Multi-Version Software Analysis to Detect Architectural Mismatches

Master’s Thesis

Koen Egelink
Multi-Version Software Analysis to Detect Architectural Mismatches

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Koen Egelink
born in Warnsveld, the Netherlands
Multi-Version Software Analysis to Detect Architectural Mismatches

Author: Koen Egelink
Student id: 9217376
Email: koen@egelink.nl

Abstract

A software repository is a special kind of information system maintaining information about one or more software components. The information may range from software code to design and requirement specifications, design decisions, test data, software versions and configurations.

Software repository data are available for most large software projects and represent a detailed record of the development of a software system. Software repositories are not only useful to record development history, this data can also be used to study various aspects of software development such as software design and architecture.

In this thesis we investigate whether the problems of increased effort encountered during code changes at ASML are caused by an inappropriate software architecture. To answer this question, we use software repository mining techniques to find evidence of architectural problems.

The thesis consists of two main parts: a literature study in software repository mining and a case study in which we conduct various mining experiments on ASML source code.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. Ing. L. Moonen, Faculty EEMCS, TU Delft
Committee Member: Dr. M.R.V. Chaudron, Faculty SAN, TU Eindhoven
Public
Preface

I would like to express my sincere appreciation to my committee members, chairman Prof. Dr. Arie van Deursen, supervisor Dr. ing. Leon Moonen and Dr. Michel Chaudron.

I would like to express my gratitude to my supervisor Leon Moonen for giving me the opportunity to finish my master’s in computer science at the Embedded Systems Institute. I would like to thank him for his excellent guidance and encouragements.

In addition I want to thank the Ideals project members for their input during the process.

I would also like to thank my fellow students Remco, Peter and Ewald who were so generous in their support at various stages of this study.

Then I really want to thank my parents Leo and Ingrid and my brothers Nico and Rob for their patience and support during my studies. A very special thanks to my girlfriend Sabine for her support and encouragements.

KOEN EGELINK
Delft, the Netherlands
October 3, 2006
Public
# Contents

**Preface** iii  
**Contents** v  
**List of Figures** vii  

1. **Multi-Version Software Analysis to Detect Architectural Mismatches** 1  
   1.1 Foreword ................................................. 1  
   1.2 Terminology ............................................. 1  
   1.3 Context .................................................. 2  
   1.4 Research Question ...................................... 2  
   1.5 ASML ..................................................... 3  

2. **Software Repository Mining** 5  
   2.1 What is Software Repository Mining? ....................... 5  
   2.2 Reasons for Software Repository Mining? .................. 6  
   2.3 Research Areas .......................................... 6  
   2.4 Typical Analysis ....................................... 7  

3. **Chapter 3** 9  

4. **Chapter 4** 11  

5. **Related Work** 13  
   5.1 Extraction and Preprocessing ............................. 13  
   5.2 Code Decay ............................................... 14  
   5.3 Software Evolution ...................................... 14  
   5.4 Cluster Analysis ........................................ 15  
   5.5 Visualization ........................................... 16  

6. **Conclusions and Future Work** 19  
   6.1 Contributions ........................................... 19  
   6.2 Conclusions ............................................. 19  
   6.3 Future work ............................................ 20
List of Figures

2.1 Typical Analysis Phases ............................. 7
Public
Chapter 1

Multi-Version Software Analysis to Detect Architectural Mismatches

1.1 Foreword

This document describes the research I have been doing for the past year to complete my masters thesis at Delft University of Technology. In this document we investigate a research question using available literature and techniques from the field of software repository mining. The research is done within the Ideals project at ASML in Veldhoven, The Netherlands.

This chapter introduces all aspects that form the context and motivations of the graduate project. This document is organized as follows:

- Introduction to ASML and its main business lithography, as well as the Ideals project and our research question.
- Introduction to and a description of the research done in software repository mining.
- Introduction to software configuration management and its concepts at ASML.
- Experiments and results.
- Discussion of related work.
- Conclusions and future work.

1.2 Terminology

The term software configuration management (SCM) is the build and process management and the control of software development. Build management manages the process and tools used for builds. Process management ensures adherence to the development process. Control of software development is done using a version control system. A version control system keeps track of the changes to software. Well known version control systems are SCCS, RCS and CVS.
1.3 Context

Complexity of Software Evolution

ASML is a world leader in the manufacture of advanced technology systems for the semiconductor industry. These systems are used by large internationals world wide. The software in these systems is written by ASML and is for the current product family under development since 1998. Since then the software has evolved into a state where simple modifications take too long to implement.

