

Comparison of Software Architecture Reverse Engineering Methods

C. Stringfellow, C. D. Amory, D. Potnuri

Department of Computer Science, Midwestern State University

Wichita Falls, TX 76308, USA

+1 940-397-4578(ph) +1 940-397-4442(fax)

A. Andrews

Electrical Engineering and Computer Science Department, Washington State University

Pullman, WA 99164, USA

+1 509-335-8656(ph) +1 509-335-3818(fax)

M. Georg

Department of Computer Science and Engineering, Washington University at St. Louis

St. Louis, MO 63130, USA

Abstract

Problems related to interactions between components is a sign of problems with the software architecture of the system and are often costly to fix. Thus it is very desirable to identify potential architectural problems and track them across releases to see whether some relationships between components are repeatedly change-prone.

This paper shows a study of combining two technologies for software architecture: architecture recovery and change dependency analysis based on version control information. More specifically, it describes a reverse engineering method to derive a change architecture from Revision Control System (RCS) change history. It compares this method to other reverse engineering methods used to derive software architectures using other types of data. These techniques are illustrated in a case study on a large commercial system consisting of over 800 KLOC of C, C++, and microcode. The results show identifiable problems with a subset of the components and relationships between them, indicating systemic problems with the underlying architecture.

Key words: software architecture, reverse engineering, maintainability

Preprint submitted to Elsevier Science

1 Introduction

As a system evolves and goes through a number of maintenance releases [13], it naturally inherits functionality and characteristics from previous releases [25,26], and becomes a legacy system. As new functionality and features are added, there is a tendency towards increased complexity. This may impact the system's maintainability. This makes it 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. (Components are defined as a collection of files that are logically or physically related.) Increasing difficulty in further evolution and maintenance is a sign of code decay.

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. Components that are difficult to maintain are certainly candidates for re-engineering efforts.

Early decay identification is desirable so that steps can be taken to prevent further degradation. The question is how to identify this decay and what to do to stop it. 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. Change reports usually contain information about the nature, time and author of the change. A change can modify 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 [1,35]: Software architecture consists of a description of components and their relationships and interactions, both statically and behaviorally [35]. Problems and possible architectural decay can be spotted via changes related

Email addresses: `catherine.stringfellow@mwsu.edu`, `damory@hotmail.com`, `d.potnuri@yahoo.com` (C. Stringfellow, C. D. Amory, D. Potnuri), `aandrews@eecs.wsu.edu` (A. Andrews), `mgeorg@arl.wustl.edu` (M. Georg).

to components and interactions of the components. A software architecture decay analysis technique must identify and highlight problematic components and relationships.

Code decay can be due to the deterioration of a single component. In this case it manifests itself in repeated and increasing problems that are local to the component. A second type of code decay is due to repeated problems that become increasingly difficult to fix and are related to interactions between components, that is, components are repeatedly fault-prone or 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.

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. [32] 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 Mayrhauser et al. [39] 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 components 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 cohesion and coupling measures.

This paper investigates ways to

- identify components and relationships between components that are change-prone or fault-prone either through an existing software architecture document in conjunction with change history, or, in its absence, through reverse architecting techniques [4,9,10,14,18,24,40,42]. This paper tries to deal with the latter situation: an obsolete or missing software architecture document and the need for some reverse architecture effort. 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 cohesion and coupling for the components and component relationships identified.
- set thresholds for these measures to determine the most problematic 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.
- verify the change architecture by comparing it to the system architecture derived using reverse architecting techniques that involve #include statements similar to the method described in [9,14,18,24], as well as the fault architecture [39,36].

Section 2 reports on existing work related to identifying (repeatedly) fault-prone components. It also summarizes existing classes of reverse architecting approaches. Few researchers have tried to combine the two [11,12]. This paper uses two steps rather than a combination, because the reverse architecting approach is used to build both a change architecture and a reverse architecture. Section 3 details the approaches used. Section 4 reports on their 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 is important to determine what components should be focused on the most in system test or maintenance. This study uses data extracted from an RCS version control system and its files. This paper ranks components, with the intention of identifying those most in need of attention. It also identifies which components tend to be changed together, whether for fixes or for other reasons. Components that are likely to change and usually involve changes to a number of other components need special attention to ensure that the job is done right. The methods described in this paper will identify components that have many relationships between files within their own component and with other components. The architecture diagrams created will visually display those problematic components.

2 Background

2.1 Tracking and Predicting Fault-prone Components

It is important to know which software components are stable versus those which repeatedly need corrective maintenance, because of decay. Decaying components become worse as they evolve over releases. Software may decay

due to adding new functionality with increasing complexity as a result of poor documentation of the system. Over time decay can become very costly. Therefore it is necessary to track the evolution of systems and to analyze causes for decay.

Ash et al. [2] provide mechanisms to track fault-prone components across releases. Schneidewind [34], Khoshgoftaar et al. [21] provide methods to predict whether a component will be fault-prone. Khoshgoftaar et al. [22] present an algorithm using a classification tree, which represents an abstract tree of decision rules to aid in classifying fault-prone models. Their results show that the classification-tree modeling technique is robust for building software quality models. They also show that principal component analysis can improve the classification-tree model to classify fault-prone modules.

Ohlsson et al. [32,31] 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. The ranks and changes in ranks are used to classify components as green, yellow and red (GYR) over a series of releases. Corrective maintenance measures are analysed via Principal Components Analysis (PCA) [15]. This helps to track changes in the components over successive releases.

According to Graves et al.[16], software change history is more useful in predicting number of future faults than product measurements of the code, such as number of lines of code and number of developers. They focus on identifying those aspects of the code and its change history that are most closely related to the number of faults that appear in modules of the code. Their most successful 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 [17] describe a methodology for historical analysis of the effort necessary for developers to make changes to software. The results of the study 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 (1) identify modules of code which are too costly to change, or (2) assess developers' expertise with different modules.

Ostrand and Weyuker [33] examine the relationship between fault-proneness and the age of the file, the relationship between a file's change status and its fault-proneness, and to what extent the system's maturity affects the fault density. They observe that the age of a file is not a good predictor of fault-proneness: as the product matured the overall system fault density decreased.

Their results show that testers need to put more effort on new files, rather than on files that already exist.

Tools and a sequence of visualizations and visual metaphors can help engineers understand and manage the software change process. In [29], Mockus et al. describe a web-based tool, which is used to automate the change measurement and analysis process. The change measures include change size, complexity, purpose and developer expertise. Version control and configuration management databases provide the main sources of information. Eick et al. [6] presents several useful metaphors: matrix views, cityscapes, charts, data sheets, and networks, which are then combined to form "perspectives". Different ways to visualize the changes and the components changed are also categorized. Changes are classified as adaptive, corrective, and perfective maintenance. The goal is to understand the relationship between dimensions (attributes associated with the software development process that cannot be controlled) and measures (the responses that the development organization wishes to understand and control). The authors conclude there are significant payoffs - 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 [19], 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 drives 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 [4,14,18,40]. Alternatively, if it can be assumed that the code embodies certain architectural cliches, an associated reverse architecting approach would include their recognition [10].

