A CONFIGURATION MANAGEMENT SYSTEM FOR SOFTWARE PRODUCT LINES

by

Cheng Thao

A Dissertation Submitted in
Partial Fulfillment of the
Requirements for the degree of

Doctoral of Philosophy
in
Engineering

at

The University of Wisconsin-Milwaukee

August 2012

A CONFIGURATION MANAGEMENT SYSTEM FOR SOFTWARE PRODUCT LINES

By

Cheng Thao

The University of Wisconsin-Milwaukee, 2012

Under the Supervision of Professor Ethan V. Munson

ABSTRACT

Software product line engineering (SPLE) is a methodology for developing a family of software products in a particular domain by systematic reuse of shared code in order to improve product quality and reduce development time and cost. Currently, there are no software configuration management (SCM) tools that support software product line evolution. Conventional SCM tools are designed to support single product development.

The use of conventional SCM tools forces developers to treat a software product line as a single software project by introducing new programming language constructs or using conditional compilation. We propose a research configuration management prototype called Molhado SPL that is designed specifically to support the evolution of software product lines. Molhado SPL addresses the evolution problem at the configuration level instead of at the code level. We studied the type of operations needed to support the evolution of software product lines and proposed a versioning model and eight cases of change propagation.

Molhado SPL supports independent evolution of core assets and products, the sharing of code and the tracking relationships between products and shared code, and the eight cases of change propagation. The Molhado SPL consists of four layers with each layer providing a different type of service. At the heart of Molhado SPL are the versioning model, component object, shared component object, and project objects that allow for independent evolution of products and shared artifacts, for sharing, and for supporting change propagations. Furthermore, they allow product specific changes to shared code without interfering with the core asset that is shared. Products can also introduce product specific assets that only exist in that product.

In order to for Molhado SPL to support product line, we implemented XML merging, feature model editing and debugging, and version-aware XML documents. To support merging of XML documents, we implemented a 3-way XML document merging algorithm that uses versioned data structures, change detection, and node identity. To support software product line derivation or modeling of software product line, we implemented support for feature model including editing and debugging. Finally, we created the version-aware XML document framework to support collaborative editing of XML documents without requiring a version repository. The version history is embedded in the documents using XML namespaces, so that the documents remain valid under the XML specification. The version-aware XML framework can also be used to support the exporting of documents from Molhado SPL repository to be edit outside and import back the change history made to the document.

We evaluated Molhado SPL with two product lines: a document product line and a the graph data structures product line. This evaluation showed that Molhado SPL supports independently evolution of products and core assets and the eight change propagation cases. We did not evaluate MolhadoSPL in terms of scalability or usability.

The main contributions of this dissertation research are: 1) Molhado SPL that supports the evolution of product lines, 2) a fast 3-way XML merge algorithm, 3) a version-aware XML document framework, and 4) a feature model editor and debugger.

Acknowledgements

I would like to thank my advisor, Prof. Ethan Munson, for his support of my research. He has been a great professor and a mentor. I learned a lot from him over the years. I would also like to thank my thesis committee members: Prof. John Boyland, Prof. Christine Cheng, Prof. Huimin Zhao, and Prof. Daniel Gervini for taking time from their busy schedule to help me. I want to thank Michael Levi Haufe for helping with implementing the Graph Product Line. This dissertation uses Fluid. It was implemented by Prof. John Boyland.

Dedication

To my parents: Kong Thao and Kao Xiong.

Contents

Acknowledgements **iv**

Dedication **v**

1 Introduction **1**

1.1 Problem Statement 3

1.2 Summary of Proposed Approach 4

1.3 Thesis Outline 5

2 Background **6**

2.1 Software Product Line 6

2.2 Software Product Line Activities 8

2.2.1 Core Asset Development 9

2.2.2 Product Development 11

2.3 SPL Adoption Approach 11

2.4 An SPL Example: Nokia Mobile Phones 12

2.5 Software Configuration Management 14

2.6 Concepts and Terminology 14

2.7 Challenges In Using SCM With SPL 18

2.8 Overview of SPL SCM Approaches 19

2.9 Summary 24

3 Approach **26**

3.1 Introduction 26

3.2 System Overview 27

3.3 Low-Level Versioning Layer 28

3.4 Component and Project Versioning 35
3.5 Product Line Versioning Layer 38
3.5.1 Shared Component 39
3.5.2 Change Propagation and supported cases 40
3.5.3 Implementing Change Propagation 43
3.6 User interfaces 45
3.7 Summary and Discussion 45

4 Three-way XML Document Merging **48**
4.1 XML Tree Semantics 49
4.2 Three-way Tree Merging 50
4.3 Versioned Tree Data Model 51
4.3.1 Versioned Data Structure 51
4.3.2 Change History 52
4.3.3 XML Documents as Versioned Trees 53
4.4 Longest Common Subsequence 54
4.5 Diff3 Merging Algorithm 54
4.6 Proposed XML Merge Algorithm 56
4.6.1 Merge Rules 56
4.6.2 Conflict Detection Rules 57
4.6.3 False Conflict Handling 57
4.6.4 Node Matching 58
4.6.5 The Algorithm 59
4.6.6 Implementation 61
4.7 Performance Evaluation 64
4.7.1 Test data and Hardware Configuration 64
4.7.2 Results 65
4.8 Related Work 70
4.9 Discussion and Future Work 72

5 Version-aware XML Documents **74**
5.1 Introduction 74
5.2 Background and Related Work 75
5.3 Implementation 76

5.4 Evaluation 78
5.5 Discussion and Future Work 80

6 Feature Model Editing and Debugging **82**
6.1 Introduction 82
6.2 Background 83
6.2.1 Feature Model 83
6.2.2 Current Support for Editing and Debugging 85
6.3 Approach 86
6.3.1 Feature Model to CNF Formula 86
6.3.2 Supporting Debugging 87
6.4 Implementation 89
6.5 Discussion and future work 90

7 Proof of Concept Evaluation **91**
7.1 DITA 92
7.2 The Graph Product Line 97
7.3 Summary and Discussion 100

8 Conclusion **103**
8.1 Approach 103
8.2 Contributions 104
8.3 Future Work 105

Bibliography **106**

List of Tables

3.1 Possible cases of change propagation. Boldface items are concrete, while items in non-bold characters are represented indirectly by shared components. 42

List of Figures

2.1 Cost in SPL in comparison to cost in single system development. . . . 8
2.2 Core asset development. The inputs are product constraints, production constraints, product strategy, preexisting assets. The outputs are product line scope, core assets, and production plan. 9
2.3 Product development. The inputs are requirements, product line scope, core assets and production plan which consists of attached process of core assets. The outputs are the products. 11
2.4 Two configurations of Version 1 and Version 2. One configuration has component **A**, **B** and **C** while the other configuration has component **A**, **B** and **D**. The repository contains all the selectable components **A**, **B**, **C**, **D**, **E**, and **F**. 15
2.5 A branching and merging example. The label inside a square is the name of the branch. A round rectangle represents a feature added. The solid line arrow represents a branch or a continuous evolution of the branch. The dotted line arrow represents a merge between two branches. 17
2.6 Core assets and products are managed under one process [1] 18
2.7 Evolution of software product line [1] 19
2.8 Generic CM model [2] 20
2.9 Variation management clusters [1] 22
2.10 Variation management of a production line [1] 23
3.1 Molhado SPL consists of multiple layers. 27
3.2 The left slot represents an unversioned slot, which has the value "a." On the right is a versioned slot, which has different value for different versions. Give a slot and a version; we can retrieve the value of that slot at that version.. 29

3.3 Values of slots at different versions. 30
3.4 Nodes and attributes can be thought of as a table where the rows are the nodes, the columns are the attributes, and the cells are the slots. A value then is a function of a node and an attribute. 31
3.5 With versions, the table can be thought of as a three-dimensional table where the third dimension is time (versions). A value is the function of a node, an attribute and a version. 32
3.6 The table representation is very sparse. Change often does not occur everywhere. For example, a version of project may only have a small change to a file. The empty spaces in the table do not use memory, as the actual implementation of the table is list. 33
3.7 Complex data structures created from nodes, slots, and attributes. . . 33
3.8 The states of a versioned graph at different versions. 34
3.9 Representing trees with attribute name. 34
3.10 Relationship between the version tree and eras. An era represents a sub tree of the version tree but excludes the root of the sub tree. In this figure, the initial era has one version, version 0 and has no root node. The era holding the root node is a special type of era. The remaining eras require a root node. Ideally, each era could represent a single saving session. 35
3.11 Nodes, attributes and versions are grouped into region, bundle, and era. The intersections of these are chunks, which represent the difference between eras (versions). 36
3.12 Project versioning 37
3.13 Product line versioning model. Green box represents shared component, yellow edge represents sharing, and blue box in products represents product specific code. 40
3.14 ersioned shared component. A shared component may refer to different version or different component at different version of the product. . 41
3.15 Product A is using **a** and **b** and product B is using **a** and **c** from the core asset project. Product A is at version PA_{v1}; product B is at PB_{v1}; and Core Assets project at CA_{v1}. 44

