This is to certify that the

thesis entitled

Configuration Management Based on Software Component Locality and System Structure

presented by

Steven R. Schafer

has been accepted towards fulfillment
of the requirements for

M.S. degree in Computer Science

Betty H. C. Cheng
Major professor

Date 5/5/95

Configuration Management Based on Software Component Locality and System Structure

By

Steven R. Schafer

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Computer Science Department

1995

ABSTRACT

Configuration Management Based on Software Component Locality and System Structure

By

Steven R. Schafer

Software configuration management encompasses the entire development and maintenance phases of a software system. Three problem areas have made it difficult to construct effective configuration management systems for the development and maintenance of software systems: file localization, lack of system structure, and broad effects of change. This thesis presents a new approach to configuration management that concentrates on the abstract, logical aspects, and architecture of a software system in order to effectively support development and maintenance of systems developed using object-oriented or structured techniques. Based on this new approach, the software configuration management system (SCM) has been developed to provide a three-tiered configuration management system that supports abstract concept localization, system structure, and fine-grained version control to systematically manage the effects of change.

To R.H.W., thanks for all you taught me.

ACKNOWLEDGMENTS

I would like to thank my family and friends, for without their never ending support and understanding none of this would have been bearable. A special thanks goes to my committee members Dr. Lionel M. Ni and Dr. Mats P.E. Heimdahl. Finally, my most sincere and dearest thanks goes to my advisor Dr. Betty H.C. Cheng. Her constant source of knowledge, motivation, and encouragement made this all possible. I will always be honored to have been her student.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

It is inevitable for software to change. Configuration management is an activity of software engineering that specifically addresses the control and management of change in software systems. More importantly, software configuration management should be an "umbrella" activity that is applied throughout the software engineering life cycle [1].

Based on studies of commonly used software development approaches, three specific steps appear to be integral to most software development processes. Specifically, design, implementation and maintenance are included in software process models such as the waterfall, iterative waterfall, spiral, and rapid prototyping models [1]. Additionally, software maintenance typically follows the same phases (design, implementation, and continued maintenance). As such, a configuration management framework should concentrate on these three phases and tightly couple them to provide flexibility and to increase the traceability between the phases [2].

We have identified three problem areas that have made it difficult to construct effective configuration management systems for the development and maintenance of software systems: file localization, lack of system structure, and broad effects of change. This thesis presents a new approach to configuration management that specifically addresses these three problems in the context of four major directions in

configuration management research [3]: configuration management system architecture, product representation, product software architecture, and domain extensions. In order to demonstrate the feasibility and practicality of our approach, we have developed a prototype Software Configuration Management (SCM) system to address current configuration management problems by providing a three-tiered configuration management system that supports abstract concept localization, system structure, and fine-grained version control to systematically manage the effects of change [2].

Thesis Statement: *A software configuration management system that concentrates on the abstract, logical aspects of a system is more effective than traditional approaches to configuration management in supporting development and maintenance of large scale software systems developed using object-oriented or structured techniques.*

1.1 Motivation

The key tasks identified in software engineering and software maintenance are complemented by a number of "umbrella" activities [1]. These activities include documentation, quality assurance, reviews, and change control. These activities are all tasks of software configuration management, a collection of techniques that coordinate and control the identification, organization, construction and modification of a software system [4]. As such, configuration management is an umbrella activity encompassing the entire cyclical model of software engineering and software maintenance. This cyclical model defines the software life cycle, which begins at the conception of the software idea, and ends only when the software system is no longer in use.

Based on the importance of configuration management, it is essential for a configuration management system to provide a solid foundation for all other software engineering activities to build upon. Five key principles have been identified as the

necessary ingredients to ensure effective configuration management [4].

- **Proactive:** Configuration management should not be viewed as a solution to the problems of software development. Correct configuration management procedures should be developed to ensure that the problems do not arise.
- **Flexible:** Configuration management procedures should be flexible to compensate for the differences in developing different software systems.
- **Automated:** Configuration management tools must be developed to ease the burden of performing the procedures manually.
- **Integrated:** Configuration management should integrate all aspects of the software development project, including planning, management, development, and maintenance.
- **Visible:** The relationships between the configuration items and how they have changed should be identifiable and accessible by all involved in the project.

The approach taken by existing configuration management systems make it difficult to adhere to these principles in an effective manner. Each step in the software life cycle is localized around a different concept, whereas software systems are developed with software components that are often organized according to their logical relation throughout the entire software life cycle. Specifically, during the implementation phase, current version control systems localize around files [5, 6], leaving the software engineer with enormous configuration and maintenance problems. Although management practices such as, placing one software component per file or one object class per file, alleviate many maintenance problems, they do not, however, successfully relieve the engineer from problems at the system level. Specific consequences of this focus on files result in broad effects of change in the system for a small change in a portion of one file, and complete versions of a system are composed of many

different versions of the files that make up the system. This approach imposes upon the developer or group of developers the cumbersome task of remembering which file versions make up a given system version. Given the size and complexity of current systems, the effects of change and the focus on files must be minimized or eliminated and replaced with a focus on a more localized concept.

Likewise, current configuration management systems put little emphasis on the structure of the system components. Most methods are only concerned with the individual components that make up a system, and not with how these components interact with each other. Many modeling and design notations, such as entity-relationship diagrams, structure charts, and object models [7] have been developed to serve as a means for describing the interaction of the components of a software system. Currently, there exist systems [8, 9] that use tree-like structures to represent subsystems or variants of the software system, however these tree structures do not represent the interaction and communication between the subsystems. Therefore, the complexity of most configuration management problems could be significantly reduced with a means to describe the software architecture that facilitates the implementation and propagation of changes [10].

This thesis emphasizes three main concepts as the basis for a new approach to configuration management systems for software development and maintenance that adheres to the principles identified for an effective configuration management system: concept locality, limited effects of change, and system structure

1.2 Contributions

There are two main contributions presented in this thesis. First, a new approach to software configuration management has been developed. A generic framework developed from the approach is based on three key concepts: concept locality, system

structure, and fined-grained version control. Based on the generic framework, specific frameworks for the object-oriented analysis and design and structured analysis and design are then developed.

The second contribution of this work is the development of a fined-grained version control model for object-oriented systems. A formal model and basic operations are presented for the version control model. Finally, a proof of concept software configuration management system, based upon the configuration management framework and version control model, is described.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information relevant to the area of software configuration management, software maintenance, and object-oriented modeling. The contributions of this work are described in detail in Chapters 3 through 6. Chapter 3 identifies the key problems with current configuration management systems and presents new perspectives of major configuration management concepts on which new systems can be based. Using these new perspectives, Chapter 4 develops a new framework for configuration management. Chapter 5 defines the fined-grained version control model for object-oriented systems. Chapter 6 presents a prototype implementation of a software configuration management system for object-oriented systems. Related work in the area of configuration management is described in Chapter 7, and Chapter 8 draws conclusions and discusses possible directions for future research.

CHAPTER 2

Background

One of the first definitions of software engineering was coined by Fritz Bauer [1]

> *The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

Today there are many more rigorous definitions, however, the underlying principle is that software engineering should use sound engineering procedures for the development of software. The use of these sound procedures suggests a disciplined approach through analysis, design, implementation, testing, and maintenance stages for the production of software. This sequential approach to software production is commonly known as the "waterfall model" to software development [1]. In attempts to model how real software is developed, other models have been developed, including rapid prototyping, the spiral model, and the recursive/parallel model [1, 11].

Study of the many software development models leads to the observation that four specific steps appear to be inherent in all of the models. Specifically, definition, design, implementation, and maintenance appear in all. The definition phase of software development focuses on *what* the software system must do. The requirements of the system and software are specified. During the design phase, *how* the requirements of the definition phase are going to be satisfied is specified, along with the

detailed architecture of the complete system. The implementation phase focuses on translating the design into a specific programming language. Finally, the error correction, enhancements, and maintenance of the software system are performed during the maintenance phase.

2.1 Software Maintenance

The life time of a software system does not end once it is delivered to the customer. There will be errors to fix, new platforms to move to, new functionality requirements, and other enhancements to incorporate into the system. Software maintenance is concerned with these changes that may be necessary due to error correction, enhancements, or changes in requirements. The types of maintenance needed on software systems fall into three basic categories [1]:

- **Corrective maintenance:** changes due to defects or errors discovered in the system.
- **Adaptive maintenance:** changes due to change in requirements or change in environment, such as a new operating system.
- **Perfective maintenance:** changes to enhance functionality, make the system more efficient, or make the system more robust for later modifications.

While performing the different types of software maintenance, the same basic software engineering steps are repeated. That is, the definition of *what* change is to be made, the design of *how* the change will be implemented, and the actual implementation of the change are performed in a sequential manner. Furthermore, after a change is implemented, additional maintenance may need to be performed. Thus, software development and software maintenance are integrated into a continuous iterative model.

2.2 Configuration Management

Configuration management is an activity of software engineering that specifically addresses the identification, organization, and modification of software systems. Because change in software is inevitable and because the change usually takes place during the maintenance phase, it is important to distinguish the difference between configuration management and software maintenance. Software maintenance begins after the software product has been delivered to the customer and the system is in need of maintenance. In contrast, configuration management is a collection of tracking, control, and management activities that begin at the conception of the software product and end only after the software is taken out of service [1]. Therefore, configuration management encompasses software maintenance and the software development phases.

Configuration management involves five main tasks. First, configuration items must be identified. A configuration item is any piece of information that needs to be managed. These items can include source code, documentation, test cases, status reports, and memos. An important task for performing configuration management is to identify exactly what entities will be under management control.

Second, due to the large amount of change to the configuration items, there is the need for some type of procedure for identifying and retrieving different versions of the configuration items. The most common approach is the use of a repository or database that contains all the configuration items. This task of configuration management is commonly referred to as version control.

In a disciplined environment, it is unwise to allow change to happen randomly. Therefore, a third important task for configuration management is to create and enforce change control policies. A common practice is to create a change control board consisting of members of the project. This board will evaluate change requests,

and determine both whether a given change is necessary, and what effects it will have on the system.

Once a change has been approved by the change control board, the development team must be notified that the change is being made, who is performing it, why it is being performed, how long it will take, and what effects it will have on the rest of the development. In addition, when the changes are complete, a record of what was done, by whom, and when it was done must be retained for future reference. This type of status reporting activity is a critical task in configuration management.

Finally, in order to ensure that changes have been made properly and the software development is proceeding smoothly, periodic reviews and audits of the project must be performed. This task may take the form of technical code reviews and walk-throughs, or reviews of status reports and other documentation to ensure that the proper procedures are followed.

With these and many other tasks, configuration management easily becomes a major component of any software development project. To aid in the understanding of the policies and procedures to be enforced, one of the earliest tasks in the development process is to create a configuration management plan. This plan is a document that describes in detail the procedures that will be implemented to address the configuration management needs of a project. The diversity of software makes it impossible to create a standard plan for all software projects. As such, the configuration management plan needs to be tailored to the project environment [12].

2.3 Object Modeling Technique

The Object Modeling Technique (OMT) [7] is an object-oriented analysis and design methodology that uses three complementary modeling techniques to describe a system. The *functional model* uses data flow diagrams to describe the data flow and

services. The *dynamic model* uses state charts to describe behavior in the form of states and transitions. The *object model* describes the static structure of object classes in the system. The object model's strength is that it supports the description of not only the design and implementation of a system, but it also enables for the analysis of the problem to be described at a very high, abstract level and then refined into the design and implementation. Taking advantage of these features, this thesis uses the object model notation to describe concepts throughout. We give an overview of the object model notation used in this thesis.

An object model contains two main concepts: an *object class* and an *association.* An object class is any distinguishable entity or concept. It may be concrete, such as a file or a chair, or conceptual, such as a list or policy. Each object class has specific data (attributes) and operations (methods) which it can perform. For example, a file might have attributes, such as contents (the information in the file), size (the number of bytes in the file), and type (binary or text file); and it may have methods, such as open, close, read, and write. In an object model, an object class is represented by a rectangle partitioned into three sections. The top section contains the class name, the middle section contains the attributes, and the bottom section contains the methods. Figure 2.1 depicts the notion graphically. For simplicity the attributes and methods are often not shown in most models.

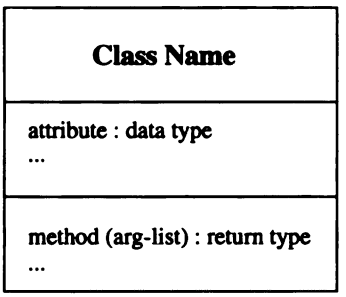

Figure 2.1. An Object Class in OMT Notation

In an object-oriented system object classes communicate with or have relationships with other object classes. These relationships are called associations. There are three types of associations in the object model notation: association (a simple relationship), aggregation (a stricter is-part-of relationship), and generalization (an is-a relationship). An association is represented by an arc (line) between the two object classes. An aggregation is represented by an arc (line) decorated on the aggregate class side of the arc by a diamond. A generalization is represented by an arc (line) decorated with a triangle pointing to the refined class. Figure 2.2 depicts each of these types of associations, where class A is associated with class B, class C is-part-of aggregate class D, and class E is a generalization of refined class F.

In addition, each association can be decorated with a multiplicity. A multiplicity specifies how many objects of one class may relate to a single object of another class. Figure 2.3 describes some of the multiplicities available.