Other ways to look at reverse architecting a system include using state machine information [12], or release history [11]. CAESAR [11] uses the release history for a system. It tries 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. It requires data from many releases. 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. This makes the simpler approaches more appealing. In this regard, Krikhaar's approach is particularly attractive [24]. The approach consists of three steps:

- (1) Define and analyze the import relation (via `#include` statements) between files[24]. Each file is assigned to a subsystem (in effect creating a part-of relation). If two files in different subsystems have an import relationship, the two subsystems to which the files belong have one as well.
- (2) Analyze the part-of hierarchy in more general terms (such as clustering levels of subsystems). This includes defining the part-of relations at each level.
- (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. Analogous to the other steps, Krikhaar [24] 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 [9]. For example, the method proposed by Bowman et al. [3] fits into this framework: the reverse architecting starts with identifying components as clusters of files. Import relations between components are defined through common authorship of files in the components (ownership relation). Bowman et al. [3] also includes an evaluation of how well ownership and dependency relationships model the conceptual relationship.

2.3 Version history analysis

Several recent studies [3,5,7,8,27,30,37,41,44] have focused on version history analysis to identify components that may need special attention in some maintenance task. Fischer et al. [7,8] describe an approach for populating a release history database with data from bug tracking and version control systems to understand the evolution of a software system. They look at problem reports

and modification reports for software features to uncover hidden relationships between the features. Visualization of these relationships can aid in identifying areas of design erosion (or code decay). Ying et al. [41] and Zimmerman et al. [44] develop approaches that use association rule mining on a software configuration management system (CVS) to observe change patterns that involve multiple files. These patterns can guide software developers to other potential modifications that need to be made to other files. Mockus and Weiss [27] introduce methods that uses modification requests to identify tightly coupled work items as candidates for independent development, especially useful when developers are at different sites. Stringfellow et al. [37] use change history data to determine the level of coupling between components to identify potential code decay.

Most of these approaches require grouping changes that are made to code in some way. Changes may be grouped according to a common author and a small time interval for checking in the changed files [5,27,37]. Ying et al. [41] also require the same check-in comment. Changes may also be grouped according to check in times for files changed to fix the same defect or respond to the same problem report [8]. The time intervals used to group changes vary. Curbranic and Murphy [5] group check ins that occur within six hours, but indicate confidence in the grouping decreases after five minutes. Zimmerman et al. [44] propose using a time interval of 3 1/3 minutes. Most of these approaches are at the file level [5,7,8,37,41], while some are finer-grained [27,44].

This paper uses an adaption of the reverse architecture technique proposed in [9] using version control history to build a change architecture. A change group consists of file changes that are checked in by the same author within a certain time interval. Different time intervals are considered. This analysis is lifted to the component level to derive a change architecture. In addition, a "fault architecture" is derived taking into consideration changes made to files for purposes of fixing a common defect, the author of the change, and the check-in time of files changed. These architectures are analyzed and compared to a reverse architecture based on `#include` statements similar to [9].

3 Approach

A software developer needs to focus on the most problematic parts of the architecture in testing or maintenance, those that are most fault-prone and/or change-prone. This study derives a software architecture using three different reverse engineering techniques for comparison purposes. The first technique is based on static relationships - the `#include` relationships, as described in [9]. Other measures of function calls and global variables may also be used, however, this study chose `#includes`, because the other two techniques work on a

file level. The second technique is based on grouping changes to files within and between components [37]. The third technique groups changes based on a common defect they are meant to fix. The following sections describe these techniques in more detail.

3.1 LOC change architecture

Stringfellow et al. [37] describe an approach to derive change architectures from RCS history data. RCS files [38] 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. As source 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.

It is also assumed that components with the highest number of lines of code (LOC) changed in their files are the most likely to require changes, and are hence change-prone. RCS logs contain data on the number of LOCs added and deleted in each change. Stringfellow et al. [37] use this data in two ways:

- Change Cohesion Measure

To rank the change-prone components, in terms of local 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 .

- Change Coupling Measure:

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 [36], the thresholds are set at 10% of the highest measures.

3.2 Fault architecture

Von Mayrhauser et al. [39] 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. If the objective is to concentrate on the most problematic parts of the software architecture, 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).

The defect cohesion and coupling measures are defined in [36] as follows:

- Multi-file Defect Cohesion Measure:

Multi-file defect cohesion counts the pairwise defect relationships between files within the same component where those files were changed as part of a defect repair:

$$Co_{\langle C \rangle} = \sum_{i=1}^n C_{i \langle C \rangle} \quad (3)$$

where

$$C_{i \langle C \rangle} = \begin{cases} 1, & f_{d_i} = 1 \text{ and only 1 component is involved in fixing } d_i \\ \frac{f_{d_i}(f_{d_i}-1)}{2}, & f_{d_i} \geq 2 \end{cases}$$

and

f_{d_i} is the number of files in component C that had to be changed to fix defect d_i .

n is the number of defects that necessitated changes in component C .

This provides an indication of local fault-proneness. Unless only one file is changed, this measure will be much larger than the basic cohesion measure. It penalizes components that are only involved in a few defects, but where each defect repair involved multiple files.

The defect coupling measures are defined as follows:

- **Multi-file Defect Coupling Measure:** Two or more components are fault related, if their files had to be changed in the same defect repair (i.e. in order to correct a defect, files in all these components needed to be changed).

For any two components C_1 and C_2 , the defect coupling measure $Re_{\langle C_1, C_2 \rangle}$ is defined as:

$$Re_{\langle C_1, C_2 \rangle} = \sum_{i=1}^n C_{1_{d_i}} \times C_{2_{d_i}}, \quad C_1 \neq C_2 \quad (4)$$

where

$C_{1_{d_i}}$ and $C_{2_{d_i}}$ are the number of files in component C_1 and C_2 that were fixed in defect d_i

n is the number of defects whose fixes necessitated changes in components C_1 and C_2 .

- **Cumulative Defect Coupling Measure:** A component C can be fault-prone with respect to relationships if none of the individual defect coupling measures are high, but there are a large number of them (the sum of the defect coupling measure is large). In this case the defect coupling measure for a component C is defined as:

$$TR_C = \sum_{i=1}^m Re_{\langle C, C_i \rangle} \quad C \neq C_i \quad (5)$$

where

m is the number of components other than C

$Re_{\langle C, C_i \rangle}$ is the defect coupling measure between C and C_i .

The multi-file defect coupling measure emphasizes pairwise coupling related to code changes for defects involving a pair of components. The cumulative defect coupling measure is concerned with components in many fault relationships with two or more components.