3.16 Product A has product specific changes to **a** resulting in version PA_{v2}. Core asset has changes to **a** resulting in CA_{v2}. Product B makes product specific changes to **c** and introduces a product specific component **d** (shown in blue) resulting in PB_{v2}. 45
3.17 Product A updates **a** with changes from the core assets project (forward change propagation). Changes to **c** in product B is pushed to the Core Assets project (backward change propagation). In addition, product specific component **d** is moved to the Core Assets project so other products can use (backward change propagation). As a result, product B now shares **d**. Change propagation is supported by merging (indicated as red) as indicated in the core assets project's version tree. 46
3.18 User interfaces 47
4.1 A three-way merge example for SVG documents. 50
4.2 Representing XML documents as versioned trees. 53
4.3 *diff3 algorithm* 55
4.4 Three-way merged of versioned tree. 62
4.5 Graphical conflict resolution interface. Conflicting elements are marked with a large square in front of the element name. The number displayed after a conflicting element helps the user identify the conflicting elements in the other trees. The number in front of the element is the node ID and allows users to match elements in different trees especially when the element name is not meaningful and there are many elements with that name. 63
4.6 Total execution time as changes increase. 66
4.7 Memory usage as changes increase. 66
4.8 Total execution time as elements increase. 67
4.9 Memory usage as elements increase. 67
4.10 Merge time as changes increase. 68
4.11 Merge time as elements increase. 68
4.12 Merge time as changes increase. 69
4.13 Merge time as elements increase. 69
4.14 Parse time as elements increase. 70
5.1 Version-aware XML document collaboration workflow example. . . . 78

5.2 This version-aware Inkscape document contains three revisions. Box 1 represents the entire version-aware document with molhado namespace defined in box 2. Box 3 contains the version data (box 4) and the version data signature (box 5). Box 6 contains the Inkscape document content which represents the latest revision. 79
5.3 vinkscape: retrieval of revision 1, 2 and 3 from a version-aware Inkscape SVG document. 80
6.1 A feature model example of a family of cars. Feature highlighted as green indicate one possible car. 85
6.2 Problems and warnings in feature model: (a) an inconsistent model, (b) B is a false optional feature, (c) the implication of B is redundant, (d) B is a dead feature. 85
6.3 Editor: highlighting errors and warning in the model. 89
7.1 Topics are assembled in DITA maps to create different deliverables. There are two DITA maps sharing topics [3]. 93
7.2 DITA topics and maps are used to create multiple deliverables from common topics [3]. 94
7.3 An output generated from the DITA sample using DITA Open Toolkit. 95
7.4 Screen shot of a DITA document product line in Molhado SPL. . . . 97
7.5 The feature model of the Graph Product Line. 99
7.6 One possible product instance of the Graph Product Line. 100
7.7 Screen shot of the Graph Product Line in Molhado SPL. 101

Chapter 1

Introduction

Software is constantly increasing in complexity. Customers are demanding more features in a software product. Software has to support varying hardware, constraints and market segments. At the same time, developers are under pressure to reduce development cost, increase product quality and shorten time to market in order to stay competitive. Software reuse is known to increase productivity and shorten time to market. Reuse is greatly helped by improvement in programming languages that supports object oriented programming and tools such as IDE, architecture and frameworks, but the problem of software development and managing their evolutions remains a challenging problem. Development of single product is well known and understood. But when introducing multiple products that share much of the same functionality, development complexity increases an order of magnitude compare to single product development [4].

A recent software development paradigm, software product line (SPL), is to alleviate many of these problems especially to reduce development cost, to shorten time to market, to support multiple market segment, and to increase quality by exploiting large scale reuse. In some domains, software products share many commonalities such as hardware, features, standards and algorithms. Product line is a well known technique in the manufacturing industry from Boeing to McDonald. In the automobile industry, cars of different models are built using many common parts. The key idea in a product line is to take advantage of reuse of existing components, design, and requirements that are costly to create from scratch. These reusable artifacts are called *core assets*. In a software product line, code implementing common feature is used in

multiple software products that share that feature.

Software configuration management (SCM) is a software engineering discipline that concerns the management of software evolution and change control. Software product line poses a different problem to SCM than a single product software development. In a single product, the evolution of the product is in the time dimension. A single product is maintained as one evolving entity. In SPL, products evolve independently of the components that are shared among the different products. Product and components have their own line of development. The evolution of the components are said to evolve in the time dimension while the evolution of the products are said to evolve in the space dimension. Space refers to the product space in the product line.

The component developers and the product developers may not be the same people. Even the developers of different products may not be the same developers. This sharing of component across products creates a network of product component dependency. A change in a component may affect multiple products and components that depend on the changed component. For example, changes in the interface of a component could break products and components using it. Dependencies are either found in the code, or maintain externally such as an external database.

A conventional approach is to use traditional SCM systems' branching capability to create products. Once the branch is created, theres no communication between the branches and the core assets. The products evolution and the core asset evolution eventually leads product and the latest core assets incompatibility. To move to the latest core assets, products have to be re-implemented [5].

Some approaches to SCM and SPL use build scripts to create products and support core assets and asset variants while the evolution is managed by conventional SCM systems such as CVS [6]. These are at file level abstraction for component variant [5, 1]. Low level abstraction creates an overhead for developers. Its hard for developers see the structure of the application without looking at the build script or some rules that forms the application. Build scripts and rules need to be managed under a SCM system. The disadvantage to this approach is that such approach does not allow for developers to work on products and core assets at the same time. Everyone is working on core assets. Products are generated by the scripts [1].

For some approaches, each core asset is maintained independently using the team's SCM system [7, 8, 9, 10]. Each core asset team may be using different SCM systems.

To avoid component and product incompatibility, it is important that developers use the latest release of core assets to prevent incompatibility. Product developers have to make their product compatible with latest release if they were to use it. Dependencies relationships are maintained externally[11].

1.1 Problem Statement

Configuration management for software product lines poses difficulties that do not exist for single products due to the multidimensional nature of SPL evolution. Conventional SCM tools are excellent at managing evolution of components and product along the time dimension, but provide no support for managing the evolution of components and products across the space dimension. In SPL, products and components evolve independently.

- Products can share components. For example, two products may share a component that implement a feature common to both products.
- There may also be relationships among components and products. For example, a component may require the existence of another component, or the exclusion of another component.
- The interaction among the components and products can easily lead to inconsistency in the configuration. For example, a change in a component may affect products using the component and other components that depend on the changed component or it may lead to incompatibility with products using an older architecture.

The current approach to software product line configuration management is to manage each component's configuration independently using conventional SCM tools such as CVS or Subversion. These SCM tools are not aware of relationships among components and sharing among components and products which can be used to enable automatic propagation of changes. To deal with variation over space, products are generated on the fly at production time using rules or build scripts specified by the developer [8, 1, 5]. In many current approaches to SPL, variants of a component can be generated by specifying a set of rule or specification that would compose the

component from reusable code. The rules or specification themselves must be managed. Creation and management of the rules themselves causes overhead for developers.

1.2 Summary of Proposed Approach

We propose an SCM tool that solves the SPL evolution problem at the configuration management level instead of at the code level as described earlier. It has a versioning model that allows for independent product and core assets development and allows core assets to be shared in products. Core assets and products are managed under one process. The versioning model also support change propagation from core assets to products and from product to core assets. The tool supports product derivation through the use of feature models.

The solution is a prototype that provides the following features:

- Managing the evolution of a software product line,
- A versioning model consisting of one core assets project and one or more product projects,
- Support code sharing in multiple products,
- Manage shared code and products relationships through feature models,
- Support change propagation from shared code to products and from products to shared code,
- Provide feature model editing and debugging,
- Enforce a main trunk for each product line,
- 3-way merging for components and product lines and
- Automatically generated branches to support product specific changes to shared code.

1.3 Thesis Outline

The lay out of this dissertation is as the following. Chapter two will give an introduction to software product line and its essential activities and configuration management concepts. The design and implementation of the prototype is described in Chapter 3. Chapter 4 describes a 3-way XML merge algorithm that is used to support merging in the prototype. Chapter 5 talks about feature model and the implementation of editing and debugging of feature model. Chapter 6 talks about storing version history inside XML documents. Evaluation of the prototype on two product lines is presented in Chapter 7. Conclusion and future work is described in Chapter 8.

Chapter 2

Background

2.1 Software Product Line

The field of software engineering works to improve software development and to manage the complexity of software development. Programming languages have continuously evolved to provide some support for this by providing language support for procedures, modules, class, components and separate compilation. At a higher level, there are various architecture styles and software design patterns that can speed up software development and improve software quality. As software development has grown more complex over the years, many problems have arisen. Often the software product being developed appears similar to products developed before and code and functionality are often duplicated in different products. In domains such as telecommunication, it is common that different products share many features and functionalities. This results in duplicate work in building the products and doubling software maintenance, for example, fixing a bug in duplicate code requires fixing other products containing the duplicate code.

Software product line (SPL) is an approach to software development that tries to address the above problems. Clements and Northrop [12] define SPL as ‘‘a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed ways.” Products in SPL refer to software products that are used internally or sold to customers. Some SPL do not produce software products but produce software components either to be used internally to

build products or to be sold to third parties that can use the components to build their own products. The goal of software product line is to reduce the time to produce a product by reuse and reduce the duplicate code problem, to increase product quality and to increase evolution manageability of the products. The emphasis of SPL is code reuse. Based on case studies, advocates of SPL say that if the products in the same product line share many components, it should reduce the cost to develop a new product and the time required to deliver the product because of reuse instead of rewriting new components. Many companies such Nokia, HP, LSI Logic, Philips, Cummins and others are using SPL engineering development to develop their products.

In *single-system development* or *one-at-a-time product development*, a product is an independent software development. Each product is maintained independently. In worst case, there is no software reuse in building new products. New products are created from scratch even though the new products may resemble the previous products. Often reuse in single-system development is called ‘‘clone and own.” The idea is to take advantage of existing software by making a copy and start developing on the new copy to make it fit the requirement. Now there are two products and they are independent and are maintained independently of each other. On the other hand, software product line is treated and maintained as a whole rather than being view as multiple products being managed independently. In addition, *core assets* such as components and other artifacts in SPL are designed for reused. In SPL, the overall health of the product line is more important than the individual product. Degrade in the SPL's health reduce reusability and makes the SPL looking like a single-system development.

