependencies from N to P, P to Q, and T to U are unexpected with respect to Fig.1a.

2. *Gratuitous dependencies*: These architectural dependencies are present in the conceptual architecture, but not in the concrete architecture. Referring to Fig.1a, the dependency from Q to P is gratuitous with respect to Fig.1b.

Forward architecture repair changes the concrete architecture to match the conceptual architecture. The forward repair actions can be categorized into two classes:

1. Repair actions that modify the EL (change source code).

2. Repair actions that modify the AL (do not change source code).

Changes that are done to the EL are lifted to the AL to alter the concrete architecture. Forward repair actions that modify the AL simply alters the system hierarchy.

There are two types of forward repair actions:

- *Kidnapping*: Moves a subsystem, module, or program entity from one container to a new container. For example, kidnapping a program entity moves it from one module to another module. Kidnapping a module moves the module from one subsystem to another subsystem.

- *Splitting*: Divides the contents of a subsystem or module into separate components. These components are subsystems, if the splitting action is applied to a subsystem, or modules, if the splitting action is applied to a module. These new subsystems or modules are then moved to other subsystems.

Reverse architecture repair changes the conceptual architecture to match the concrete. Reverse architecture repair modifies the AL, but it does not change source code. Reverse repair actions alter a conceptual architecture by:

- Adding or removing subsystem/module interactions. Adding interactions to the conceptual architecture will cause certain unexpected dependencies to become expected. Removing interactions will remove gratuitous dependencies from the conceptual architecture.

- Changing the system hierarchy. Reverse repair actions that alter the system hierarchy include kidnapping, splitting, and merging subsystems.

Table 1 summarizes the repair actions and the categories they belong to. It should be noted that performing a repair action that changes a system hierarchy for a conceptual architecture sometime results in a mismatch with the system hierarchy for the concrete architecture, and *vice versa*. After any modifications to a system hierarchy, we will likely want to *coalesce* the SH for the conceptual architecture and the SH for 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[4]), 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.

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

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[5]. Forward repair actions involve only moving and regrouping program entities (and their containers); forward repair cannot remove dependencies between

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

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

program entities (*e.g.*, procedural calls between two functions).

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 diagnose 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 analyzing 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 outsiders who have 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, non-developers of either systems. 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 want to make minimal changes to the system's internals, and consequently we favoured repair actions that did not change the source code. Also, we assume that unexpected dependencies are the result of poor implementation, and hence we prefer to use forward repair actions to remove unexpected dependencies rather than reverse repair actions.

5.1. The Linux Kernel

We analyzed release 2.0.0 of the Linux kernel which is a large system consisting of approximately 350 KLOC[6] spanning over 900 source files.

5.1.1 Initial Architecture for Linux

We used the conceptual architecture for the Linux kernel 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 initial conceptual and concrete architectures for the Linux kernel are shown in Fig.3a and Fig.3b, respectively.

[6] Thousands of lines of source code not counting comments and empty lines.

In our architecture diagrams, we use a dashed box to group subsystems; an arrow ending (beginning) at a dashed box represents arrows to (from) many or most of its contained boxes. For example, the File System, Memory Manager, Network, Process, and Inter-Process Communication subsystems all depend on the Library subsystem.

In the concrete architecture diagrams, hollow arrow heads represent unexpected dependencies. The arrow head size indicates the strength of the dependency between the subsystems; more particularly, the number next to an arrow head gives the number of subsystem and module level dependencies in the direction of the arrow.