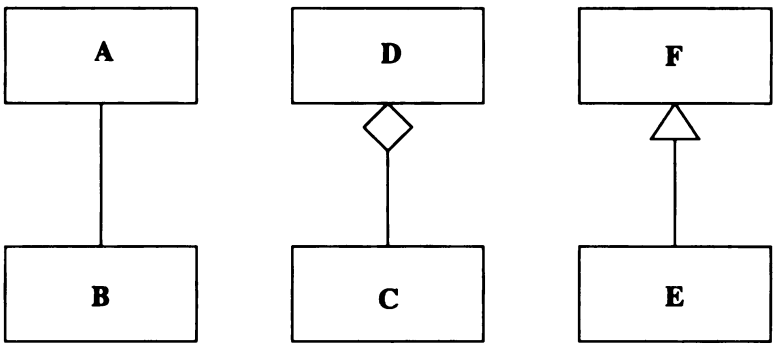

Figure 2.2. Three Types of Associations in OMT Notation

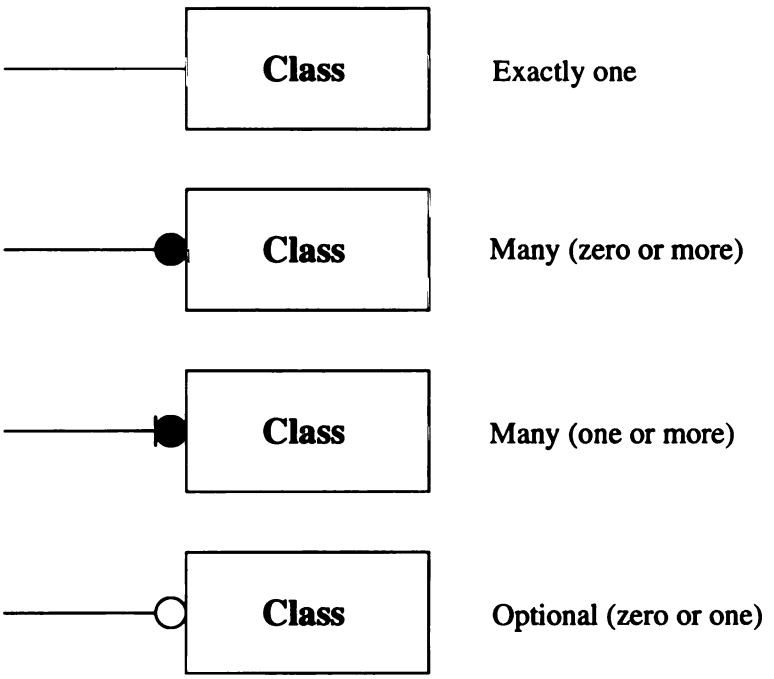

Figure 2.3. Multiplicities in OMT Notation

CHAPTER 3

A New Perspective on Configuration Management Concepts

The management and control of all items produced during the software development process continues to become more difficult as the size and complexity of software systems grows, compounded by the frequency in staff changes for a given project. Configuration management systems are developed to alleviate this difficulty. Numerous sources including software engineering texts [1, 13], configuration management texts [4, 14], papers [15, 16], and standards attempt to define the functional requirements for a configuration management system. Dart [16] defines the functionality requirements of a configuration management system as follows.

- **Component:** identifies, classifies, stores, and retrieves the components which make up the software product.
- **Structure:** defines the system architecture or structure of the software product.
- **Construction:** supports the construction of the software product and its artifacts.

- **Auditing:** reviews the progress of the software product and its process, to ensure policies are followed.
- **Accounting:** gathers statistics about the software product and development process.
- **Controlling:** controls how and when changes are made to the software product, artifacts, the development process, or management policies.
- **Process:** supports the management of the software product development.
- **Team:** enables a team of developers to develop and maintain a family of software products.

However, these requirements are fine-grained, where continued focus on individual requirements will not lead to significant advances in the area of software configuration management. By grouping the requirements into the following categories we are better positioned to identify the problems and propose solutions for configuration management.

- **Version Control:** Version control is the most important area of software configuration management. A *repository* is a fundamental notion of a configuration management system. The repository is the centralized library of files that provides version control for the system [16]. Furthermore, version control encompasses all other areas of software configuration management. The configuration management system requires a repository to store and retrieve all configuration management information, including source code, object code, diagrams, documentation, executables, and test cases and procedures. Therefore, advances in version control will have significant benefit to all configuration management systems.

- **Component and Item Identification:** Proper identification of all components and items of a software product, along with easy access to the items in the repository is an essential property for a software configuration management system.

- **Software System Structure:** Architectural representation of the complete software system architecture and the mechanisms for construction of the system and other artifacts, such as documents, is necessary for a software configuration management system.

- **General Management:** The general management of software development, including auditing, accounting, and controlling along with the team and process management is important to software configuration management. However, these areas of configuration management are less influenced by technological advances.

From these categories, technological advances would produce the greatest benefit in component identification, software system structure, and version control. Specifically, current configuration management system difficulties arise from three major problems: file localization, lack of system structure, and broad effects of change inherent in version control systems. As such, the focus of configuration management research needs to be placed on these concepts.

3.1 Localization

Configuration management systems must concentrate on three key phases (design, implementation, maintenance) of the software life cycle. One of the key factors that contributes to the difficulty of configuration management is the specialized focus of each of the three phases. That is, the system requirements deal with functionality,

system design and implementation deal with the system architecture, and system version control deal with files. More concisely, each step in the software life cycle is localized around a different concept, where *localization* refers to the process of gathering and placing items in close physical proximity to one another [11]. Software systems are developed with software components that are often organized into modules, abstract data types, or object classes according to their logical relation [17]. If we extend the concept of localization to include the placement of components in close logical proximity of each other, it seems only natural that a software configuration management system should localize its information around the different abstract software concepts.

Experience has shown that this lack of locality presents difficult problems, however, current systems are still developed using this approach. Specifically, version control systems localize around files [5, 6], leaving the software engineer with enormous configuration and maintenance problems at the system level. Commonly asked questions with current configuration management systems include [10]:

- What version of the file is this?
- How many files were affected by fixing this one bug?
- Are all the correct versions of files used in the current release?

In addition, by localizing configuration management information on the specific components of a system, the relationship between the components and their artifacts, such as requirements, test cases, or status accounting information, is preserved. Product representation is another area to be considered for configuration management investigations, of which this type of relationship information is included. For example, consider an object-oriented system in which a specific object class has been modified. By localizing on object classes, the key component of object-oriented development, it is straightforward to modify the design and/or test cases for that object

class while modifying its implementation code. In contrast, localizing on files would mean modifying multiple files, one or more for the implementation code, one or more for the design, and one or more for the test cases, which potentially leads to enormous file management problems. Therefore, given the size and complexity of current systems, this focus on files must be minimized or eliminated and replaced with a focus on a more localized concept if advancements in configuration management are to be made.

3.2 System Structure

Another problem with existing configuration management systems is that a complete version of the system may be composed of many different versions of the files. Additionally, a different version of the system may be composed of the same or different versions of the files that make up another version of the system. This approach imposes upon the developers the cumbersome task of remembering which file versions make up a given system version.

Likewise, current configuration management systems put little emphasis on the structure of the system components. Most methods are only concerned with the individual components that make up a system, and not with how these components interact with each other [3]. Clearly, the interaction between the components in a software system is essential to the understanding of the system as a whole. Furthermore, the complexity of most configuration management problems could be significantly reduced with a means to describe the software architecture that facilitates the implementation and propagation of changes [10].

Localization around the components of a software system provides the ability to represent important architectural information, such as data flow or control flow connections among the components The two most popular software development

methodologies, object-oriented analysis and design, and structured analysis and design provide graphical notations to depict the relationships among the components of a system. Specifically, object models and structure charts, respectively depict the system architecture and interaction among the components of the software system. Incorporating these types of concepts into a configuration management system would address many difficulties of current systems.

3.3 Fine-Grained Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software engineering process [1]. A version is essentially a snapshot of a configuration object at a moment in time. A configuration object can be any item created during the software development process, such as source code, documentation, maintenance reports, or test data. The storage, creation, and retrieval of versions of a configuration object are the major tasks associated with version control. Specifically, version control addresses questions critical to software development and maintenance [4].

- How should the system be structured so that different systems can be built to meet the requirements of different users?
- How should an old version of the system be preserved, for example, to investigate a fault?
- How can a version of the system be built so that it contains certain fixes but not others?
- How can many versions of an item be stored efficiently?

Investigation of these questions lead to the discovery of some important issues that help to identify requirements for version control. Specifically the issues, *the structure*

of the system, *a version of the system*, and *versions of an item* suggest that version control must be able to handle the individual items that make up the system, the system as a whole, and the structure of the system. As will be shown, much work is being done in the area of system version control, however, no method specifically addresses all of these requirements in a single framework. In addition, the emphasis of the version control remains on the individual files that make up the system. A consequence of this focus on files is that a small change to a part of the file results in the whole file being considered as changed. As such, the effects of a small change affects a large portion of the system. The effects of these changes need to be reduced to a minimum.

Many problems arise in version control due to the emphasis on files and the lack of emphasis on system structure. Subsequently, any understanding of the system and the abstraction obtained during the design of the system is completely lost during configuration management activities. Since a system can be composed of many files, an individual component may be implemented in one or more files, and many components may be contained in a single file, it becomes impossible to determine the system structure and how changes, updates, and fixes will affect the system as a whole. A simple change or modification to a file results in the entire contents of the file being considered modified. As such, components in that file may be unnecessarily marked as changed. The effects of the change can be enormous and unidentifiable. A version control system must alleviate these problems by allowing the structure of the system to be specified and the effects of change to be traced and limited.

Based on these difficulties, requirements for version control within a configuration management system can be stated succinctly with three high-level objectives. First, system level version control with the ability to describe and manage the structure of the system. Second, fined-grained version control to limit the affects of change and track the consequences of change on the system as a whole. Finally, localized version

control around the logical components being used. Chapter 5 develops a version control mechanism for object-oriented systems that meets these requirements.

CHAPTER 4

General Configuration Management Framework

Emphasis on three main concepts defines the basis for a new approach to a configuration management system: concept locality, limited effects of change, and system structure. Based on these concepts, requirements for a configuration management system can be stated as follows.

- Localization around the logical components being used,
- System level mechanisms with the ability to describe and manage the structure of the system, and
- Fined-grained version control to limit the affects of change and track the consequences of change on the system as a whole.

Analyzing the requirements leads to an OMT object model that is used to define the central structure of the configuration management framework. Figure 4.1 shows that a software *system* is made up of one or more *localized components* and one *system structure.* Additionally, the system structure defines the relationships between the one or more localized components.

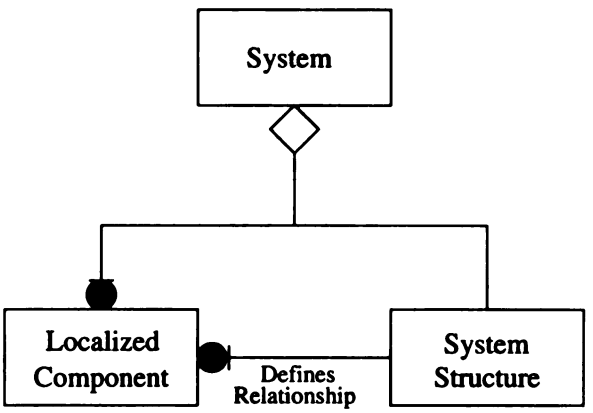

Figure 4.1. Analysis Model of General Framework Requirements

The relationship between configuration management and version control is tightly coupled. Configuration management is based and built upon a version control layer (repository). As such, the general framework depicted in Figure 4.1 is built upon the fine-grained version control mechanism needed to satisfy the third requirement. Furthermore, the actual nature of the fine-grained version control is dependent upon the localized concept. For example, in an object-oriented system an object class could be the localized concept and the version control mechanism must have specific functionality to support management of object classes. In contrast, in a structured system, a module or function may be the localized concept and the version control mechanism must have specific support for modules or functions. Therefore, it is difficult to develop a general model for the version control mechanism. However, for completeness, Chapter 5 will develop the fine-grained version control mechanism specific to an object-oriented system and show how the configuration management system makes use of it. As such, the remainder of this chapter will concentrate on all other aspects of configuration management.

Specific configuration management operations and concepts can now be added to the general framework of Figure 4.1. Continuing with an object-oriented analysis shows that the attributes and operations of each object class in Figure 4.1 provide the desired configuration management operations. Figure 4.2 shows a subset of the possible attributes and operations that can be added to the framework to construct a complete configuration management system. For example, each class may have a *description* or *design details* that may be collected and formatted into a design document and printed with a *print design document* method. In addition, *bug reports* and *status reports* could be attributes of the system as a whole, or attributes of the individual localized components. Finally, a method of the system may be to *build the executable.* The exact attributes and methods of the object classes in Figure 4.2 are organization and project dependent, based on the needs of the particular software system. The general framework allows the flexibility of each organization or project to tailor the configuration management to its particular configuration plan by introducing new attributes and methods for the system, system structure, and localized component.

Based on this configuration management framework, the following sections present specific applications of the general framework for object-oriented systems and structured systems, respectively.

4.1 Framework for Object-Oriented Systems

With the flourish of object-oriented analysis and design methods [7, 18, 19, 20, 21], the central abstract concept for localization is easily defined as an *object class.* An object class comprises its data (attributes), operations (methods), and interconnections (associations) to other object classes in the system. Figure 4.3 depicts the specific model based on the general framework. Notice the additions of two object classes,

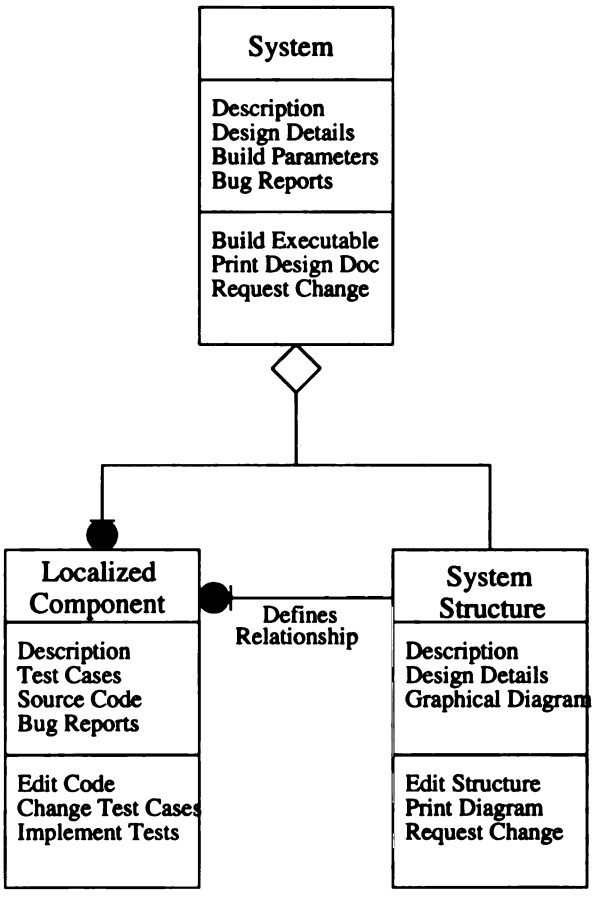

Figure 4.2. Addition of Configuration Management Operations

attributes and *methods*, since an object class is made up of many attributes and many methods. The model shows that a software *system* is composed of one or more *object classes* and an *object model* that defines the relationships between the object classes. Additionally, an object class is composed of zero or more *attributes* and *methods*.