Adopting SPL approach to development adds up-front or *initial investment* cost to build the product line. Whether to adopt an SPL approach to development will largely depend on the number of products and whether the company is able to afford the initial investment to build the software product line. Figure 2.1 shows the expected the cost and benefit of the SPL development approach compared to single-system development approach. The dashed line shows the cost of SPL as the number of products increase. The solid line shows the cost of single system development approach as the number of products increases. The intersection of the two lines is called *break-even point* where the cost of both approach are the same. The pay-off from SPL approach exceeds single-system development after the *break-even point*. According to Clements and

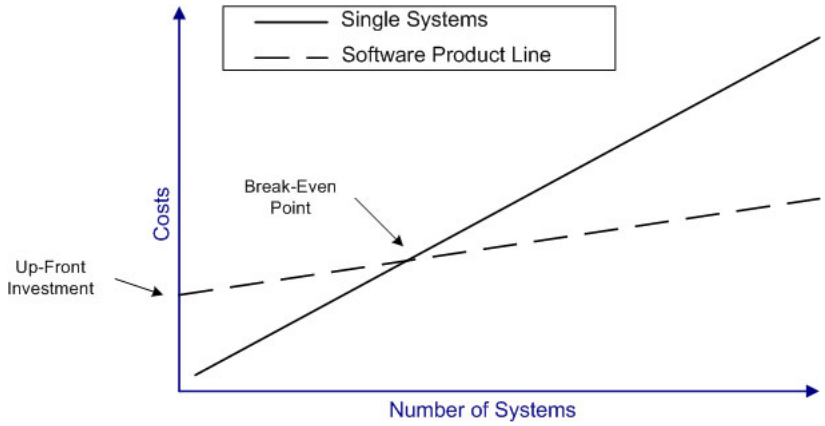


Figure 2.1: Cost in SPL in comparison to cost in single system development.

Northrop [12], the break-even point is around three products. This chapter gives an overview of SPL terminology and of the essential activities in SPL engineering. This dissertation uses terminology from the book *Software Product Lines: Practices and Patterns* [12]. Other authors [13] use somewhat different terms with notable differences between North American and European researchers. We will note the difference in terminology where appropriate.

2.2 Software Product Line Activities

There are three essential activities in software product line engineering: *core asset development*, *product development* and *management*. *Core asset development* produces *core assets* and *product development* uses the *core assets* to create the products. Both activities are under the supervision of the *technical and organizational management*. In this dissertation, we will only focus on core asset development and product development. Core asset development and product development can occur in either direction. Often they are performed in concert with each other. New products are built from core assets and core assets are extracted from existing products. Most core assets are created for the intention of being used in new products. There is the notion of a strong feedback loop between the core assets and the products as they each influence the development of the other.

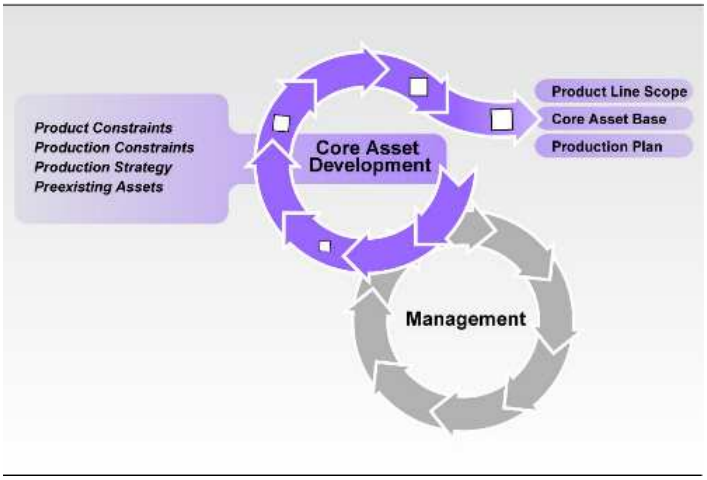


Figure 2.2: Core asset development. The inputs are product constraints, production constraints, product strategy, preexisting assets. The outputs are product line scope, core assets, and production plan.

2.2.1 Core Asset Development

Core asset development (also called *domain engineering*) is the process of developing the core assets and its goal is to establish a production capability for creating products. *Core assets* (also called *platform*) may include any aspect of software development including requirements, reusable software components, architecture, performance modeling and analysis, test cases, test plans, and documentation. The process of developing core assets follows an iterative process or the classic waterfall cycle. Core asset development makes use of *product constraints*, *architecture style, pattern and framework*, *production constraints*, *production strategy*, and *preexisting assets* which will be described below. The results of core development are the *product line scope*, the *core assets*, and the *production plan* described below. Figure 2.2 depicts core asset development.

Product constraints describe the commonalities and variabilities among the products in the product line in terms of behavior features, standards observed, performance limit, interface and environment constraints, quality, and security constraints. *Architecture style, pattern and framework* will influence how the core assets are developed. An architecture may constraint how components interact. Patterns and framework forces a development paradigm on developing the core assets. *Production constraints* specify what commercial, military, or company standards apply to the products,

the infrastructure on which the products must be built if any, which legacy and off-the-shell components could be reused. Production constraints will impact the core asset development. *Production strategy* describes how the core assets are developed (in house vs off-the-shelf components) and the approach to develop the products (automatically generated vs assemble). An example of production strategy is whether the products will be developed bottom up or top down. Top down approach starts with creating the core assets and then move to creating the products. Bottom up starts with products and create the core assets to form the product line. The product strategy also specifies how the core assets will be managed. *Preexisting assets* are existing assets that may be reused. Legacy products can be mined for core assets if needed.

The results from the core assets development activity are: *product line scope*, *core assets*, and *production plan*. *Product line scope* describes what products will constitute the product line. It could be simple as a list of products or more comprehensive such as a list of variability and commonality in terms of features, operations and performance among the products. The process of defining the product line scope is called *scoping*. The scope of a product line evolves with the market and the company's plan. *Core assets* are the basis for producing products in a product line. Core assets include reusable components and their associated requirements, test cases, and many other artifacts that are time consuming to produce. Note that commercial off-the-shell components are also part of the core assets if adopted. Each core asset has an *attached process* that describes how it will be used in the development of actual products. For example, the attached process may say that the core asset is required by the product line, what variations are supported and so on. The attached processes are also part of the core assets and they later are combined to form the *production plan*. A production plan specifies how a product is created using the core assets. It also specifies the product line's constraints that must be followed by the products. The attached processes of core assets are combined to form part of the production plan. It defines the relationships between the core assets and the products.

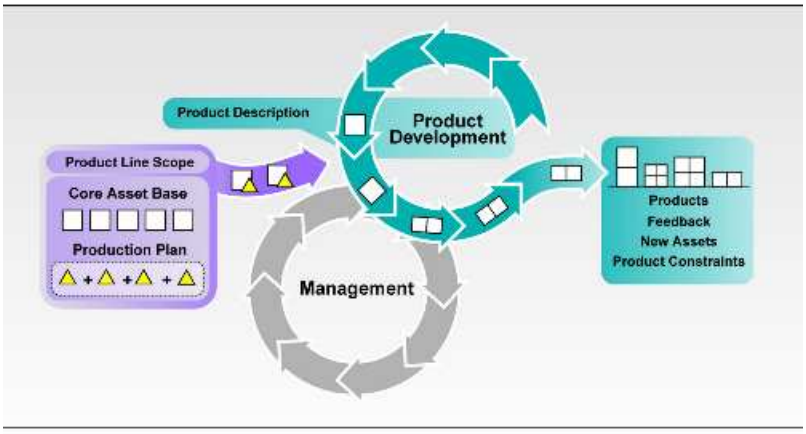


Figure 2.3: Product development. The inputs are requirements, product line scope, core assets and production plan which consists of attached process of core assets. The outputs are the products.

2.2.2 Product Development

The second activity in SPL is *product development* (also called *application engineering*) which is the process where the products are created using the developed core assets. This process may affect the core asset development, for example, a product may require or introduce new core assets. Producing a new product that has an unexpected commonality with another product may lead to creating a core asset that can be shared in future products. Product development makes use of the output from core asset development: product line scope, requirements, core assets and production plan. The *product line scope* determines whether the product under consideration can be included in the product line. This is also referred to as *product space*. The *production plan* describes how the product is to be built from the core assets. The ultimate goal of product development is products.

2.3 SPL Adoption Approach

When an organization starts a software product line by developing the core assets, they are said to take a *proactive* approach [14]. The proactive approach usually suits larger organization with resources because proactive approach requires a significant up-front investment to produce core assets that are generic for the product line. In

addition, they require good domain knowledge and prediction of future products.

The opposite of proactive approach is the *reactive* approach. In the *reactive* approach, the organization builds core assets on demand as new products are being introduced. It is an incremental approach and is best used when new products and features are difficult to predict.

Organizations that begin with one or a few existing products and use them to make core assets for future products are said to take a *extractive* [14] approach. The reactive approach is considered more suitable for smaller organizations with less resources or already have some products and would like to move to SPL development. It costs less to start a product line under the reactive approach because the core assets are not developed up front.

2.4 An SPL Example: Nokia Mobile Phones

Nokia Mobile Phone is one of the largest mobile phone makers [11]. They sell phones over 130 countries. In 2004, there were 2 billion users of Nokia phone. The Nokia product line consists of different series: s30, s40, s60, and s80. In 2002, the estimated number of new products produced by the Nokia product line was 25 to 30 products a year [11]. Nokia makes phones of with varying number of keys, display size, color depth, set of features, languages, input methods, and protocols (CDMA, TDMA, GSM, and more). In addition, the phones are divided into low-end and high-end segment. The phone and software must be backward compatible with accessories already on the market. It is believed that their SPL development approach is one of the factors that has allowed them to be world's largest phone maker.