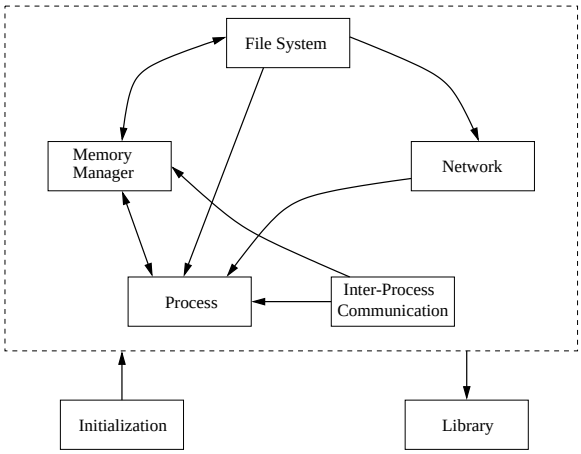
5.1.2 Linux Architecture Repair

Our goal in repairing the Linux architecture was to reconcile its conceptual architecture and its concrete architecture. Since there were no gratuitous dependencies in the initial conceptual architecture, but many (502) unexpected dependencies, we started the Linux architecture repair process with forward repair. Our strategy for repairing the Linux concrete architecture was to repair one module or subsystem at a time, removing all unexpected dependencies entering it. We wanted to choose small subsystems first. We were not very confident with our approach to architecture repair and repairing small subsystems quickly revealed any improvements to the architecture. We also hoped that while repairing each subsystem, we might *accidentally* remove other anomalies so that the entire repair process would be shortened.

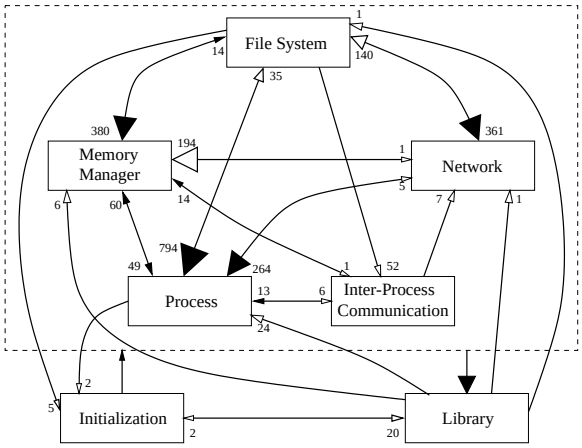
The repair scenario depicted in Fig.2 (unexpected dependencies are represented by dashed edges) illustrates how anomalies are accidentally removed from the architecture. Suppose we are trying to remove the unexpected dependency from $S1$ to $S3$. Also, suppose that we can justify moving C to $S1$. The result is that when we remove the unexpected dependency from $S1$ to $S3$ by moving C to $S1$, we accidentally remove the unexpected dependency from $S3$ to $S2$.

We selected the Inter-Process Communication (IPC) subsystem to repair first since it was the smallest subsystem, containing 11 modules and 2900 LOC. We analyzed all unexpected dependencies entering IPC and discovered that many of them ended at one particular subsystem within IPC. This pattern suggested that this subsystem may not belong in IPC. Since we knew the subsystem causing the anomalies, we quickly read the source code comments of the modules contained in this subsystem to determine whether it belongs to IPC. We discovered that this subsystem was handling Linux loadable modules[7]. We concluded that it would

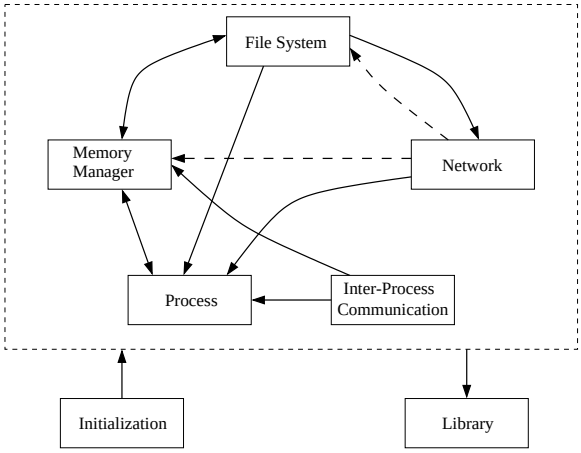
[7] Linux loadable modules are *plug-and-play* components such as sound