"Complexity is the prime obstacle to the successful incorporation of embedded software products. If no appropriate measures are taken, design teams for embedded systems will continue to expand and their designs will take increasingly longer to complete, if they can be successfully at all. Since embedded systems are among the most complex in the world, ASML faces these problems head-on. [14]"

Idiom Design for Embedded Applications on Large Scale (Ideals)

This research is carried out within the Ideals project. The Ideals project aims to

"develop a software design methodology that realizes the structured composition of software from separate modules, while handling system-wide interacting aspects of a problem domain. [14]"

These aspects are also called crosscutting aspects or concerns. The improved handling of crosscutting concerns in embedded software systems should lead to:

- Reduced code size and complexity, because the aspects are not replicated over the code;
- Improved possibilities for maintenance and change, by making changes in existing aspects simpler to realize;
- Improved efficiency of software designers by allowing them to reason and design at the higher abstraction level of crosscutting concerns, instead of at the level of implementation.

In the Ideals project, the software embedded in ASML wafer scanners is taken as a case study and acts as driver for the project.

1.4 Research Question

The basis of our thesis project lies in the investigation whether the problems of increased effort encountered during code changes is caused by an inappropriate software architecture. The research question that we investigate in this report is

can we find evidence of architectural problems using software repository mining?
Software architectures consist of a description of components and their relationships and interactions, therefore architectural problems can be detected via defects to relationships and interactions between components. If there is an architectural problem, this analysis will show modules which are candidates for refactoring/re-engineering thereby reducing the required effort for development and maintenance.

1.5 ASML

ASML Company Profile

This company profile is taken from the ASML homepage [32].

ASML is a world leader in the manufacture of advanced technology systems for the semiconductor industry. The company offers an integrated portfolio for manufacturing complex integrated circuits (also called ICs or chips).

ASML designs, develops, integrates, markets and services advanced systems used by customers - the major global semiconductor manufacturers - to create chips that power a wide array of electronic, communications and information technology products.

With every generation, the complexity of producing integrated circuits with more functionality increases. Semiconductor manufacturers need partners that provide technology and complete process solutions. ASML is committed to providing customers with leading edge technology that is production-ready at the earliest possible date. ASML technology is supported by process solutions, enabling customers to gain and sustain a competitive edge in the marketplace.

ASML's corporate headquarters are in Veldhoven, the Netherlands. Manufacturing sites and research and development facilities are located in Connecticut, California and the Netherlands. Technology development centers and training facilities are located in Japan, Korea, the Netherlands, Taiwan and the United States. Additionally, ASML provides optimal service to its customers via over 50 sales and service organizations in 16 countries.

Founded in the Netherlands in 1984, the company is publicly traded on Euronext Amsterdam and NASDAQ under the symbol ASML.

TWINSCAN

Our research focuses on software embedded in ASML TWINSCAN lithography systems. ASML systems, called steppers and Step & Scan tools (scanners), use a photographic process to image nanometric circuit patterns onto a silicon wafer, much like a camera prints an image on film.

The ASML TWINSCAN lithography system is dual-phase system that allows exposure of one wafer while simultaneously measuring another wafer. A wafer is placed
on each of the two wafer stages. A wafer stage is a supporting structure for the wafer. The wafer is exposed or measured horizontally (measuring) and vertical (leveling) to check the wafers position.
Chapter 2

Software Repository Mining

This chapter focuses on the ongoing research on the subject of software repository mining. The first part of this chapter gives a general introduction to research in software repository mining. The second part is a summary of related work that can help us in trying to answer our research question.

2.1 What is Software Repository Mining?

To answer this question, we first need to understand what is meant with a "software repository":

"A software repository is a special kind of information system maintaining information about one or more software components and supporting a number of control functions. The information may range from software code to design and requirement specifications, design decisions, test data, software versions and configurations. Control functions may include check-in/check-out facilities, work flow and notification mechanisms and others specifically tailored for software development, evolution, reuse or re-engineering. [19]"

Examples of common software repositories are

- source control systems, store changes to the source code as development progresses
- defect tracking, follow the resolution of software defects
- archived communications between project personnel record rationale for decisions throughout the life of a project

Of course every company has can have its own specific repositories. Software repository mining is about extracting useful information from (combined) software repositories. Which information is useful is to someone or a company is another question. This document aims to answer the question what information is useful to the group leader of the developers working on the software component used in the case study, currently Ed de Gast.
2.2 Reasons for Software Repository Mining?