There is no generic structure for an object-oriented system, since each object-oriented development methodology has its own slightly different representation of the system architecture. Therefore, for completeness purposes, a specific example, the OMT object model, that describes the structure of the object classes in OMT is used, as shown in Figure 4.3.

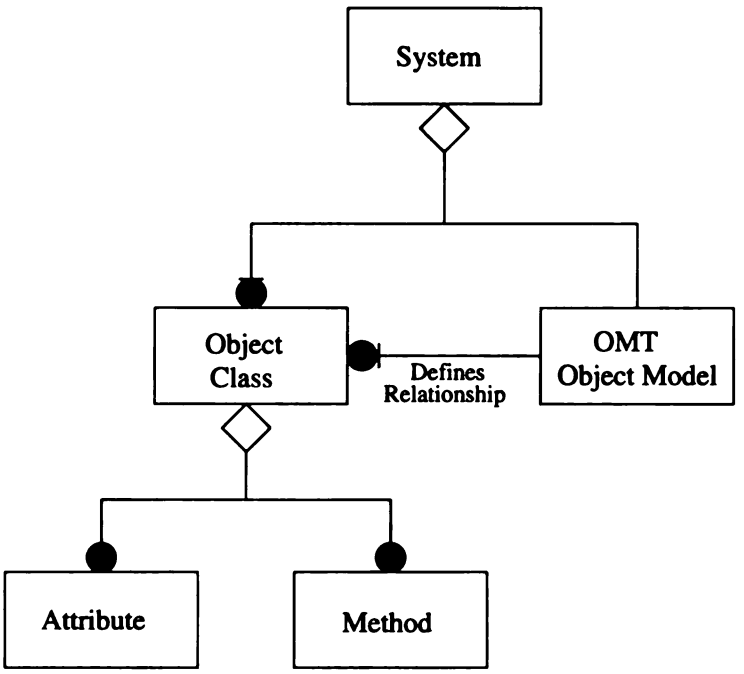

Figure 4.3. General Framework for Object-Oriented Systems

Specific configuration management operations and concepts can now be added to the object-oriented system framework of Figure 4.3. The *system* as a whole may have a description, requirements, bug reports, and status reports as attributes, with methods including, build executable, request change, and print design document. The *object class*, *attribute*, and *method* classes may all have similar attributes to the *system* class. In addition, the attribute class may have attributes for data type and description of the data type if it is a complex structure. The method class may have attributes for source code, arguments, and return type, and a method to compile the source code into object code. The *object model* may have attributes for the diagram, design decisions, or descriptions of the diagram, along with methods for producing a postscript diagram or editing the diagram. Again, as with the general framework, the exact attributes and methods of each class in the model must be defined by the

organization or project.

Based on this configuration management framework for object-oriented systems, Chapter 5 develops the fine-grained version control mechanism. Chapter 6 describes the implementation of a prototype proof of concept system that integrates the configuration management framework upon the version control mechanism.

4.2 Framework for Structured Systems

The widely and successfully used structured design method [22] is based on functional components (modules). The foundation on components easily defines the central localized concept to be a *module.* Figure 4.4 depicts the specific model based on the general framework. Unlike the object-oriented development methodologies, the generic structure of a structured system is a *structure chart.* A structure chart is a graphical means of showing the component structure of a software system [13]. Therefore, Figure 4.4 shows that a software *system* is composed of one or more *modules* and a *structure chart* which defines the relationships between the modules.

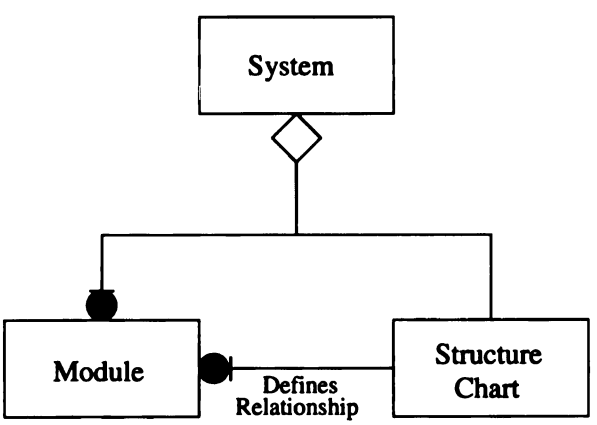

Figure 4.4. General Framework for Structured Systems

Specific configuration management operations and concepts can now be added to the structured system framework of Figure 4.4. The *system* as a whole may have a description, requirements, bug reports, and status reports as attributes, with methods including, build executable, request change, and print design document. The *module* class may have attributes for description, source code, arguments, and return type and a method to compile the source code into object code. The *structure chart* may have attributes for the diagram, design decisions, or description of the diagram, along with methods for producing a postscript diagram or editing the diagram. As with the general and object-oriented framework, the exact attributes and methods of each class in the model must be defined by the organization or project.

CHAPTER 5

Object-Oriented Version Control

One of the primary requirements needed by a configuration management system is a fine-grained version control mechanism. Based on our object-oriented configuration management framework defined in the previous chapter, a user concentrates only on the system and object classes. The concept of version control should only be visible at the abstract level of the system. In order to provide the fine-grained version control needed to limit the effects of change, a three-tiered version control model for object-oriented systems has been developed. Additionally, this model satisfies the requirements for object localization and system structure. Specifically, object class locality, a three-tiered fine-grained version control mechanism, and a system revision history based on the system structure, respectively are used to address each requirement.

5.1 Object Localization

An object-oriented system is already localized around one key concept, an object class. Accordingly, the version control model is based on the concept of an object class and uses the property that a software system is composed of many object classes with some specified inter-object structure. An object class is typically defined as an entity

that has data (attributes) and operations (methods). By performing object-oriented analysis on the definition of an object class and the definition of an object-oriented system, we have the basis for the model for object-oriented version control. Figure 5.1 shows an analysis model of an object-oriented software system in OMT notation [7]. This model states that an object-oriented *system* is composed of one or more *object classes*, where each object class is composed of zero or more *attributes* and zero or more *methods*.

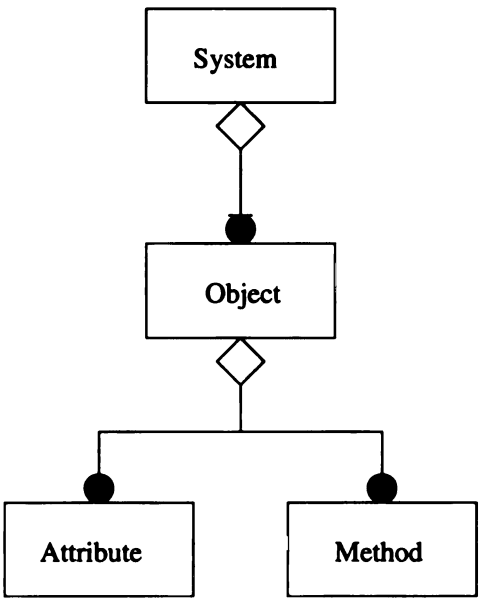

Figure 5.1. Analysis Model for an Object-Oriented Software System

Based on this model, it is a simple transformation to an analysis model to capture the notion of version control of an object-oriented system. The addition of "*version of*" to each object in the model yields the analysis model for a version of an object-oriented software system. That is, a *version of* an object-oriented *system* is

composed of one or more *versions of objects*, each object being composed of zero or more *versions of attributes* and zero or more *versions of methods.* Figure 5.2 depicts the transformation to the analysis model of a version of an object-oriented software system. Therefore, based on this strategy, all version control is centered around an object, where a system is composed of objects and their versions.

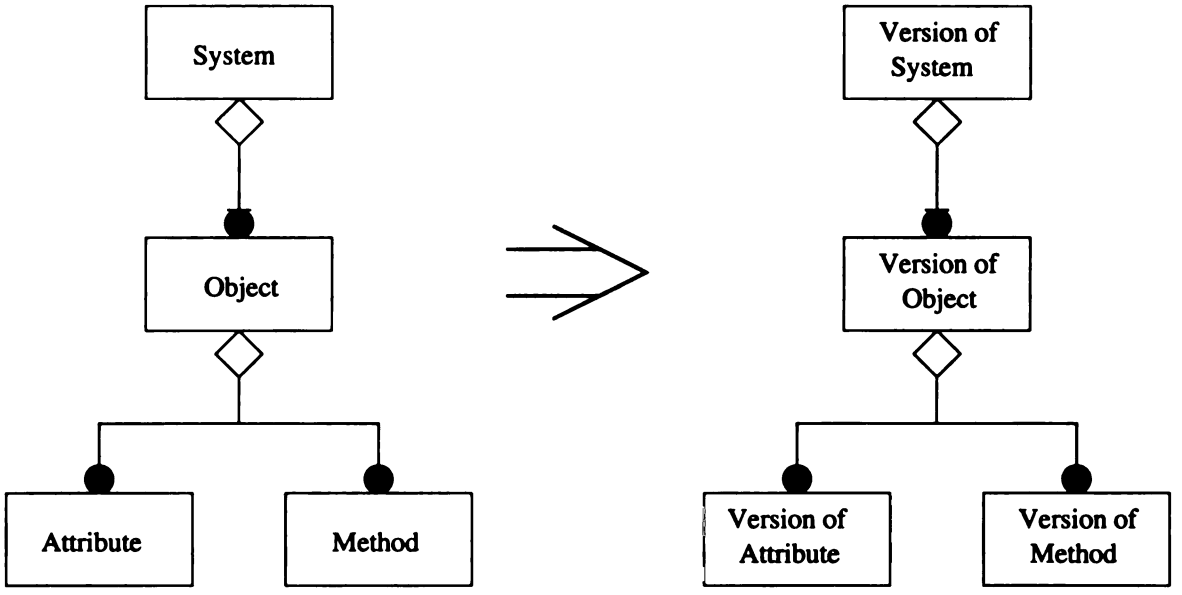

Figure 5.2. Transformation from an Analysis Model for an Object-Oriented Software System to an Analysis Model for a Version of an Object-Oriented Software System

5.2 Three-tiered System Revision History

The next step is to incorporate into the model the concept of a system revision history. The tree representation of version control seems to allow the most flexibility. Not only does it have proven success with systems such as RCS [23], other systems

have used the tree approach to model such hierarchies [8]. Additionally, a version of the system revision history must have the ability to define a system structure. The prototype described in Chapter 6 uses the OMT object model, as did the example in the previous chapter, for the system structure. However, a generic use of system structure for object-oriented systems is now explicitly maintained in order to facilitate the applicability to other object-oriented modeling approaches. Therefore, a *system version* has a *structure* for the *objects* of which it is composed.

Figure 5.3 adds both the structure and tree representation to the evolving model of object-oriented version control. The model now shows the integration of localization, system version control, and structure into a single version control framework. Each *version of the system* is a node of a *system version tree* that contains zero or more nodes. Each version of the system is then composed of one or more *versions of objects* and a *structure* that defines the relationship of those objects to produce the specific version of the system. Finally, each version of an object is composed of zero or more *versions of attributes* and *versions of methods.*

Based on this model, there is the need to determine and limit the effects of change. We build on concepts developed by Magnusson et al [17] to develop a fine-grained approach to the system level version control model. In order to accomplish this task, the granularity and the hierarchical levels for the model need to be determined.

The system revision history is the lop level (tier) of the version control model and is represented by a tree. Further application of the localized concept leads us to consider an object class, where a given system version can have many versions of an object class, thus making object class versions the second level (tier) of the revision hierarchy. Attributes, considered in isolation, are not typically interesting. Attributes are usually simple data types and undergo very little configuration management. However, if attributes are changed, the change should be considered as part of an object class change, which would be captured by the object class version tier.

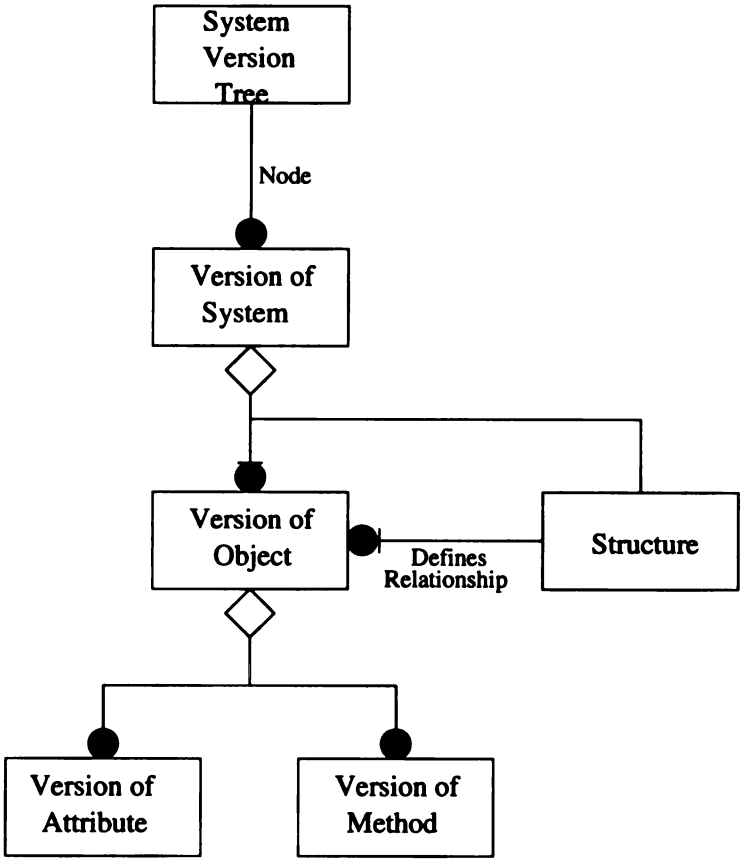

Figure 5.3. Analysis Model for System Level Version Control of an Object-Oriented Software System

In contrast, methods for an object class may undergo many changes. There may be new methods, errors to fix in existing methods, modifications to existing methods, etc. In fact, methods have been the target of many "file" version control approaches [6, 23]. Accordingly, the individual methods become the final and most fine-grained version control level (tier) in our revision hierarchy. In order to proceed to a finer level of granularity, individual statements or tokens of a particular programming language need to be considered, which would violate the objective to keep this model generic with respect to implementation.