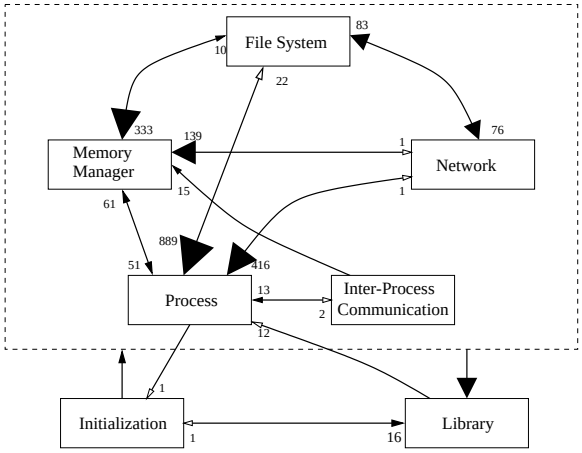
(a) Initial conceptual architecture



(b) Initial concrete architecture



(c) Final conceptual architecture



(d) Final concrete architecture

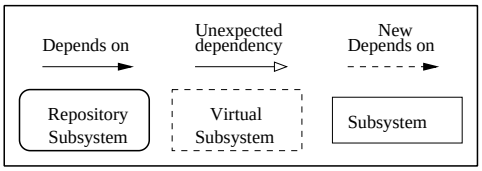


Figure 3. Repair history for the Linux architecture.

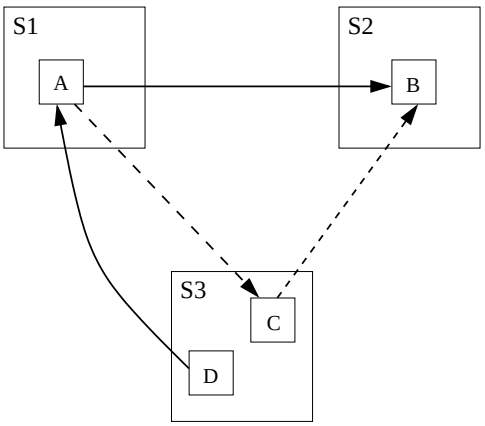


Figure 2. Accidentally removing dependency from $S3$ to $S2$ when $S1$ kidnaps C.

be good to change the system hierarchy for the concrete architecture so that Process contained the subsystem handling Linux loadable modules. Hence, a kidnapping action was performed on the subsystem which moved it from IPC to the Process subsystem.

The next subsystem which we selected to repair was the Network subsystem. We began by trying to remove seven unexpected dependencies from IPC to Network (see Fig.3b). We discovered that all unexpected dependencies ended up at a module called `ipc.h`. We also discovered that 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 [15], this pattern strongly suggested that `ipc.h` was placed in the wrong subsystem. It also suggested that a kidnapping action could be used as a repair action. After reading the source code comments for `ipc.h`, we decided to modify the system hierarchy by moving the module to IPC. Under the modified system hierarchy, all seven unexpected dependencies were eliminated from the concrete architecture.

A similar pattern was found when we analyzed the five unexpected dependencies from Process to Network (see Fig.3b). Like before, we performed a kidnapping action and the results were the removal of more unexpected dependencies.

For the other subsystems, such as Memory Manager, File System, and Process, architecture analysis revealed that many program entities were being declared within the "wrong" modules, and hence causing unexpected dependencies. Most of these unexpected dependencies were removed by kidnapping program entities and splitting modules.

At this point in our repair process, we could not perform any forward repair actions without altering the program's semantics (*i.e.,* redesigning and reimplementing components within the Linux kernel). However, there were still 194 unexpected dependencies from Network to Memory Manager and 140 unexpected dependencies from Network to File System. Since forward architecture repair was not feasible, we resorted to reverse architecture repair. Analyzing the unexpected dependencies from Network to File System, we discovered that sockets were extensions of the inode data structure, and hence, many procedures which operate on sockets reused inode handling routines. With this knowledge, we expect the Network subsystem to depend on the File System and so we added the subsystem interaction from Network to File System to the conceptual architecture. As the result, all dependencies from the Network to the File System were no longer unexpected in the concrete 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 (see Fig.3c and Fig.3d).

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 502 anomalies down to 40. The remaining 40 architecture anomalies could not be removed without either adding dependencies to the conceptual model which may contradict the developer's intuition (*e.g.,* adding a dependency from the Process subsystem to the Network subsystem), or modifying program entities. Modifying a program entity is considered a risky action that is likely to introduce system faults, and hence, such an action was avoided.

In our repair of the top-level architecture of the Linux kernel, we performed both forward and reverse architec-

support, printer support, device drivers, *etc.*

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

Table 2. Frequency of repair actions applied to the Linux architecture.

ture repair. Table 2 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 interactions from Network to File System and from Network to Memory Manager. We also decided to document the dependency of the Math-Emulator subsystem, within the Library subsystem, on the Memory Manager rather than show it in our conceptual and concrete architectures.

