Deriving a Fault Architecture to Guide Testing

C. Stringfellow, A. von Mayrhauser*
Computer Science Department
Colorado State University
Fort Collins, CO 80523
avm,stringfe@cs.colostate.edu
970-491-7016(ph) 970-491-2466(fx)

Abstract

Defect analysis of software components can be used to guide testing with the goal of focusing on parts of the software that were fault-prone in earlier releases or earlier life cycle phases, such as development. We replicate a study that adapted a reverse architecting technique using defect reports to derive fault architectures. Our case study uses defect data from three releases of a large medical record system to identify relationships among system components based on whether they are involved in the same defect report.

We investigate measures that assess the fault-proneness of components and component relationships. Component relationships are used to derive a fault architecture. The resulting fault architecture indicates what the most fault-prone relationships are in a release. We also apply the technique in a new way. Not only do we derive fault architectures for each release, we derive fault architectures for the development, system test and post release phases within each release. Comparing across releases makes it possible to see whether some components are repeatedly in fault-prone relationships. Comparing across phases makes it possible to see whether development fault architectures can be used to identify those parts of the software that need to be tested more. We validate our predictions using system test data from the same release. We also use the development and system test fault architectures to identify fault-prone components after release and validate our predictions using post release data.

1 Introduction

Early identification of fault-prone components and fault-prone component relationships is desirable so that steps can be taken to more thoroughly expose the nature of problems. This may involve more intensive testing of fault-prone components and relationships, or performing code decay analysis. Code decay analysis involves identifying code that increasingly becomes problematic over time and more difficult to maintain, and taking steps to prevent further degradation.

We are particularly interested in whether problems identified in earlier releases or earlier phases of development are indicators of problems in system test or post release. The rationale for such a relationship is that parts of the software with severe or systemic problems during development carry a higher risk of not being completely fixed at the start of system test, that they are more likely to exhibit long term problems, and that these problems are more likely to be severe. If this is the case, one can use defect reports from development to guide testing. Specifically, components that are fault-prone or are in fault-prone relationships during development should be system tested at the earliest possible time. This would shift higher defect intensities earlier in the test cycle, giving developers more time to fix remaining problems.

*corresponding author

A software architecture describes the components and their relationships and interactions. One can identify system components and the relationships between components through either an existing, up-to-date software architecture document, or in the absence of such a document, through reverse architecting techniques [3, 7] that analyze the code. The problem with a software architecture based on code, is that while it identifies the components and the relationships between components, it does not necessarily indicate those that will be the most problematic. This paper applies a reverse architecting approach adapted from [10, 14, 15] to derive "fault architectures." Relationships and interactions of components can be identified by defect reports and common codes fixes in response to defect reports. One can identify and highlight problematic components and relationships, and ignore components and component relationships that are not problematic. Identifying problematic parts of the system will enable system testers to focus testing on these parts of the system. Persistent problems involving multiple components over several releases may indicate the need for rearchitecting.

Defect analysis identifies both components and relationships between components that are problematic. A defect cohesion measure at the component level is an indicator of problems local to the component, while a defect coupling measure between two components is an indicator of relationship problems between components [15]. High values in either are undesirable, indicating problems. The problems they indicate are of different types. High defect cohesion measures identify components that have problems locally, that is, they have internal problems. High defect coupling measures identify relationships between components that are broken.

Ohlsson et al. [10] use a simple defect cohesion measure that looks at the number of defects reported for a component to identify the most fault-prone components across successive releases and a defect coupling measure that uses the number of common code fixes for the same defect report to identify the most fault-prone relationships between components. In [15], variants of the cohesion and coupling measures are provided to more clearly distinguish defect fixes that involve single versus multiple file changes in a component. These measures are more sensitive to the number of files that had to be changed in each component in order to fix a defect. If the objective is to focus on the most problematic parts of the software architecture, these measures may be used with filters.

This paper

- investigates the two defect cohesion measures to identify fault-prone components.
- investigates the two defect coupling measures to identify components that have many fault relationships.
- proposes and investigates a method of setting the threshold based on order of magnitude, that is within 10% of the largest measure.
- applies the fault architecture technique to analyze defect reports of three releases of a large medical record system.
- performs across-release and within-release analysis of fault-prone components.

Section 2 presents some background and an existing approach that has been used to identify components that are fault-prone or are in fault-prone relationships. Section 3 presents the approach used in this paper. Section 4 presents the results of applying this approach in a case study using defect reports from three releases of a large medical record system. Section 5 draws conclusions.

2 Background

2.1 Identifying Fault-Prone Components

Much of the research in the area of defect analysis on software components has focused on identifying fault-prone components [5, 6, 8, 9, 11] or predicting the number of defects remaining in components [2, 16, 17] based on their characteristics. It is useful to know which components are fault-prone during prior release or in earlier life cycle activities, as these components should be tested more intensely. Data from previous releases can be used, but if this data is not available, data from earlier life cycle phases may be helpful.

Ash et al. [1] provide a method to track fault-prone components across releases. Schneidewind [11], Khoshgoftaar et al. [5] provide methods to predict whether a component will be fault-prone. Other methods [9, 10, 14] combine prediction of fault-prone components with code decay analysis - looking at relationships between components. These techniques rank components based on the number of defects in which a component plays a role. The ranks and changes in ranks over successive releases are used to classify components as green, yellow or red (GYR). This paper is more concerned with fault-prone component relationships, rather than components alone.

2.2 Reverse Architecture

Reverse architecting is a type of reverse engineering. Tilley et al. [13] describe an approach to reverse engineering that is used in aiding program understanding during software evolution. The process of reverse engineering identifies system components and their dependencies and generates abstractions to make them more understandable. This involves extracting artifacts from the source code (or in our case, the defect data) and representing them in a manageable form so the system's structural and functional characteristics can be analyzed.

A reverse engineering approach should consist of the following phases [13]:

1. Extraction Phase: This phase extracts information from sources such as source code, documentation, and documented system history (e.g., defect reports and change management data).

2. Abstraction Phase: The phase abstracts the extracted information based on the objectives of the reverse engineering activity. The potentially very large amount of extracted information is distilled into a manageable amount.

3. Presentation Phase: This phase presents the abstracted information into a representation that is understandable to the user.

The purpose of the reverse architecting activity drives what information is extracted, how it is abstracted, and presented. If we are interested in a high level architecture of the system, then we would not want to extract too much information during phase 1, otherwise there would be too much information to abstract.

By applying reverse engineering techniques, Krikhaar [7] derives an architectural model for several complex systems using metrics that measure import relations (e.g. #include statements) and use relations (e.g. call statements, type, constant and variable usage). The architecture is a description of the system and its components, as well as their relationships. The main goal is to create a representation of the system at a higher level of abstraction to provide a better understanding of the architecture to more easily assess and identify parts of the system that may require maintenance or enhancement.

Reverse architecting concerns activities that make software architectures explicit using reverse engineering techniques. Reverse architecting follows the reverse engineering phases. Depending on the required models, the appropriate steps are executed. The approach consists of three steps:

1. Extract the import relations from files, which are assigned to subsystems by use of a directory structure (creating part-of relationships). Import relations are then derived at the subsystem level as follows: If two files in different subsystems have an import relationship, the two subsystems to which the files belong have one as well. Results are then presented.

2. Analyze the part-of hierarchy for the system (e.g. files, clusters, subsystems) in general. The part-of relations are extracted at file level and derived for the various levels. The results are presented at various different levels of abstraction.

3. Extract and analyze use relations at the code level. Use relationships include not only import statements, but call and called-by relationships, global and shared variables, constants and structures. These and other use relations are best extracted using source code parsers. Analogous to part-of relations, the use relations are abstracted for various levels and the results are presented.

There are many ways to adapt this framework to a specific reverse engineering architecting objective, in our case, identifying fault-prone relationships between components.

Feijs et al. [3] describe a relational approach to support the analysis of software architectures. The relational approach supports many techniques that apply to analysis and structuring tasks, including:

- Lifting. Import and use relations at a higher level are obtained by union of import and use relations at a lower level according to the part-of relation.

- Checking rules. For example, typically each .c file in a C program must import at least one .h file; and it is not allowed that an .h file import other .h files (unless they are libraries).

- Impact analysis that allows one to determine which files require re-test as the consequence of changing a component.

- Finding unused and unavailable components.

- Studying alternative structures.

- Identifying components in top and bottom layers.

- Checking for cycles in uses relationships. This allows one to check for dependencies in software layers. For example, higher levels may use lower layers, but lower layers may not use higher levels.

The extraction-abstraction-presentation model is useful in reverse engineering. It allows visualization of the architecture at the highest level, while making sure that the high level views and the real system are related in an accurate way. This makes it easier to perform improvement activities. At any point in time, it is possible to see and correct the evolving system's structure.

Gall et al. [4] use a reverse architecting technique to identify the logical coupling of modules across releases. Their work differs from other work in that they do not focus on source code metrics or design metrics. Their technique builds an architecture based on logical coupling using change data rather than syntactical coupling, which is usually based on code or design. Modules are logically coupled if they have identical change behavior during software development. The architecture can be easily described using graphs where nodes represent subsystems and weighted edges represent the amount of coupling. Determination of interrelationships among modules aids in identifying modules that should undergo restructuring, re-engineering or in our case more testing. The advantage of this technique is that it does not require analyzing possibly large amounts of code, but instead analyzes release change data which is more manageable and usually available.