By combining the three levels of granularity with the decision to use tree-based representation of versions, integration of the fine-grained based version control into the model is accomplished. Also, observing that each individual object and each individual method have version trees and that the system has one version tree, the complete model for object-oriented version control is defined. Specifically, *object-oriented version control* consists of a *tree of system versions*, one or more *trees of object versions*, and zero or more *trees of method versions*. Figure 5.4 depicts the complete analysis model.

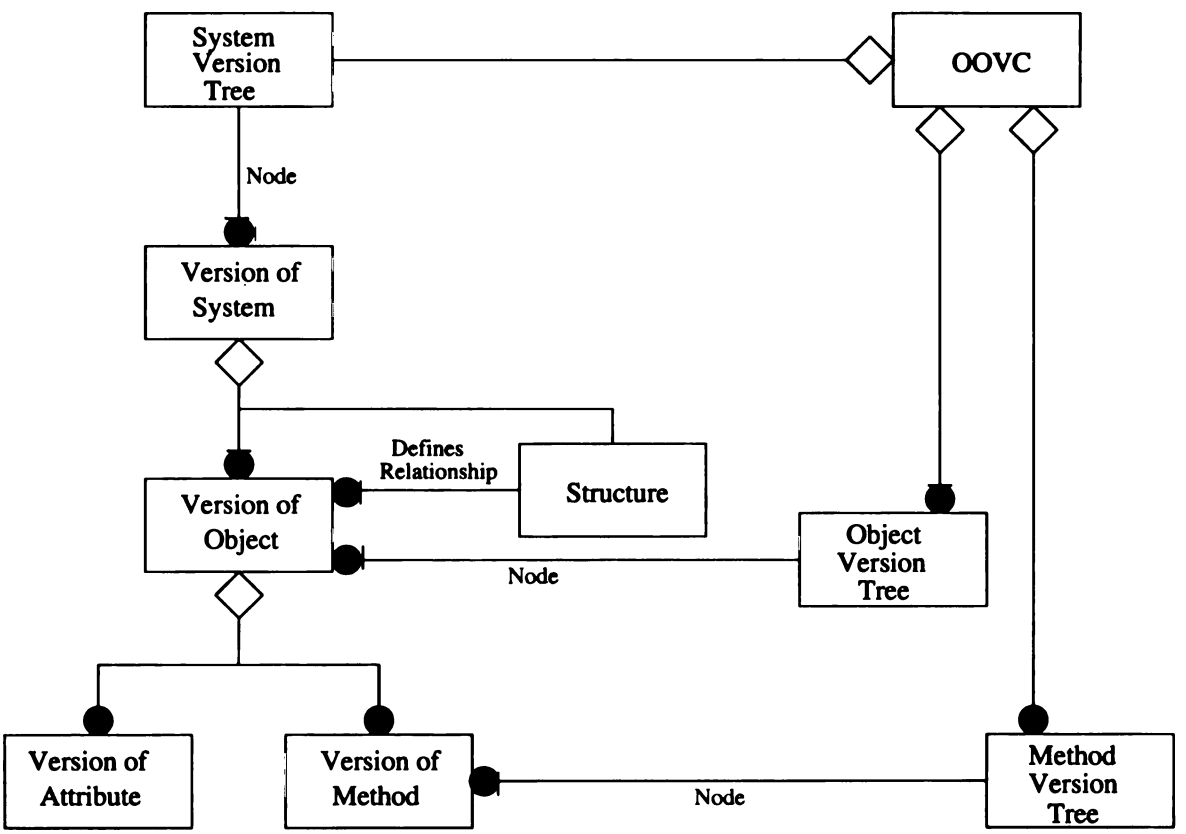

Figure 5.4. Analysis Model for Version Control of an Object-Oriented Software System

5.3 Formal Model of Object-Oriented Version Control

At first glance, the analysis model depicted in Figure 5.4 may not appear to be intuitive. This problem is an inherent weakness in many informal and graphical modeling techniques such as OMT. Therefore, in order to supplement the graphical notation, concise, formal definitions are presented in order to eliminate ambiguity and confusion.

Formal Model

1. We define object-oriented version control as a three-tuple $OOVC(T, O, M)$, where T is a tree, O is a non-empty set of trees, and M is a set of trees.
2. We define T to be a tree of system versions. Each vertex of tree T is a three-tuple $V_T(N_T, S, C)$, where N_T is the vertex identifier, S is the system structure, and C is a non-empty set of two-tuples. Each vertex represents an individual system version.
   (a) Each member of the set C is identified by C_i where $1 \leq i \leq cardinality(C)$.
   (b) We define each C_i to be a two-tuple $C_i(O_j, N_O)$, where O_j is an identifier to the object tree in set O and N_O is an identifier to the vertex N_O of object tree O_j, where $1 \leq j \leq cardinality(O)$.
3. We define O to be a non-empty set of trees where each tree represents an individual object class. Each member of set O is a tree identified by O_j where $1 \leq j \leq cardinality(O)$. Each vertex of tree O_j is a three-tuple $V_O(N_O, A, F)$, where N_O is the vertex identifier, A is a set of attributes, and F is a set of two-tuples. Each vertex of tree O_j represents an individual object version.
   (a) Each member of the set F is identified by F_k where $0 \leq k \leq cardinality(F)$.
   (b) We define each F_k, $k \geq 1$, to be a two-tuple $F_k(M_l, N_M)$, where M_l is an identifier to the method tree in set M and N_M is an identifier to the vertex N_M of method tree M_l, where $0 \leq l \leq cardinality(M)$.
4. We define M to be a set of trees where each tree represents an individual method of some object class. Each member of the set M is identified by M_l

where $0 \leq l \leq cardinality(M)$. Each vertex of tree M_l, $l \geq 1$, is a two-tuple $V_M(N_M, B)$, where N_M is the vertex identifier and B is the body of the method. Each vertex of tree M_l represents an individual object version.

The above definition is depicted pictorially in Figure 5.5, where a few links have been omitted and other links labeled for clarity. The dashed boxes depict the object-oriented version control three-tuple. The trees represent either a system version tree, object version tree, or method version tree based on its corresponding level (T,O,M). The dotted lines represent the connections between the levels and correspond to the C and F tuples depending on the level.

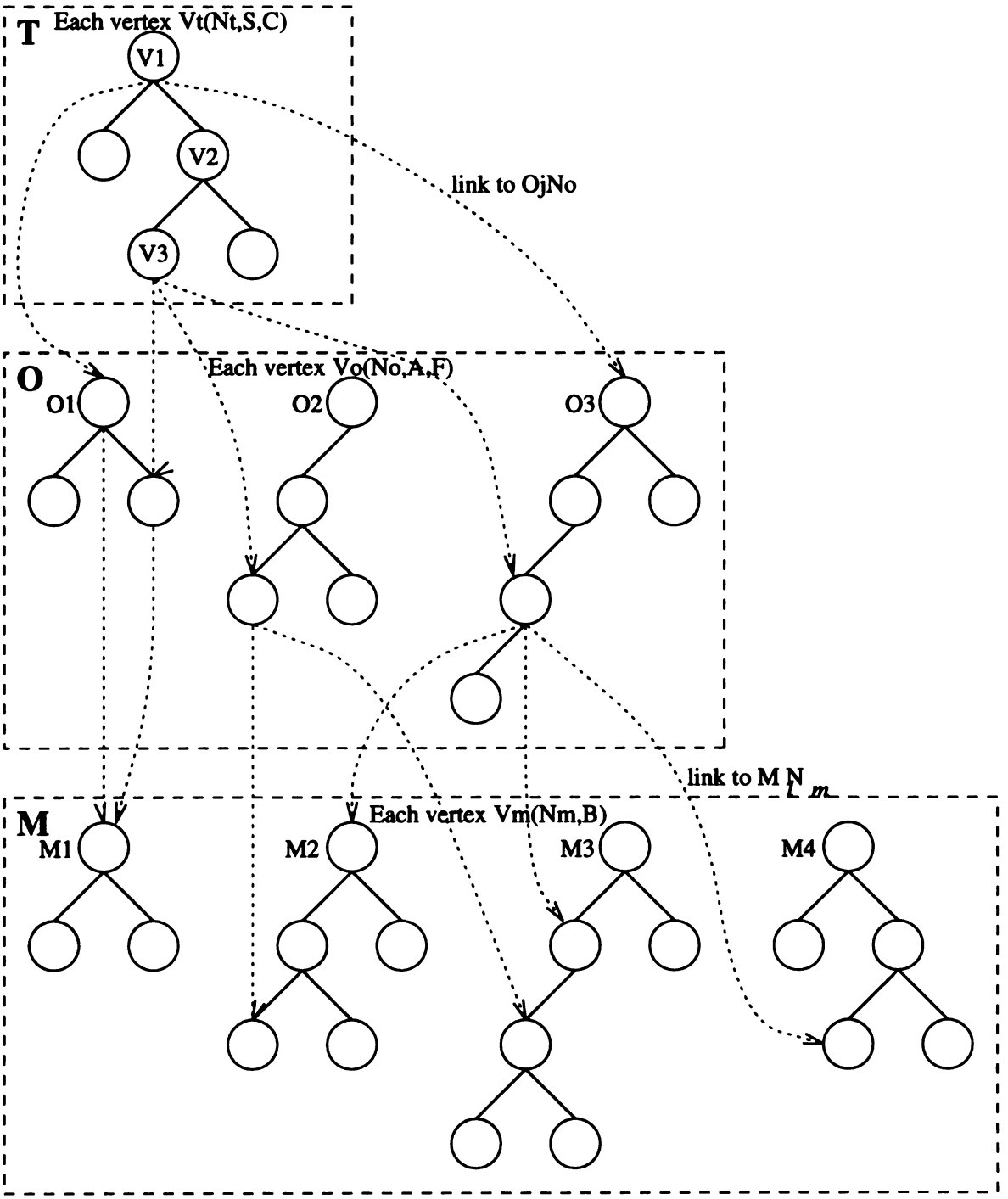

Figure 5.5. Graphical Depiction of Formal Model

5.4 Operations on the Version Control Model

The two most essential operations for a version control model are storage and retrieval, more commonly known as *checkin* and *checkout* [23], of the configuration items. These two operations need to be defined for the object-oriented version control model previously developed. For clarity, the notation and graphical representation of the formal model defined above will be used. Since describing the retrieval of configuration items from the version control model is largely based on the understanding of the contents of the version control model, the storage, or checkin, of items is first described.

Figure 5.6 depicts the version control model of an initial creation of a software system. As can be seen, the system has one system version $V1$ with two object classes $O1$ and $O2$, each being version $V1$. Object class $O1$ has one method, version $V1$ of method $M1$. Object class $O2$ has three methods, versions $V1$ of methods $M2$, $M3$, and $M4$.

Suppose that an error is discovered in method $M1$. In order to correct the error, a new version of the method is created and then stored in the version control model. Figure 5.7 shows the resulting version control model after this operation. Method $M1$ now has a new version $V2$. Notice that object class $O1$ also has a new version $V2$. This upward propagation of version creation is due to the logical implications of modifying the method. Version $V2$ and version $V1$ of object class $O1$ are slightly different in that a method of the object class has changed. Likewise, the system also has a new version $V2$ for similar reasons. Version $V1$ of the system has an error, whereas version $V2$ does not have the error. This propagation is reasonable in practice, since a change or revision of part of a system actually results in a new version of the system as a whole. Notice, however, that the object class $O2$ and its methods have not changed, limiting the propagation of change and storage space needed to store multiple versions of the system. The version propagation can be stated as:

A version at any level of the version control model is changed and a new version is created if its contents or the contents of any of its parts are changed.

In order to fully understand the storage mechanism, consider a more complex example. Suppose that a major enhancement of the system is created which requires the introduction of a new object class and the modification of other object classes. Figure 5.8 shows one such example where a new system version $V3$ is created. Object class $O1$ undergoes no change, a new object class $O3$ is introduced with one method $M5$, and object class $O2$ undergoes changes to two of its methods $M2$ and $M3$.

