

Deriving Change Architectures from RCS History

C. Stringfellow, C. D. Amory, D. Potnuri
Department of Computer Science
Midwestern State University
Wichita Falls, TX 76308, USA
catherine.stringfellow@mwsu.edu
mail@dennisamory.com
d_potnuri@yahoo.com

M. Georg
Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873, USA
mgeorg@holly.ColoState.edu

A. Andrews
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164 USA
aandrews@eecs.wsu.edu

ABSTRACT

As software systems evolve over a series of releases, it becomes important to know which components show repeated need for maintenance. Deterioration of a single component manifests itself in repeated and increasing problems that are local to the component. A second type of deterioration is related to interactions between components, that is, components are repeatedly change-prone in their relationships with each other. The latter requires changes to code in multiple components and is a sign of problems with the software architecture of the system. Software architecture problems are by far more costly to fix and thus it is very desirable to identify potential architectural problems early and to track them across multiple releases. This paper uses Revision Control System (RCS) change history to determine which system parts are the most change-prone, both locally and in their interactions with other parts of the systems. Relationships among system components are identified based on whether they are involved in the same group of changes, and how many lines of code are changed. The resulting change architecture figures show what the system's most change-prone components and relationships are. We illustrate our technique on a large commercial system consisting of over 800 KLOC of C, C++, and microcode.

KEY WORDS

software architecture, reverse architecting, software maintenance

1 Introduction

It is important to track the evolution of a system and its components, particularly those components that are becoming increasingly difficult to maintain as changes are made over time. The identification of these components

serves two objectives. First, this information can be used to direct efforts when a new system release is being developed. This could mean applying a more thorough development process, or assigning the most experienced developers to these difficult components. Secondly, the information can be used when determining which components need to be re-engineered at some point in the future.

Deterioration of a single component manifests itself in repeated and increasing problems that are local to the component. A second type of deterioration is related to interactions between components, that is, components are repeatedly change-prone in their relationships with each other. The latter requires changes to code in multiple components and is a sign of problems with the software architecture of the system. Software architecture problems are by far more costly to fix and thus it is very desirable to identify potential architectural problems early and to track them across multiple releases.

Change reports are a major source of commonly available information that can be used to identify decay. Change reports are written when developers modify code during system development or maintenance, and usually contain information about the nature, time and author of the change. A change can affect code in one or more components. If changes are local to a component, that is only files belonging to the component are changed, the component's change is said to have cohesion. By contrast, if changes involve multiple components, the change shows coupling. This concept of looking at a change as either local to a component or as coupling components with regards to code modification is analogous to describing structure or architecture of software [1, 17]: Software architecture consists of a description of components and their relationships and interactions, both statically and behaviorally [17]. Problems and possible architectural decay can be spotted via changes related to components and interactions of the com-

ponents.

High change cohesion and coupling measures indicate problems, although of a different sort. High change cohesion measures identify components that are broken in their functionality, while high change coupling measures highlight broken relationships between components.

There are choices as to which cohesion and coupling measures to use. Ohlsson et al. [16] identify the most problematic components across successive releases, using simple coupling measures based on common code fixes as part of the same defect report. Von Mayrshauser et al. [20] use a defect coupling measure that is sensitive to the number of files that had to be fixed in each component for a particular defect. These defect cohesion and coupling measures can be computed for all components and component relationships that contain faults. However, usually only the most fault-prone components and component relationships are of concern, since they represent the worst problems and the biggest potential for code decay.

Change management data is useful in measuring properties of software changes and such measures can be used in making inferences about cost and change quality in software production. With this in mind, the primary interest is to identify components and component relationships that exhibit problems most often, i. e. with the highest change cohesion and change coupling measures.

This paper investigates ways to

- identify components and relationships between components that are change-prone through reverse architecting techniques [3, 5, 8, 11, 13, 21, 22]. Reverse architecting in this context refers to the identification of a system's components and component relationships without the aid of an existing architecture document.
- measure change cohesion and change coupling for the components and component relationships.
- set thresholds when distinguishing between change-prone and non-change-prone components and component relationships. The components and component relationships that are change-prone form the part of the software architecture that is problematic. This is called the change architecture.
- compare the change architectures derived by grouping changes by different time intervals.

Section 2 reports on existing work related to using change process data. It also summarizes existing classes of reverse architecting approaches. Section 3 details our approach. Section 4 reports on its application to a sizable embedded system. The results show identifiable problems with a subset of the components and relationships between them, indicating systemic problems with the underlying architecture. Section 5 draws conclusions and points out further work.