5.2. The VIM Text Editor

We analyzed the VIM text editor, release 5.3, which is another large open source system, consisting of 100 KLOC across 115 source files. VIM has all the basic features of `vi` [8], plus additional features such as a graphical user interface, multiple windows and buffers, multi-level undo, and wildcard expansion.

5.2.1 Initial Architecture for VIM

Initially, we used Lee's conceptual architecture for VIM which was created using system documentation, domain knowledge about text editors, and experience using VIM [11]. This conceptual architecture for VIM, shown in Fig.4a, is modelled as a *repository style* [17] architecture, with central data structures found in the Global subsystem. To simplify the diagram, the Global subsystem is drawn as a box with rounded corners to indicate that all other subsystems depend (or are permitted to depend) it. Table 3 gives a description for each subsystem.

Figure 4b shows the concrete architecture for VIM. Interestingly, the structure does not reflect a true repository style architecture in that there are dependencies from the Global subsystem to other subsystems.

5.2.2 VIM Architecture Repair

We used almost the same strategy to repair the VIM concrete architecture as we did to repair the Linux concrete architecture. That is, we repeatedly selected a subsystem and attempted to repair all unexpected dependencies entering it. However, unlike the Linux repair in which we chose small subsystems first, for the VIM architecture we selected the subsystems that we believed would have the greatest improvement 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 initial conceptual architecture, and so by repairing the major subsystems, we could quickly determine whether the conceptual architecture is a reasonable model of the system structure.

We examined the Global subsystem first. 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. Lee did not initially place the modules containing these data structures in the Global subsystem because he grouped them together with modules with similar names (*e.g.,* `foo.h` was grouped together with `foo.c`). To remove the unexpected dependencies leaving Global, we kidnapped these modules from Command, GUI, and Terminal to Global. Removing the dependencies leaving the Global subsystem caused many other unexpected dependencies to be accidentally removed from the concrete architecture.

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

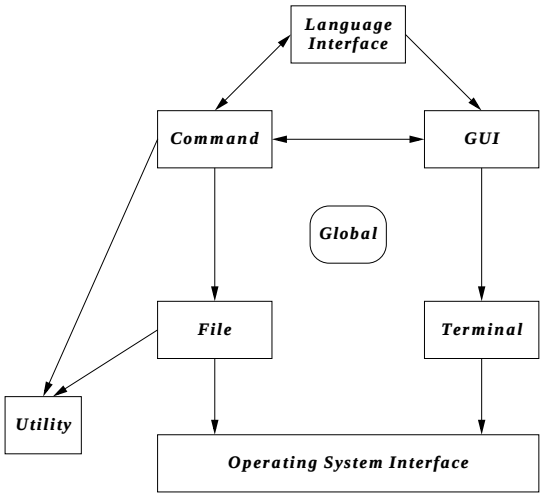
a. handle indentation for C and Lisp programs,