Stringfellow et al. [36] determine defect cohesion and defect coupling measures for all components. Then they filter based on the defect coupling measure to focus on the most problematic relationships. A component is considered fault-prone in a release, if it is among the top 25% in terms of defect reports written against the component. The 25% threshold provides a manageable number of

problematic components for further analysis. Similarly, a threshold may distinguish between component relationships that are fault-prone and those that are not. The threshold is set to an order of magnitude less than (or 10% of) the maximum value of the defect coupling measure. Component relationships were fault-prone depending on how often a defect repair involved changes in files belonging to multiple components. When identifying fault-prone component relationships, the method only identifies components as having fault-prone relationships, if these relationships rank in the top 25%. This omits components in the fault architecture with fault relationships that rank lower, but which may still have a large number of them.

Stringfellow et al. [36] adapted the technique to highlight both the nature and magnitude of the architectural problem. They include components in the fault architecture if the number of fault relationships is within an order of magnitude of the number of fault relationships of the highest ranked component. The defect cohesion measure by Von Mayrhauser et al. [39] does not differentiate between defects that require modifying one file in a component, or dozens. The approach in Stringfellow et al. [36] distinguishes between single and multiple file changes related to a defect repair. This provides a ranking of components with respect to how locally fault-prone they are and this simplifies issues related to validation of the measures [20].

The fault relationship can be abstracted to the subsystem level: Two subsystems are related, if they contain components that are related. This represents Krikhaars lift operation [24]. Von Mayrhauser et al. [39] and Stringfellow et al. [36] apply this lifting method when going from the subsystem to the system level. Changes in patterns, or persistent fault relationships between components, across releases, are indicators of systemic problems between components and thus architecture.

Von Mayrhauser et al. [39] and Stringfellow et al. [36] also produce a series of fault architecture diagrams, one for each release. The fault architecture diagrams are aggregated into a Cumulative Release Diagram and show the components that occur in at least one fault architecture diagram [36].

This study works with data from RCS change history and does not include data from a defect tracking system. Therefore, to derive a "fault architecture", changes by the same author and for a time interval are grouped together and the changes are assumed to fix a common defect. This allows the fault architecture technique described in Stringfellow et al. [36] to be applied.

3.3 Comparing architectures

The approach to compare the architectures from process data consists of the following steps:

- Extract desired data from logs in RCS source files.
- Group all related changes and create an aggregate change architecture diagram for the different time intervals. Change-prone components and relationships are identified by analyzing the number of lines of code (LOCs) changed, as in [37].
- Create a fault architecture diagram by analyzing files changed to fix defects, as in [36].
- Create a reverse architecture diagram by analyzing `#include` statements in files, as in [24].
- Compare the architecture diagrams.

Relationships, either change-prone or fault-prone, that appear in change or fault architecture diagrams identify components that are becoming increasingly difficult to maintain over time. Change-prone or fault-prone relationships between components that do not have `#include` relationships, especially indicate an increased complexity in the system's architecture that affect future evolution and maintenance. This is a sign of decay.

The architecture techniques are applied in this study to a large commercial flight simulation system consisting of over 800 KLOC of C, C++, and microcode. The system has 58 components, each of which consists of one to 108 files that are logically related. Over half of the components contain 3 files or less. For the purposes of this paper, a component is defined as a set of files located in the same directory. The component files are organized into several directories with several subdirectories in these directories, every directory except the home directory contains files. Files in the subdirectories of a component directory are not considered part of the component. Figure 1 shows the directory structure of the system. Numbers in the nodes represent the number of header and source code files in the component.

4 Analysis of architectures

4.1 Reverse architecture

Figure 2 shows the reverse architecture diagram for the flight simulation system based on `#include` relationships. This figure focuses on subsystem S3 (a

subsystem with many .c and .cpp files). The nodes in the diagram represent the components. Edges represent a relationship between two components. The strength of the relationships between components in this diagram are indicated by the number of #includes. For example, file(s) in component s3/N include files from S1 - it might be that one file in S3/N includes 14 different files in S1, or 14 files in S3/N include one file in S1, or several files in S3/N include several files in S1. The layout for this figure also serves as a master for the other architecture diagrams for comparison purposes.

Most of the architectural dependencies are related to header files and corresponding source files. Many of the components in S3 include files in components S1 and S2/I. As it happens, S1 and S2/I contain many of the header files. Few components in Subsystem S3 include files from other components in S3, due to the fact that components in this subsystems tend to implement different features of the system, so little coupling is expected. Component S3/c does include a file from component S3/w. These components implement features for weapons and countermeasures, and one might expect that a countermeasure could involve a weapon. Component S2/S/s concerns scenarios, and contains both the definition and implementation of scenarios within the same component. Similarly, components S3/N/d, S3/w, S3/c, and S2/S/l contain header and corresponding source files.

Figure 3 focuses on the relationships between components in subsystems S2 and S3. For example, file(s) in S3/s includes file(s) from S2/p.

Figure 4 shows the #include architecture of the S2 subsystem locally. The shaded components in the diagram are also in Figure 2. This diagram shows the internal relationships between components of subsystem S2 and the S2 components that include files in the S1 subsystem.

The #include architecture diagrams in this study reveal a great many relationships in the system, too many to be understandable. The benefit of the #include architecture is mainly to help validate relationships found using other reverse architecture methods and to aid in the understanding of those relationships.

4.2 LOC change architecture

Each RCS file includes data regarding the number, date, time, author, and nature of each change. In addition, the number of lines of code deleted and added for each change is available. Files that were checked into RCS within a given time interval are considered part of the same group, and are assumed to be related to fixing a particular problem or implementing an additional feature.

Figure 5 shows the aggregate change architecture diagram using time intervals of 2 minutes, 7 minutes, 10 minutes and 30 minutes for purposes of grouping check-ins. It probably does not make sense to group files checked-in too far apart, as they are most likely not related to fixing one particular problem. The 5 minute interval resulted in the same change-prone relationships as the 2 minute interval, so that data is not presented.

A shaded node indicates a component is locally change-prone, that is, its grouped changes result in many lines of code changed in files within that component only. The numbers in brackets in these nodes are the change cohesion measures and indicate the number of lines of code changed in files within that component, hence they are a local change measure. Locally change-prone components are those whose change cohesion measures are in the top 10%. Focusing attention on these components in testing or maintenance should yield the most benefits. Available time and effort may dictate a different threshold, but this study follows the recommendation in [36] in setting the threshold.

Regardless of which time interval is used for grouping changes (2 or 30 minutes), components S2/p, S3/s, S3/N/d, S3/w and S3/a are considered locally change-prone. Component S3/N/r locally change-prone only in the 2 minute interval – it has very close check-in times for files with a large number of LOC changed. If changes are related by check-in times of more than 5 minutes apart, then S3/N/r is not one of the highest ranking change-prone components. The assumption, however, is that very close check-in times for files indicates that the changes are related.