Being able to identify troublesome components in system test is important to determine what components

should be focused on the most. This study works with change data from RCS change history. This paper focuses on ranking components according to those most in need of attention. It also identifies which components tend to be changed together. Components that are likely to change when any number of other components are changed need special attention to ensure that the job is done right. Finally, change architectures are created that will visually display the relationships between components.

2 Background

2.1 Identifying and Analyzing Change-prone Components

It is important to know which software components are stable versus those which repeatedly need corrective maintenance. The latter components tend to become worse as they evolve over releases, due to the addition of new functionality with increasing complexity and out-of-date documentation of the system. Over time these problems can become very costly.

Ohlsson et al. [16, 15] combine prediction of fault-prone components with analysis of decay indicators. It ranks components based on the number of defects in which a component plays a role. Corrective maintenance measures are analysed to track changes in the components over successive releases.

According to Graves et al. [9], software change history is useful in predicting the number of future faults. Their model measures the fault incidence of a module based on the sum of contributions from all forms of changes, with large and recent changes receiving most weight. Graves and Mockus [10] show that four types of variables are critical contributors to estimated change effort: the size of the change, the developer making the change, the purpose of the change, and the date the change was opened. The conclusions drawn from this study are important as they can be used in project management tools to identify modules of code which are too costly to change.

Tools and a sequence of visualizations and visual metaphors can help engineers understand and manage the software change process. In [14], Mockus et al. describe a web-based tool, which is used to automate the change measurement and analysis process. Version control and configuration management databases provide the main sources of information. Eick et al. [4] presents several useful and different ways to visualize the changes and categorize the changes made to components as adaptive, corrective, and perfective. The authors conclude there are significant pay-offs - both intellectual and economic - in understanding change well and managing it effectively.

2.2 Reverse Architecture

Reverse architecting is a specific type of reverse engineering. According to [12], a reverse engineering approach should consist of the following:

1. Extraction: This phase extracts information from source code, documentation, and documented system history (e. g. defect reports, change management data).
2. Abstraction: This phase abstracts the extracted information based on the objectives of the reverse engineering activity. Abstraction should distill the possibly very large amount of extracted information into a manageable amount.
3. Presentation: This phase transforms abstracted data into a representation that is conducive to the user.

Objectives for reverse architecting code drive what is extracted, how it is abstracted, and how it is presented. For example, if the objective is to reverse architect with the associated goal to re-engineer (let's say into an object oriented product), architecture extraction is likely based on identifying and abstracting implicit objects, abstract data types, and their instances [3, 8, 11, 21]. Other ways to look at reverse architecting a system include using state machine information [7], or release history [6]. CAESAR [6] uses the release history for a system to capture logical dependencies instead of syntactic dependencies by analyzing common change patterns for components. This allows identification of dependencies that would not have been discovered through source code analysis. This method could be seen as a combination of identification of problematic components and architectural recovery to identify architectural problems.

If one is interested in a high level fault architecture of the system, it is desirable not to extract too much information during phase 1, otherwise there is either too much information to abstract, or the information becomes overwhelming for large systems. In this regard, Krikhaar's approach is particularly attractive [13]. The approach consists of three steps:

1. Define and analyze the import relation (via #include statements) between files[13]. Each file is assigned to a subsystem (in effect creating a part-of relation).
2. Analyze the part-of hierarchy in more general terms (such as clustering levels of subsystems).
3. Analyze use relations at the code level. Examples include call-called by relationships, definition versus use of global or shared variables, constants and structures. Krikhaar [13] also abstracts use relations to higher levels of abstraction.

Within this general framework, there are many options to adapt it to a specific reverse architecting objective [5]. For example, the method proposed by Bowman

et al. [2] starts with identifying components as clusters of files. Import relations between components are defined through common authorship of files in the components (ownership relation). Use relationships are defined as call-called-by relationships (dependency relation) of functions in components. Bowman et al. [2] also includes an evaluation of how well ownership and dependency relationships model the conceptual relationship.

Von Mayrhauser et al. [20] combine the concepts of fault-prone analysis and reverse architecting to determine and analyze the fault-architecture of software across releases. The basic strategy to derive the fault architecture uses defect cohesion measures for components and defect coupling measures between components to assess how fault-prone components and component relationships are. These measures are used with thresholds to identify

- The most fault-prone components only (setting a threshold based on the defect cohesion measure);
- The most fault-prone components relationships (setting a threshold based on two defect coupling measures). Component relationships are fault-prone depending on how often a defect repair involved changes in files belonging to multiple components.