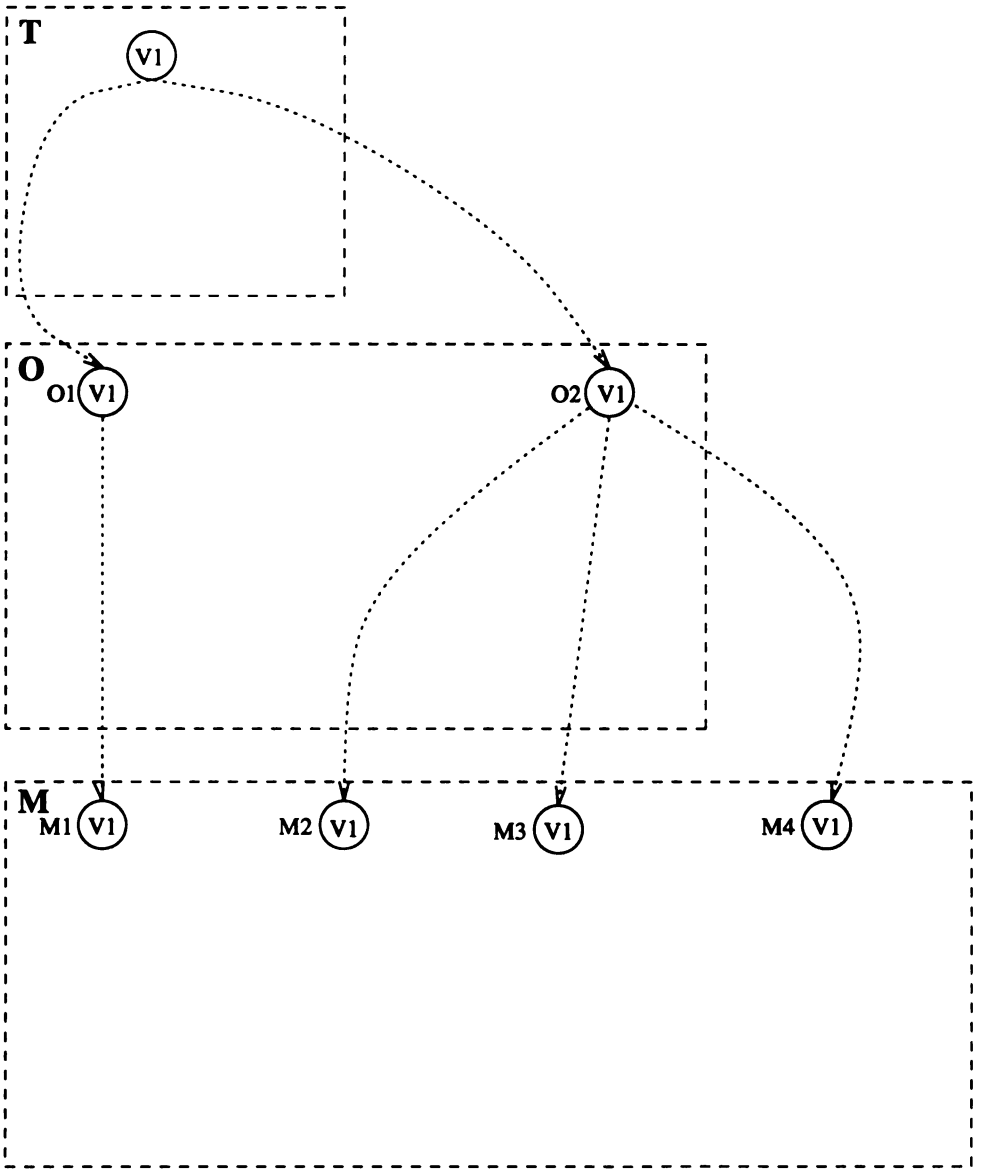

Figure 5.6. Initial Version of System

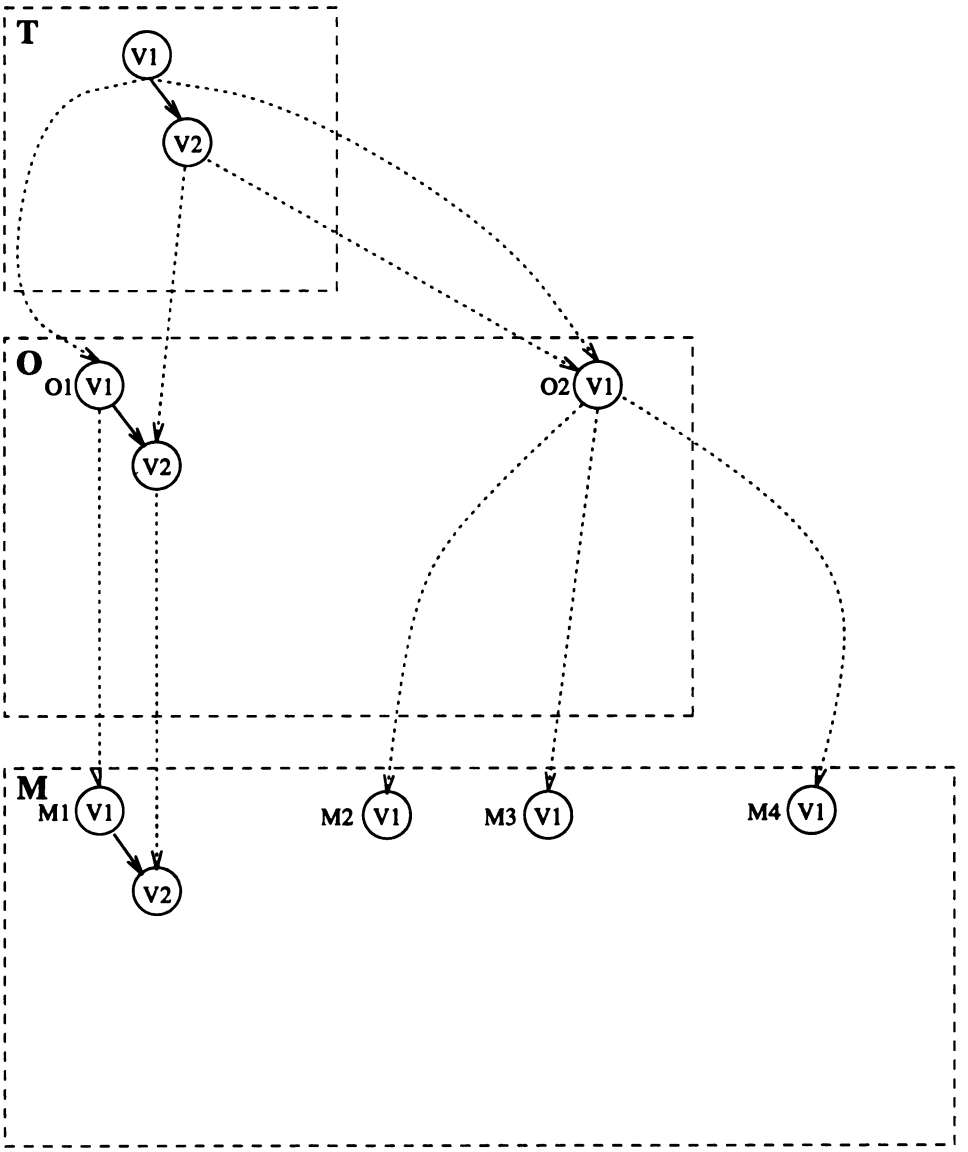

Figure 5.7. First Revision of System

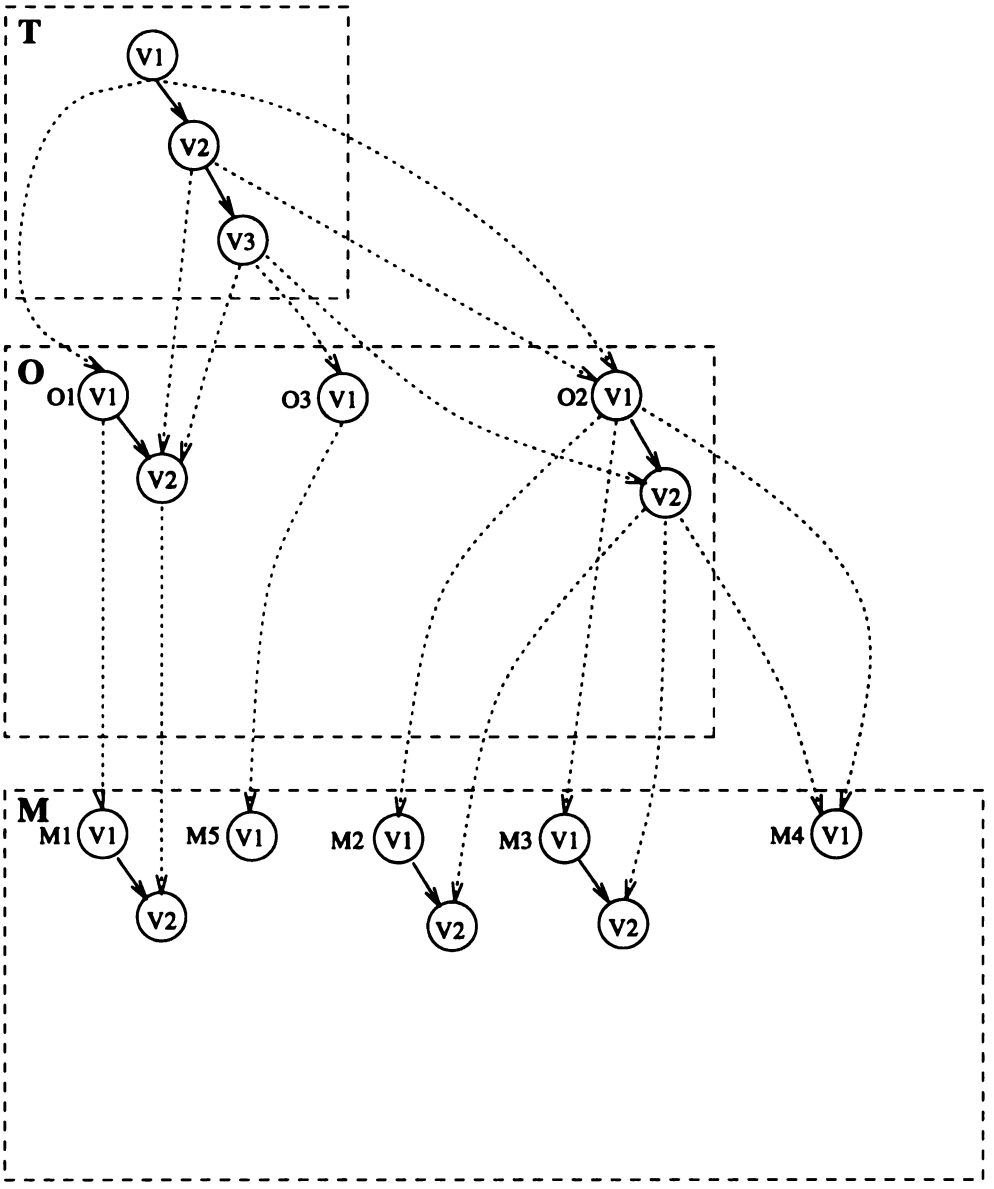

Figure 5.8. Second Revision of System

Based on the understanding of the storage model, the retrieval of configuration items can now be described in more detail. Due to the version propagation stated above, any change to a system or part of a system will result in a new version of the system. Therefore, it seems illogical to retrieve an individual object class or method since any change to that item will result in the need for higher levels (tiers) of the model to be retrieved and there is no backward referencing mechanism. That is, an individual method does not *know* to which object classes it belongs, and likewise, an object class does not *know* to which versions of the system it belongs. Therefore, the only retrieval operation visible to the user of the version control model is at the system level. That is, the user only retrieves versions of a system. All other retrieval is handled internally by the version control model. Specifically, a particular version of a system retrieves its object classes, and object classes retrieve their methods. Additionally, a version of a system retrieves all other information contained in its node such as the system structure, and object classes and methods retrieve all information contained in their nodes such as attributes and source code, respectively.

CHAPTER 6

Implementation Example

We have developed a prototype Software Configuration Management (SCM) system to aid in the proof of concept. The SCM prototype was developed such that the prototype system provided the configuration management for itself. This bootstrapping approach has enabled us to use the SCM system to further develop and maintain itself and also to assess the usefulness and scalability of the system to a larger example. In order to describe the overall framework and use of the system we provide a simple scenario.

6.1 User Interface

Upon invocation of the SCM system and the loading of a project, the software engineer (herein referred to as the user) is presented with the system revision history, as shown in Figure 6.1. It is important to note that the versions *do not* represent file versions as we typically expect in systems such as RCS [5] and SCCS [6]. The versions represent *system versions.* For example, in Figure 6.1, version 1.1 may represent the Solaris Operating System version, version 1.2 the SunOS 4.1.x version, and version 1.3 the HP-UX version of the *system.* Thus, the branches (such as 1.1.2.2) may represent the latest release of the Solaris version of the system.

The system version tree is modeled after the RCS versioning mechanism [23], in which a particular version can have one main trunk of revisions and many parallel branches (variants). Each version of the system has attributes and methods associated with it. For this particular implementation, a system version has the attributes *name* and *description* and methods *create revision*, *create variant*, *change version name*, *edit/read description*, and *build version*, as shown in the menu of choices in Figure 6.1.

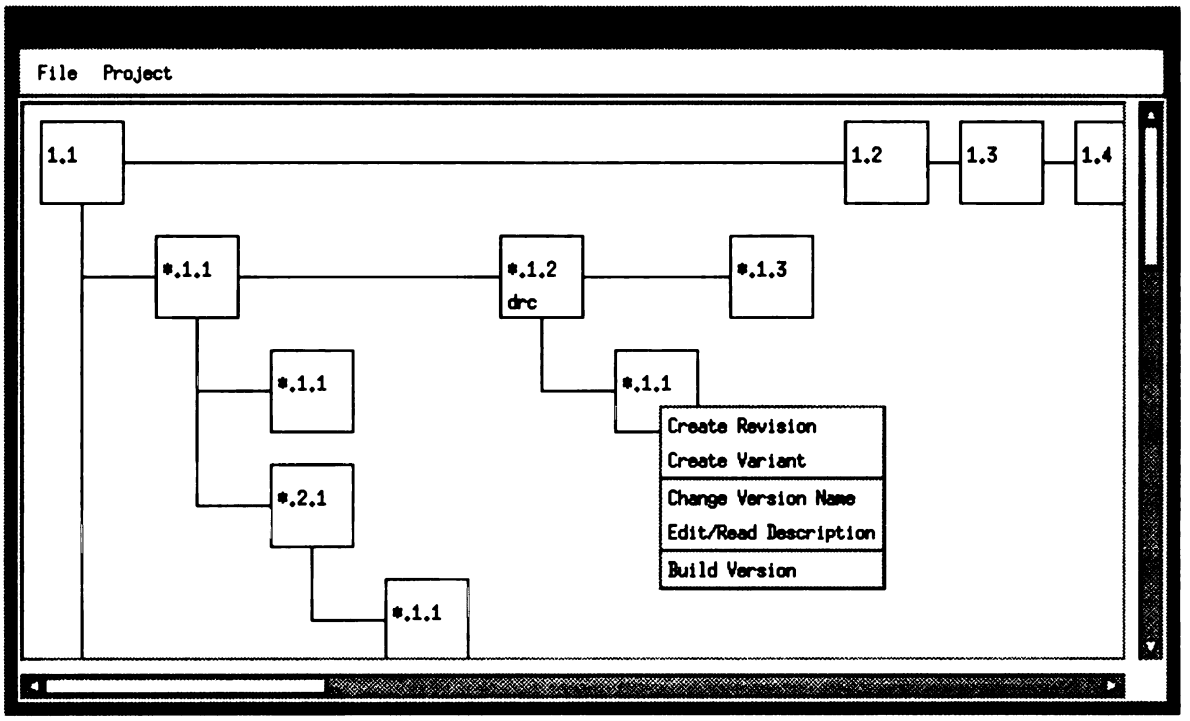

Figure 6.1. System Revision History Tree

The user selects the operation to perform by selecting the choice from the menu. By selecting *create revision* or *create variant* the user selects which version of the system to modify. Once the user selects a system version for modification, the OMT object model for the version is displayed. For example, suppose the user selects

a version (from Figure 6.1) of the current project. Then the object model for that system version will be displayed as shown in Figure 6.2. Now the user can proceed with any stage of the life cycle (design, implementation, maintenance) by concentrating on the object classes of the system. Each object could have a variety of relevant information, including a description, processing narrative, data dictionary, methods, attributes, preconditions, postconditions, etc. that correspond to the relevant sections of documentation or source code for the system. This particular implementation has only an *object name*, *description*, *header information*, *attributes*, and *operations* (methods) associated with each object class, as seen in the menu in Figure 6.2. The user can then change or modify any of these items including changing the object model itself. The list of editing operations on the left side of the window of Figure 6.2 show the options available in this particular implementation. The most common modification will be to change the methods of the object. The methods of an object typically refer to the actual source code. Modifications at this level allow the user to concentrate on a very focused task, a single method for an object, and provides the fine-grained version control of the system since only the method is changed. In addition, this approach allows multiple users to modify different methods of the same object class without file locking conflicts.