2.3 Fault Architecture

One can build a fault architecture in two ways:

1. Use an existing architecture and mask the components and relationships that are not fault-prone using the same measures of fault-proneness in [9, 14, 15].

2. If an existing architecture does not exist, reverse architecting techniques may identify it by using fault data extracted from the system.

Von Mayrhauser, et al. [10, 14] developed an adaptation of reverse architecting based on the need to represent defect relationships between components and the ability to focus on the most problematic parts of the architecture by quickly filtering out information.

Methods in [10, 9, 14] combine prediction of fault-prone components with code decay analysis. They look at relationships between components and identify the relationships that are fault-prone to indicate underlying systemic architecture problems. In [10], the authors identify software components that are fault-prone across releases to analyze code decay. Components are fault-prone based on the number of times they have been fixed. A small number of components were identified as problematic in terms of having a high number of fault relationships. How many fault relationships a component has with others is based on the number of other components involved in the same defect fix.

The approach [14] consists of the following steps :

1. Identify fault-prone components. Apply GYR [9] to identify fault-prone components over successive releases. Components with problems in several releases indicate possible code decay. The authors consider a component fault-prone, if it is among the top quartile in terms of defect reports in a given release. Several factors determine the threshold chosen. One of the goals is to provide a manageable amount of data.

2. Create a fault component directory structure. A component is a collection of files within the same subdirectory. The directory structure of the software provides the "part-of" relationship. Leaves in this structure are the fault-prone components identified by GYR in step 1. Internal nodes represent subsystems (subdirectories at higher levels in the directory structure) that contain fault-prone components. Since only fault-prone components are included, this does not represent the entire directory structure. This directory structure is referred to as a *Fault Component Directory Structure*.

3. Determine fault-prone relationships. If the number of fault relationships is quite high, narrow the number of components with fault relationships. This study set the threshold to the top 10% components. Fault relationships among these components that are above a threshold set to an order of magnitude (10%) less than the largest number of fault relationships are considered fault-prone.

4. Create a fault architecture diagram for each release. Abstract fault relationships to the next higher subsystem level. Subsystem levels are based on the level of depth in the *Fault Component Directory Structure*. Two subsystems are fault related, if they contain components that are fault related. (This represents Krikhaar's second step [7].) Nodes represent components. Arcs between two nodes show that components are fault-prone in their relationship. Weights on the arcs indicate the number of defect reports associated with two components or subsystems.

   The resulting fualt architectures indicate for each release the components that have the most fault relationships. Changes in patterns or persistent fault relationships between components across releases indicate systemic problems related to components and system architecture.

5. Aggregate the fault architecture diagrams into a cumulative release diagram. Nodes in the cumulative release diagram aggregate nodes that occur in at least one fault architecture diagram. Two nodes are related, if there is a fault relationship between corresponding nodes in at least one fault architecture diagram. Annotations on the edges indicate the releases in which the relationships had problems.

Results of the case study showed trends across releases. The study identified problematic component relationships. The cumulative release diagram identified persistent problems. Fault architecture diagrams and cumulative release diagrams draw attention to fault relationships that may need corrective maintenance, re-architecture, or more testing.

2.4 Defect Cohesion and Defect Coupling Measures

In design, cohesion is a measure of the internal consistency within components. It is a single component measure - an attribute of individual modules describing the extent to which the individual parts of a component perform the same task. In testing, defect cohesion refers to a measure that quantifies the number of individual parts of a component, e.g. files, that had to be changed to correct the same defect. Defect cohesion measures the fault-proneness internal to a component.

In design, coupling measures the degree of interaction between components. Two components are coupled when parts of one component use parts of another. In testing, defect coupling measures the degree of interaction of components in terms of the work needed to repair the same defect. Two components are related or "coupled" if some or all of their files are changed to repair a defect. Defect coupling measures the fault-proneness of the relationship between components.

Whereas high cohesion and low coupling desirable in design, low defect cohesion and low defect coupling are desirable in testing: They indicate a lower degree of fault-proneness internal to and between components.

Von Mayrhauser et al. [15] provide variants of the defect cohesion and defect coupling measures to more clearly distinguish defect fixes that involve single versus multiple file changes in a component. These measures are more sensitive to the number of files changed in each component to fix a given defect. They also present several options for abstracting the extracted fault-coupling relationships between components to the subsystem level.

2.4.1 Defect Cohesion Measures

The defect cohesion measures are defined as follows:

- Basic Defect Cohesion Measure:

  The basic defect cohesion measure for component C is defined as:

$$Co_{<C>} = d \tag{1}$$

  where

  d is the number of defect reports written against a component.

- Multi-file Defect Cohesion Measure:

  Merely counting defects does not differentiate between a defect in a component that requires modifying one file or many files. If we assume that a defect is more complex when its repair involves more of the component's files, a more detailed defect cohesion measure is needed.

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:

$$Co_{<C>} = \sum_{i=1}^{n} C_{i_{<C>}} \tag{2}$$

where

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

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.

2.4.2 Defect Coupling Measures

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 $Re_{<C_1,C_2>}$ is defined as:

  $$Re_{<C_1,C_2>} = \sum_{i=1}^{n} C_{1_{d_i}} \times C_{2_{d_i}}, \qquad C_1 \neq C_2 \tag{3}$$

  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:

  $$TR_C = \sum_{i=1}^{m} Re_{<C,C_i>} \qquad C \neq C_i \tag{4}$$

  where

  m is the number of components other than C

  $Re_{<C,C_i>}$ 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.

The primary use of these measures is to provide an ordinal scale to rank components. The exact values of the measures or the range of values is secondary. Ohlsson et al. [10] use the values to rank the components and then identify the top 25% in this rank order as fault-prone. The multiplicative nature of the multi-file defect cohesion and coupling measures magnifies individual differences among components. This accentuates differences between components that are close in measurement values for the defect cohesion and defect coupling measures and reduces ties in ranks.

2.5 Determining Fault-Prone Components and Fault-Prone Relationships

The basic strategy in [10, 14, 15] 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 filters to identify

- the most fault prone components only (setting a threshold based on the defect cohesion measure);
- the most fault prone component relationships (setting a threshold based on the defect coupling measure).

Von Mayrhauser et al. [14] consider a component fault-prone in a release if it is among the top 25% in terms of defect reports written against the component. In general, one would set the threshold based on available resources, quality, and objectives of the analysis (most problematic versus all components that have problems). The 25% threshold provided 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 was set to an order of magnitude less than (or 10% of) the maximum value for the defect coupling measure. Setting the threshold is a subjective decision and depends on the objectives of the investigation and the number of fault relationships.

Von Mayrhauser et al. [15] 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 C can be fault-prone with respect to relationships for two reasons:

1. The defect coupling measure is high for a particular pair of components $< C, C_i >$.
2. None of the individual defect coupling measures are high, but there are a large number of them (the cumulative defect coupling measure is large).

It is this second reason that prompted them to determine a threshold based on the sum of the defect coupling measures for a component. In their study, the threshold for including a component in the fault architecture is set as 10% of the highest TR_C measure. The threshold for including a fault relationship in the fault architecture is set at 10% of the highest $Re_{<C,C_i>}$.

These two thresholds identify the components and component relationships that provide the lowest level of the *Fault Architecture*. The fault architecture may have components with a high TR_C measure, but a low $Re_{<C,C_i>}$ measure. In this case, the component has no fault-prone relationships, to denote situation 2 above.

3 Approach

In order to validate that an assessment tool is useful, it is important to empirically evaluate whether it works on more than one project. We chose a very different project (medical record system) from that used in a prior case study (system software) [14, 15]. We add to the study in [14, 15] by applying the approach to a new data set. In addition, rather than just applying one defect cohesion measure and one defect coupling measure, the approach applies all measures of Section 2.4 so as to compare how results might differ. Unlike the prior case study, which only used post-release data, this study uses defect data from development, system test and post-release.

3.1 Release Analysis

The steps in the approach are:

1. Determine both defect cohesion measures of Section 2.4 and identify fault-prone cmoponents.

   Rather than choose between the two defect cohesion measures described in [10, 15], our approach uses both to identify problematic components that should be considered for more intense testing or for rearchitecting.

   This study considers a component, C, fault-prone, if

   $$Co_C \geq 0.1 * Co_{max}$$

   where

   Co_C is defined in 1 or 2.

   $Co_{max} = \max \{ Co_{<C_i>}, 0 \leq i \leq n\}$

   n is the number of system components.

2. Determine both defect coupling measures and identify fault-prone component relationships.

   Unlike the prior studies [10, 14, 15], this study sets the threshold to an order of magnitude less (or 10% less) than the highest defect coupling measure. This is an alternative to setting the threshold to some percentage of fault-relationships. The purpose is to focus on fault-relationships with the highest ranks. This reduces the number of relationships, so that testing or reengineering efforts may focus on components that are much worse than others.

   A component is considered fault-prone, if either of the following applies:

   (a) A fault relationship between two components C_1, C_2 is fault-prone, if

   $$Re_{<C_1,C_2>} \geq 0.1 * Re_{max}$$

   where

   $Re_{max} = \max \{ Re_{<C_i,C_j>}, 0 \leq i,j \leq n, \ i \neq j\}$

   n is the number of system components.

   (b) A component C is fault-prone, if

   $$TR_C \geq 0.01 * TR_{max}$$

where

$TR_{max} = \max \{ TR_{<C_i>}, 0 \leq i \leq n\}$
n is the number of system components.

3. Create a *Fault Component Directory Structure.*

   The fault-prone component directory structure shows the components identified as fault-prone according to either one of the defect cohesion measures.