Nokia requires that each features be configurable (on/off and various settings), able to change behavior after product release, and plug-in-play. The product line must support the evolution of both product hardware and software. Nokia defines a software product line as a group people developing a specific set of features for wide range of products [11]. In this view, each product line is a people driven process that produce components or products. They have software product lines for different level of software: DSP, architecture, user interface, features, and etc. They often produce new release so others in the Nokia phone development can use them. There's a product line for each feature to be included in the final product to be sold. Nokia software

product line consists of a large set of components such as phone book, SMS, snake, clock, calculator and so on. The view on a phone may consist of text, bar, graphic, formatted text components. Thus, most of Nokia product lines are product lines that create components to be used by product lines that create the actual products to be sold.

Nokia defines four challenges that their product line has to deal with [11]. They are language, UI, hardware, and feature selection challenges. To support different languages, they separate the language knowledge from the code by using the observer design pattern to provide abstraction between appearance and behavior. The variability of the user interface is handled by a window manager which is designed to separate behavior and appearance to the largest extent possible. The hardware consists of different number of keys, sound playback, display size, color depth, and types of wireless connection. Nokia abstracts the hardware problem through the use of the Window Manager. The Window Manager disconnects the hardware from the relevant data. The feature challenges consist of: variability in features, new features created continuously, the fact that most feature change over time, and the importance of feature memory usage. Nokia's solution to the feature challenge problem is to use of a client/server architecture. The architecture provides a nice encapsulation of resources and allows for adding and removing of features easily.

As for feature selection, it is done both by static selection or dynamic selection. Static selection of a feature can be done by including the feature in the build. Partial feature selection can be done by use of compile time flags. Dynamic selection of features can be done through dynamic setting, configuration file and dynamic discovery service.

Since there are so many components that are depended upon by other components and products, Nokia needed a systematic way to keep track of these dependencies. They maintain a global database of dependencies which allows Nokia to notify affected parties on changes. With the database, bug fixes can be distributed faster to projects that use the affected components. The dependencies are also useful when making changes as they need to ensure interoperability.

2.5 Software Configuration Management

Software Configuration management (SCM) is the discipline in software engineering that deals with managing the evolution of large and complex software systems [15, 16, 17, 18]. Without an SCM system, a software product being developed by multiple software developers may have the following problems: the double maintenance problem, the shared data problem and the simultaneous update problem. Double maintenance problem arises when two or more developers make copies of the code of the same software and start developing separately and independently. When a bug is found in the original software, each of the developer would need to update their own copy to include the fix resulting duplicated work. The shared data problem arises when multiple developers are sharing a single copy of the software and modifying the software in parallel. The changes one developer makes may interfere with the progress of other developers. For example, one developer may break the software or introduce bugs which forces the rest of the developers to wait until the software is fixed before they can continue. The simultaneous update problem occurs when two or more developers update the same file or component simultaneously resulting in one developer's work being overwritten by the other's work. The purpose of configuration management is to overcome the above problems and to control what changes are permitted.

SCM provides two different needs: management support and development support. SCM in management support deals with controlling changes to software products while SCM in development support deals with facilitating developers in performing coordinated changes to software products. This research concerns with SCM in development support especially in the domain of software product line. This section describes concepts and terminology in SCM and the challenge of SCM in supporting the development of software product line.

2.6 Concepts and Terminology

A software system is composed of different modules, components, or files. The list of the items that the system is composed of is called the *configuration.* For example, a configuration of a system may consist of component **A**, **B**, and **C** while another configuration may consist of **A**, **C**, and **D** as shown in Figure 2.4. The process of

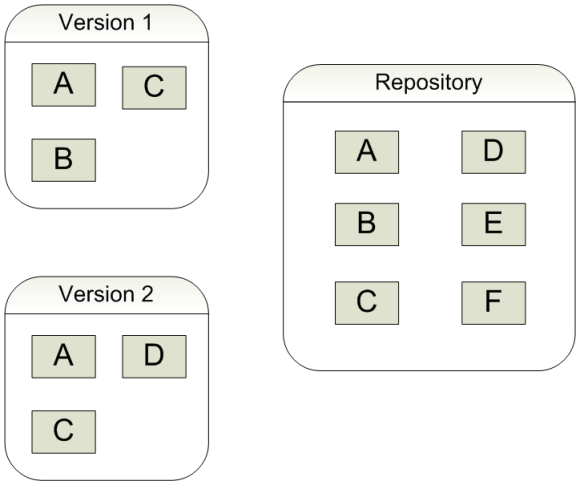


Figure 2.4: Two configurations of Version 1 and Version 2. One configuration has component **A**, **B** and **C** while the other configuration has component **A**, **B** and **D**. The repository contains all the selectable components **A**, **B**, **C**, **D**, **E**, and **F**.

selecting what components, files and modules go into a software is called *configuring.* The components, files and modules are selected from a *repository* that contains all existing versions of components and files. Configuring is also known as *product derivation* or *product instantiation.*

Version is a state of a software artifact which may include files, components, requirements and test cases. A version is immutable in that only a new version of the artifact is created when modifying an existing version. The previous version remains unchanged. There are two different kinds of versions: *revision* and *variations.* A revision is a new version intended to supersede the previous version. A revision may be a bug fix, a new feature or an improvement. Revision follows a linear order such that revision $n + 1$ supersedes revision n. The second type of version is variation. A version of a component may be a variation of another. A variation serves a different purpose than that of a revision. A variation is an alternative that can be substitute with another variation of the same component. For example, there are two variations of the same component where each is used in different operating systems with different user interface API. Note that version refers to both revisions and variants. The rest of the writing will use the term version unless there's a need to distinguish revisions and variations.

Instead of storing the entire file for each revision, SCM systems store *deltas* between

two revisions to conserve disk space. A delta is a sequence of editing commands that transform one revision to the other. One of the revision is stored in full and the other revisions are constructed by applying the deltas to the fully stored revision. SCM systems use two types of deltas: *forward* or *backward* (also called *reverse*) deltas. In forward delta, the oldest revision is stored in full and newer revisions are constructed by applying the deltas. For example, to construct revision n, the deltas stored in revision n is applied to revision $n-1$. In backward delta, the newest revision is stored in full and older revisions are constructed by applying the deltas. To compute revision $n-1$, the delta in $n-1$ is applied to revision n. Another way to represent revisions is through the use of *interleaved deltas*. A file containing interleaved deltas is divided into blocks of lines where each block contains header information specifying which revision the block belongs to. To create a revision, only the blocks belonging to that revision are selected. An approach to represent the variation of a file is through the use of *conditional compilation*. A single file contains all the variations of that file. The differences are separated by macros that the compiler understands. When the compiler is invoked on a file with the name of the variation, the compiler will select only the code in the file that are relevant to the supplied version name. The problem with using conditional compilation to present variation is when there are many variations, the code become very difficult to read.

To avoid the share data and simultaneous update problem, SCM system has a *baseline* and provides *private workspace* for developers. The baseline (sometimes called the *database*, *codebase* or *repository*) contains the all versions of the shared project. The baseline contains all the necessary components of all versions that will be used to construct the software system that is being built. Usually the baseline is tightly controlled as to ensure integrity. Software developers can work on the project by *checking out* the project consisting of versions of files that the developers want to work on into their private workspaces. Changes made in the private workspace are not visible to other developers. When a developer is done with the files, he can *check in* or *commit* the changes into the baseline. Once the changes have been checked in, other developers can see the changes and choose to update their own private workspace with the new changes.

Sometimes a developer may check out various versions of files of the project. The developer would have to know which version of individual files he would need. This

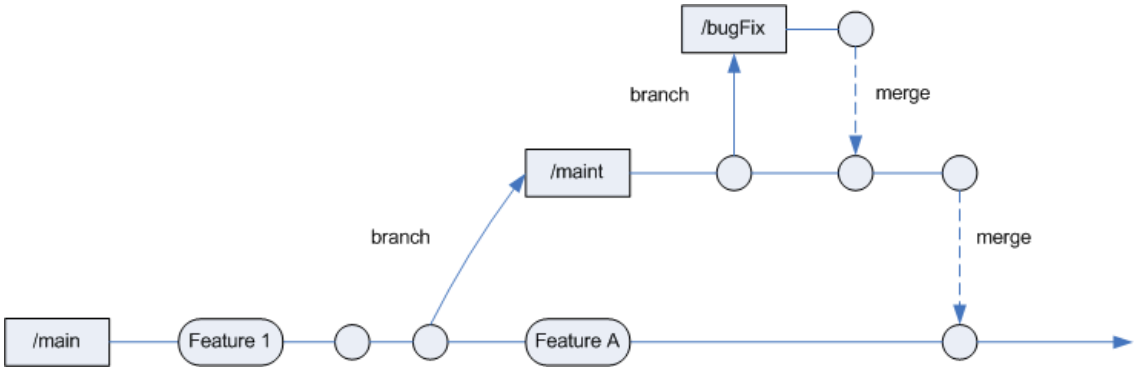


Figure 2.5: A branching and merging example. The label inside a square is the name of the branch. A round rectangle represents a feature added. The solid line arrow represents a branch or a continuous evolution of the branch. The dotted line arrow represents a merge between two branches.

will make the task of checking out projects complicate. *Project version tagging* is a way group a set of files of different versions and gives the set of version of files a symbolic name. The label stays fixed with the set of versions even as newer versions of those files are being added. A developer can check out the set of files by using the label thus simplify the process of selecting all the correct versions of files. Another form of tagging is *project file tagging* which tags a set of files but ignores the versions of the files. Usually checking a project using a label created by project file tagging gives the latest version of the tagged files.