Software repositories contain valuable information that is not only useful to record development history but also to analyze that history as is done in our research at ASML. Reasons for doing software repository mining are numerous. Some are enumerated below, accompanied by an illustrative example.

- Assist in program understanding and visualization. A visualization can help to better understand for example relationships between modules and helps to reduce the amount of data to be investigated by identifying hotspots.

- Predict and estimate the reliability and quality of software systems. If for example the number of bug reports per module is taken then this could be an indication for quality or reliability of a module.

- Study the evolution of software systems. Since software repositories contain information created over time, the evolution of the system can be recreated.

- Discover patterns of change and refactoring. Source code shows syntactical dependencies, software repositories show logical dependencies. Patterns of change can show for example which entities are changed together a lot. An entity can be anything from a declaration to a module or component.

- Understand the origins of code cloning and design changes. This will give more insight in reasons and motivations for a copy-pasted code.

- Model software processes for development, defect repair, etc. Bug fixes combined with a communication archive can be used to maybe track down the common path that is followed to fix a bug. This path can be adapted to changing needs and standardized.

- Assist in project planning and resource allocation. Effort information from the past can help to judge the effort needed for pending bug fixes or changes in functionality.

2.3 Research Areas

Lets take the example of creating visualization of relationships between modules. This process might involve extraction of information from a repository, integrating extracted information from other repositories and a visualization of this information to help interpreting the results. This process is made up of 'building blocks', these building blocks are categories in which research is done. The International Workshop on Software Repository Mining (MSR2004) distinguished six categories of research relevant to software repository mining:

- **Fact Extraction** How to get useful data from the available software repositories. This includes the quality of the extraction (quality of the extracted data) and the unknown quality of data in software repositories.
2.4 Typical Analysis

A typical software repository mining analysis is done in four stages. Figure 2.1 gives an overview of the stages involved in a relatively simple analysis.

The analysis starts with an idea which is the motivation for the analysis. To answer our research question the idea started simple and gradually after several iterations of the analysis phases, new ideas were born and where much more specific formulated.

The first phase is the extraction phase, this is also the most important phase since the extraction will directly influence the results. If the information extracted is wrong or mangled the result is unreliable. Extraction also involves preprocessing of the data, if for example branches are not of interest to you they can filtered out.

Second phase is the analysis of the extracted data. Examples are mathematical analysis or analysis of relations between entities.

To better understand a visualization the results of the analysis can be used. Existing visualization techniques can be used or new developed techniques that suits the needs.

The last phase is the interpretation of the results. Interpretation is difficult if knowledge of the domain you’re working in is lacking. It is very important to find a ‘guru’ to do or to help with interpretation of the results.

- **Integration & Presentation** Integration of various sources of information and visualization techniques of the results after the data is mined.
- **System Understanding & Change Patterns** Use metrics for better understanding of the system and assist in future development. Good examples are logical coupled modules and the growth of the lines of code of a module.
- **Defect Analysis** Modeling defects and software reliability through mined data from software repositories.
- **Process & Community Analysis** Gain insight in software development team organization and structure.
- **Software Reuse** Techniques to facilitate software reuse in large teams.

![Figure 2.1: Typical Analysis Phases](image-url)
Concluding Remarks

The research on the subject of software repository mining ranges from assisting in program understanding to assisting in project management. Our research will focus on trying to understand software architecture evolution, using research from two sections mainly: infrastructure and extraction and research on integration and presentation.
Chapter 3

Chapter removed from public thesis.
Public
Chapter 4

Chapter removed from public thesis.
Chapter 5

Related Work

This chapter discusses technologies related to software repository mining. It is divided in five sections, the first section discusses work related to extraction and preprocessing of data from software repositories. The second section discusses related work covering code decay. Code is decayed if it is harder to change than it should be. The third section covers software evolution, the fourth covers cluster analysis. The final section discusses related work about visualizations. Although some related work covers multiple sections, we discuss them in the section that we think is most appropriate.