After completing the modifications, the user can then submit the changes into a new system revision in the revision hierarchy and rebuild the system. In the actual implementation of the build, the system will eventually have to be based on the concept of files. The SCM system automatically generates the corresponding C++ files and Makefile needed to build the system. The user specifies which system version to build by selecting the version on the system revision history graph (Figure 6.1). Throughout the modification and build processes, there is no representation of the actual files presented to the user. The user concentrates only on system level versions and object classes providing abstract concept localization.

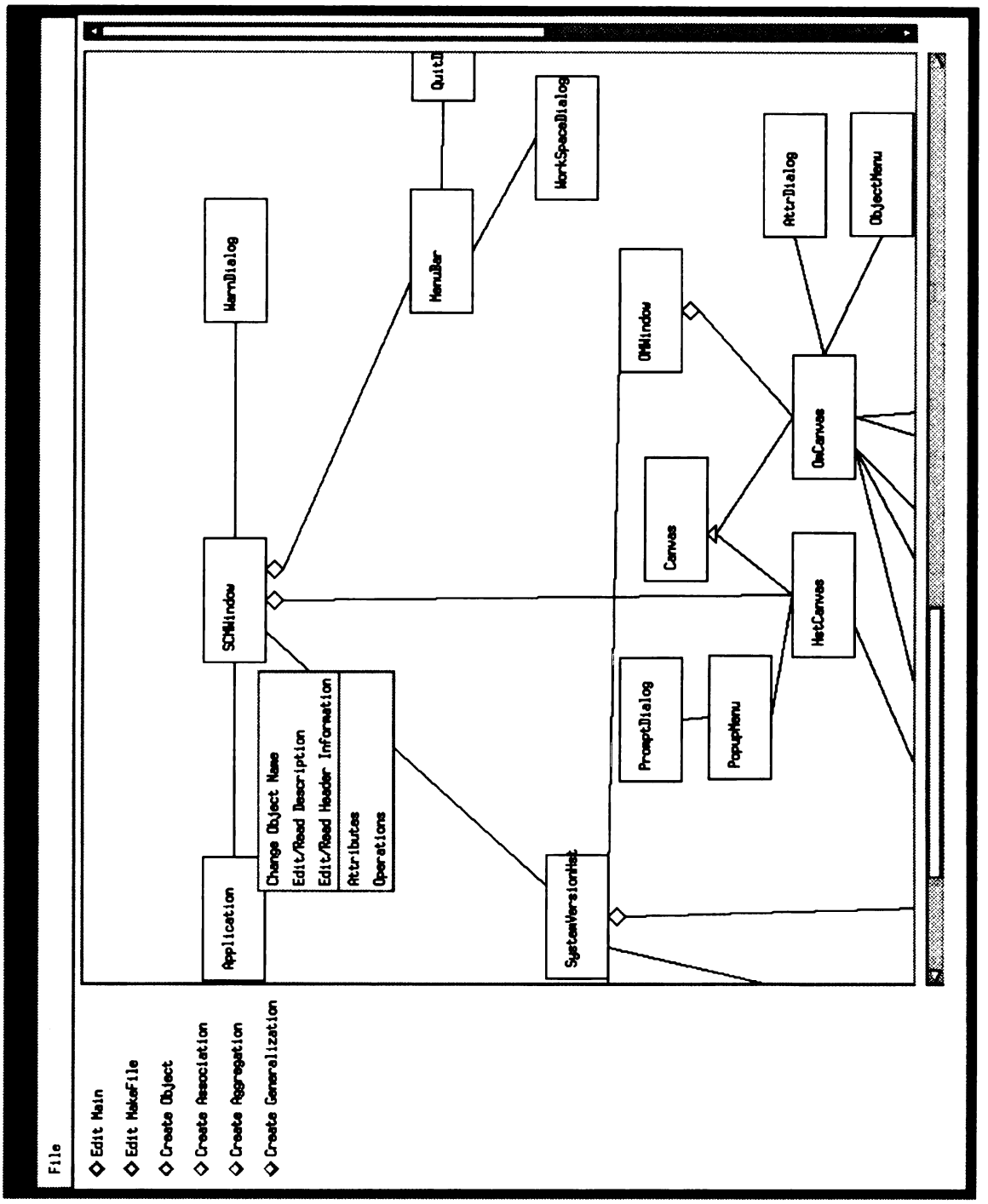

Figure 6.2. Object Model in Current SCM Prototype

It can be seen from the prototype SCM and scenario that the fine-grained version control, system structure, and concept localization concepts of the configuration management framework are provided and produce a usable development and maintenance environment based on a strong configuration management foundation.

6.2 Code Construction

One primary function of the SCM system has not been discussed thus far. This function is the construction of source code files for compilation. In order to do so, the system not only must extract each fine-grained piece of code from the version control mechanism, it must also construct the files in a manner that preserves the system structure. Our current prototype constructs C++ code based on the OMT object model. In order to generate the code structure, specific semantics must be explicitly defined for the three types of associations that are allowed in an OMT object model. These semantics are specific to a C++ implementation of an OMT object model and are not general rules for all implementation languages or system structures.

Review of the OMT methodology shows that there are three distinct association types between object classes. Therefore, we give our interpretations of what each type means in a C++ implementation.

- **Association:** Associations are inherently bidirectional [7], but they do not have to be implemented as such. That is, a pointer may be used in a unidirectional fashion. The difficulty is that it is not straightforward or possible to determine the direction of the association without background knowledge and experience. Therefore as a convention, we implement all associations as bidirectional.

- **Aggregation:** We treat aggregation as a tightly coupled form of association with some added semantics [7]. These added semantics include *transitivity*, *antisymmetry*, and that characteristics (associations, attributes, operations) may

propagate to the component class. From this definition, we give the following rules for code construction:

1. All associations of the assembly class are propagated to the component class(es), and recursively to all sub-component classes.
2. The reverse of item 1 is not true.

- **Generalization:** In C++, the standard inheritance constructs provide our implementation semantics. That is, if an object class is associated to a subclass object, then by default it is associated to the superclass indirectly through the subclass. Additionally this relationship is also recursive in nature as with aggregation. However, the reverse is not true. That is, an object class associated to a superclass is not considered to be associated to its subclasses. This interpretation suggests that a generalization is only unidirectional.

Based on these rules, the system structure of the OMT object model can be preserved by correct construction of the C++ header files. Figures 6.3 and 6.4 give the object model and source code, respectively, for an example that makes use of aggregation and association. As seen in Figure 6.3, class A is associated to class B and has component class C. Class C is part of class A and is associated to class D. Based on the above rules the following structure needs to be preserved.

- Class A is associated to classes B and C.
- Class B is associated to class A.
- Class C is associated to classes A, B, and D.
- Class D is associated to class C.

Figure 6.4 gives skeletons of the C++ headers that preserve this structure. Associations are implemented as pointers to the associated objects, for example class A is

associated to class B and C. Therefore, class B and class C are declared, then class A is defined with pointers to class B and class C. The definitions of the associated classes are included at the end to avoid circular includes in the preprocessor due to the bidirectionality of the association. In order to implement the propagated association of class B to C from the aggregate class A, the same implementation is used except class C does not contain a pointer to class B. The declaration and definition of class B is propagated to class C due to the propagation, however, the pointer to B is not since this would imply a direct association to class B. The definitions of classes B and D are straightforward.

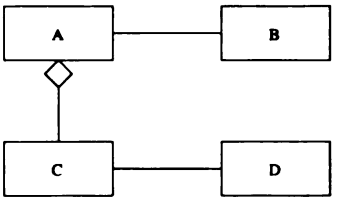

Figure 6.3. Example of Aggregation and Association

```
#ifndef __A_H__
#define __A_H__

class B;
class C;

class A
{
      B  *theBobject;
      C  *theCobject;
};

#include"B.h"
#include"C.h"
#endif
```

```
#ifndef __B_H__
#define __B_H__

class A;


class B;
{
      A  *theAObject
};


#include"A.h"

#endif
```

```
#ifndef __C_H__
#define __C_H__

class A;
class B;
class D;
class C
{
      D  *theDobject;
      A  *theAobject;
};

#include"A.h"
#include"B.h"
#include"D.h"
#endif
```

```
#ifndef __D_H__
#define __D_H__

class C;


class D;
{
      C  *theCobject
};


#include"C.h"

#endif
```

Figure 6.4. C++ Header File Example of Aggregation and Association

Similarly, Figures 6.5 and 6.6 give the object model and source code, respectively, for an example that makes use of generalization and association. As seen in Figure 6.5, class A is associated to class B and has subclass C. Class C is refinement of class A and is associated to class D. Based on the above rules the following structure needs to be preserved.

- Class A is associated to class B.
- Class B is associated to class A.
- Class C is associated to class D and a subclass of class A.
- Class D is associated to class C.

Figure 6.4 gives skeletons of the C++ headers that preserve this structure. The definitions of classes A, B, and D are straightforward. In order to implement the subclass C, the complete definition of class A need to be included before the definition of class C. The remainder of the definition is straightforward.

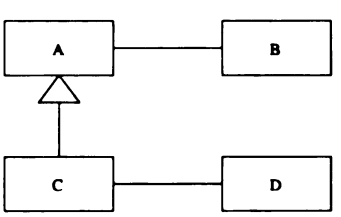

Figure 6.5. Example of Generalization and Association

```
#ifndef __A_H__
#define __A_H__

class B;

class A
{
      B  *theBobject;
};

#include"B.h"

#endif
```

```
#ifndef __B_H__
#define __B_H__

class A;

class B;
{
      A  *theAObject
};

#include"A.h"

#endif
```

```
#ifndef __C_H__
#define __C_H__

#include"A.h"
class D;

class C : public A
{
      D  *theDobject;
};

#include"D.h"
#endif
```

```
#ifndef __D_H__
#define __D_H__

class C;


class D;
{
      C  *theCobject
};

#include"C.h"
#endif
```

Figure 6.6. C++ Header File Example of Generalization and Association

CHAPTER 7

Related Work

Many configuration management systems have been developed with a breadth of functionality features. A commonality of these systems is the use of a repository, or version control system to manage the various configuration items. Therefore, we give an overview of the approaches used to provide version control as well as an overview of various configuration management systems.

7.1 Version Control Systems

Advances in software engineering methods, such as techniques for structuring systems, higher level programming languages, and object-oriented analysis and design methods have enabled programmers to better structure their applications by partitioning them into manageable pieces [10]. These pieces are the main configuration items for a version control mechanism. We first present an overview of methods for version control of these individual items that we will call components. These methods range from the simple text-based editor management [24] to the more complex file delta methods [5, 6, 23] and extensive database management systems [25]. The primary weakness of these, what we will call *component-based approaches*, is the absence of system abstraction and comprehension. Numerous *system-based approaches*

have been developed to eliminate this weakness [8, 9, 17, 25, 26]. We point out the major inadequacies for object-oriented systems, based on the requirements for object-oriented version control, while presenting an overview of these approaches. Finally, we give an overview of related work in alternative methods to version control that are not based on the concept of a version.

7.1.1 Component Version Control

We first consider the component-based approaches to version control. These approaches divide the software system into components. Each component could range from a single function of source code to an entire design document, the granularity and structure of the components is determined by the development team. The most common decomposition and implementation of these methods is based on files. Each component is stored in its own file, and the version control mechanism manages the individual files. Database management systems have also been used, however, at the component level the techniques are essentially modeling versions in the database with records or tuples extended with timestamps or a unique version number [25]. Configuring a system then consists of selecting a version for each component of the system. This approach to constructing a system becomes a major problem for these methods.

Text Representation

Narayanaswamy [24] proposes a text-based approach to version control based on contexts that the developer describes as a hypertext-like structure. A *context specification* is used to describe a particular variant or version of the file. In order to create a version, a special notation is used to describe the version where there is accompanying text to assist the tracking of the version. The major disadvantage to this approach is the highly complex and potentially confusing structure of the text file. The approach is similar to using #ifdef and #ifndef in C, and in our experience has led to great

confusion. In addition, this approach does not address or satisfy the requirements for an object-oriented version control method.

Set Representation

Rochkind [6] proposes what can conceptually be viewed as a set-based approach to version control. The approach treats each component as a set of related sequences, with each member of the set representing a version of the component. Every time a component is changed, a new version is created and inserted into the set. Each version is identified by an identification tag in the form *release.level*, where release represents the release number and level represents the level number of the version. The set is stored in a repository file using a *delta* storage technique, where a delta represents "the changes". This approach saves substantial space since only the changes are stored. The main drawback of this approach is the need to select the correct versions of each component for the system. Since this approach is based on components, it could satisfy the locality requirement of our model. However, current practice shows that a file is still the most common component.

Tree Representation

Tichy [5, 23] proposes what can conceptually be viewed as a tree-based approach to version control. The approach is similar to the set-based approach described previously, but adds functionality based on the tree structure and improves the efficiency of retrieval. The revision tree has a main branch, called the trunk, along which revisions are numbered similarly to the set approach (e.g. 1.1, 1.2). However, a revision may have one or more branches.

The tree is stored in a repository file using a *delta* storage technique. The delta technique used here is different from the one used in the set approach. This approach uses a backward delta for revisions along the trunk, whereas the set approach used a

forward delta. That is, the latest version of the component is kept current and intact, and to generate an old version the deltas are applied in a reverse order to arrive at an older version. The justification here is that the latest version is more often desired, thus it should be retrieved the quickest. However, this approach does not work for branches. To generate a version on a branch, the backward deltas are followed to the branch, then the branch is followed in a forward direction. Thus, branches are stored as forward deltas as with the set approach. It has been found that this approach to implementing the deltas does improve the performance of extracting versions of a component. Again, the main drawback of this approach is the requirement placed on the developer to select the correct versions of each component for the system. Since this approach is based on components, it could satisfy the locality requirement of our model. As with the set approach, current practice shows that a file is the most common component.