   Components carry the prefix **d**, if they are identified as fault-prone using only the basic defect cohesion measure 1. They carry the prefix **f**, if they are identified as fault-prone by only the multi-file defect cohesion measure 2. If there is no prefix, the component is identified as fault-prone using both measures. Components marked in bold are fault-prone because of fault relationships. The components not in bold are fault-prone components with internal problems instead of fault-prone relationships with other components.

4. Create the *Component Level Fault Architecture Diagrams.*

   Components are included in the diagrams, if they have been identified as fault-prone with respect to at least one of the defect coupling measures. In addition, those components that are fault-prone according to the defect cohesion measures are included in an inset to the diagram for comparison purposes. Components that are fault-prone according to both a defect cohesion measure and a defect coupling measure are indicated in bold.

   Analysis of the component level diagrams can give developers and testers an indication of what kind of problems the system has and where they are. For example, one can determine if most problems are local or involve relationships between components.

   Comparing across releases makes it possible to identify components that are repeatedly fault-prone or in fault-prone relationships. Developers and testers can then focus their attention on them in terms of either re-engineering them or testing them more intensely.

5. Lift.

   The lift operation abstracts the fault-prone relationships at the component level to the subsystem level. These are presented in *Fault Architecture Diagrams* for each release and a *Cumulative Release Diagram.* A subsystem is defined as the level immediately below the system level (or root).

   Figure 1 shows an example of a fault component directory structure. In this example, S_1 and S_2 are subsystems. Not all components are at the same level. A component belongs to a subsystem, if there is a path from the subsystem to the component. In the example, component C_{11} belongs to S_1. Components C_{21}, C_{22}, and C_{23} belong to S_3.

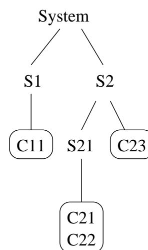

Figure 1: Example of a Fault Component Directory Structure Diagram.

The steps the lift operation are:

(a) Determine defect coupling measures for the subsystem level.

For any two subsystems, S_1 and S_2, the defect coupling measures are defined as follows. Let $C_{11}, \ldots, C_{1n}$ be components in subsystem $S1$ and $C_{21}, \ldots, C_{2n}$ be components in subsystem $S2$, then

$$Re_{<S_1,S_2>} = \sum_{i=1}^{n} \sum_{j=1}^{m} Re_{<C_{1i},C_{2j}>} \tag{5}$$

where $Re_{<C_{1i},C_{2j}>} \geq$ threshold for multi-file defect coupling measure for the release.

$$TR_S = \sum_{i=1}^{n} TR_{C_i} \tag{6}$$

where $TR_{C_i} \geq$ threshold for cumulative defect coupling measure for the release.

(b) Create subsystem level diagrams.

We construct *Fault Architecture Diagrams* for the three releases using the defect coupling measures for the subsystem level. Nodes in the diagram represent subsystems that occur in at least one fault architecture diagram. Arcs indicate the magnitude of the problem between the subsystems. Nodes in the diagram represent subsystems that occur in at least one fault architecture diagram. Arcs indicate the magnitude of the problem.

Next we aggregate the fault architecture diagrams into a cumulative release diagram. The *Cumulative Release Diagram* illustrates persistent problems within and between subsystems. This diagram aggregates subsystem level relationships across releases. The arc annotations in the diagram describe fault relationships between subsystems that persisted across releases.

The questions we address are:

- What are the sybsystems with the majority of problems?
- Are problems between subsystems or are they local within the system?
- Do problems in one release occur in other releases?

Analysis of the subsystem level diagrams can give developers and testers an indication of whether problems exist between subsystems or are local within systems. In addition they indicate problems that occur over several releases.

3.2 Single Phase Analysis

Studies in [9, 10, 14, 15] used defect measures from successives release to identify components that are fault-prone or are in fault-prone relationships across releases to identify possible code decay. By constrast, this study also applies these techniques to identify components that are fault-prone or in fault-prone relationships during development, system test and post release to determine those components that should be tested more intensely.

This study derives and analyzes component level fault architecture diagrams for each of the development phases. The questions we address are:

- Is system test finding all problems?
- Do problems in development appear in test?
- What is the nature of the problems that are detected in the field?

To investigate this we perform across-phase analysis between earlier phases of development and post-release, comparing fault problems in development, system test, and post-release. Comparing across development phases within a release makes it possible to see whether some components are repeatedly in fault-prone relationships, whether the problems are prevalent in particular phases, and whether they are repaired. For example, few fault-prone relationships during development, but many during testing indicate that development is creating defects, but either does not find them or does not correct them during development. A preponderance of fault-prone relationships during post-release indicates insufficient system testing. The objective of this analysis is to assess the quality of the product and identify possible architectural problems. By analyzing earlier defect data, the information may be used to guide testing, that is to improve its effectiveness.

We investigate whether development fault architectures can identify the parts of the software that need to be tested more intensely. We validate these assessments using system test data from the same release. We also use the development and system test fault architectures to identify fault-prone components after release and validate our assessments using post release data.

4 Case Study

4.1 Data

The defect data come from a large medical record system, consisting of 188 software components. Each component contains a number of files. The components vary in the number of files they contain, ranging from 1 to over 800 files. Defect data covered three releases of the system. All releases added major functionality to the product. Of the 188 components, 99 had at least one defect in Releases 1, 2 or 3. Seven new components were added in Release 1, 5 new components were added in Release 2, and 3 new components were added in Release 3. In addition, one of the new components in Release 1 saw major enhancements in Release 2. Many other components were modified in all three releases.

The data is based on defect and fix reports. A defect report records the following:

- release identifier
- phase in which defect occurred (development, test, post release)
- defective entity (code component, type of document by subsystem)
- whether the component was new for a release
- the date the defect was reported.

A fix report records the following:

- release identifier
- defect identifier for which the fix is being made
- file identifier for the file being changed to correct the defect
- the date the change was made.

Table 1: Number of components identified as fault-prone in Releases 1 – 3.

	Release 1	Release 2	Release 3
Basic defect cohesion measure	17	18	3
Multi-file defect cohesion measure (Eq. 2)	6	10	2
Both combined	18	24	5

4.2 Determine Defect Cohesion Measures

This study identified the most fault-prone components in each release using the two ways to measure defect cohesion. Table 1 shows how many components were identified as fault-prone in each release. The last row shows the number of components that are considered fault-prone by at least one of the defect cohesion measures.

If there is a large overlap in the number of components that are considered fault-prone by both rankings, one can assume that the two measures are similar. In this case it means that components with a lot of defect reports also require changes in multiple files. If there are a lot of components that are fault-prone according to one method and not the other, the measures are different. In this case it means a component with few defect reports may not require changes in multiple files to repair its defects. Further a component that required many multiple file changes may have few defects reported.

Table 1 shows that the basic defect cohesion measure results in identifying 17 fault-prone components in Release 1, while the multi-file defect cohesion measure identifies 6 fault-prone components. Between the two methods, there is an overlap of five components. Thus, the multi-file measure mostly flags components that have already been identified as fault-prone by the basic measure. However, the basic measure includes components not flagged by the other measure. In Release 2, the two defect cohesion measures identify 18 and 10 fault-prone components with an overlap of four components between the two methods. Table 1 shows that in Release 3, the defect cohesion measure identifies three components, the multi-file defect cohesion measure identifies two components. There is no overlap between the fault-prone components in Release 3. (The *Fault Component Directory Structure* shown in the next section displays these components.)

The two methods for computing the defect cohesion measure identify a few of the same components as fault-prone. There are sufficient differences between what the two measures identify as fault-prone components that it warrants using both methods. This is more noticeable when looking at the *Fault Component Directory Structure* (see Figure 2).

Table 2: Number of releases in which components were fault-prone.

	Number of Times Fault-Prone	0	1	2	3
Number of Components	Basic defect cohesion measure	161	15	11	1
	Multi-file defect cohesion measure (Eq. 2)	174	10	4	0
	Both combined	156	21	11	1

Table 2 shows many of the 188 components were never identified as fault-prone by either method for computing defect cohesion. The last row shows that 33 components were identified as fault-prone in at least one release using one of the defect cohesion measures. At most one or two components were identified as fault-prone in all three releases and these were components that had defect cohesion measures much, much higher than any other components.

4.3 Fault Component Directory Structure

Figure 2 shows the *Fault Component Directory Structure.* Components in the diagram include those identified as fault-prone using either the basic defect cohesion measure (labeled with prefix 'd') or the multi-file defect cohesion measure (labeled with prefix 'f') in at least one release. Several components are identified as fault-prone using both measures (no label).