Stringfellow et al. [18] adapted the technique to highlight both the nature and magnitude of the architectural problem. Their approach distinguishes between single and multiple file changes related to a defect repair. Stringfellow et al. [18] also consider two reasons why a component can be fault-prone with respect to relationships:

1. The defect coupling measure is high for a particular pair of components.
2. None of the defect coupling measures is high, but there are a large number of them (the sum of the defect coupling measures are large).

Von Mayrhauser et al. [20] and Stringfellow et al. [18] produce a series of fault architecture diagrams, one for each release, and then aggregate them into a Cumulative Release Diagram and show the components that occur in at least one fault architecture diagram.

This paper proposes an adaptation based on the need to represent change relationships between components and the ability to focus on the most problematic parts of the architecture. In this study, the data consists of RCS files [19] from a large flight simulator. These files include the date and time of each check-in, the name of the person who made the change and the number of lines added and deleted from the file. The component files are organized into several directories with several subdirectories in these directories, every directory except the home directory may have code files in it. For the purposes of this paper, a component is defined as a set of files located in the same directory. A component, however, does not include files in the subdirectories found in its directory.

3 Approach

The approach to derive the change architecture from RCS change data consists of the following steps:

- Extract desired data from logs in RCS source files.
- Put all related changes into groups using different time intervals.
- Create change architectures by analyzing the number of lines of code (LOCs) changed.
- Aggregate the change architecture diagrams for comparison of different time intervals.

As sources files are checked in and out of RCS, a log is automatically inserted into the file. Each log in a file has a unique number and includes the date, time, and author of the change. A perl program traverses the application system's directory to search for the source files, open them, and extract information from the logs.

In order to perform any meaningful analysis on that data, all related changes must be put into the same group. Two assumptions are made to determine which changes are related. The first assumption is that all related changes are made by the same programmer. Secondly, it is assumed that related changes are checked in within a certain time interval.

RCS also contains data on how many LOCs are added and deleted in each change. This data is used in two ways:

- To rank the change-prone components, in terms of local changes. The components with the highest number of lines of code changed in its files are the most likely to require changes. The change cohesion metric, for a component C , is defined as follows:

$$Coh(C) = \sum_{i=1}^n FLOC_i \quad (1)$$

where

n refers to the number of files in a component

$FLOC_i$ refers to the number of lines of code changed for each of the n individual files in component, C .

- To rank the change-prone relationship of components. The change coupling metric is defined, for a pair of components C_j and C_k , as:

$$Coupling(C_j; C_k) = \sum_{i=1}^m GLOC_i \quad (2)$$

where

m refers to the number of groups of related changes, where a group is a set of changes checked in to RCS by the same author within a time interval, and

$GLOC_i$ refers to the number of lines of code changed in each group of related changes that changed code in both components, C_j and C_k , $j \neq k$.

Once the measures are collected, change architecture diagrams are created, showing the components and relationships with the highest values, which are considered most change-prone. As in [18], the thresholds are set at 10% of the highest measures.

4 Results

The RCS history data in this study come from a large commercial flight simulation system consisting of over 800 KLOC of C, C++, and microcode. The system has 59 components, each of which consists of one to many files that are logically related. Each RCS file includes data regarding the number, date, time, and author of each change. In addition, the number of lines of code deleted and added for each change is available.

Changes that were checked into RCS within a given time interval are part of the same group. This study uses time intervals of 2 minutes and 30 minutes, respective to group changes. Figure 1 shows the change architecture diagram using a time interval of 2 minutes for grouping changes. Nodes represent components that are change-prone or in change-prone relationships. The diagram shows components whose change cohesion measures are in the top 10%, in terms of number of lines of code changed in files within that component. The numbers in the node indicate the number of lines of code changed. The diagrams show components whose change coupling measures are in approximately the top 10% of the relationships in terms of number of lines of code changed. The numbers labeling the edges indicate the number of lines of code changed in files in the components with the change relationship.

Figure 2 shows the change architecture diagram using a time interval of 30 minutes for grouping changes. It probably does not make sense to group changes too far apart, as they are most likely not related to fixing one particular problem.