b. format files,

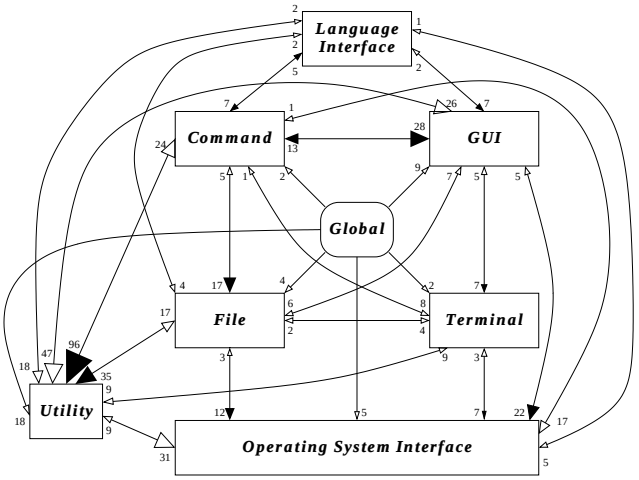
c. manipulate the editing screen,

d. map keyboard and mouse input,

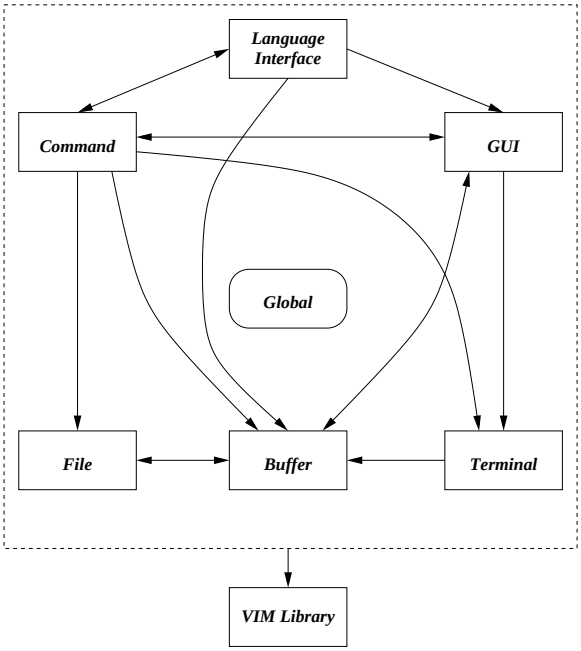
e. handle regular expressions and wildcard expansions,



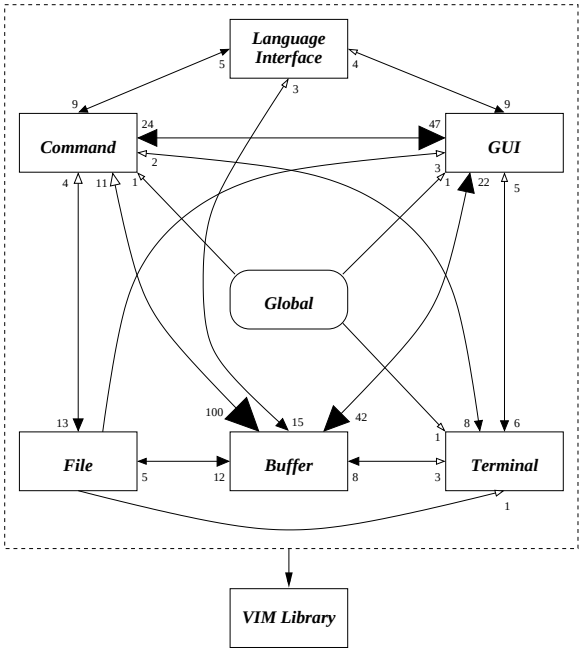
(a) Initial conceptual architecture



(b) Initial concrete architecture



(c) Final conceptual architecture



(d) Final concrete architecture

Figure 4. Repair history for the VIM architecture.

Subsystem Name	Description
Command	Processes user commands.
File	Contains file read/write operations and program entities to manipulate file 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 3. Subsystem descriptions for VIM.

f. manipulate strings and manage memory[8].

It was clear that `misc1.c` and `misc2.c` implemented features for many different subsystems, the obvious subsystems being Command, File, GUI, and Terminal. We split these two miscellaneous modules according to the dependencies (*i.e.*, to or from Command, File, GUI, and Terminal) and moved the new modules to the corresponding subsystems. After this splitting action, `misc1.c` contained program entities to handle regular expressions which we then moved to the module `regexp.c`[9]. After the splitting action, the module `misc2.c` contained program entities for memory management and string manipulation. We moved the string routines to a new module called `vim_stdlib.c` and the memory management routines to a new module called `memory.c`. After all these actions, both the "miscellaneous" modules were empty so they were removed from the system.

The two new modules, `vim_stdlib.c` and `memory.c`, were used by many of the modules in the VIM system. If we kept these modules in Utility, we would have to change our conceptual architecture to include dependencies from the GUI, Terminal, and OS Interface to Utility. We decided to change the system hierarchy by splitting Utility into two subsystems: Utility and VIM Library. The Utility subsystem still contained the regular expression engine, the multi-undo routine, and the implementation of the character sets. The VIM Library subsystem contained the implementation to manage memory and strings. We also changed the conceptual architecture to allow all other subsystems to depend on VIM Library.