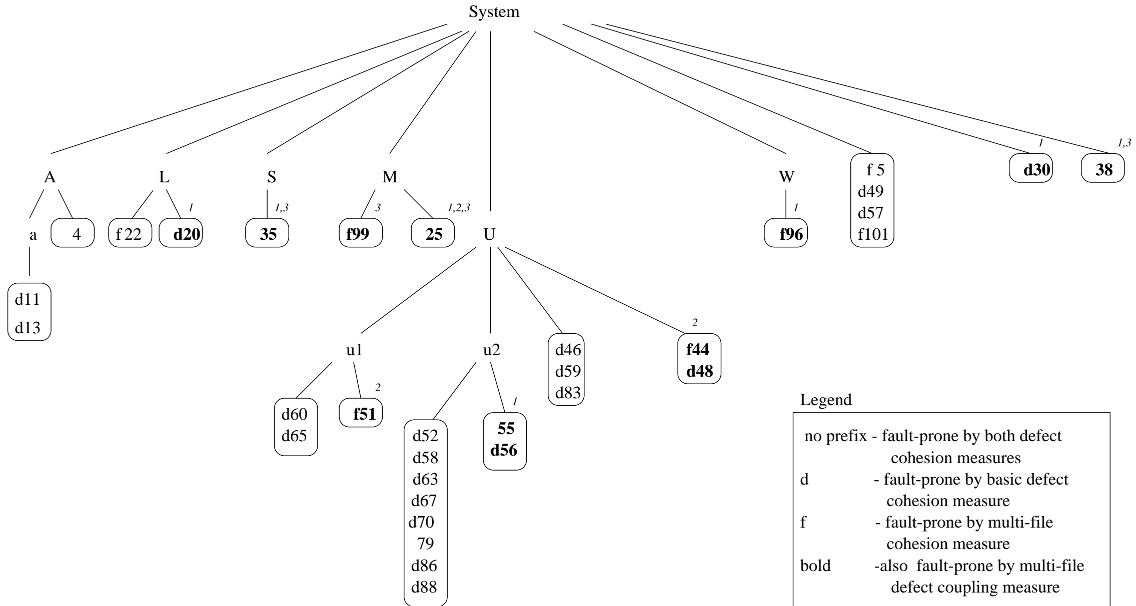

Figure 2: Fault Component Directory Structure.

Not all the components identified as fault-prone using one method were fault-prone using another. The two methods for computing the defect cohesion measure occasionally identify different components as fault-prone. There are 32 components that are fault-prone using either the basic defect cohesion measure or the multi-file defect cohesion measure in the three releases. Of these 32, 20 are fault-prone according to the basic defect cohesion measure, that is they had a lot of defect reports, but not a lot of file changes. Six components are fault-prone according to the multi-file defect cohesion measure, indicating they had a lot of file changes but not a lot of defect reports. Six components are fault-prone according to both measures, that is they had a lot of defect reports and a lot of file changes. Since there is not a large overlap of components that are fault-prone by both measure (6 out of 32), this indicates that the measures identify different components and both measures should be used to determine components that require more intense testing or reengineering, rather than choosing one method over another.

It should be noted that components vary in size and in the number of files they contain. Component 38, for example, which has 230 files, may be considered a subsystem. Component 79, which is several levels down in the existing hierarchy, consists of 183 files. Component 86, which is at the same level as component 79, consists of 1 file.

4.4 Determine Defect Coupling Measures

Table 3 summarizes the results of applying the defect coupling measure. Column 2 identifies the number of components involved in fault relationships for each release. Column 3 gives the number of fault relationships (arcs) between components. This shows that between Release 1 and 3, the number of fault relationships decreased to less than one quarter of the number of such relationships in Release 1. Similarly, the number

Table 3: Fault relationship information

	Number of Components	Number of Relations
Release 1	65	245
Release 2	74	61
Release 3	34	56

of components in fault relationships was cut almost in half by Release 3.

To determine which are the most problematic of these relationships, we apply the order of magnitude threshold. Table 4 shows the results. Column 2 identifies the number of components involved in fault-prone relationships for each release using the multi-file defect coupling measure. Column 3 identifies the number of fault-prone relationships between components. Column 4 identifies the number of components involved in fault-prone relationships using the cumulataive defect coupling measure. (The number of fault-prone relationships is the same for both measures.) Column 5 identifies the number of components in fault-prone relationships using either measure.

Table 4: Fault-prone relationship information using the defect coupling measures

	$Re_{<C,C_i>}$ measure		TR_C measure	Combined
	Components	Num. Relations	Components	Components
Release 1	15	10	21	21
Release 2	10	7	10	10
Release 3	4	2	6	6

These are much more manageable numbers of components to focus attention. Table 4 also shows that using both defect coupling measures to identify components with a lot of fault relationships does not result in more components than using the TR_C measure alone. In this study, then, using only the TR_C measure to identify components for more intense testing is sufficient.

Based on these results, we updated the *Fault Component Directory Structure* in Figure 2, marking components in fault-prone relationships with other components in bold. Bold components are annotated with the release identifiers in which they were considered relationship fault-prone. The components not in bold are fault-prone components with internal problems instead of fault-prone relationships with other components.

4.5 Component Level Fault Architecture Diagrams

Figures 3 – 5 show the *Fault Architecture Component Level Diagrams* for Releases 1, 2, and 3. Note that the *Fault Architecture Component Level Diagrams* include components at different levels of the existing logical structure. In addition, the components are of various sizes in terms of the number of files they include. The diagrams show the components in fault-prone relationships (arcs are annotated with the Re_{C,C_i} value). Components are also included, if they have high cumulative defect measures (The TR_C measure is shown in parenthesis). Components in bold are also fault-prone according to one of the defect cohesion measures.

Figure 3 shows ten fault-prone relationships involving 15 components in Release 1. Six components have no fault-prone defect relationships with any other component. These components, 10, 46, 51, 65, 79 and 114 are included, because they have high cumulative defect measures. The other 15 components

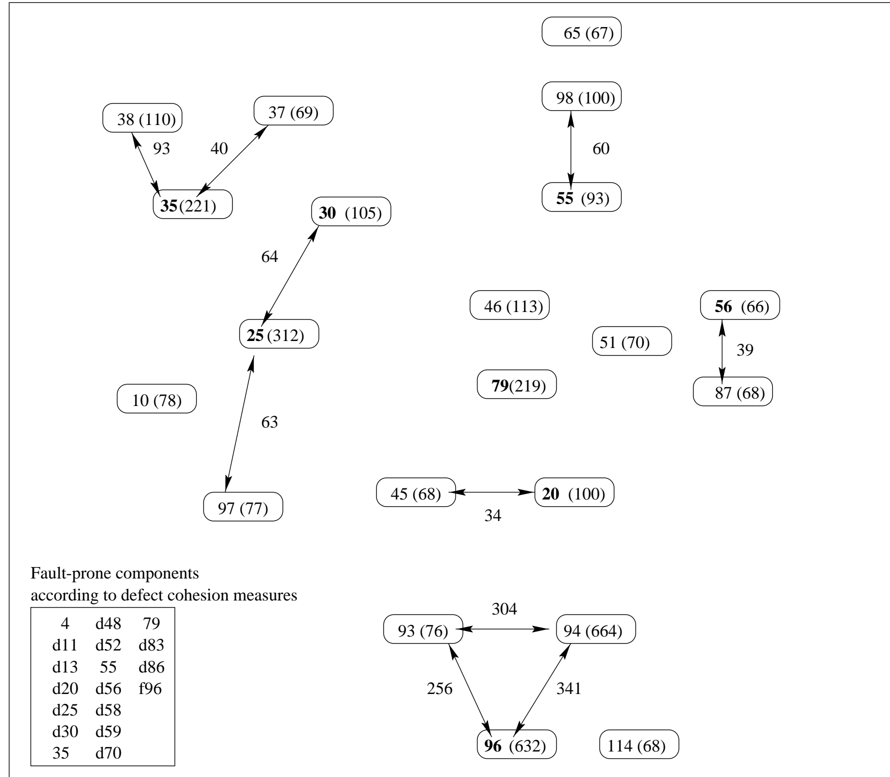

Figure 3: Release 1 Component Level Fault Architecture.

were fault-prone according to both defect coupling measures. Several components were involved in more than two fault-prone relationships. Components 25 and 35 were involved in two fault-prone relationships. Components 93, 94 and 96 were in fault-prone relationships together. All other components had only one fault-prone relationship.

In Release 2 there are seven fault-prone relationships involving ten components. Component 25 is again in a fault-prone relationship, but this time with a different component. All of the components were fault-prone based on either of the two measures. No additional components were identified based on the cumulative defect coupling measure. In effect, both measures included exactly the same components in Release 2.

Figure 4 shows fewer components and fewer fault-prone relationships, indicating that some prior problems have been successfully fixed. This trend continues into Release 3. There is, however, one component (25) that was relationship fault-prone in all three releases. There are two fault-prone relationships involving four components and two other components that have high cumulative defect coupling measures. Component 35 reappears again in fault-prone relationships (it had disappeared in Release 2).

Using the $Re_{<C,C_i>}$ by itself did not include any more fault-prone components than the TR_C measure. Since all components considered fault-prone by the $Re_{<C,C_i>}$ measure are also fault-prone by the TR_C measure, the TR_C might be considered a good choice for a defect coupling measure all on its own.

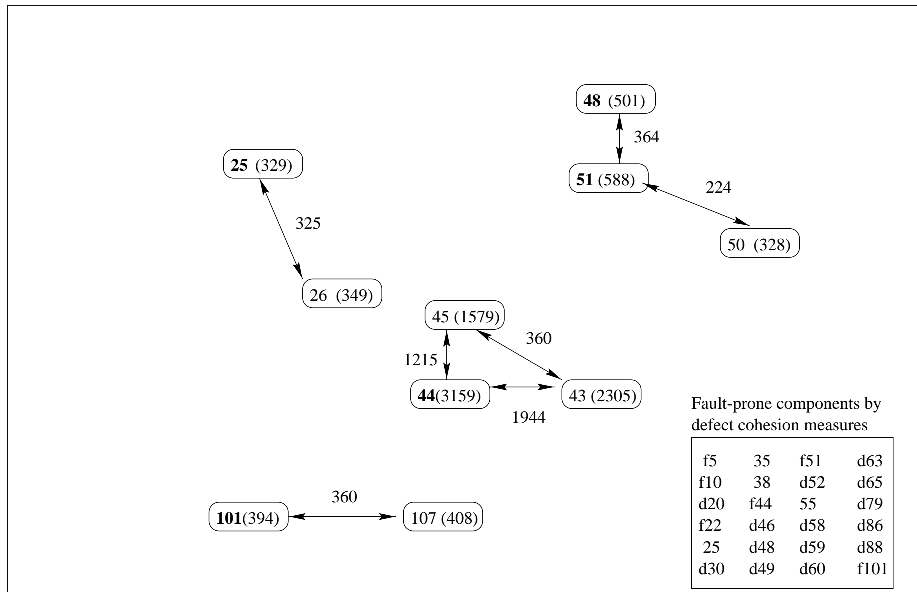

Figure 4: Release 2 Component Level Fault Architecture.