To avoid the simultaneous update problem, SCM provides a lock mechanism where if one developer wants to modify a file, he or she would check the file out and lock it so that others can not modify the file. The problem arises when another developer needs to modify the file in order to continue his or her work. Since the file is locked, the second developer is stalled. Worst, the developer who locked the file may be out for a long vacation. A different approach to the lock mechanism is the *branching* and *merging* mechanism. Branching and merging allows developers to work in parallel and may modify the same files. In the lock mechanism, versions grows in linear order. With branching, versions now grow in a tree order where the branches are the versions are derived from versions in the main trunk. Branching can also apply to the baseline. Merging is the processing combining the changes of multiple versions of a file into a new version. For example, a developer checks out a project as a different branch to fix a bug and now wants to propagate the fix to the main trunk, he would merge his

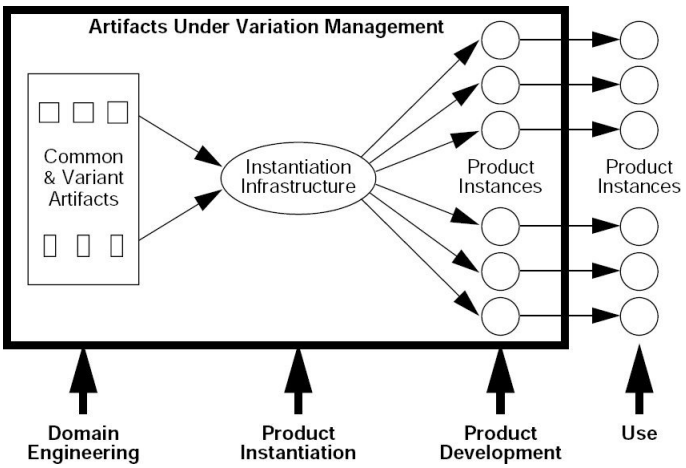


Figure 2.6: Core assets and products are managed under one process [1]

version with the version in the main trunk producing a new version containing the bug fix as shown in Figure 2.5. Merging is usually done with the help of a textual merge tool employing line based differencing. Merging of two versions of the same file may or may not have conflicts. Merge conflicts usually result from two changes to the same lines of code or close enough that the merge tool is unable to decide what do with the changes.

2.7 Challenges In Using SCM With SPL

Conventional SCM systems such as CVS [19, 6], Subversion [20], and SCCS [21] are designed to manage configurations of a single software product. The nature of software products in an SPL is different than that of a single product in the traditional sense of software development. In single product software development, all the artifacts making up the product are managed together as a single file hierarchy.

In SPL, configuration management becomes a multidimensional problem. Products of a given SPL contain core assets and custom assets. The core assets are shared among the products thus forming complex relationships among core assets and products. Changes to core assets may affect other core assets and multiple products. The core assets and the products must be configuration managed under one process as shown in Figure 2.6. Krueger [1] refers to the evolution of a component as ‘‘change in the time dimension” and the evolution products as ‘‘change in the space dimension.” Traditional

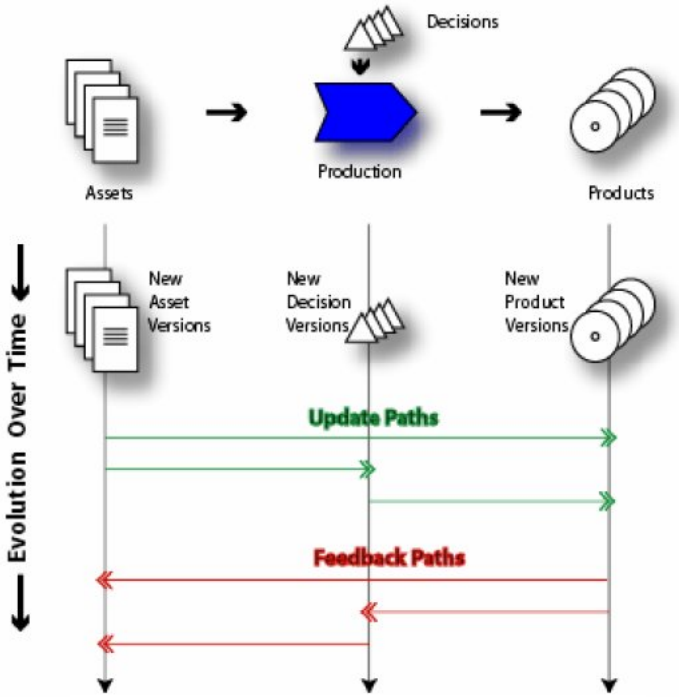


Figure 2.7: Evolution of software product line [1]

SCM systems are designed for managing the evolution in the time dimension, but lack support for managing evolution in the space dimension. Figure 2.7 shows the evolution of the core assets, decision inputs, the products as a two dimensional problem and the different update paths that can occur. A change in a core asset may propagate to the products through the *update path*. Application development may find bugs or introduce common code that leads to the core asset being updated through the *feedback path*.

2.8 Overview of SPL SCM Approaches

This section gives an overview of different approaches and proposals to deal with configuration management in SPL. The approaches and proposals all rely on conventional SCM tools such as CVS and Subversion.

Figure 2.8 is the generic SCM model described by Clements and Northrop [12]. In the model [2], core assets, custom assets, and product instances or derived product are kept under configuration management. For each *product instance* under SCM,

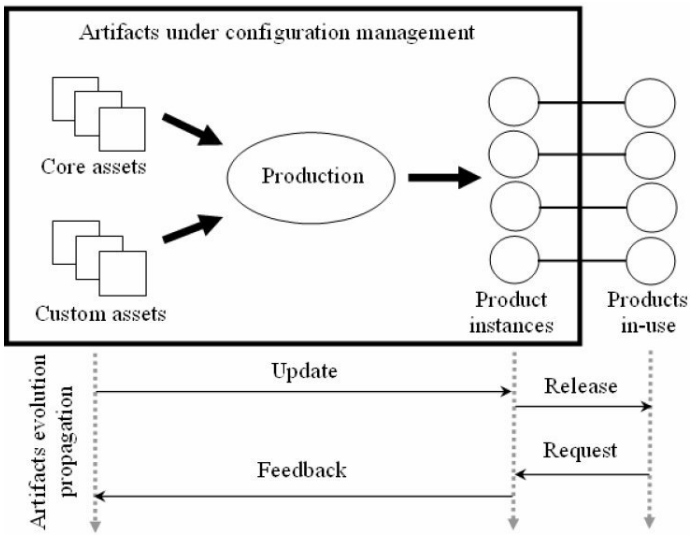


Figure 2.8: Generic CM model [2]

there's a corresponding *product-in-use.* A product-in-use is a product that is released to users. The relationship of product instance and product-in-use is a one-to-one relationship. This model allows for the independent development of core assets and products. Because each product-instance is kept under SCM, we can retrieve any version of the product. Changes made to the core and custom assets are propagated to the product instances through the update path. The updated products are then release again to the user through the release path. User change request in the product is handled through the request path. Changes made to the product are propagated to the core assets and custom assets through the feedback path.

Staples [22] describes four problems that could arise from SPL due to their nature. The first is that SPL configuration management is a multi-dimensional problem and thus it is difficult to deal with traditional SCM tools. Second is that changes to the SPL architecture can affect multiple core assets because the core assets interfaces depend on the architecture. Third is that products have different release constraints and their release may depend on conflicting release constraints such as schedule and quality. Last is that a product line can ''decay'' from having multiple similar functionality implemented in multiple custom components.

He provides some guidelines to deal with the SCM problem of SPL. He suggested that new product versions do not have to use the most recent of core assets that meet the requirement. In some case, it may be desirable to use older versions when

new core assets introduce incompatibility. Staples also suggested that a product can use a product specific branched version of a core asset, but this should only be done sparingly and temporary. Again, this can help developers deal with new incompatible core assets.

van Gurp [23] proposes combining variability tools and Subversion to support product derivation and configuration management. He argued that subversion is suitable because Subversion supports file system based versioning, copy by reference, and annotations by associating properties (name value pairs) with any artifact under version management. The annotations can be used to link variability model information and artifacts in the version system.

van Ommering [9] at Philips describes the configuration management approach in their Koala component based product population using conventional SCM tools. A product population is software product line with enough difference to require variant architectures. The Koala components have three types of interfaces: provides interfaces, requires interfaces, and diversity interfaces. A provides interface specifies the environment and what the component can do, and a require interfaces describe what the component needs. The diversity interfaces are used to fine-tune the component into a configuration. Components are organized in packages where each package implements a specific sub-domain functionality. Components can be public or private within a package. The Koala components can be combined to form compound components and compound components can be combined to form (more) compounded component. A product is constructed by instantiating components and binding the diversity interface, and other interfaces. Product variation is managed through steps below.

There is a team assigned to each package development and may use their own SCM system. A package team issues formal releases of the package and clients of the package can download them from the intranet in a ZIP format. Each formal release must be compatible with the previous release package. This is called the ‘‘golden rule” and it allows them to simplify their configuration management problem. When the ‘‘golden rule” fails, there’s the ‘‘silver rule” which states that clients of the package should build with the new release of the package.