The next subsystem we repaired was the File subsystem. When we analyzed unexpected dependencies from GUI to File, we discovered that we had misunderstood the purpose of the module `buffer.c`. Originally, we had thought that the module managed the content of a text file. However, the results of our analysis indicated that `buffer.c` also manages the contents of the display screen, such as text colouring. With this discovery, we decided that a buffer is a separate concept from a file or a screen. We decided to modify the system hierarchy by splitting 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. They also removed all but one dependency between File and Terminal.

With the new Buffer subsystem, we returned to the Utility subsystem and attempted to continue to repair unexpected dependencies entering it. With the new Buffer subsystem, we were able to move the multi-undo module from Utility to Buffer since the multi-undo feature was implemented as a list of buffers. Another repair action which we performed was to rename Utility to Expressions and to move the subsystem to Buffer. The reason we moved 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.

After the kidnapping actions done by the Global subsystem (described above), the OS Interface contained only one module, `os_unix.c`. This module contains machine dependent routines to process files, handle the graphical interface (*e.g.*, 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 move the new modules to the corresponding subsystems. This repair action removed the OS Interface subsystem and added, into each subsystem, an OS interface layer.

[8] Vim implements its own memory management and string manipulation procedures to achieve portability.

[9] `regexp.c` implements the regular expression engine for VIM.

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

Table 4. Frequency of repair actions applied to the VIM architecture.

5.2.3 Repair Summary

Our repair work for VIM lead to a better conceptual architecture (*e.g.*, the concept of buffers are now shown in the architecture). Furthermore, we were able to improve the modularity of three modules (`misc1.c`, `misc2.c`, and `os_unix.c`), each averaging 3000 LOC. The result of our repair work is summarized in Table 4. We were able to remove approximately 88% of the architectural anomalies, from 313 to 39 (see Fig.4). The remaining 39 anomalies could not be removed without redefining several program entities. Since we are not VIM developers, we did not attempt to modify those program entities involved in the anomalies.

6. Related Work

Krikhaar *et al.* propose a two-phase approach to architecture improvement [10] which is closely related to our work. In the *architecture impact analysis* phase (Phase I), system structural changes are tested on the architecture model to determine the resulting architecture. If the resulting architecture is desirable, the structural changes are applied to the system to obtain the new architecture. Implementing the changes to the system constitutes the second phase, which is called the *transformation* phase.

Our approach to recovering and analyzing the architecture of a software system is similar to the reflexion model approach suggested by Murphy *et al.* [14]. The reflexion model allows a developer to derive an system's architecture that reflects his/her mental model. The mental model is revised until it more accurately describes a static source model (*i.e., the approach does not modify the source code*). Since the architecture is a *reflexion* of the developer's intuition, understanding the system structure is much easier. Murphy *et al.* used the terms *absent* and *divergent* instead of *gratuitous* and *unexpected*. The repair approach presented in this paper allows a developer to change the mental model (conceptual architecture) to match the source model (concrete architecture), or to change the source model to match the mental model.

7. Conclusions

We have presented two open source systems whose architectures we repaired using forward and reverse repair actions. For both systems, we were able to reconcile their concrete architectures and their conceptual architectures. The repair process was done fairly quickly with the Linux architecture repair requiring 16 person-hours and the VIM repair requiring 80 person-hours.

For both case studies, we have shown that our approach to architecture repair can easily identify problem areas in a system's architecture. With the Linux architecture, the main problem area was the System Hierarchy (as suggested by the high count in the number of subsystem and module kidnapping actions). With the VIM architecture, the problem areas included the modules `misc1.c`, `misc2.c`, and `os_unix.c`, and the initial conceptual architecture.

7.1. Future Work

One area of future research is to seek external validation of our work from the Linux and VIM communities. In particular, we would like to know whether a repaired architecture does improve program comprehension, and hence facilitates software development and maintenance. 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.

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. of ICSE-21, 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] W. Joy. An Introduction to Display Editing. Electrical Engineering Computer Science, University of California, Berkeley, CA, USA.

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

[10] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A Two-phase Process for Software Architecture Improvement. *Proc. of ICSM*, September 1999.

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

[12] 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.

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

[14] 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.

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

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

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

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

[19] J. B. Tran and R. C. Holt. Forward and Reverse Architecture Repair. *Proc. of CASCON '99, Toronto*, pages 15–24, November 1999.

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