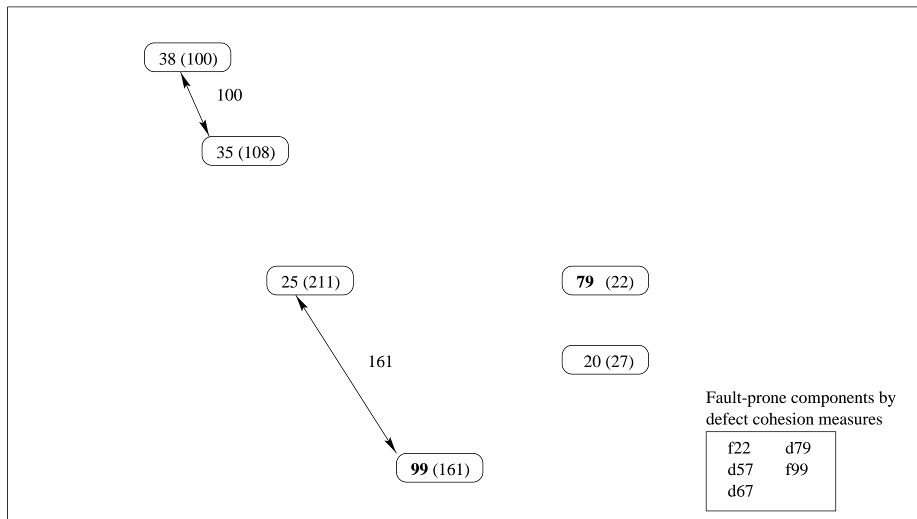

Figure 5: Release 3 Component Level Fault Architecture.

4.6 Interpreting Component Level Diagrams

The component level diagrams indicate that the most fault-prone parts of the software are not due to relationship problems, but to local problems, that is, problems internal to the components.

Analysis of the fault component directory structure shows that 32 components out of 188 are locally fault-prone. This is a low number, less than 20% of the components have problems. Out of these 32 components, almost half are also relationship fault-prone in at least one release. The fault-prone components are at various levels in the directory structure.

In looking at the component level fault architecture diagrams, one can see that many components that are in fault-prone relationships are also locally fault-prone (8 out of 21 in Release 1, 5 out of 10 in Release 2, and 2 out of 6 in Release 3). In addition, several components are in more than one fault-prone relationship.

More attention should be focused on these components.

A cross-release analysis of the component level fault architecture diagrams indicates that the number of components in fault-prone relationships and those with a large number of fault relationships decreases in successive releases. This indicates that components are being repaired. The number of fault-prone components increased in Release 2. Problematic parts of the software had less to do with relationships between components and more to do with internal problems These problems should be less difficult to fix. In Release 3, both kinds of problems continue to decrease. Cross-release analysis also reveals that a few components are consistently problematic: Components 25, 35, 38, and 45 are fault-prone in at least two releases. Components 25 and 35 are fault-prone both in terms of relationship problems and local problems. System developers and testers should focus on these components in successive releases.

4.7 Lift the Fault Relationships to the Subsystem Level

For purposes of performing the lift operation, Table 5 shows to which subsystem components in fault-prone relationships belong. Components 30, 38, 101, and 107 are also considered subsystems. Subsystems A, 5,

Table 5: Subsystem containment of components with many fault relationships.

Subsystem	Components
S	35, 37
M	25, 97, 99
W	93, 94, 96, 114
U	44, 45, 46, 48, 50, 51, 55, 56, 79, 87, 98
L	20, 22
X	26
B	43

10, 49 and 57 do not contain components that are fault-prone or are in fault-prone relationships.

4.7.1 Create the Fault Architecture Diagrams

The fault architecture diagrams illustrate subsystems that have a majority of the problems. They also indicate whether relationship problems are between subsystems or more within subsystems.

Figures 6 – 8 show there are only six fault-prone relationships between subsystems in the three releases. There are few fault relationships between components, hence there are few between subsystems, and the degree of fault-proneness is small.

Figure 6 shows that in Release 1, subsystems M, S, 30, 38, U, and L have fault-prone relationships with other subsystems. Subsystems S, M, U also have fault-prone relationships internal to the subsystem. W has only internal fault-prone relationships.

Figure 7 shows that in Release 2, we have three fault-prone relationships between subsystems. The first is between subsystems M and X. The second is between subsystem 101 and subsystem 107. The third is between the subsystems U and B.

Figure 8 shows that in Release 3, we have only one fault-prone relationship between subsystems. It is between subsystems S and 38. Subsystem M has fault-prone relationships that are internal to the subsystem.

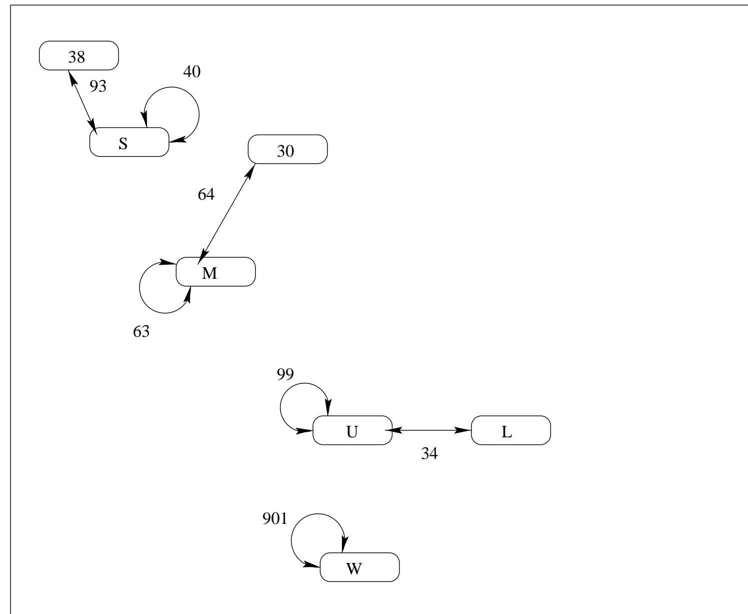

Figure 6: Release 1 Fault Architecture Diagram.

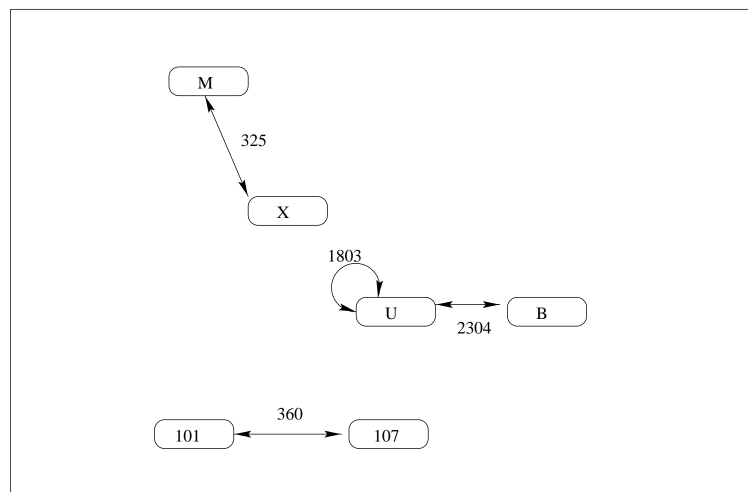

Figure 7: Release 2 Fault Architecture Diagram.

4.7.2 Create Cumulative Release Diagram

So far, we have only analyzed each release individually. When successive releases are to be tested, it might be useful to have a test guideline on how to treat components and relationships that have been fault-prone in prior releases. The purpose of a test guideline would be to focus attention on components in system test that had fault-prone relationships in the prior release. Figure 9 shows the *Cumulative Release Diagram* for the system in this study.

4.8 Interpret Subsystem Level Diagrams

The subsystem level diagrams reveal that the most problematic relationships between subsystems were different for each release, save for one or two relationships. Only six fault-prone relationships exist between

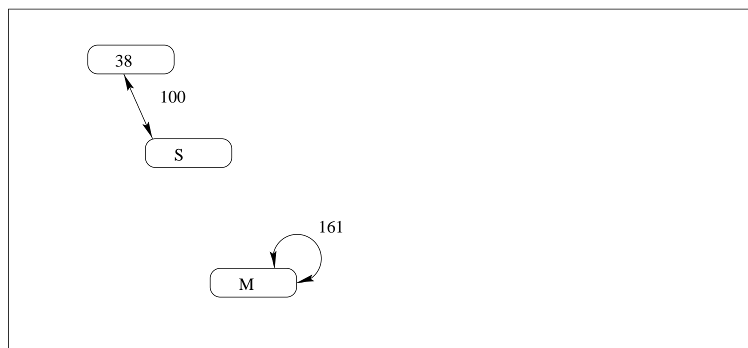

Figure 8: Release 3 Fault Architecture Diagram.