7.1.2 System Version Control

Even though they do not meet any of our criteria for an object-oriented version control model, the major disadvantage of all approaches presented in the previous section is that they lack adequate support for system structure. This deficiency is due to the lack of correspondence between versions of different components for a given instance of a system. As such, it is difficult to perform the builds of complete system versions. Next, we describe different approaches that address the issue of system versions in a version control system.

Partial Order

Plaice and Wadge [26] define a partial order on the versions so that the most relevant version of a component can be used when the specified version is unavailable. They point out an important fact, and disadvantage, to the component-based approaches.

That is, building a system version is not simply combining all components with the same version. Their approach solves this problem by applying two concepts. First, version labels for components are defined to have a global, uniform significance. Secondly, the authors define a partially ordered algebra for the versions. The partial order is the refinement relation: $V \sqsubseteq W$, read as "V is refined by W". This partial order allows the use of the most relevant version of a component if that component does not have the specified version available. This approach appears to be a step in the direction of formalizing version control. However, issues such as how to structure the components and limiting the effects of change are not addressed. In addition, the locality of the components is not specified.

Unified Framework

Katz [25] proposes a version model whose aim is to provide a unified framework for version control of engineering databases, that can be tailored for the needs of a given environment[1] The framework is based on four key concepts: component hierarchies, version histories, configurations, and equivalences.

The component hierarchy is the standard **is-a-part-of** or **aggregation** tree-like relationship. Figure 7.1 shows a simple example. The version history is the typical version tree of revisions and variants, called alternatives in this approach. Figure 7.2 shows the version history model. Each node is represented by *name[version#].type.* The main line of descent in the tree is a derivate, and siblings are alternatives. A configuration is the result of a version history combined with a component hierarchy, as depicted in Figure 7.3. System components often contain a variety of representations, all of which are needed to fully describe the component. For example, an object might have a source code, a documentation, and a test data representation.

[1]The term *database* does not necessarily mean a database management system. The current implementation of this approach is file based. However, the author does give a sketch as to how this approach can be implemented in a relational database system with the addition of foreign keys.

This relationship is modeled as an equivalence as shown in Figure 7.4.

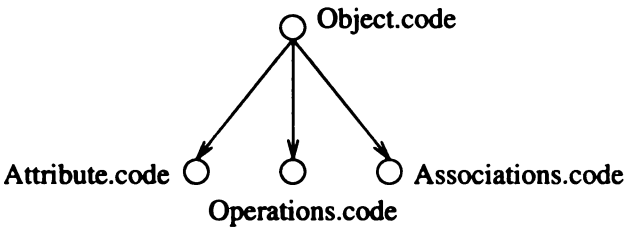

Figure 7.1. Unified Framework Component Hierarchy

This method is similar in concept to what this thesis is proposing, however the Katz method does not currently support different interconnections between object classes, or any other object-oriented concepts. The system structure is incorporated into the framework. Therefore, it would require significant changes to incorporate an object-oriented structure into this framework.

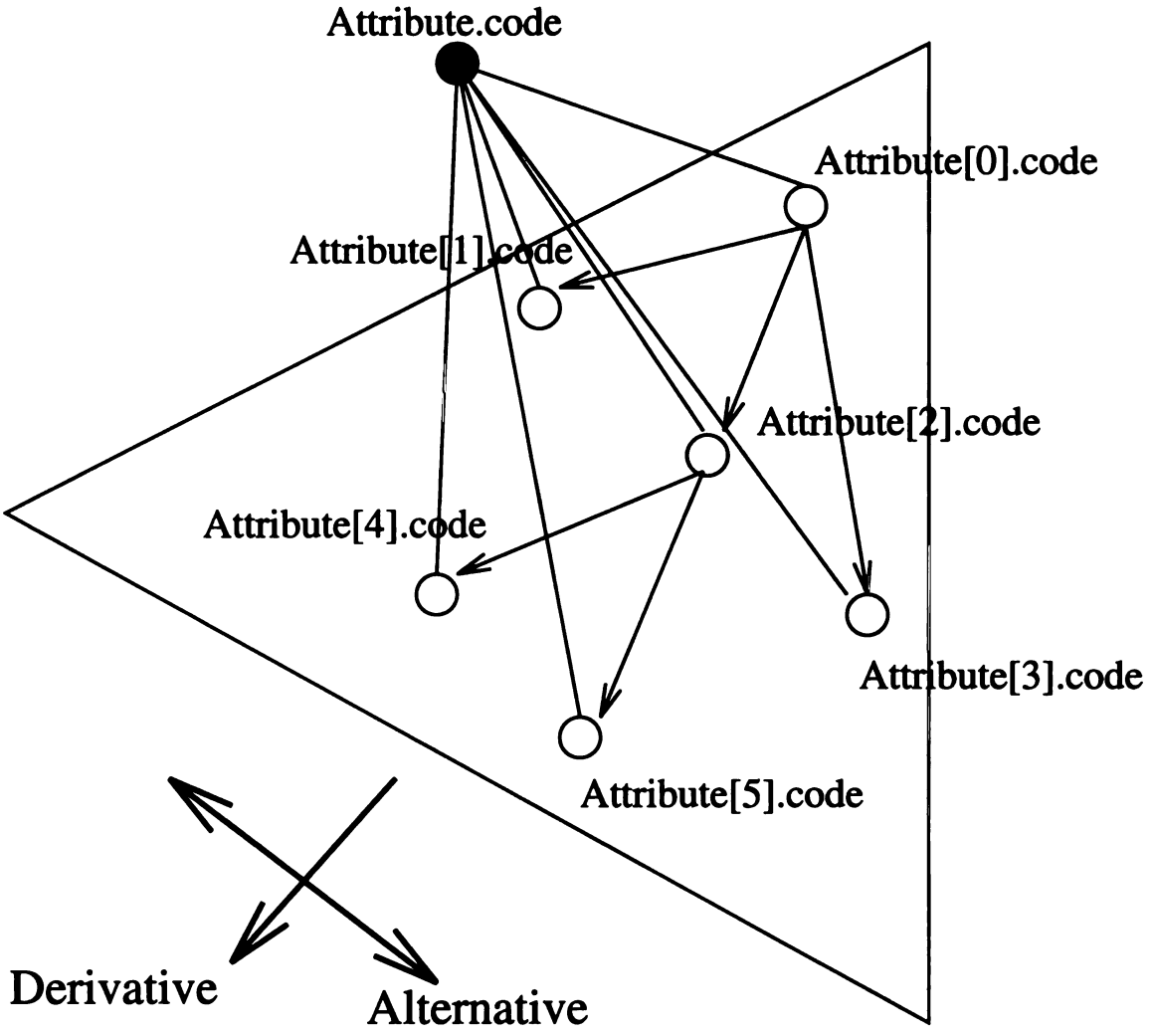

Figure 7.2. Unified Framework Version History

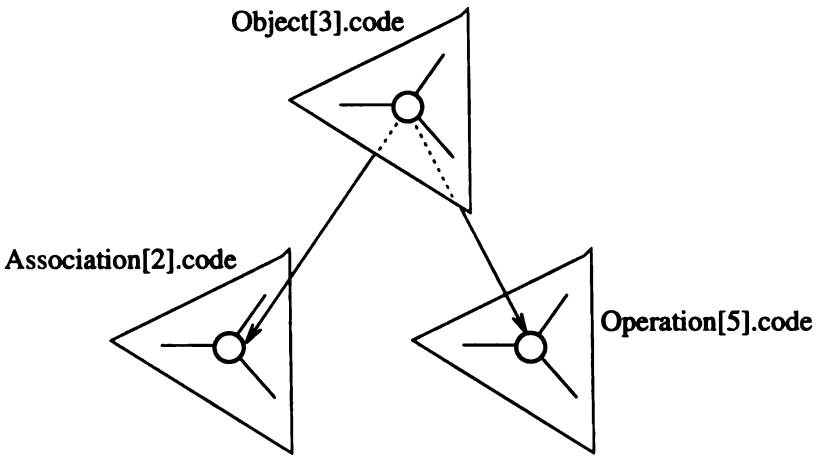

Figure 7.3. Unified Framework Configuration

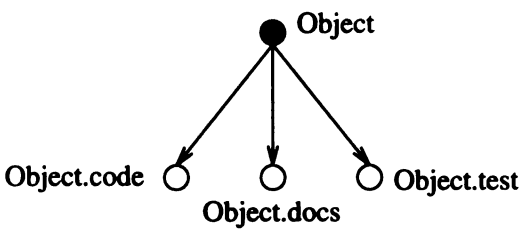

Figure 7.4. Unified Framework Equivalence

Inverted Approach

Miller et al. [8] attempt to solve system building problems with an inverted three level approach built up from a system variant base. The first level is the *Variant Level* which contains a composition of system variants. A *variant* is an instance of a complete system and is a structured collection of components. Figure 7.5 depicts a system version with two variants, the X10/X11 variant containing components 1,2,3,4 and the NeWS variant containing components 1,2,3,5. In general, any component in the system can have zero or more *attribute* terms which will define the variants. For example in Figure 7.5, if the attribute **Win** is set to **NeWS** when building a version of the system, then the 1,2,3,5 variant of that system will be built.

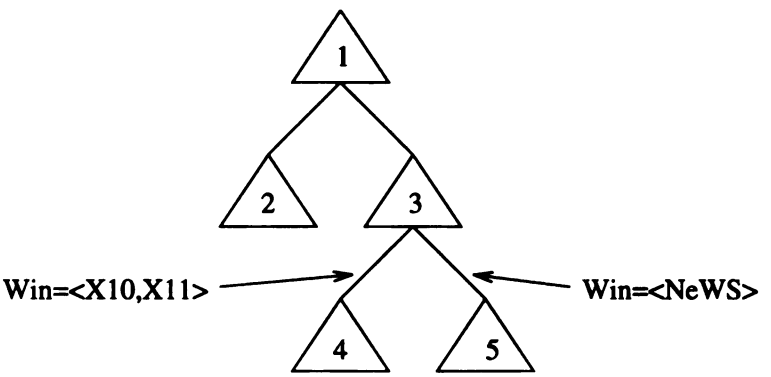

Figure 7.5. Inverted Approach Variant Level

The second level is the *Revision Level* which serves as the system version control level. It provides a revision tree of the development paths of system versions. Each version contains all system variants for that version. Figure 7.6 gives a graphical view of the Revision Level.

The third level is the *Transaction Level.* This level manages the development

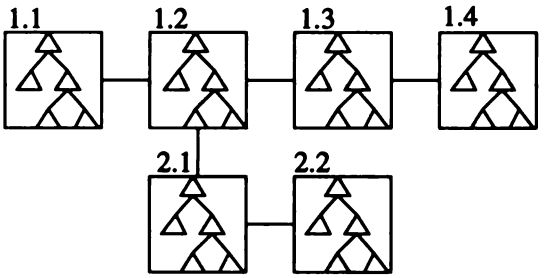

Figure 7.6. Inverted Approach Revision Level

of the revisions of the software system. This approach gives the software developer an intuitive understanding of the system versions, and is only lacking an adequate system structure for object-oriented systems and mechanisms to limit the effects of change. In addition, it is not clear as to what type of localization is inherent in the variant level.

Orthogonal Approach

Reichenberger [9] proposes a three-dimensional model of system version control called an "orthogonal organization" of variants and revisions which represents an *object pool.* Accordingly, the approach is localized around the concept of an *object.* Figure 7.7 conveys the most important concepts of this approach. The three dimensions are *variant, revision,* and *component.* All variants, components, and revision coexist in the same structure each having a unique identifier. To define the system structure, a hierarchical *project structure tree* depicts the component structure for each revision of the system. Again, it seems that this approach is only lacking an adequate system structure for object-oriented systems and mechanisms to limit the effects of change. However, the storage required to store the three dimensional structure could become quite extensive.