Components with a high change coupling measure indicate a change-prone relationship with another component. The numbers labeling the edges indicate the number of lines of code changed in files in the components with the change relationship for 2, 7, 10, and 30 minute intervals. There were over 60 change relationships between components identified, but the change architecture diagram focuses on those that have the highest measures. It shows components whose change coupling measures are in approximately the top 10% of the relationships in terms of number of lines of code changed. If the change coupling measure was not in the top 10a time interval, it is indicated by a '-'. This indicates that for this time interval grouping that the relationship would not be considered one of the most change-prone.

The change-prone relationships between components S2/p and S2/s, S2/p and S2/I, and S2/s and S2/I occur for all time intervals. Note that S1 does not appear to be in a change-prone relationship for the two minute grouping, but there is a big difference in the number of lines of code changed in the components' file in the 7 minute group, and then not much more within 10 minutes and 30 minutes groups. In the change architecture diagrams that group changes that occurred within 30 minutes, the number of relationships

between components S2/I/1 and S2/p pop back into the top 10%.

Figure 5 illustrates that for this case study, grouping changes checked-in within two minute intervals will identify almost all change-prone components and relationships. Again, the assumption is that very close check-in times for changed files indicates that the changes are related. However, the change coupling measures show that if one considers the set of all relationships (not just those that are change-prone) for the time interval of 30 minutes, it is a superset containing all relationships grouped by smaller time intervals (and yet does not add anymore change-prone relationships). Assuming that changed files checked in as far apart as 30 minutes are related implies that either more files are changed and take longer to check in, indicating the degree of coupling is high or that some change being made to the components' files is more complicated. If the changes also involve many LOCs, these components will be categorized as change-prone. Focus should then be directed at these components as their changes take more effort.

In comparing the change architecture with the `#include` architecture, the components with the highest number of change relationships have the highest number of `#include` relationships. That is probably because changes are made to header files and their corresponding source code files (which belong to different components) most frequently. This supports work by Ying, et al.[41] which found very strong (and obvious) relationships in changes touching related `c` and `h` files. More interesting, components S2/p and S2/I/1 show a high change relationship measure, without a high number of `#includes`. In looking at the functionality of the components, this makes sense as component S2/p deals with pages and component S2/I/1 deals with labels, and labels appear on pages. Components S2/p and S2/S do not include files from each other, yet they show the fourth highest change coupling measure. (However, they both include a file belonging to component S2/I that contains only typedefs and defines.) 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.

When changes occurring within 10 minutes are grouped, S2/I/1 does not show up in a change-prone relationship. That means changes are quickly checked-in in less than 7 minutes or the changes take 30 minutes or more to check-in. Comparison of change architecture diagrams that group changes occurring within intervals of 2 minutes through 30 minutes found that in the most change-prone relationships, 75-80% of check-ins occur within 2 minutes. The relationship between components S3/s and S3/E is not change-prone in the 2 minutes change architecture diagram, but it is in the remaining change architecture diagrams. Note this relationship between components S3/s and S3/E concerns groups of

check-ins that are performed by the same programmer within 7 minutes. Of most interest, is the change-prone relationship between the components S3/s and S3/E, which do not include any files from each other. In addition, these two components seem to involve very different functionality (one relates to surfaces and the other to aircraft engine function). Why would a programmer work on two components that do not include files from one another? 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 diagram.

4.3 Fault architecture

The fault architecture methods employed by Stringfellow et al. [36] are applied to reverse engineer the software architecture for the same flight simulator system. Strictly speaking these are not fault architectures, since the data used is RCS change data, and not defect data. The techniques discussed in [36] assume that data relating to fault architecture are available.

Figure 6 shows components considered to be fault-prone and components in fault-prone relationships. Components' total relationship coupling measures (TRc) and their defect coupling relationship measures (Re) are indicated beside nodes and on edges. The inset in the figure lists the fault-prone components that are locally fault-prone, that is in fixing each fault multiple files within the component had to be changed. This figure is derived by grouping file check-ins in which changes fix a defect using a 15 minute time interval. In shorter time intervals, the fault-prone relationship between S2/I/s and S2/p did not exist. A time interval of 20 minutes up to 59 minutes shows one additional fault-prone relationship between components S2/S/s and S2/p. Otherwise, the only differences in the fault-architectures for different time intervals is an increase in the measures (and not the components).

The fault architecture diagram for this system is very similar to the change architecture diagram. This diagram is compared to the diagram in Figure 5. Components S1, S2/p, S3/s, S3/N/d, S3/w, S3/a, S2/I, and S2/S are not only change-prone; they are also fault-prone. Components S3/E and S2/I/s are considered change-prone, but not fault-prone. Components S3/h, S3/c, S2/S/s, and S3/N/c are fault-prone, but not change-prone.

Relationships that are both fault-prone and change-prone include those between components S1 and S3/s and S2/p and S2/I/s, as well as the triangular relationship between components S2/p, S2/I, and S2/S. Only one more relationship appears to be change-prone and that appears between S3/s and S3/E. The relationships between components S2/p and S2/S/s and S3/N/d

and S3/N/r are fault-prone, but not change-prone.

The change architecture diagram indicates the same problematic components and component relationships that fault-architecture diagram does. A few problematic components are identified by one method only. This shows that using either fault data or change data, one may reverse architect a system and produce similar architectures identifying the same problematic components. If, however, both fault and change data are available, thorough software developers may want to use both methods to pick up a few more problematic components.

5 Future work

Mockus and Votta[28] hypothesize that one can use the textual description of a software change to derive the reason for that change. Additional future work with change architecture will involve determining the change architecture based on the type of change. Their study also found strong relationships between the type and size of a change and the time required to carry it out. Future work may focus on changes that take more effort. The change architecture diagram shows us that most check-ins for code changes, when grouped by programmer and time, are quickly accomplished within two (to five) minutes. The number of lines of code changed does not increase dramatically when looking at changes grouped within 30 minutes. More interesting, perhaps, would be to focus on changes that occur within 30 minutes, but exclude all those that occur within 2 minutes. These components may be more highly coupled (involving many files) or the changes made to the components' files are more complicated causing some files to be checked in further apart in time.

6 Conclusions

The data used in this study are derived from RCS change history which, by its very nature, deals with changes at the file level. This means that any researcher seeking to find component-based changes or change relationships using RCS has to find some means of first determining the files in each component, then the various relationships between components. The RCS change history information for files must be lifted to components: This is not difficult to do with a few scripts, but a configuration management system that stores the history of components in addition to the files would be helpful.