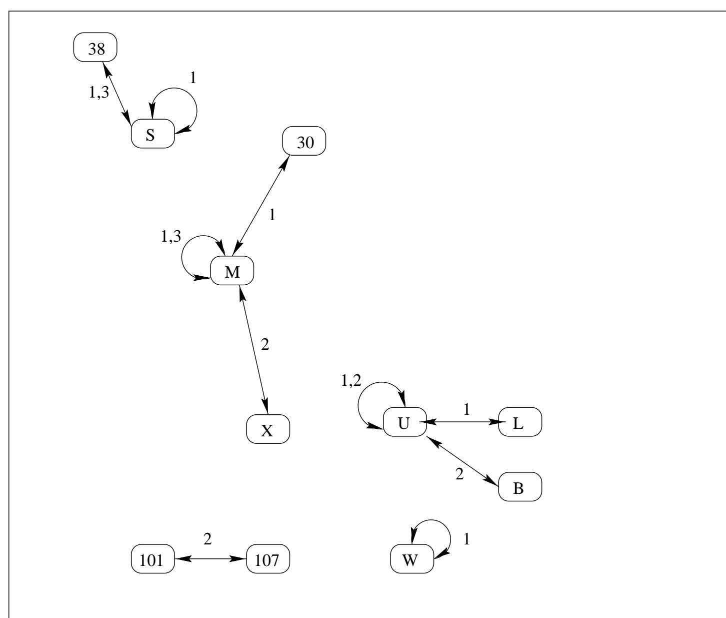

Figure 9: Cumulative Release Diagram using the multi-file defect coupling measure.

subsystems in the three releases. Three subsystems, 38, U and M are in fault-prone relationships with two other subsystems. In this system, the most problematic parts of the software are internal to the subsystems, rather than between subsystems.

Results indicate that a test guideline that would focus attention in system test on components in a system that had fault-prone relationships in the prior release does not help in this case study. Figure 9 shows only one fault-prone relationship between subsystems occurring more than once. This is the fault relationship between subsystems S and 38. This relationship is fault-prone in Release 1 and Release 2. There are two reasons why such a guideline does not help in this study. They are:

1. There are few fault-prone relationships.

2. Problems, once identified, are fixed.

4.9 Single Phase Analysis

4.9.1 Create Component Level Fault Architecture Diagrams for Development and Test

Clearly, the number of reported defects for development phases is much lower than the number of reported defects for the entire release.

Row 1 in Table 6 shows the number of fault relationships that exist in development, system test, and post release in Releases 1 – 3. Row 2 shows the number of relationships that are fault-prone according to either defect coupling measure. Row 3 shows the number of components that are in fault-prone relationships according to either measure.

Table 6: Fault relationship information for all releases by development phase.

	Release 1			Release 2			Release 3		
	dev	test	post	dev	test	post	dev	test	post
# fault relationships	211	126	27	20	7	1	12	12	4
# fault-prone relationships	8	32	9	7	7	1	3	4	1
# components in fault-prone relns.	11	33	14	10	10	2	4	6	1

Figures 10 and 11 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for development and system test for Release 1. The numbers annotating the arcs are the $Re_{<C,C_i>}$ measure. The numbers in parentheses are the TR_C measure. Of the eleven components that are in fault-prone relationships in development, seven are also in fault-prone relationships in system test. These components include: 25, 30, 35, 38, 96, 97, and 98. The components with which they are relationship fault-prone, however, are not necessarily the same. Only the two relationships between components 35 and 38 and components 25 and 30 are fault-prone in both development and system test. Components 25 and 35 are in fault-prone relationships in multiple releases as well.

Figures 12–13 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for development and system test for Release 2. In Release 2, none of the fault-prone relationships in development exist in system test. The components in fault-prone relationships are also different. A test guideline that recommends testing components in fault-prone relationships in development more intensely would not work well in Release 2.

Figures 14 and 15 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for development and system test for Release 3. In Release 3, only 1 component that is in a fault-prone relationship in development is also in a fault-prone relationship in system test. This is again component 25. The components with which it has fault-prone relationships are different in development and system test. Because component 25 is in fault-prone relationships in five phases in the three releases with many other components, this component may require attention.

4.9.2 Create Component Fault Architecture Diagrams for Post Release

Figures 16 to 18 show the *Fault Architecture Component Level Diagrams* using both defect coupling measures for post release for Releases 1, 2 and 3. The numbers annotating the arcs are the $Re_{<C,C_i>}$ measure. The numbers in parentheses are the TR_C measure.

Of the nine fault-prone relationships in post-release in Release 1, five were also fault-prone in system test. Seven out of 11 components in fault-prone relationships were also fault-prone in development or system test. This indicates that system test was identifying and testing most of the components that have relationship problems in post-release. These problems, however, did not get fixed before release.

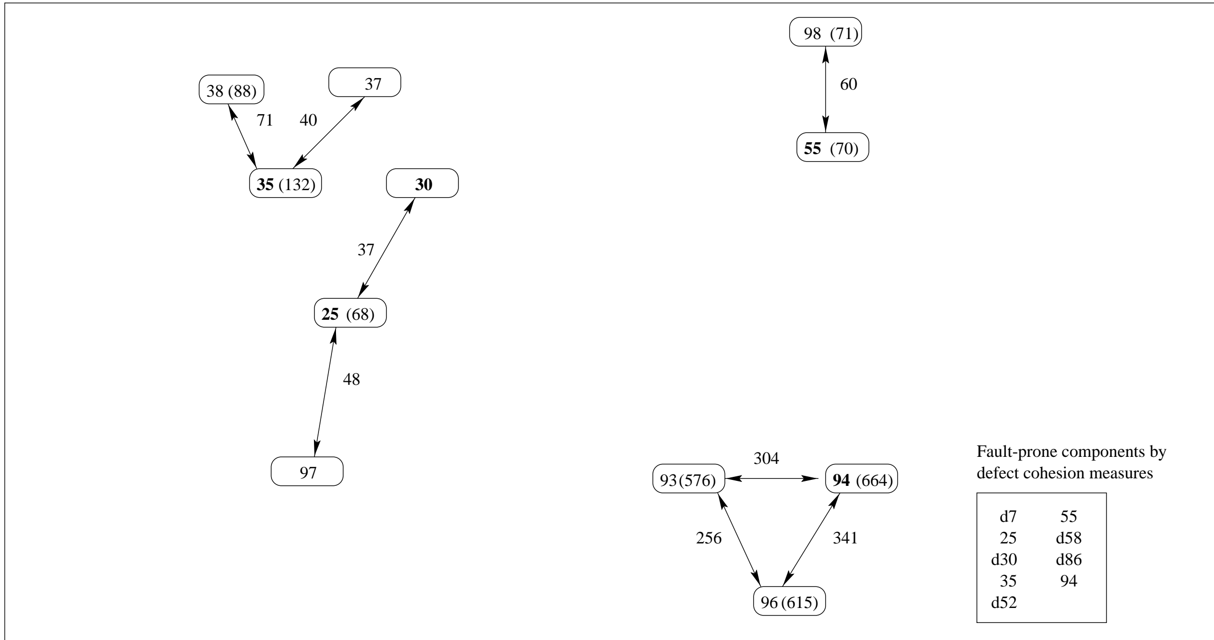

Figure 10: Fault Architecture Component Level Diagrams for development in Release 1.

Release 2 had only one fault relationship in post-release. Most of the problems in Release 2 were internal to the components. Component 26 was, however, in a fault-prone relationship in development.

In Release 3, there were four fault relationships. Only one relationship was fault-prone. This relationship was between components 35 and 38, the same relationship that was fault-prone in system test in Release 1. It was not, however, fault-prone in development or system test in Release 3. This indicates that system test did not adequately test this relationship.

4.10 Interpret Component Level Fault Architectures for Development Phases

The most problematic relationships tended to be different for the development and system test phases of each release. This indicates problems identified in development are usually fixed before system test. The most problematic relationships in post release tended to be different than those in development and system test. This indicates some problems are getting through, but because the defect coupling measures are small in post release, there are few problematic relationships.

There are only four components that are repeatedly in fault-prone relationships throughout the development and system test phases of all three release. These are not new components and easily identifiable by the high number of defects all the way through the life cycle. These components include 25, 35, 38 and 99. Obviously, such brittle components are causes for concern.

Harder to identify before release are the components that are normal during system test, but are in fault-prone relationships after release. The analysis quite strongly points out that any component that was in a fault-prone relationship during test was **not** fault-prone after release. This is good news: the problems that were identified during testing were corrected before release. Development and system testing are doing a very good job eliminating problems. None of the components in fault-prone relationships are new, but they may have have been affected by changes or enhancements. One possible cause for missing these components may be because regression test did not test them thoroughly enough. The defect data is not detailed enough to explain this phenomenon. A suggestion for improving testing would be to assess and improve impact analysis and regression testing.