5.1 Extraction and Preprocessing

In [7] Fischer introduces the population of a Release History Database that combines version and bug report data. Combining the modification report information with the bug report database is useful in several ways: detection of logically coupled files, for example files which are coupled by the appearance of the same bug ID in several files distributed over the source tree; identification of error prone classes with affected components or products; or estimation of code maturity with respect to the probability of remaining bugs and discover rate of bugs in earlier releases of the system. Our approach also combined version (SCCS) and bug report data (CR/PR). Although we did not use a preconstructed database, we did use datafiles for storage of combined information.

Fischer gives a good overview about the problems and considerations when extracting and preprocessing the available data. Extraction and preprocessing are important steps because the quality of the result depends on the extracted data.

Preprocessing and extracting data from a version control system can be based on the description of the change that was entered. These log or change messages can be a valuable resource [17] to maintain and manage a software project. In [15] Hassan concludes that researchers should investigate techniques and approaches to improve the quality of the change messages and make them more accessible for developers as they evolve software systems. In our experiments we noticed that analyses are more reliable when developers describe their changes more accurately. We also think that when developers see that change descriptions can be used in an analysis that they benefit from, they are more likely to enter proper change descriptions.
Another technique is used by Alonso in [25]. Alonso developed a framework for the analysis and exploration of software repositories that relies on database techniques. The framework was not applied to source code archives, only to a mailing list archive.

5.2 Code Decay

Early work in software repository mining of Eick [5] found evidence of decayed code. Code is decayed if its more difficult to change than it should be in respect to the cost of the change, the time needed to complete the change and the quality of the changed software. Eick uses a medical metaphor of software as a patient which enables him to reason in terms of causes, symptoms, risk factors and prognoses. Others [10, 26, 11, 22, 9, 16] also analyze quantified risk factors through ‘code decay indices’ but do not mention possible causes like Eick does. Eick lists among others inappropriate architecture, violations of the original design principles and time pressure as possible causes for code decay. Examples of quantified risk factors are the number of lines added/deleted, age, growth, history of frequent changes and creative combinations of these and others that fit the goal of the research. To find evidence of possible problems in the software architecture we also looked for symptoms. We quantified risk factors as module hotspots and the crosscuttingness of change requests and problem reports.

5.3 Software Evolution

Software repositories keep track of the history of a product. This history allows to reconstruct the evolution of the product. In [20] Kemerer tries to find an explanation for the characteristics of a long series of time-ordered observations of a large software product. A version control system keeps track of (time-ordered) changes to a product. Kemerer first classifies events in the history of the software into types of changes and then evaluates several statistical methods to explain the characteristics. Kemerer also looks at types of changes that frequently occur. Classification to find reasons for software changes is also done by Mockus in [24]. Mockus distinguishes between corrective, adaptive, perfective, inspective and unclassified changes. These types of changes are also the reason for a change.

Graves [12] and Von Mayrhauser [34] both do a fault analysis. Graves uses change management data from a very large, software system and explores the extent to which measurements from the change history are successful in predicting the distribution of incidences of faults over modules. Von Mayrhauser tries to find software architectural problems by analyzing defect reports. Fault relationships are identified based on whether they are involved in the same defect report, and for how many defect reports this occurs. A similar approach is used by Biemann [3]. Biemann also includes metrics that indicate properties of a class’s relationship with other classes. We used a likewise approach to answer our research question. Our defect reports do not mention the files that they modify. Our approach includes an extra step in which defect reports are first linked to files. This technique enabled us to find modules that are frequently changed together and can be candidates for reengineering or restructuring.
Ducasse developed a meta-model [4] having in its center the notion of history, and argues that we need such a meta-model to reason about evolution of software systems. In solving our research question we did not use meta-models.

In [30] Van Rysselberghe explores the possibilities of clone detection on the differences of successive revisions of a file in a version control system. This heuristic allows to reconstruct evolution processes of existing software systems by exploiting techniques to detect duplication in large amounts of data. To answer our research question we looked at techniques that help us to evaluate the software architecture.

We did not focus on the historical analysis of the effort necessary by developers to make changes to software [13]. ASML already 'diagnosed' themselves with increased development effort. The increased development effort is the reason we are looking for evidence of a faulty software architecture.