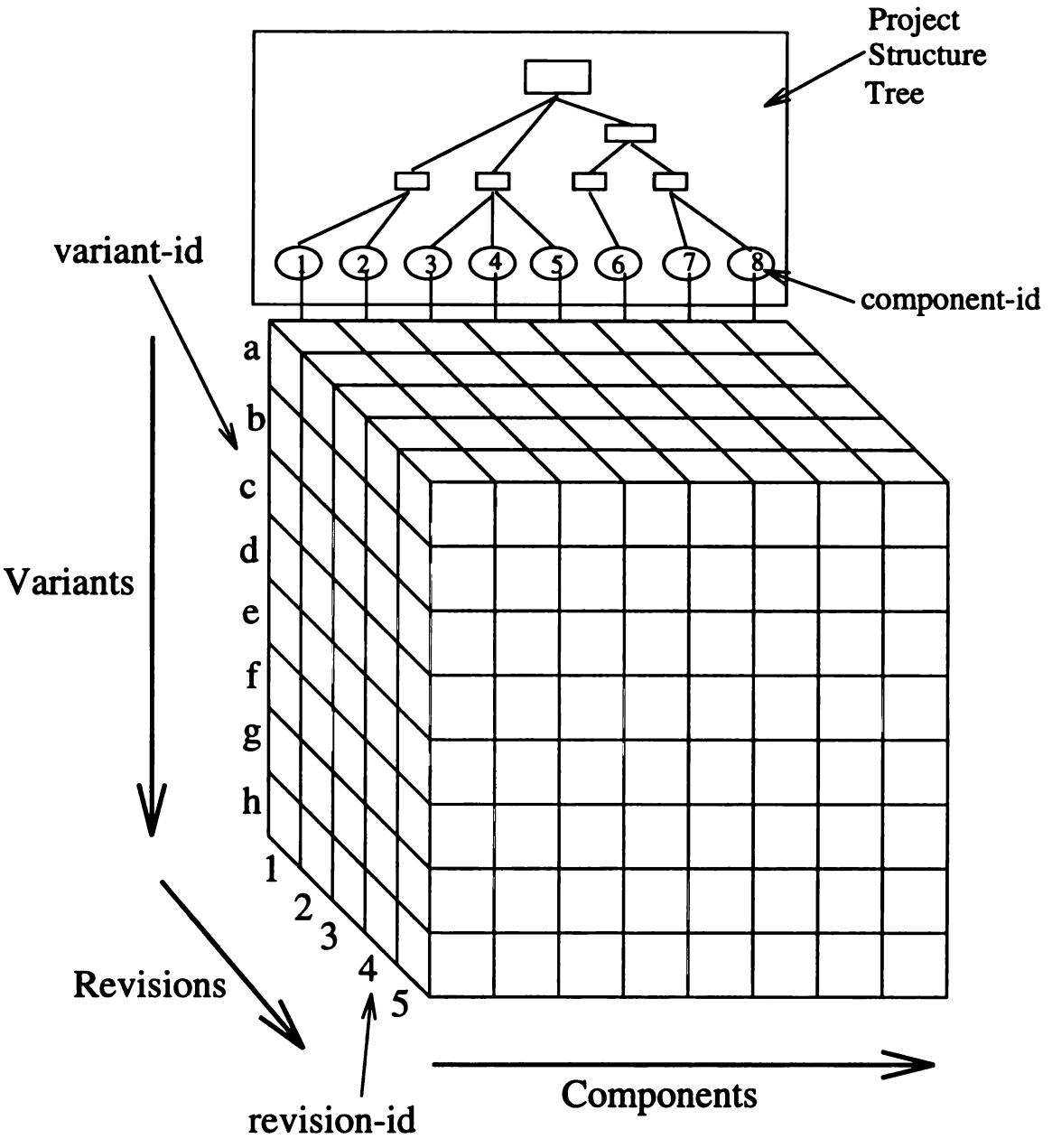

Figure 7.7. Orthogonal Version Management Concepts

Hierarchical Revision Graph

Magnusson et al. [17] propose a fine-grained approach to system version control that will limit the effects of change. All aforementioned approaches for system version control create some structure of system components. However, none specify how this structure is defined or the granularity of the structure. The basis for the fine-grained version control is what Magnusson et al. call a *RevisionTree* server which is an engineering database for storing hierarchical information.[2]The fine-grained nature of the tree is accomplished by allowing each node of the tree to be as fine-grained as the developer wishes. Thus, theoretically, a node could contain as little as one line, or one token of source code. Therefore, if that node is changed, then the effects of the change are limited by the chosen granularity. In addition, storage of changes is kept to a minimum, due in part to the fine granularity, but most importantly to the sharing of nodes between the revisions of the graph. This approach satisfies the fine-grained nature and limits the effects of change, however, this approach has no object-oriented system structure or concept of localization.

7.1.3 Alternatives to Version Control

Up to this point all methods mentioned have one property in common. They are all based on the primary concept of a *version.* Even though, this approach appears to be the most common focus, based on the amount of work in this area, methods based on other concepts need to be investigated for their adequacy to the object-oriented paradigm. Lie et al. [27] present a change-oriented method where *functional change* is the primary concept. Clemm [28] presents a *job-oriented* method where the software process itself is the primary concept. These approaches, all of which are not based on versions, do not lend themselves to the basic principles and requirements identified

[2]This is not necessarily a database management system. The authors do not describe the database storage mechanism in detail.

for object-oriented version control

7.2 Configuration Management Systems

The first generation of configuration management systems were generally version control systems with extensions or operating system support for the other configuration management activities. We have seen that this limited approach to configuration management is not acceptable for effective control and management of change in software development. Furthermore, these approaches do not adhere to the five necessary principles [4] nor meet the functionality requirements for effective configuration management [16]. Although advancements are still being made in version control, the current trend in configuration management systems is focused on systems that largely satisfy the functionality requirements for configuration management. The major advancements in this area are predominantly in research and commercial products such as Aide-De-Camp (ADC), Change and Configuration Control (CCC), and Domain Software Engineering Environment (DSEE). Below we discuss the strengths and weaknesses of these systems, contrasting them to our configuration management framework.

ADC

ADC is a commercial configuration management system based on a database repository of configuration management information. It satisfies the main functionality requirements of a configuration management system. However, its localized concept is a source code file. System structure information is defined in attributes and relationships between the files and in the notion of a *change set* between system versions. A *change set* is a set of logical changes that can apply to one or multiple files in a system version to create a new system version. In addition, limited system struc-

ture information, such as file dependencies, is derived by scanning the source code to produce Makefiles and to perform dependency analysis. Although, ADC provides the functionality to meet the major configuration management tasks, it does not alleviate the problems inherent with file localization and lack of system structure.

CCC

CCC is a commercial configuration management system that satisfies the main functionality requirements of a configuration management system. However, its localized concept is a source code file with version control of the individual files based on the tree-based approach. The notion of a *project* binds files together to form complete systems. The system structure is based on the *project* along with some file dependency analysis. Again, CCC provides the functionality to meet the major configuration management tasks, however, it does not alleviate the problems inherent with file localization and lack of system structure.

DSEE

DSEE is a configuration management environment composed of four components. The *History Manager* acts as a repository and stores versions of the source files and allows increased capabilities for selecting versions of the files. The *Configuration Manager* is concerned with the system definition and construction. The *system model* used is a description describing the relationships between the files and dependencies between tools and derived objects, such as object files. The *Task Manager* and *Monitor Manager* components provide the functionality of other configuration management activities. Although these four components satisfy configuration management system requirements, DSEE does not address the problems inherent with file localization and lack of system structure.

Investigation of the above configuration management systems reveals that the

common focus for system structure is the dependencies, tools, and procedures needed to derive an executable system from multiple files containing source code. The vast number of approaches to defining this system structure make it difficult to evaluate and compare the different approaches. Render and Campbell [29] propose a object-oriented semantic model to configuration management as one approach to alleviate this difficulty. Their model is based on two object classes: a configuration item and a derivation relation. An *item* defines the configuration item and all attributes of the item. Items can be specialized into subclasses, such as, an *aggregate item* (items composed of other items), a *derived item* (items derived by other items), *versioned item* (items that have alternate implementations), or a *version* (a concrete implementation of a versioned item). A *derivation relation* defines the relationship between items. A derivation relation can be specialized into subclasses, such as, a *derivation procedure* (a relation between kinds of items), or a *derivation execution* (a relation between specific instances of items).

This model does focus around a localized concept, the configuration item, defines the structure of those items, and enables easy integration of other software engineering procedures, such as project management [30]. However, the localized concept and system structure do not supply the important design and implementation information, such as component interaction, data flow, or control flow, needed to effectively maintain the software product, since there is no knowledge of a software component or architectural relationship.

CHAPTER 8

Conclusions and Future Work

A new approach to software configuration management has been developed based on three key concepts: concept locality, system structure, and fine-grained version control. Concept locality is based on the property that a software system is being composed of many abstract components, such as object classes, abstract data types, and modules, that are the focus of the entire software development and maintenance life-cycle. These components interact with each other in a specified manner that is defined by a system structure. Therefore, the basis for the general configuration management framework is that a software system is composed of many software components and a system structure that defines the relationships between the components.

Specific frameworks for object-oriented analysis and design and structured analysis and design are developed based on the general framework. In an object-oriented system the software components are object classes and system structure is commonly defined by diagrams, such as an OMT object model. Therefore, an object-oriented system is composed of many object classes and an object model that defines the associations between the object classes. In contrast, a structured system is composed of modules with relationships between each module defined by diagrams such as a structure chart.

Based on the object-oriented framework, a fine-grained version control model has

been developed. This model used a three-tiered approach to limit the effects of change to the software system. The top tier is the software system as a whole. The software system is composed of many object classes that make up the second tier. Finally, the third tier is the methods of each object class. This fine-grained approach enables a specific method of an object class to be modified and viewed as changed without affecting any other methods or object classes in the system.

As a means to investigate the feasibility and practicality of the framework, a prototype software configuration management (SCM) system has been developed [2]. Scalability to large-scale software systems has been investigated by using the SCM system to provide configuration management for itself. Preliminary observations have shown that the SCM system provides effective configuration management of object-oriented systems. The system tightly couples design, implementation, and maintenance of software systems, where the user can only access and change implementation information based on the design structure, which enables traceability between the design and implementation of a software system. In addition, the system allows the software engineer to concentrate on abstract concepts such as a system version, an object model, object classes, and operations without the burden of dealing with configuration management issues and file management. Also, the SCM framework supports the ability to limit the effect of changes in the system by providing a fine-grained version control mechanism.

Further work based on the framework could lead to the construction of a complete software development environment and could easily be extended to provide mechanisms for reverse and re-engineering by generating an object model from source code [31]. By adding requirements analysis or the definition phase of the software life-cycle, traceability throughout the entire life-cycle could be attained. Possible candidates for a requirements model could be an analysis object model. In addition, a diagram based on formal specifications [32, 33, 34] that could derive or be transformed

into a design object model could be used.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] R. S. Pressman, *Software Engineering - A Practitioner's Approach.* McGraw-Hill, Inc., 1992. ISBN 0-07-050814-3.

[2] S. R. Schafer and B. H. C. Cheng, "Configuration Management: Design, Implementation and Maintenance through the OMT Object Model," Technical Report MSU-CPS-95-8, Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, 48824, March 1995. Submitted for publication.

[3] André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf, "Does configuration mangement research have a future?," April 1995. To appear in *Proceedings of the 5th International Software Configuration Management Workshop*, Seattle, USA.

[4] D. Whitgift, *Methods and Tools for Software Configuration Management.* Baffins Lane, Chichester, West Sussex PO19 1UD, England: John Wiley and Sons Ltd., 1991. ISBN 0-471-92940-9.

[5] W. F. Tichy, "Design, implementation, and evaluation of a revision control system," in *Proceedings of the 6th International Conference on Software Engineering*, pp. 58–67, September 1982.

[6] M. J. Roekind, "The source code control system," *IEEE Transactions on Software Engineering*, pp. 364–370, December 1975.

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1991. ISBN 0-13-629841-9.

[8] D. B. Miller, R. G. Stockton, and C. W. Krueger, "An inverted approach to configuration management," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, pp. 1–4, October 1989. Published in Software Engineering Notes, Volume 17, Number 7, 1989.

[9] C. Reichenberger, "Orthogonal version management," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, pp. 137–140, October 1989. Published in Software Engineering Notes, Volume 17, Number 7, 1989.

[10] S. A. Dart, "The past, present, and future of configuration management," tech. rep., Software Engineering Institute, Carnegie Mellon University, July 1992. CMU/SEI-92-TR-8, ESC-TR-92-8.

[11] E. V. Berard, *Essays on Object-Oriented Software Engineering.* Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1993. ISBN 0-13-288895-5, Volume 1.

[12] N. M. Bounds and S. A. Dart, "Configuration Management (CM) Plans: The Beginning to Your CM Solution," tech. rep., Software Engineering Institute, Carnegie Mellon University, July 1993.

[13] I. Sommerville, *Software Engineering.* Addison-Wesley, 1992. ISBN 0-201-56529-3.

[14] H. Ronald Berlack, *Software Configuration Management.* John Wiley and Sons, 1992. ISBN 0-471-53049-2.

[15] S. Dart, "Spectrum of Functionality in Configuration Management Systems," tech. rep., Software Engineering Institute, Carnegie Mellon University, December 1990. CMU/SEI-90-TR-11, ESD-90-TR-212.

[16] S. Dart, "Concepts in configuration management systems," in *Proceedings of the Third International Workshop on Software Configuration Management*, pp. 1–18, 1991.

[17] B. Magnusson, U. Asklund, and S. Minör, "Fine-grained revision control for collaborative software development," in *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 33–41, December 1993. Published in Software Engineering Notes, Volume 18, Number 5.

[18] G. Booch, *Object-Oriented Design.* Benjamin-Cummings, 1990.

[19] P. Coad and E. Yourdon, *Object-Oriented Analysis.* Prentice-Hall, 1990.

[20] B. Meyer, *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[21] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis.* Yourdon Press, 1988.

[22] L. L. Constantine and E. Yourdon, *Structured Design.* Prentice Hall, Inc., 1979.

[23] W. F. Tichy, "RCS - a system for version control," *Software - Practice and Experience*, vol. 15, pp. 637–654, July 1985.

[24] K. Narayanaswamy, "A text-based representation for program variants," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, pp. 30–33, October 1989. Published in Software Engineering Notes, Volume 17, Number 7, 1989.

[25] R. H. Katz, "Toward a unified framework for version modeling in engineering databases," *ACM Computing Surveys*, pp. 375–408, December 1990.

[26] J. Plaice and W. W. Wadge, "A new approach to version control," *IEEE Transactions on Software Engineering*, pp. 268–276, March 1993.

[27] A. Lie, R. Conradi, T. M. Didriksen, E.-A. Karlsson, S. O. Hallsteinsen, and P. Holager, "Change oriented versioning in a software engineering database," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, pp. 56–65, October 1989. Published in Software Engineering Notes, Volume 17, Number 7, 1989.

[28] G. M. Clemm, "Replacing version-control with job-control," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, pp. 162–169, October 1989. Published in Software Engineering Notes, Volume 17, Number 7, 1989.

[29] H. Render and R. Campbell, "An object-oriented model of software configuration management," in *Proceedings of the 3rd International Workshop on Software Configuration Management* (P. H. Feiler, Ed.), (Trondheim, Norway), pp. 127–139, June 1991.

[30] H. S. Render, R. N. Sum Jr., and R. H. Campbell, "Integrating configuration and project management in an object-oriented software development environment," in *Proceedings of FedCASE '89*, (Gaithersburg, Maryland), Oct. 1989.

[31] G. C. Gannod and B. H. C. Cheng, "A Two Phase Approach to Reverse Engineering Using Formal Methods," *Lecture Notes in Computer Science: Formal Methods in Programming and Their Applications*, vol. 735, pp. 335–348, July 1993.

[32] B. H. C. Cheng, E. Y. Wang, and R. H. Bourdeau, "A graphical environment for formally developing object-oriented software," in *Proc. of IEEE 6th International Conference on Tools with Artificial Intelligence*, November 1994.

[33] R. H. Bourdeau, E. Y. Wang, and B. H. C. Cheng, "An integrated approach to developing diagrams as formal specifications," Technical Report MSU-CPS-94-26, Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, 48824, September 1994.

[34] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, pp. 684–707, September 1994.