Variation is handled in the component through the use of preprocessor `\#ifdefs } and conditio`
He also describes their branch management called temporary variation in the small

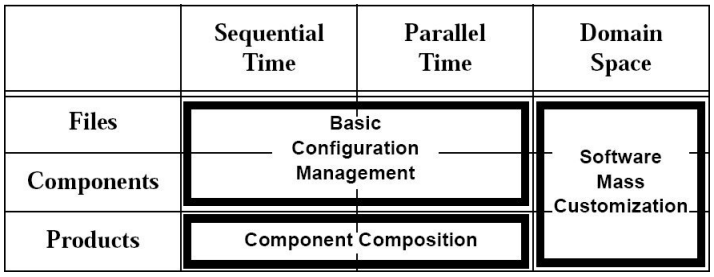


Figure 2.9: Variation management clusters [1]

and in the large. Temporary variation in the small refers to branching for the purpose of multiple developers making small fix or adding new feature in parallel and later merging the changes. Temporary variation in the large refers to branching of a package for a specific product just before the product is released. The idea is to avoid introducing bugs when new features are added to the package.

Krueger [1] describes a "divide and conquer" approach to SPL configuration management. He describes this as "variation management" as compare to configuration management in single product development. The problem of variation management is divided into subproblems and some of the sub problems are solved by traditional SCM tools as shown in Figure 2.9. In Figure 2.9 the rows represent the granularity of software artifacts. The columns represent different type of variation: sequential time, parallel time, and domain space. Variation in sequential time is the evolution of software artifacts over time in a single development path which refers to revision in traditional SCM. Variation in parallel time refers to evolution of software artifacts in parallel development path which refer to branching in SCM. Variation in domain space is the difference in the products in the product line such as features and quality. Variation of files are composed to create variations of components. Variation of components in turn are composed to create variations of products.

Files and components under sequential time and parallel time are managed by conventional SCM tools while products are supported by component composition tools. The domain space is handled by mass customization tools. Composition and mass customization will be described below. Krueger describes this process as variation management of the production line. In Krueger's approach, products are simply transient outputs. All changes are made to the common and variant artifacts. In contrast, in the conventional approach, derived products evolve and are managed as

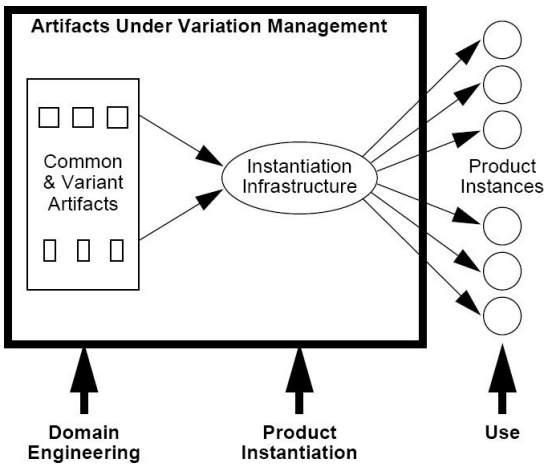


Figure 2.10: Variation management of a production line [1]

separate entities. In Krueger's approach, there is only one product to manage and that is the production line. Figure 2.8 shows the conventional approach while Figure 2.10 shows Krueger's approach.

Component composition is the process of composing different components to form a product. Software customization is the process of specializing a product. The component composition and software customization layer uses the configuration management layer to supply the correct version at a fixed point in time. To avoid the combinatoric problem of component composition, Krueger suggests using the ‘‘context’’ approach. A context represents a possible composition of components versions. The context can be implemented as a simple list in a file that tracks the ‘‘latest and greatest" composition of components for a product. The list is kept under SCM as it evolves and branching occurs. Customized products are instantiated from customized components which are instantiated by selecting the appropriate variation points in the domain space. The variation points in the domain space are maps to common, variant files, and components.

van Deursen et al. [8] describe a feature-based product line instantiation using source-level packages in their DocGen product line. Product line instantiation is the same as product derivation. Packages are developed separately. A package either implements a feature, or functionality shared by other packages. A product is assembled by merging the sources of the packages through a technique called source tree composition. Variation management is handled through the use of a feature

description which is a textual representation for the feature diagrams of the Feature Oriented Domain Analysis method FODA [24]. An FDL definition generates all possible feature configurations or product instances. Instantiating a DocGen product for a particular customer has the following step: selecting variable features to be included, selecting the corresponding packages, setting appropriate configuration switches, and writing customer-specific code for features that can not be expressed as simple switches. The authors did not specify what they used for managing the evolution of the packages, packages definitions, and features descriptions. Since the packages are developement independently, any conventional SCM systems can support it.

2.9 Summary

We have described some approaches and proposals to SCM problems in SPL. The generic model of SCM in SPL is that core assets, and derived products are kept under an SCM system. The core assets and the derived products would evolve independent of other products as seen in Figure 2.6. There's the notion of change propagation or forward feedback from core assets to the products, and from products or backward feedback to the core assets.

Staples helps define the problems arise in SCM from SPL and proposes very general ways to lessen the problem. In Staples proposal, generic SM systems will suffix. van Gurp proposes coupling variation modeling tools with Subversion to support product derivation. He has yet to have a prototype to prove that the idea works.

van Ommering describes the Koala component based software and conventional SCM systems in implementing a product population. In this approach, new release of packages should be backward compatible. Krueger describes an approach that uses conventional SCM tools. The core assets and product line instantiation infrastructure are kept under SCM. Products are generated and are not kept under SCM. All changes are made in the core assets and custom assets. van Deursen describes a package based product line which is similar to van Ommering and Krueger's approach. Ommering's approach is not generic in that it relies on the Koala technologies. Variation within Kaola is implemented using `\#ifdef } preprocessor directives which can lead to`maintainability [2
14]. In Krueger's approach, there's no separation between the core developers and the

product developers. Since only the production is under SCM, it does not allow for large numbers of independent product evolution branches that are maintained under SCM. In van Ommering, Krueger, and van Deursen's approach, forward propagation is automatic. Since changes occur in the core assets, products that uses the latest get the new changes. Dependency among components and products are manually maintained.

Chapter 3

Approach

3.1 Introduction

This chapter describes our approach and the design and implementation of Molhado SPL, a configuration management system designed for software product lines. Our approach is closer to the generic CM model describe in Chapter 2. All core assets and products are managed under one single process which allows for change propagation. Rather than introducing new programming language constructs to support code composition or the use of conditional compilation to force a software product line into a single product, we propose an approach that models software product lines and keeps core assets and products development independent and yet under the same configuration management system.

The goal of Molhado SPL is to support software product line evolution namely, supporting independent evolution of core assets and products, managing relationships among shared assets and products using the shared assets, and supporting change propagation between products and core assets. Molhado SPL supports software product line evolution by using a versioning model that has a single core assets project and one or more product projects and supports code sharing from the core assets project with the product projects.

Currently, none of the conventional SCM tool such as CVS, Subversion and GIT is specialized for supporting software product line evolution. For example, in CVS the versioning model supports single product development paradigm. CVS does not support code sharing among projects nor does it have any knowledge about software

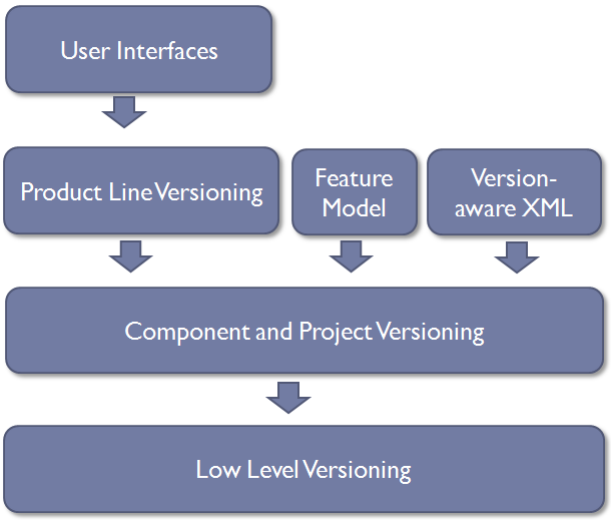


Figure 3.1: Molhado SPL consists of multiple layers.

product lines. Subversion and GIT can share the repository of a project with another project but it has no notion of what products and core assets are. They lack the versioning model and the operations to support software product line evolution. In Molhado SPL, core assets and products are treated as first class entities. Code sharing is built into the versioning model and the evolution of code sharing information is maintained. Once an asset is shared, Molhado SPL manages that information automatically and is the basis for supporting change propagation.

3.2 System Overview

Molhado SPL is implemented in four layers as showed in Figure 3.1. The layers are the low-level versioning layer, the component and product versioning layer, the product line versioning layer, and the user interface layer. The low-level versioning layer provides low-level data structure versioning services. It provides objects with properties whose values are different at different points in time. These objects form more complex data structures such as trees and graphs. In addition, the low-level versioning layer provides the necessary utility objects for storing and loading objects' change histories. The low-level versioning layer by itself is not very useful in terms of supporting software configuration. To support software configuration management, Molhado SPL needs to model software and document projects; hence, the creation of the component

and project versioning layer. The component and project versioning layer provides files, projects, merging, and higher configuration management operations. From the component and product versioning layer, we created the product line versioning layer, which models a software product line, enables code sharing and supports the evolution of software product lines. The interface layer provides the necessary operations and user interface that let users perform configuration management operations and some graphical interfaces for visualization. All of these layers together allow Molhado SPL to support the evolution of product lines. The feature model and version-aware XML modules are built on top of the component and project layer. Feature models are used to represent possible products in a software product line and are described in Chapter 5. The version-aware XML document framework layers allow XML documents to contain their change histories without the need for a repository. The version-aware XML framework is described in Chapter 6.

3.3 Low-Level Versioning Layer

The low-level versioning layer, implemented by John Boyland, is part of the Fluid project at Carnegie Mellon. The goal of the Fluid project is to support programming analysis, program transformation and code refactoring. The low-level versioning layer provides versioned data structures such as sequences, trees and graphs and the loading and storing of versioned data structures. The following are the internal representations of the low-level versioning level.