The approach taken by Pinzger in [27], combines information gained from static and some dynamic analyses of the source code with release history data into single views on different abstraction levels.

## 5.4 Cluster Analysis

Early investigations in software repository mining and the efficiency in software production by Ball in [2], showed that clustering of classes based on the development history can explore relationships between classes. Logical couplings by Gall in [10, 8] are a similar approach. This clustering technique calls these relationships between classes logical couplings, since source code reveals syntactic dependencies and software repository mining reveals logical dependencies. The dependencies and interrelationships between classes and modules affect the maintainability of object-oriented systems [28]. In our opinion not only object oriented systems are affected but modularized systems in general. Our case-study at ASML involves a large modularized code base in plain c. Components are build up of modules, modules inside the component have relationships to other modules inside the component but also relationships to other modules in other components. Two entities, entities can files or modules or classes, are logical coupled if they are changed together. If two entities often are modified together then there is high coupling, the opposite of low coupling. The question how many is often? is addressed by Ball in but not by Gall. Ball further introduces a probability measure that is not further motivated:

"For each class \( C \), let \( C_{mr} \) be the number of modification records that touch class \( C \). If \( C \) and \( D \) are two unique classes let \( CD_{mr} \) be the number of modification records that touch both classes \( C \) and \( D \). \( CD_{mr} \) defines are relationship between the two classes, which translate to a link in a graph where the nodes are classes. A probability measure can also be associated with the link: \( CD_{MR}/\sqrt{C_{mr} \cdot D_{mr}} \)

Why the square root of the product of \( C_{mr} \) and \( D_{mr} \) is used, and not for example the sum of \( C_{mr} \) and \( D_{mr} \) to the power of 0.7, is not motivated. The square root is probably used because it ‘penalties’ files or modules that have many commits. If for example one module has 100 commits and another module has 16 commits, then the square root transforms them to 10 and 4. This way modules with fewer commits, which can be
younger modules, are given a better chance to survive the threshold. In [10] Gall uses a threshold on the absolute numbers of times two entities are changed together. Our approach uses different technique where the age of the modules is not ignored. Our approach is to assign the edge the maximum value of the relative number of changes of both the nodes, and then apply a standard threshold/cut-off technique.

An aspect of coupling is the distinction between internal and external coupling. Internal coupling is a dependency that happens between to entities in respective parts of the system. E.g. the relation between classes of a single component and its modules. The connections between classes within this component and any other part of the system like another module or subsystem are considered external coupling. Ball unlike Gall does not explicitly define internal or external coupling.

Another important difference is the input that is used to determine coupling. Ball uses modification records [18]. Modification records are hooks for integrating a problem report or change request. This functionality is available in ASMLs version control system, but not used. Gall uses patch-sets [23].

A fine-grained analysis [36, 37] of revision histories allows to detect coupling between program entities such as functions, methods, or attributes. Our focus will be on a less detailed level, to find evidence of problems in the software architecture our focus was on module or subsystem level.

Another approach is used by Anquetil in [1]. To cluster files into subsystems, Anquetil extracts concepts or subsystems from filenames. ASML uses filenaming conventions that allow the clustering of files by a simple dictionary sort on the filenames.

Logical coupling is an interesting topic for maybe all software products. Applied to the ASML software component LO (Lot Operations), this technique revealed the logical software architecture. This in addition to the syntactical coupling of modules which is extracted from the source code.

As already mentioned in the introduction to the Ideals project, ASML is dealing with an increasing complexity of modifications. We applied this technique to the ASML software, in this case the LO component. We did see an architectural change, but we are not sure that the changing modification complexity can be linked to an architectural mismatch.

5.5 Visualization

In [6] Fischer combines different types of data in a release history database and creates views from this combined data. The data stored are problem and modification data. Analyzing problem and modification data reveal how and where in the source tree features reside, how they have co-evolved over time, and to what degree they are coupled to each other. Dependencies such as logical coupling between problem reports can be used to group features to sets of closely related files and classes. Fischer creates feature-view that projects problem reports onto features and project-view that projects problem reports into the project tree structure.