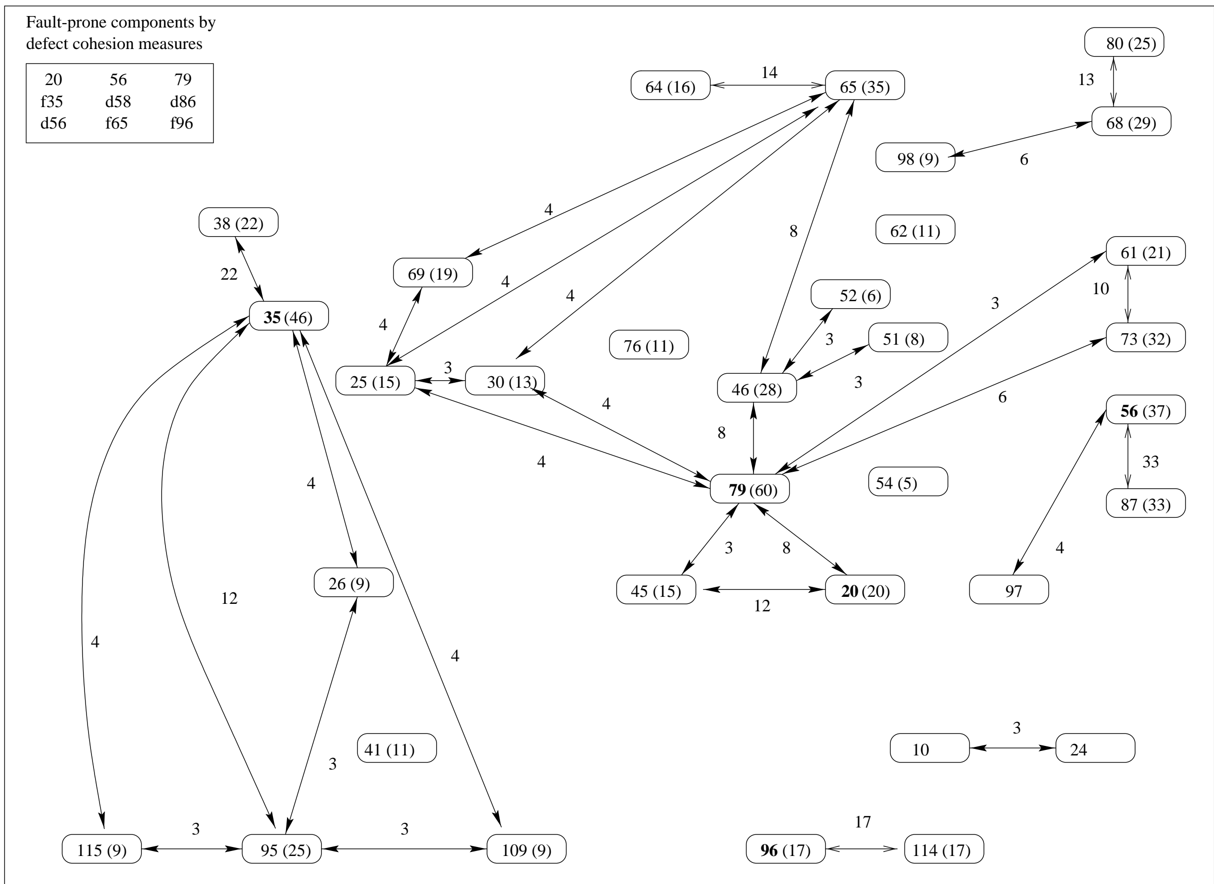

Figure 11: Fault Architecture Component Level Diagrams for system test in Release 1.

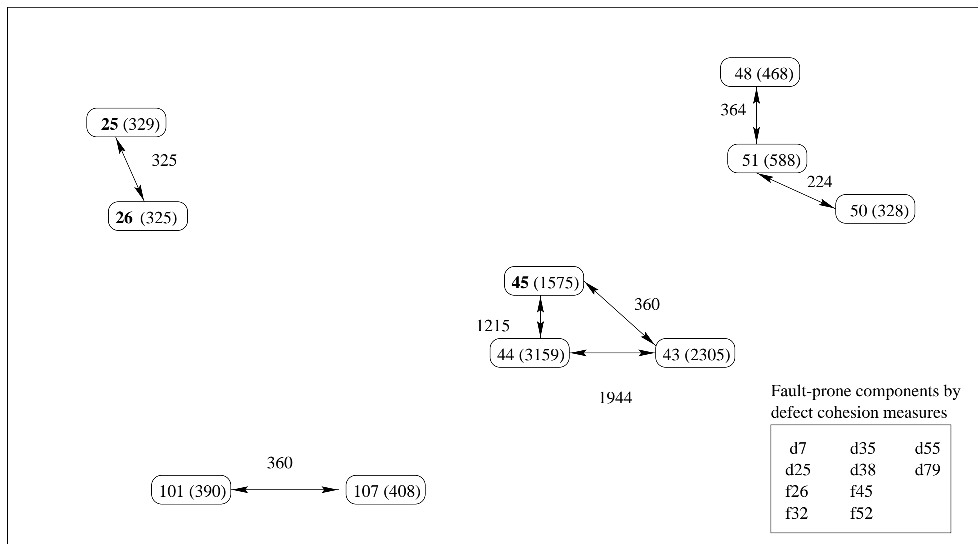

Figure 12: Fault Architecture Component Level Diagrams for development in Release 2.

Using these diagrams we performed a within-release analysis to determine whether additional test guidelines based on this data might be helpful. The following guidelines were helpful.

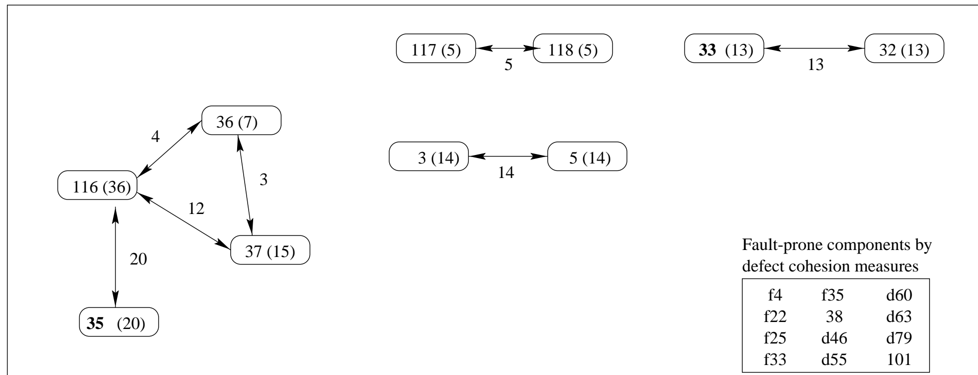

Figure 13: Fault Architecture Component Level Diagrams for system test in Release 2.

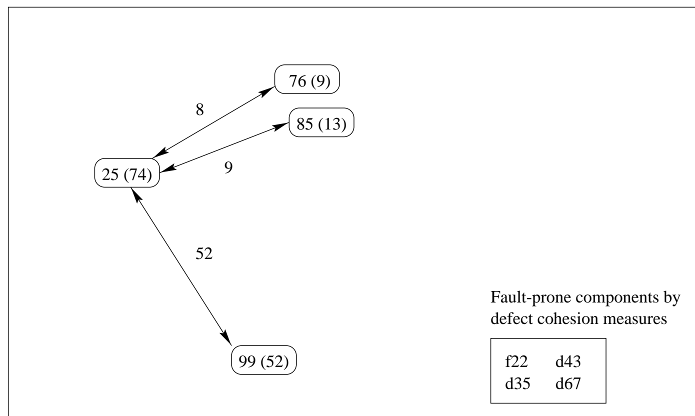

Figure 14: Fault Architecture Component Level Diagrams for development in Release 3.

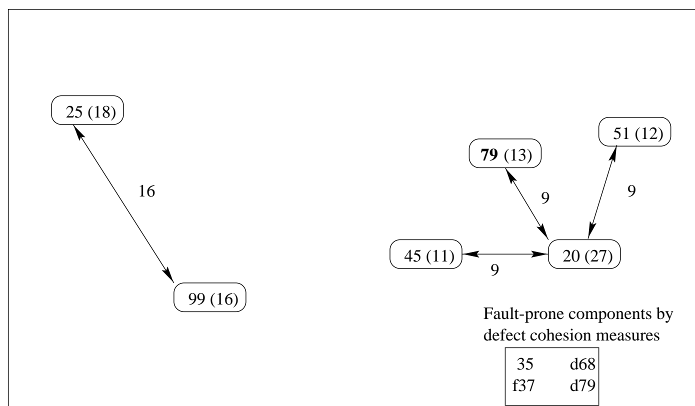

Figure 15: Fault Architecture Component Level Diagrams for system test in Release 3.

1. Test components that are fault-prone during development more thoroughly. They are likely to be fault-prone during system test.

2. Test components that are fault-prone during development as early in the drop as possible. It will give development more flexibility to fix defects and allow greater savings in test time when using statistical stopping rules.