A *version* is a global point in a tree-structured time. A version also represents a state or a snapshot of all the versioned data structures in the system at a particular time. Any states of the versioned data structures can be retrieved for any version. Versions are organized into a version tree. In the version tree, the parent version is an older state of the versioned data structures in the system while the children represent newer states. The branches in the version tree represent parallel changes or alternative newer states originated from the parent state.

A *slot* holds data such as strings, integers and references to other data structures in the versioning system. A slot can be either versioned or unversioned. An unversioned slot can only hold a single value regardless of how many versions there are in the system. Once a value of an unversioned slot is overwritten by a new value, the old

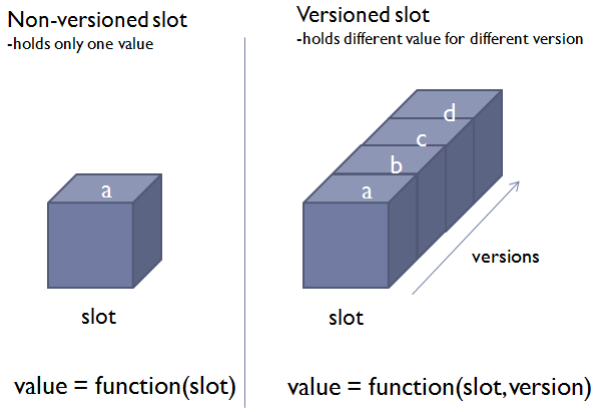


Figure 3.2: The left slot represents an unversioned slot, which has the value "a." On the right is a versioned slot, which has different value for different versions. Give a slot and a version; we can retrieve the value of that slot at that version..

value cannot be restored. In Figure 3.2, the image on the left shows a unversioned slot having the value "a." A versioned slot can have different value at different version as shown the left image of Figure 3.2. Updating the value of a versioned slot creates a new version but the old value can be retrieved using the previous version. In other words, a versioned slot remembers all its values in the version tree timeline and any of those values can be retrieved. Note that in the versioning system, there are many slots and some slots may not have been assigned a value for a version or have not been changed at certain version. Slots that have never been assigned value have undefined values. A versioned slot that has not changed for the new version simply uses the previous value as the value of the new version.

In the versioning system, there are multiple slots and the versions apply to all the slots in the system. Figure 3.3 demonstrates how the values of versioned slots can be retrieved for any given version. In Figure 3.3 example, there are three slots (*S0, S1*, and *S2*) and five versions (*V0, V1, V2, V3,* and *V4*). In a real environment, we may have thousands of slots and hundreds of versions. The y-axis represents the slots and the x-axis represents the versions. Some slots may not have values for some versions. This means either they have not been assigned values or their values have not changed for that version. For example, *S1* has not changed in *V1* and *V2*. When we retrieve the value for *S1* at version *V2*, we simply use the value of the most recent version. If there is no value for such version, we go the next most recent version and repeat until we find a value, which is the value at *V0* for *S1*. The versioning environment can be

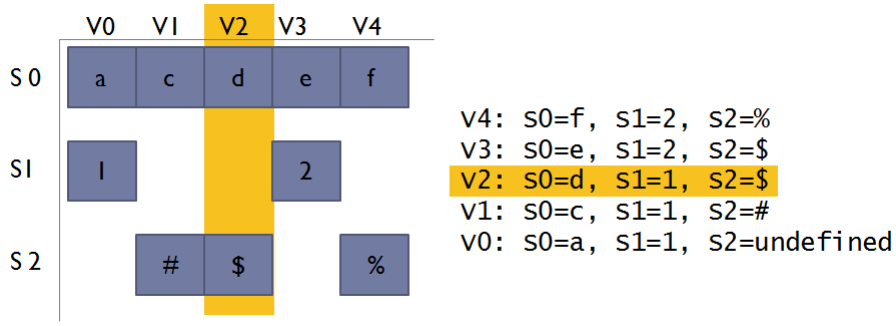


Figure 3.3: Values of slots at different versions.

set to any version and the values of any slot of the environment can be retrieved for that version. For example, if we were to retrieve the values of all slots for version *V2*, then *S0* would have **d**, *S1* would have **1**, and *S2* would have **$** as shown in Figure 3.3.

A *node* is an object in the versioning system. It has an identity and carries no other information. Its intention is to be a node in a tree, or graph but it can stand by itself. The nodes give us the notion of objects in the versioning system but the nodes alone cannot have values. *Attributes* help give node properties with values, which can be versioned. In other words, an attribute is a name mapping from a node to a slot. An attribute defines the type data a slot can store. Currently, there are attributes for string, integers and references to other version structure. The type of attributes can be extended to support other type of data storage. Nodes and attributes can be thought of as a table where the y-axis represents the nodes, the x-axis represents the attributes, and the cells are the slots as shown in Figure 3.4. When we include versions, the table becomes a three dimensional table where the third dimension represents the versions as shown in Figure 3.5. In actuality, the three dimensional table is very sparse as shown in Figure 3.6. A slot that has no value for a version does not use any storage; thus, it uses no memory. In Figure 3.6, values that do not exist for a version are represented as empty spaces. When we say an attribute of a node has a new value, we are referring the slot mapped by the node and attribute having assigned a new value.

A sequence is a linear data structure that holds multiple values of the same type. It is made from a set of slots. It can be either versioned or unversioned based on the type of slots it uses. Trees and graphs represent the most complex data structures in the versioning system. Nodes in a tree or graph have a *parent* attribute and a *children* attribute. The *parent* or *parents* attribute maps to a sequence that holds the parent

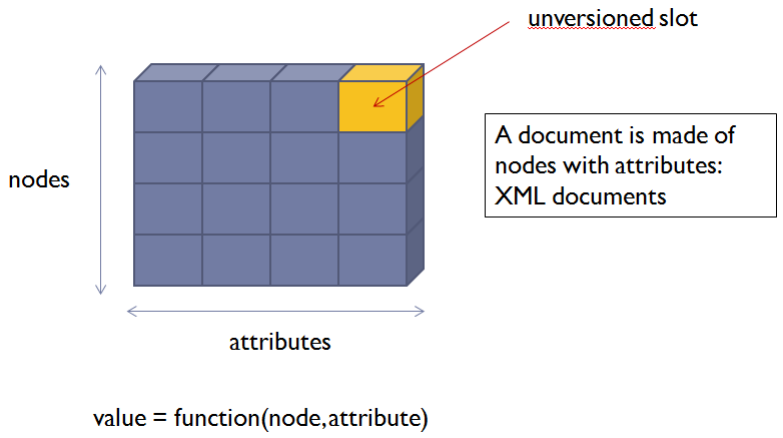


Figure 3.4: Nodes and attributes can be thought of as a table where the rows are the nodes, the columns are the attributes, and the cells are the slots. A value then is a function of a node and an attribute.

nodes. The *children* attribute maps to a sequence that holds the children nodes. If the sequences are versioned, the tree or graphs structure is versioned. The node in the tree or graph can be assigned attributes in addition to the default *parents* and *children* attributes. This allows us to model arbitrary graphs and trees by introducing new attributes. For example, the version tree is created using an unversioned tree data structure and one of its properties is a version object. A tree can be used to represent a project tree, a directory tree, a document tree in an XML document or an abstract syntax tree of a Java class. A graph can be used to represent a graph model such as a feature model or a UML diagram.

Figure 3.8 shows a versioned graph. At *version 0*, the graph consists of node **a**, node **b** and an edge connecting nodes **a** and **b**. At *version 1*, node **c** is added and is connected to **a**. At *version 2*, node **d** is added connecting to **b**. In *version 3*, the connection edge between **a** and **b** is removed. The structure of the graph at any version can be retrieved. This applies to trees and sequences as well. In actuality, the newer version only has the deltas (or differences) from the previous version. For example, *version 3* only has information about the edge between **c** and **d** and the removal of the edge between **a** and **b**. The graph for a version is computed from previous versions in additional to any new information for *version 3*. It is an efficient way to represent multiple trees or graphs that share change history. This idea is further demonstrated in the XML merging chapter. Figure 3.9 shows how two version

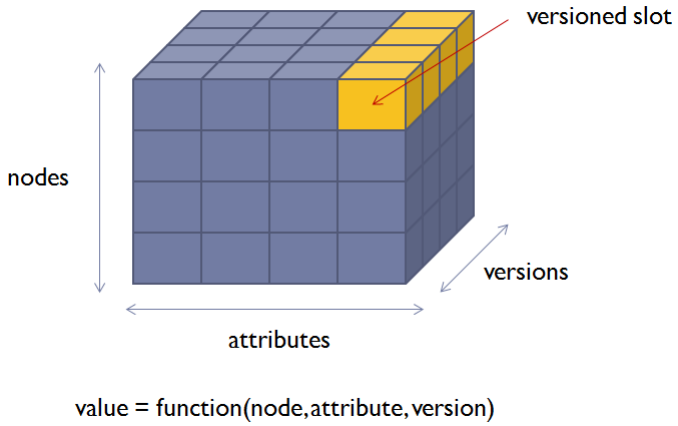


Figure 3.5: With versions, the table can be thought of as a three-dimensional table where the third dimension is time (versions). A value is the function of a node, an attribute and a version.

of trees are represented internally. Each node has a *name* attribute. Attributes that do not have values are given the value **undef**. In *version 2*, node **d** is deleted and node **e** is moved. In abstract, these are structural changes, but in implementation, they are value changes in attributes as shown in the tables to the left of the trees.