In [33] Van Rysselberghde creates a two-dimensional view of a version control system. A dot in this view represents a change committed for a file on a given time. This visualization of change history allows the recognition of unstable components, coherent entities, design evolution and productivity fluctuations. Based on the visualization
alone, no problems nor answers can be identified. The visualization only gives indications of interesting evolution situations. We extended this visualization technique by adding different colored dots at places (file and date) where a CR or PR had modified the source code. This extension allowed us to visualize the crosscuttingness of CRs and PRs.

In [29] Riva explores the possibilities of adding a third dimension to visualizations. The visualization ideas presented here provide a lot of valuable information. This visualization is applied to the ASML software component LO, the results were presented in the case-study presented in section ??.

Fischer uses problem and modification report data to visualize feature evolution. We used a similar technique to visualize the evolution of crosscuttingness of problem and modification report data.

The evolution matrix used by Lanza in [21] allows the visualization of the evolution of classes in object oriented software systems. The evolution matrix shows for each class a metric, for example size, in several versions of the software. When classes are replaced with a suitable entity, maybe a subsystem, we think this technique can be applied to non-OO software systems.

The evolution spectrograph visualization [35] of Wu combines time, files/modules and the number of modified files per file/module coded in colors to visualize software evolution. For example, frequently modified files will show a red colored horizontal strip. Red because of the frequent modifications and horizontal because throughout the lifetime of the file it was frequently changed. This visualization technique combines three dimensions in a two-dimensional graph.
Chapter 6

Conclusions and Future Work

This chapter gives an overview of the project’s contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

6.1 Contributions

This research contributes to the Ideals project by supporting other researchers in their findings. Additionally, it provides a piece of individual research that can help ASML in managing their software complexity. Keywords in solving our research question are source model extraction, software repository integration, visualization of integration, system understanding and change patterns.

6.2 Conclusions

In this document we investigated whether the software architecture was the cause of increasing development effort. To answer this question we performed a number of experiments:

- We looked the distribution of the commits. An area showing many commits, may have a high development activity. This way we can pinpoint possible ‘problem’ modules.
- Analysis of commits by type of commit and the commit behavior
- Next we looked at the crosscuttingness of a CR or PR. The number of files and modules that were modified by a CR or PR. This gave us an impression of the impact of an average change to the source code.
- We saw module hotspots, hotspots are modules that are involved in many CRs and PRs.
- We looked at the amount of parallel development to find areas with a lot of development activity. This can be an indication of a problematic area.
• We looked at the number of unique CR and PR identifiers that could be found in the SCCS log messages and compared them to the identifiers from the CR/PR database. We also looked at the origin of a CR/PR.

• We also did an experiment to see if the logical software architecture matched the software architecture based on static function calls.

With all these experiments we have given developers with a large knowledge of the domain, tools to help them judge the current state of the software architecture. We cannot and do not want to judge the software architecture, since we do not have enough knowledge of the domain. We provided tools for others to help them in their judgment. The only thing we can judge is the individual results of our experiments, and our interpretation of the results is positive. The crosscuttingness of CRs and PRs was acceptable. The logical software architecture matched the software architecture based on static function calls in a large degree.

6.3 Future work

During our experiments we came across some elements that we could not resolve given the limited time available. To future researchers we would like to make the following recommendations:

• Create a working prototype of our theoretical solution for linking gates to specific CR/PR identifiers.

• Verify that the results of the logical couplings using the theoretical solution of linking gates to CR/PR identifiers is indeed better.

• Further analyze the impact on the comparison of logical couplings to the call graph, when static as well as dynamic calls are included.

• Extend all experiments to other releases of ATLAS to get a better overview of the evolution of (ATLAS) components.

• Map changes to functions and analyze functions instead of files. The strict configuration management at ASML makes it easier for experiments to add/change functions rather than files, resulting in noise.

• Further extend the idea of directed coupling between logical coupled modules.
Bibliography


[18] IBM. Aix version 4.3 general programming concepts: Writing and debugging programs, September 1999.


Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

Commit: Action that puts changes to source code under version control

CR: Change Request

CVS: Concurrent Versioning System

IP: Company-wide improvement proposal for implementing customer problem reports and change requests

Log Message: Describing message of the committed changes

PR: Problem Report

RCS: Revision Control System

SCCS: Source Code Control System

SCM: Software Configuration Management
Public
Appendix B

How CodeManager Merges SCCS Files

The following pages are taken from the SPARCworks/TeamWare & ProWorks/TeamWare Users Guide [31].