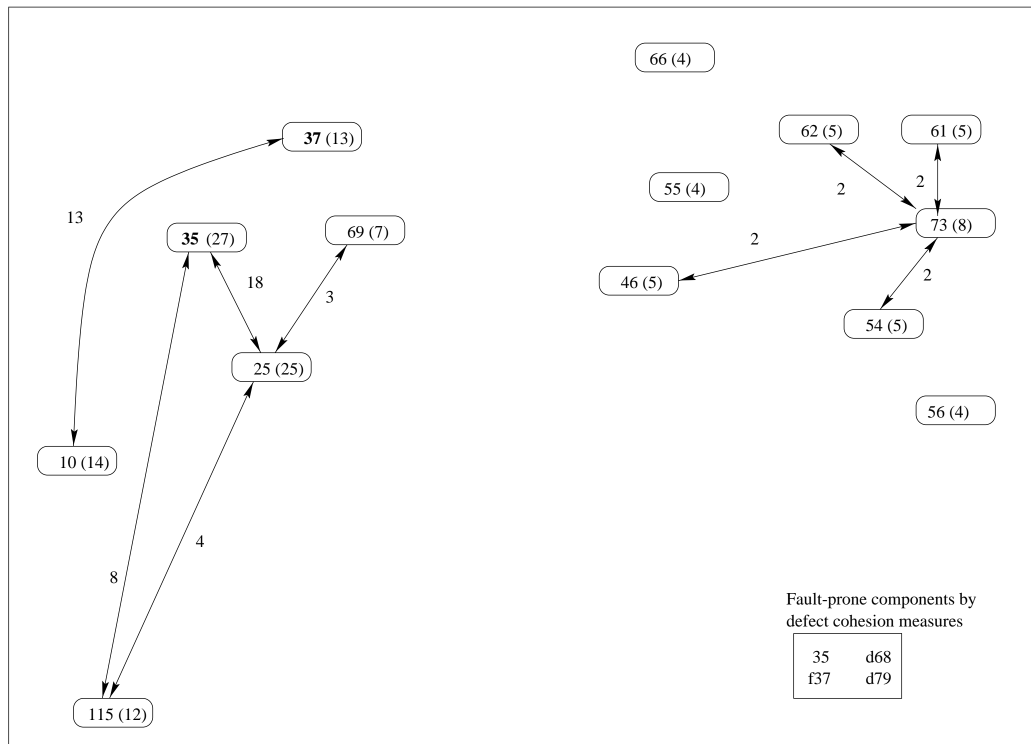

Figure 16: Fault Architecture Component Level Diagrams for post release in Release 1.

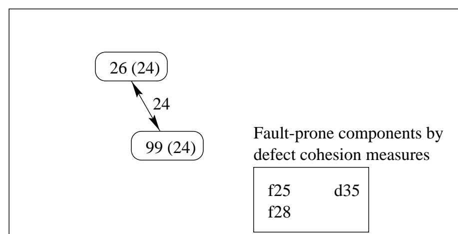

Figure 17: Fault Architecture Component Level Diagrams for post release in Release 2.

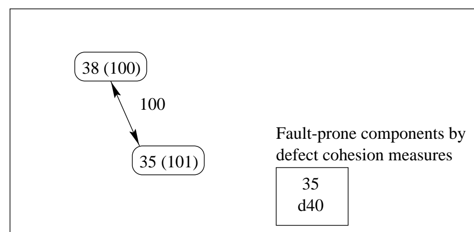

Figure 18: Fault Architecture Component Level Diagrams for post release in Release 3.

3. Between releases, pay particular attention to components that had development and post release problems, but where system testing found few problems. Likewise, evaluate the components that had problems in development, test, and after release in the prior release.

4. Improve impact analysis of enhancements on existing components and determine whether improve-

ments in regression testing could have prevented relationship problems from slipping through.

5. Specifically test components 25, 35, and 38 more intensely and earlier. They are fault-prone repeatedly according to the defect cohesion measures and the defect coupling measures.

This being a case study, we do not expect these guidelines to improve every project, although they certainly are sensible. The project we studied has certain characteristics that need to be taken into account when determining whether applying these guidelines would improve system test performance:

- Few problems remain undetected. The number of post release problems is very low.
- Most problems that are detected are fixed before release.
- In each release, the code is of very high quality. There are only nine fault-prone relationships between subsystems in the three releases.

Overall, development and system test are doing a very good job at testing and repairing components. Within release analysis therefore will not help guide system test on prevention of post release defects.

5 Conclusions

This paper replicated a study to evaluate the usefulness of using defect reports to derive fault architectures. Unlike the previous study [15], it uses both defect cohesion measures to identify fault-prone components and both defect coupling measures to identify components in fault relationships. It also different in that it set the threshold to an order of magnitude less than the largest measure, rather than to some percentage of components. Defect reports are easily available and can be used to identify parts of the software that are fault-prone. We applied several measures that identified the most fault-prone parts of the system in three releases. We also applied the measures to identify the most fault-prone relationships between components in the development, system test and post release phases of each release.

The methods for computing the defect cohesion measures and the defect coupling measures differed in how they treated defect reports. The basic defect cohesion measure is based simply on the number of defect reports for the component. The multi-file defect cohesion measure is based on defect fix reports and is sensitive to the number of files changed. The defect cohesion measures identified different components as fault-prone, hence we recommend using both.

In terms of the defect coupling measures, we investigated the multi-file defect coupling measure and the cumulative defect coupling measure. The multi-file defect coupling measure identifies components in fault-prone relationships, while the cumulative defect coupling measure identifies components that are in many fault relationships. In this study, the multi-file defect coupling measure did not identify many more problematic components than the cumulative defect coupling measure. The multi-file defect coupling measure does, however, identify pairwise coupling problems between components, while the cumulative defect coupling measure identified components with defect coupling problems with many other components.

Based on the measures, we created *Fault Architecture Component Level Diagrams* and *Fault Architecture Diagrams* for each release. We also created *Fault Architecture Component Level Diagrams* for the development, system test and post release phases for each release, as well. The fault architecture technique visualized problems due to architecture fairly well. It identified the most problematic component relationships in every release and for several phases of each release. Using these diagrams we were able to determine guidelines that should be helpful to software developers and testers.

Even the "almost perfect" project can benefit from the analysis and guidelines derived using fault architecture techniques. For high quality development environments like the one we are currently analyzing, the key issue for testers is where to put emphasis, and where not to. This technique enables developers

and testers to determine the most problematic parts of the software. They can then focus their attention on these parts. This has the greatest potential for being more efficient without sacrificing effectiveness.

References

[1] D. Ash, J. Alderete, P.W. Oman and B.Lowther, "Using Software Models to Track Code Health", *Procs. International Conference on Software Maintenance*, ICSM'94, (September 1994), Victoria,, British Colombia, Canada, pp.154 – 160.

[2] S. Eick, C. Loader, M. Long, L. Votta, S. VanderWeil, "Estimating Software Fault Content Before Coding", *Procs. 14th International Conference on Software Engineering*, (1992), Melbourne, Australia, pp.59 – 65.

[3] L. Feijs, R. Krikhaar, R. van Ommering, "A Relational Approach to Software Architecture Analysis", *Software Practice and Experience*, vol. 28, no. 4, (April 1998), pp. 371 – 400.

[4] H. Gall, K. Hajek, M. Jazayeri, "Detection of Logical Coupling Based on Product Release History", *Procs. of the International Conference on Software Maintenance*, (November 1998), Washington, D.C., pp. 190 – 198.

[5] T. Khoshgoftaar, R. Szabo, "Improving Code Churn Predictions During the System Test and Maintenance Phases", *Procs. International Conference on Software Maintenance*, (September 1994), Victoria, British Colombia, Canada, pp. 58 – 66.

[6] T. Khoshgoftaar, E. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software", *Procs. Ninth International Symposium on Software Reliability Engineering*, (November 1998), Paderborn, Germany, pp. 344 – 353.

[7] R. Krikhaar, "Reverse Architecting Approach for Complex Systems", *Proceedings of the International Conference on Software Maintenance*, (September 1997), Bari, Italy, pp. 1 – 11.

[8] N. Ohlsson, M. Helander, C. Wohlin, "Quality Improvement by Identification of Fault-prone Modules Using Software Design Metrics", *Procs. International Conference on Software Quality*, ICSQ'96, (1996), Ottowa, Canada, pp.1 – 13.

[9] M. Ohlsson, C. Wohlin, "Identification of Green, Yellow and Red Legacy Components", *Procs. International Conference on Software Maintenance*, ICSM'98, (November 1998), Bethesda, Washington, D.C., pp.6 – 15.

[10] M. Ohlsson, A. von Mayrhauser, B. McGuire, C. Wohlin, "Code Decay Analysis of Legacy Software through Successive Releases", *Procs. IEEE Aerospace Conference*, (March 1999), Track 7.401.

[11] N.F. Schneidewind, "Software Metrics Model for Quality Control", *Procs. International Symposium of Software Metrics*, Metrics'97, (November 1997), Albuquerque, New Mexico, pp.127 – 136.

[12] C. Stringfellow, A. von Mayrhauser, "Quantitative Analysis of Development Defects to Guide Testing: A Case Study", submitted to *Journal of Software Quality*, 1999.

[13] S. Tilley, K. Wong, M. Storey, H. Muller, "Programmable Reverse Engineering", *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 4, (December 1994), pp. 501 – 520.

[14] A. von Mayrhauser, J. Wang, M. Ohlsson, C. Wohlin, "Deriving a Fault Architecture from Defect History," *International Conference on on Software Reliability Engineering*, (November 1999), pp. 295 – 303.

[15] A. von Mayrhauser, J. Wang, M. Ohlsson, C. Wohlin, "Choices for deriving a Fault Architecture from Defect History," accepted by *Journal of Software Maintenance.*

[16] C. Wohlin, P. Runeson, "Defect Content Estimations from Review Data", *Procs. International Conference on Software Engineering*, (April 1998), Kyoto, Japan, pp. 400 – 409.

[17] C. Wohlin, P. Runeson, "An Experimental Evaluation of Capture-Recapture in Software Inspections", *Journal of Software Testing, Verification and Reliability*, vol. 5, no. 4 (1995), pp. 213 – 232.