The #include architecture diagrams in this study reveal a great many relationships in the system, too many to be understandable. The benefit of

the `#include` architecture is mainly to help validate relationships found using other reverse architecture methods and to aid in the understanding of those relationships. In this study, most of the change-prone relationships involve components in which one has a `.cpp` file and the other has the corresponding `.h` file. These are obvious strong relationships. The change architecture diagram and the fault architecture diagram also effectively reveal relationships between components that are not obvious by looking at the `#include` architecture diagrams. Both the change and fault architecture methods found the same triangular relationship, that is not so obvious. Code inspection revealed that in these relationships, the components have files that include one or two `.h` files from a third component. The change architecture method also found a relationship between two components implementing very different features. The software developers might want to investigate this unusual coupling.

The results show identifiable problems with a subset of the components and relationships between them, indicating systemic problems with the underlying architecture. By comparing the different software architectures reverse engineered using change or fault data, problematic components and the problematic relationships between them are validated. The change architecture and fault architecture, which focus on a certain percentage of change-prone or fault-prone components and their relationships will allow a software developer to focus attention on these components in testing and maintenance tasks.

References

- [1] R. Allen, D. Garlan, Formalizing Architectural Connection, Proceedings International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos CA, 1994, pp. 71-80.
- [2] D. Ash, J. Alderete, P. Oman, B. Lowther, Using Software Models to Track Code Health, Proceedings International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1994, pp. 154-160.
- [3] I. Bowman, R.C. Holt, Software Architecture Recovery Using Conway's Law, Proceedings CASCON'98, 1998, Mississauga, Ontario, Canada, pp. 123-133.
- [4] G. Canfora, A. Cimitile, M. Munro, C. Taylor, Extracting abstract data types from C programs: a case study, Proceedings International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1993, pp. 200-209.
- [5] D. Cubranic, G. Murphy, Hipikat: Recommending pertinent software development artifacts, Proceedings International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos CA, 2003, pp. 408-418.

- [6] S. Eick, T. Graves, A. Karr, A. Mockus, P. Schuster, Visualizing Software Changes, *IEEE Transactions on Software Engineering*, **28** 4, 2002, pp. 396–412.
- [7] M. Fischer, M. Pinzger, H. Gall, Analyzing and relating bug report data for feature tracking, *Proceedings 10th Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, 2003, pp. 90–101.
- [8] M. Fischer, M. Pinzger, H. Gall, Populating a release history database from version control and bug tracking systems, *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, 2003, pp. 23-33.
- [9] L. Feijs, R. Krikhaar, R. van Ommering, A relational approach to software architecture analysis, *Software Practice and Experience*, 1998 **28**(4), pp. 371–400.
- [10] R. Fiutem, P. Tonella, G. Antoniol, E. Merlo, A cliché-based environment to support architectural reverse engineering, *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, 1996, pp. 319–328.
- [11] H. Gall, K. Hajek, M. Jazayeri, Detection of logical coupling based on product release history, *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, 1998. pp. 190–198.
- [12] H. Gall, M. Jazayeri, R. Kloesch, W. Lugmayr G. Trausmuth, Architecture recovery in ARES, *Proceedings Second International Software Architecture Workshop*, ACM Press, New York NY, 1996, pp. 111–115.
- [13] D. Gefen, S. Schneberger, The non-homogeneous maintenance periods: a case study of software modifications, *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, 1996, pp. 134–141.
- [14] J. Girard, R. Koschke, Finding components in a hierarchy of modules: a step towards architectural understanding, *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, 1997, pp. 58–65.
- [15] R. Gorsuch, *Factor Analysis*, Second Edition, Laurence Erlbaum Associates, Hillsdale NJ, 1983.
- [16] T. Graves, A. Karr, J. Marron, H. Siy, Predicting fault incidence using software change history, *IEEE Transactions on Software Engineering*, **26** 7, 2000, pp. 653–661.
- [17] T. Graves, A. Mockus, Inferring Change Effort from Configuration Management Databases, *Proceedings of the 5th International Symposium on Software Metrics*, Bethesda, MD, 1998, pp. 267–273.

- [18] D. Harris, A. Yeh, H. Reubenstein, Recognizers for extracting architectural features from source code, Proceedings Second Working Conference on Reverse Engineering, IEEE Computer Society Press, Los Alamitos CA, 1995, pp. 252–261.
- [19] S. Tilley, K. Wong, M. Storey, H. Muller, Programmable reverse engineering, International Journal of Software Engineering and Knowledge Engineering, 4(4), 1994, pp. 501–520.
- [20] B. Kitchenham, S. Pfleeger, N. Fenton, Towards a Framework for Software Measurement Validation, IEEE Transactions on Software Engineering, 21(12) 1995, pp. 929–944.
- [21] T. Khoshgoftaar, R. Szabo, Improving code churn predictions during the system test and maintenance Phases, Proceedings International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1994, pp. 58–66.
- [22] T. Khoshgoftaar, E. Allen, Improving Tree-Based Models of Software Quality with Principal Components Analysis, Proceedings of the 11th International Symposium on Software Reliability Engineering, San Jose, CA, 2000, pp. 198–209.
- [23] T. Khoshgoftaar, E. Allen, Modeling Fault-Prone Modules of Subsystems, Proceedings of the 11th International Symposium on Software Reliability Engineering, San Jose, CA, 2000, pp. 259–269.
- [24] R. Krikhaar, Reverse architecting approach for complex systems, Proceedings International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1997, pp. 1–11.
- [25] M. Lehman, L. Belady, Program Evolution: Process of Software Change, Academic Press, Austin TX, 1985.
- [26] N. Minsky, Controlling the evolution of large scale software systems, Proceedings International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1985, pp. 50–58.
- [27] A. Mockus, D. Weiss, Globalization by chunking: a quantitative approach, IEEE Software, 18 2, 2001, pp. 30–37.
- [28] A. Mockus, L. Votta, Identifying Reasons for Software Changes Using Historic Databases, Proceedings of the International Conference on Software Maintenance, San Jose CA, 2000, pp. 120–130.
- [29] A. Mockus, S. Eick, T. Graves, A. Karr, On Measurement and Analysis of Software Changes, Technical Report BL011 3590-990401-06TM, Bell Laboratories, Lucent Technologies, 1999.
- [30] A. Mockus, R. Fielding, J. Herbsleb, Two case studies of open source software development: Apache and Mozilla, ACM Transactions on Software Engineering, 11 3, 2002, pp. 309–346.