Components S2/p, S3/s, S3/N/d, S3/w and S3/a are in both change architecture diagrams. No matter which time interval is used for grouping changes (2 or 30 minutes), these components are considered locally change-prone. Component S3/N/r is present only in the 2 minute diagram – it is change-prone, however, it is quickly modified.

The relationships between components S2/p and S2/S, S2/p and S2/I, and S2/S and S2/I exist in both the change architecture diagrams. Note that S1 does not appear to be change-prone within 2 minutes, but there is a big difference in the number of lines of code changed within 30 minutes.

It turns out the components with the highest number of relationships in the change architectures also have the highest number of includes. That is probably because changes are made to .h files and .cpp files most frequently. The relationship between components S2/p and S2/I/1 has a high change relationship measure, but does not have high number of includes: This still makes sense because com-

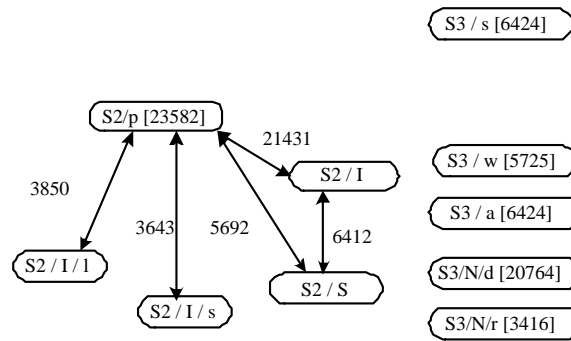


Figure 1. LOC Change Architecture Diagram using 2 minutes to group changes.

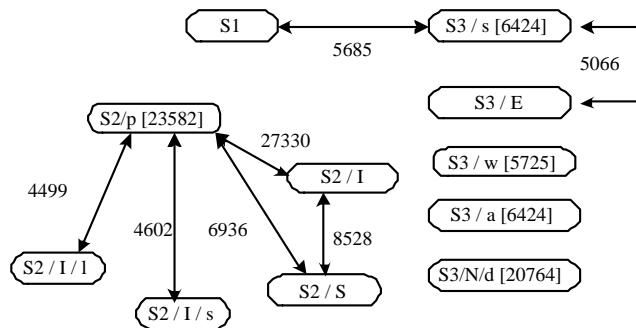


Figure 2. LOC Change Architecture Diagram using 30 minutes to group changes.

ponent S2/p deals with pages and component S2/I/I deals with labels, and labels appear on pages. Components S2/p and S2/S have no includes between them, yet it has the fourth highest change coupling measure. This relationship requires further investigation.

There exists a triangular relationship between components S2/p, S2/S and S2/I in the change architecture: Further inspection of code reveals that both S2/p and S2/S include 13 common .h files from S2/I. This is another change-prone relationship on which software developers might want to concentrate.

Comparison of change architectures that group changes occurring within intervals of 2 minutes and 30 minutes indicates that in the most change-prone relationships, 75-80% of code is changed within 2 minutes. While fewer lines of code are changed in 30 minutes, some changes take a longer time to check-in the change. The relationship between components S3/s and S3/E is not change-prone in the 2 minutes change architecture, but it is in the 30 minute change architecture. The components S3/s and S3/E do not include any files from each other. Why would a programmer work on two components that do not have any includes in between them? Inspection of the relationship between components S3/s and S3/E reveals that both include the same two files from S2/I. This triangular relationship is *not* visible in the change architecture.

5 Conclusions

There are a few change-prone relationships between components that do not include files from each other. Code inspection revealed that in a couple of these relationships, the components have files that include one or two .h files from a third component. This might indicate some coupling among different kinds of components, which software developers might want to investigate. The change architectures also show us that most code changes, when grouped by programmer and time, are quickly accomplished within two minutes. The number of lines of code changed does not increase dramatically when looking at changes grouped within 30 minutes.

Further work will look at grouping changes in 5, 7, 10, 15 minutes intervals to determine whether change-prone components and relationships are sensitive to the time interval chosen. In addition, changes to code occur for several reasons: corrective, adaptive and perfective maintenance. The RCS history data in this case study contain logs for each change indicating the reasons for the change. Focusing on components that are change-prone for corrective reasons may be a better indicator of code decay.

Additional work is required to compare the change architecture with other software architectures, such as those derived using use relationships and fault architectures, similar to [20, 18] to determine if similar architectures are created using different types of data.