- [31] M. Ohlsson, A. von Mayrhauser, B. McGuire, C. Wohlin, Code decay analysis of legacy software through successive releases, Proceedings IEEE Aerospace Conference, IEEE Press, Piscataway NJ, 1999.
- [32] M. Ohlsson, C. Wohlin, Identification of green, yellow and red legacy components, Proceedings International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1998, pp. 6–15.
- [33] T. Ostrand, E. Weyuker, Identifying fault-prone files in large software systems, Proceedings of the 7th IASTED International Conference on Software Engineering and Applications, Marina Del Rey, CA, 2003, pp. 228–233.
- [34] N. Schneidewind, Software metrics model for quality control,” Proceedings International Symposium of Software Metrics, IEEE Computer Society Press, Los Alamitos CA, 1997, pp. 127–136.
- [35] M. Shaw, Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, Upper Saddle River NJ, 1996.
- [36] C. Stringfellow, A. Andrews, Deriving a Fault Architecture to Guide Testing, Software Quality Journal, **10**(4), 2002, pp. 299–330.
- [37] C. Stringfellow, C.D. Amory, D. Potnuri, M. Georg, and A. Andrews, Deriving Change Architectures from RCS History, Proceedings of the Eighth IASTED International Conference on Software Engineering and Applications, Cambridge, MA, 2004, pp. 210–215.
- [38] W.F. Tichy, RCS—A System for Version Control, Software—Practice & Experience, 15, 7 1985, pp. 637–654
- [39] A. Von Mayrhauser, J. Wang, M. Ohlsson, C. Wohlin, Deriving a fault architecture from defect history, Proceedings International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos CA, 1999, pp. 295–303.
- [40] A. Yeh, D. Harris, H. Reubenstein, Recovering abstract datatypes and object instances from a conventional procedural language, Proceedings Second Working Conference on Reverse Engineering, IEEE Computer Society Press, Los Alamitos CA, 1995, pp. 227–236.
- [41] A. Ying, G. Murphy, Predicting source code changes by mining change history, IEEE Transactions on Software Engineering, **30**(9), 2004, pp. 574–586.
- [42] E. Younger, Z. Luo, K. Bennett, T. Bull, Reverse engineering concurrent programs using formal modeling and analysis, Proceedings International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1996, pp. 255–264.
- [43] L. Zhang, H. Mei, H. Zhu, A configuration management system supporting component-based software development, Proceedings International Computer Software and Applications Conference, IEEE Computer Society Press, Chicago, IL, 2001, pp. 25–30.

- [44] T. Zimmerman, P. Weisgerber, S. Diehl, A. Zeller, Mining version histories to guide software changes, Proceedings International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos CA, 2004, pp. 563–572.

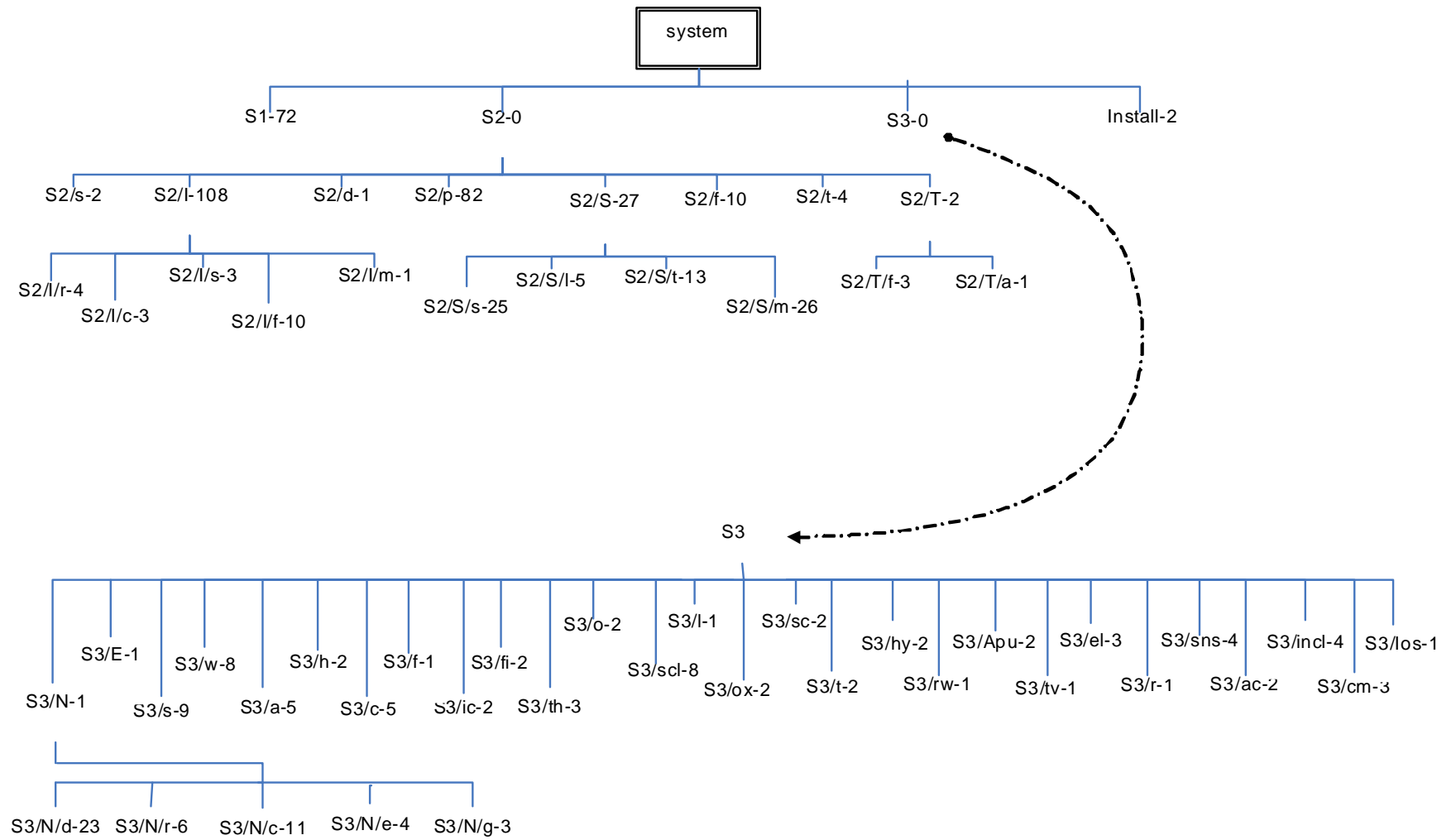


Fig. 1. Directory Structure.