References

- [1] Allen R., and D. Garlan, "Formalizing Architectural Connection," *Proc. Intl. Conf. on Software Engineering*, IEEE Computer Society Press: Los Alamitos CA, 1994, pp. 71-80.
- [2] Bowman I., and R.C. Holt, "Software Architecture Recovery Using Conway's Law," *Proc. CASCON'98*, November-December 1998, Mississauga, Ontario, Canada, pp. 123-133.
- [3] Canfora G., A. Cimitile, M. Munro, and C. Taylor, "Extracting abstract data types from C programs: a case study," *Proc. Intl. Conf. on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1993, pp. 200-209.
- [4] Eick, S., T. Graves, A. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes," *IEEE Trans. on Software Engineering*, **28** 4, April 2002, pp. 396-412.
- [5] Feijs L., R. Krikhaar, and R. van Ommering, "A relational approach to software architecture analysis," *Software Practice and Experience*, 1998 **28**(4), pp. 371-400.
- [6] Gall H., K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," *Proc. Intl. Conf. on Software Maintenance*, IEEE Computer Society Press: Los Alamitos CA, 1998, pp. 190-198.
- [7] Gall H., M. Jazayeri, R. Kloesch, W. Lugmayr and G. Trausmuth, "Architecture recovery in ARES," *Proc. Second Intl. Software Architecture Workshop (ISAW-2)*, ACM Press: New York NY, 1996, pp. 111-115.
- [8] Girard J., and R. Koschke, "Finding components in a hierarchy of modules: a step towards architectural understanding," *Proc. Intl. Conf. on Software Maintenance*, IEEE Computer Society Press: Los Alamitos CA, 1997, pp. 58-65.
- [9] Graves, T., A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. on Software Engineering*, **26** 7, July 2000, pp. 653-661.
- [10] Graves, T, and A. Mockus, "Inferring Change Effort from Configuration Management Databases," *Proc. of the 5th Intl. Symposium on Software Metrics*, March, 1998, Bethesda, MD, pp. 267-273.
- [11] Harris D., A. Yeh, H. Reubenstein, "Recognizers for extracting architectural features from source code," *Proc. Second Working Conf. on Reverse Engineering*, IEEE Computer Society Press: Los Alamitos CA, 1995, pp. 252-261.
- [12] Tilley S., K. Wong, M. Storey, and H. Muller, "Programmable reverse engineering," *Intl. J. of Software Engineering and Knowledge Engineering*, 1994, **4**(4), pp. 501-520.
- [13] Krikhaar R., "Reverse architecting approach for complex systems," *Proc. Intl. Conf. on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1997, pp. 1-11.
- [14] Mockus, A., S. Eick, T. Graves, and A. Karr, "On Measurement and Analysis of Software Changes", Technical Report BL011 3590-990401-06TM, Bell Laboratories, Lucent Technologies, 1999.
- [15] Ohlsson M., A. von Mayrhauser, B. McGuire, and C. Wohlin, "Code decay analysis of legacy software through successive releases," *Proc. IEEE Aerospace Conf.*, IEEE Press, Piscataway NJ.
- [16] Ohlsson M., and C. Wohlin, "Identification of green, yellow and red legacy components," *Proc. Intl. Conf. on Software Maintenance*, IEEE Computer Society Press: Los Alamitos CA, 1998, pp. 6-15.
- [17] Shaw M., and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall: Upper Saddle River NJ, 1996.
- [18] Stringfellow, C. and A. Andrews, "Deriving a Fault Architecture to Guide Testing," *Software Quality Journal*, **10**(4), December, 2002, pp. 299-330.
- [19] Tichy, Walter F., "RCS-A System for Version Control, Software-Practice & Experience."
- [20] Von Mayrhauser A., J. Wang, M. Ohlsson, and C. Wohlin, "Deriving a fault architecture from defect history," *Proc. Intl. Symposium on Software Reliability Engineering*, IEEE Computer Society Press: Los Alamitos CA, 1999, pp. 295-303.
- [21] Yeh A., D. Harris, and H. Reubenstein, "Recovering abstract datatypes and object instances from a conventional procedural language," *Proc. Second Working Conf. on Reverse Engineering*, IEEE Computer Society Press: Los Alamitos CA, 1995, pp. 227-236.
- [22] Younger E., Z. Luo, K. Bennett, and T. Bull, "Reverse engineering concurrent programs using formal modeling and analysis," *Proc. Intl. Conf. on Software Maintenance*, IEEE Computer Society Press: Los Alamitos CA, 1996, pp. 255-264.