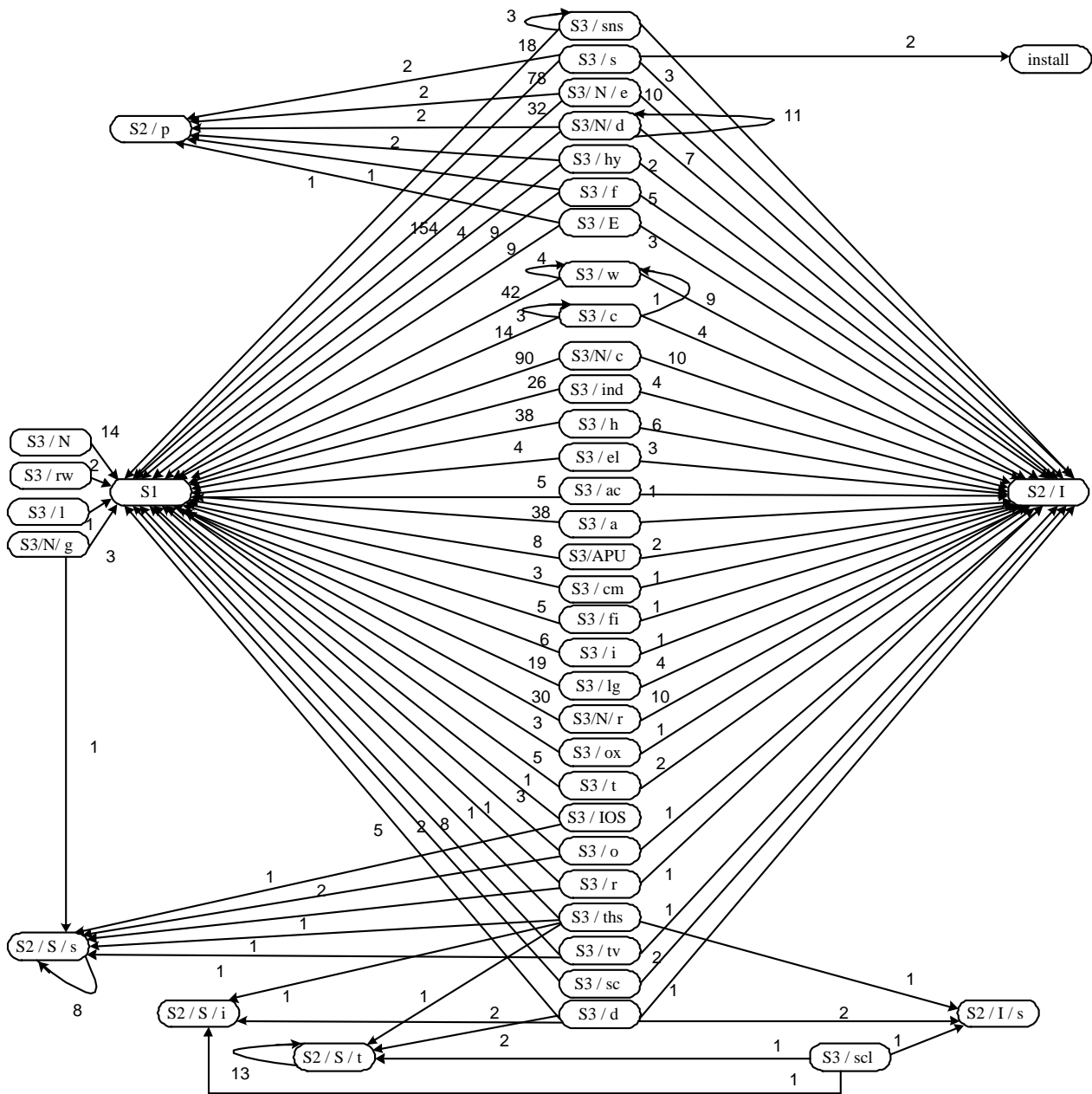


Fig. 2. #Include architecture.

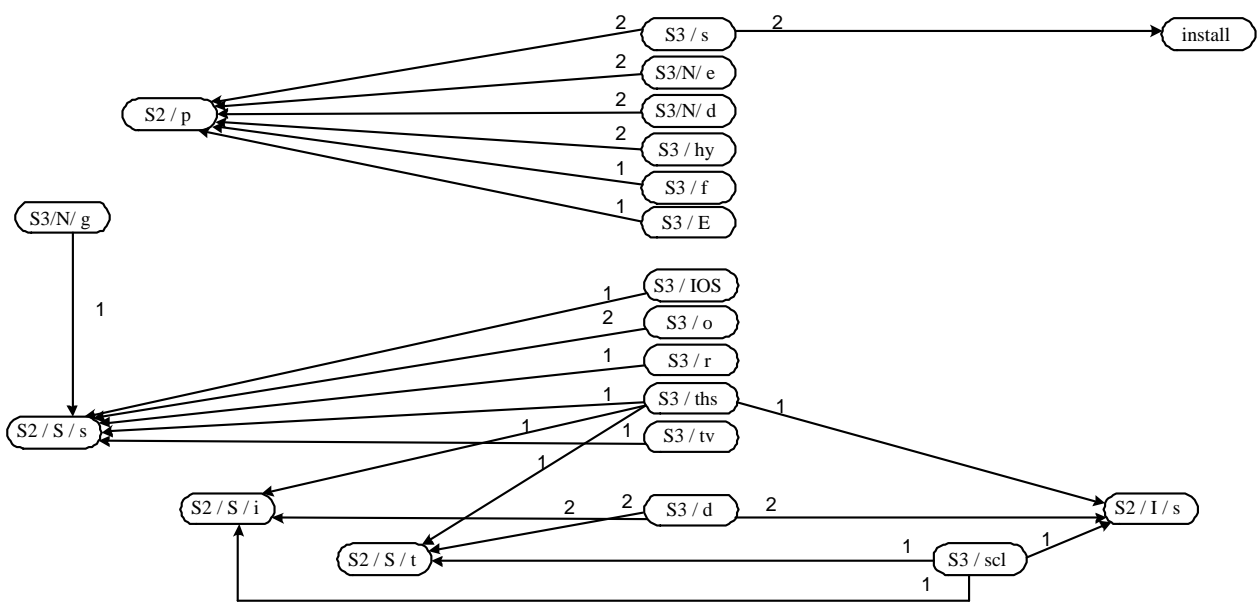


Fig. 3. #Include architecture for subsystems S2 and S3.

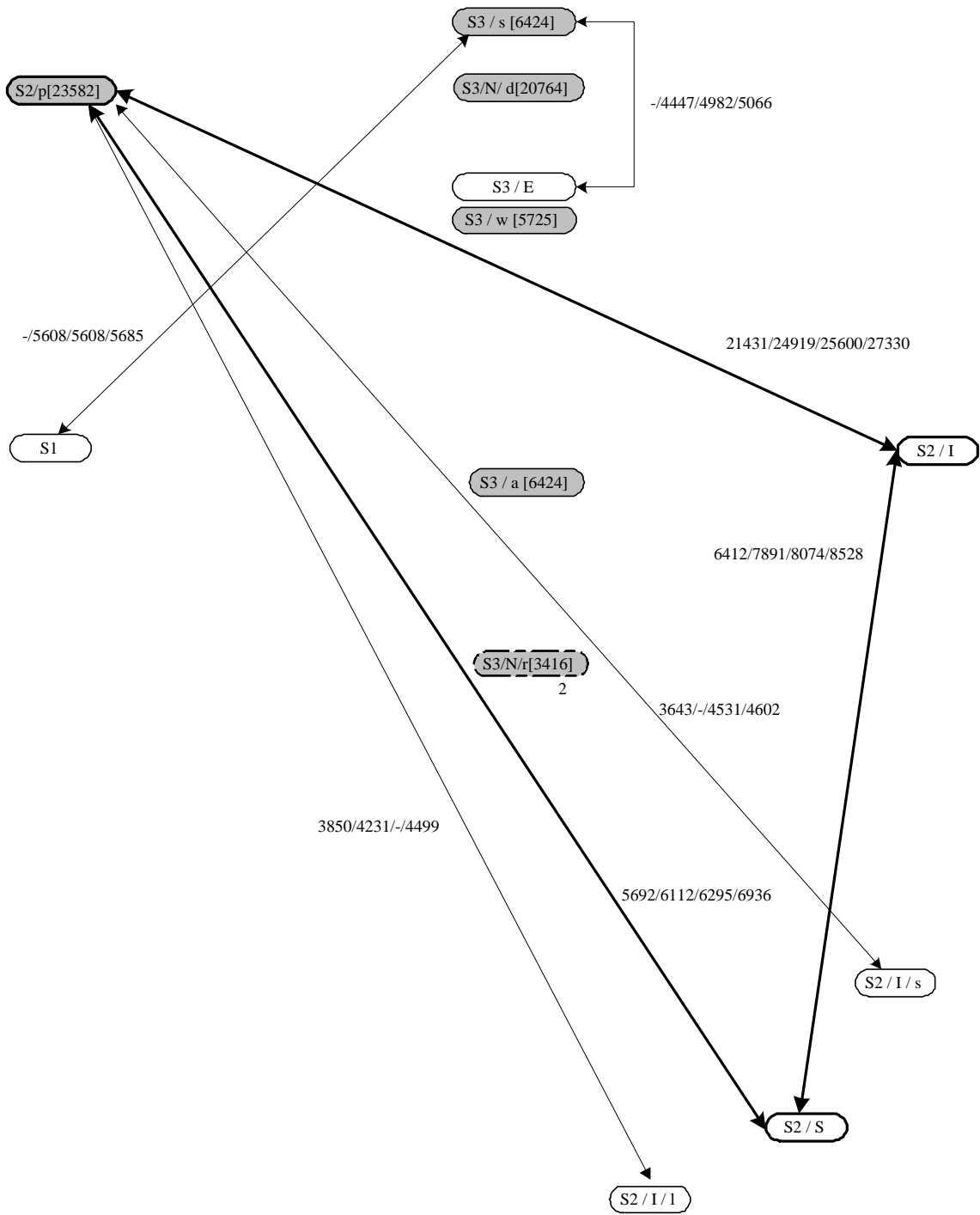


Fig. 5. Aggregate LOC Change Architecture Diagram.

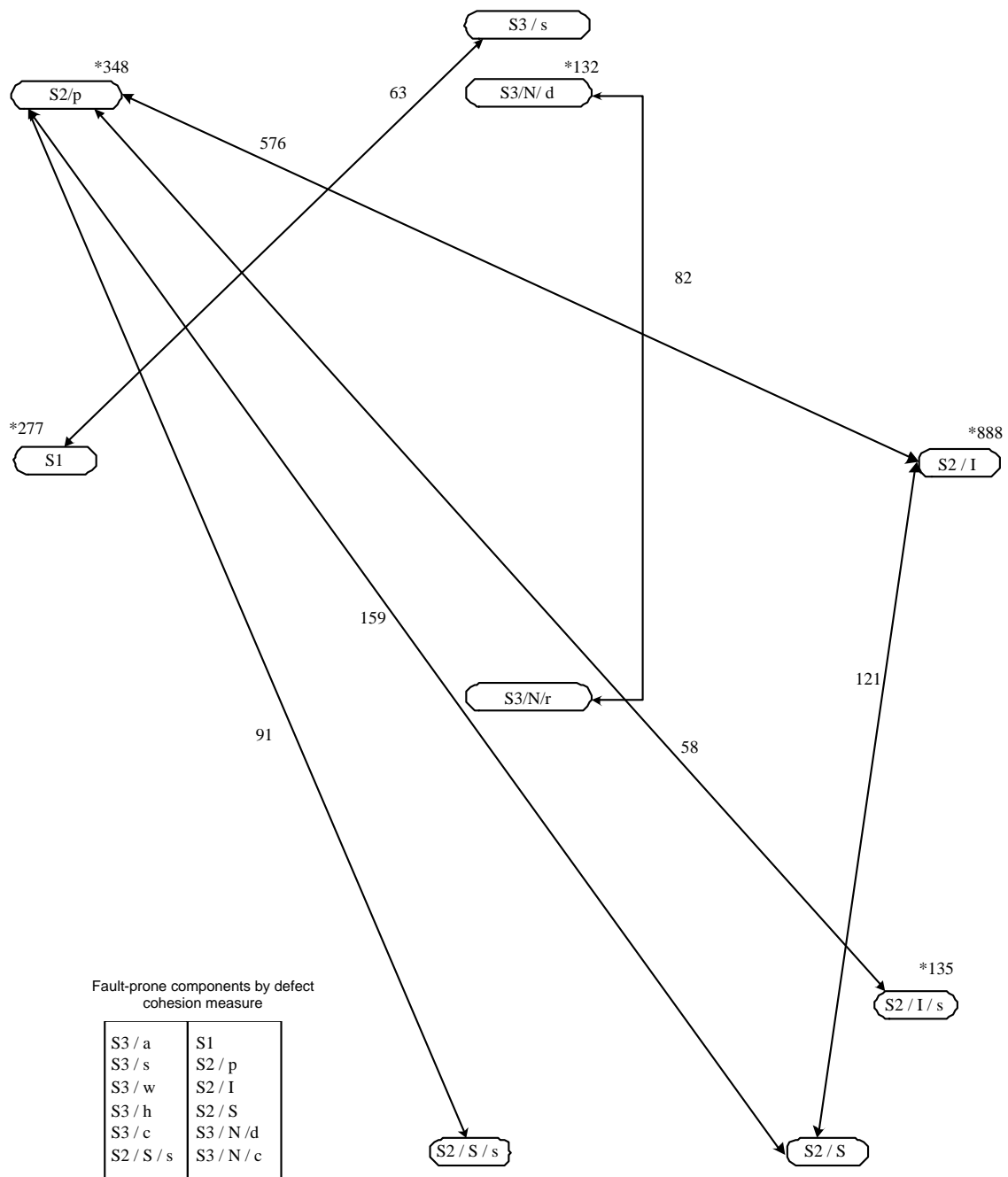


Fig. 6. Fault Architecture Diagram.