In the versioning system, once a node is created it can never be removed from the system. Once a value has been assigned to a versioned slot for a version, the value will never change for that version. Removing or updating the value of a versioned slot simply produces a new version. The previous version still has the previous value. To delete a node from a tree or graph, it is detached from the graph and this produces a new version of the graph. The node can be referenced but in the current version of the graph, the node simply has no parent or children.

There are utilities objects that do not perform versioning but provide support for recording changes to a tree between versions and support for access and version manipulation across multiple threads.

- A change record is used in trees or graphs to track nodes that change between any two versions. A tree or graph may be associated with multiple change records. Change records can also be made to listen to changes in a particular attribute other than attributes assigned to trees and graphs.

- A version tracker is used to support versioning when multiple threads are

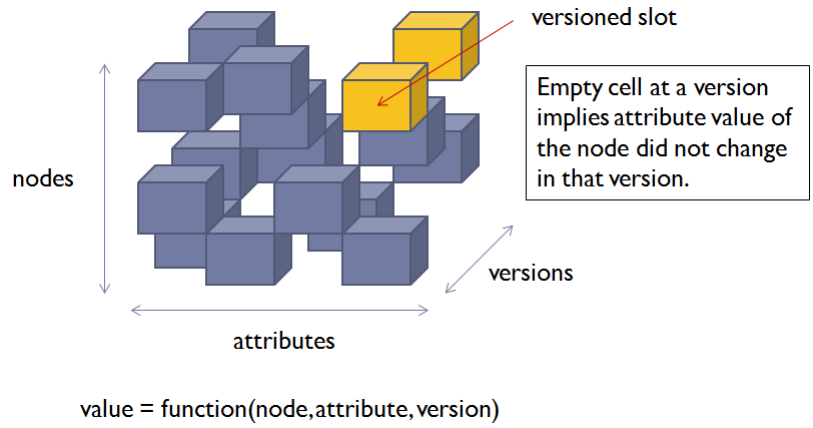


Figure 3.6: The table representation is very sparse. Change often does not occur everywhere. For example, a version of project may only have a small change to a file. The empty spaces in the table do not use memory, as the actual implementation of the table is list.

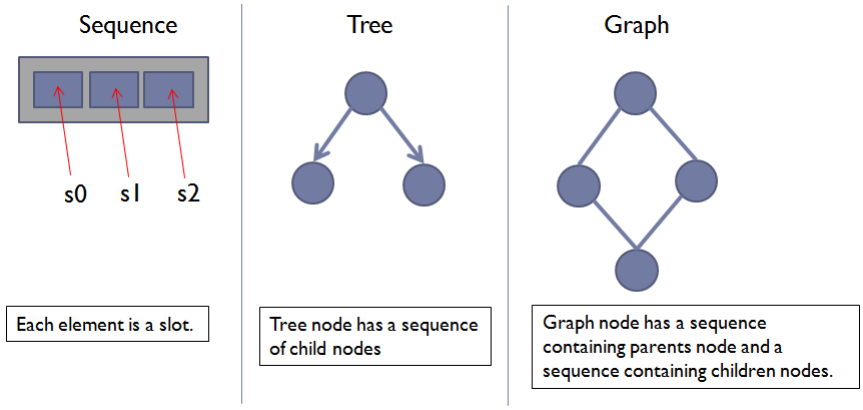


Figure 3.7: Complex data structures created from nodes, slots, and attributes.

involved that may read and update values. Different threads manipulating a version will cause the version to branch and a version tracker is used to keep the versions linear. Without them, changes in one thread will not be seen in another thread. A version tracker can be thought of as a way to synchronize changes to a version.

Several objects are responsible for storing and loading of both versioned and unversioned data. Each object is assigned a 62 bits unique ID in the versioning environment. Those objects are as follows:

- Eras are responsible for saving and loading the version tree. Versions are

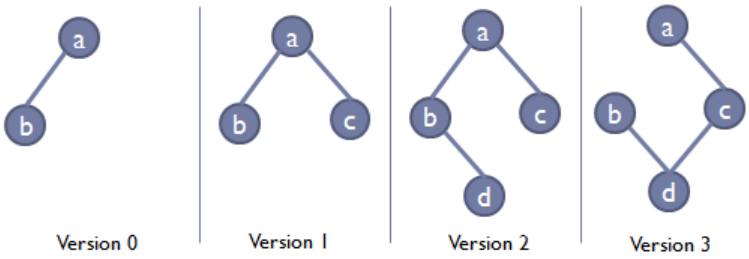
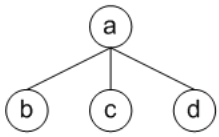
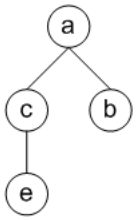


Figure 3.8: The states of a versioned graph at different versions.



(a) Version 1

node	name	children	parent
a	"a"	b,c,d	*undef*
b	"b"	*undef*	a
c	"c"	*undef*	a
d	"d"	*undef*	a



(b) Version 2

node	name	children	parent
a	"a"	c,b	*undef*
b	"b"	*undef*	a
c	"c"	e	a
d	*undef*	*undef*	*undef*
e	"e"	*undef*	c

Figure 3.9: Representing trees with attribute name.

organized into eras and are stored by era such that an era will load only those versions that are associated with that era. The version tree is composed of one or more eras with each era responsible for sub trees of the version tree. Figure 3.10 shows the relationship between era and versions.

- Regions are responsible for saving and loading nodes. Nodes are partitioned into regions and no node may exist in more than one region. Regions have unique IDs.

- Bundles are responsible for saving and loading attributes. Attributes are partitioned into bundles and each attribute must be assigned a unique name. Bundles must have unique IDs.

- Chunks are responsible for saving and loading slots and their values. Chunks

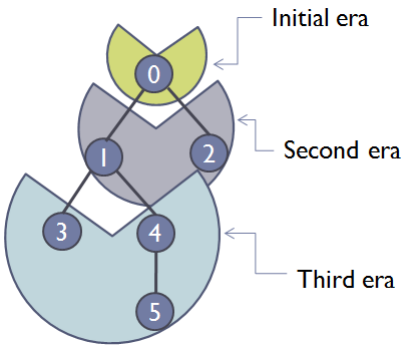


Figure 3.10: Relationship between the version tree and eras. An era represents a sub tree of the version tree but excludes the root of the sub tree. In this figure, the initial era has one version, version 0 and has no root node. The era holding the root node is a special type of era. The remaining eras require a root node. Ideally, each era could represent a single saving session.

represent the difference between versions or eras. The difference is represented as a reverse delta such that to retrieve the current state of the system, we would need to retrieve the previous state if the current state does not have all the information. To load a chunk, we need an era, a bundle, and a region. Figure 3.11 shows how nodes, attributes, and versions are organized into region, bundles, and eras.

3.4 Component and Project Versioning

Because Molhado SPL is designed to work with software product line, *components* are created to represent files and directories in a file system. The component object is responsible for saving and loading of the content of the component. At the minimum, it has a name, a creation date, a modification date, and who creates it. A *directory component* has a name and its type but has no content. Its children are stored in the tree representing the file system called the *component tree*. A *file component* represents a file such as a text file that has content. It has a name, a tree representing the content, and a region to hold, store, and load the nodes. It is associated with a bundle with a set of attributes that define the content of the component. All components of the same type share the same bundle but each component has its own region as they are each responsible for their own storage. The tree in the component

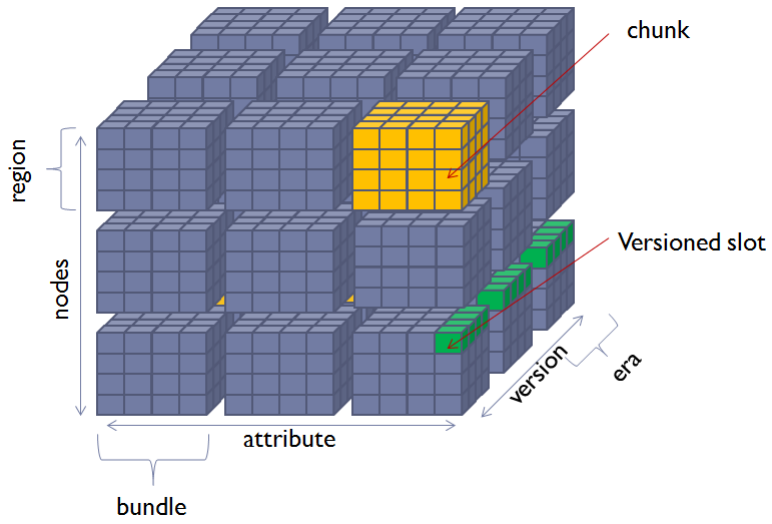


Figure 3.11: Nodes, attributes and versions are grouped into region, bundle, and era. The intersections of these are chunks, which represent the difference between eras (versions).

represents the file content of the component. An XML document is represented by tree with the necessary attributes to support element names, and an arbitrary number of XML attributes and values. A text file is represented by a tree of one level deep, where the children of the root node represent the lines. Originally, Java files were represented as an abstract syntax tree but we later decided to represent them as text files.

In order to support any kind of collaborative editing, merging of changes is needed. The low-level versioning layer provides no merging capabilities and we have not found a way to support it at the low level. We created two forms of merging: tree merging and sequential merging to support component merging. The tree merging is specific for XML document trees and the sequential merging is for text files. The XML merging and XML representation are described in Chapter 4. The sequential merging is the typical textual 3-way merging that is implemented by the standard 3-way diff tool, which is used by conventional configuration management tools.

A *project* represents a software or document project. It can be thought of as a versioned file system. It has a tree that represents the directory tree of the entire project. Storing and loading of a project is handled by the project object. Each tree node points to a component. Each project has its own version space thus allowing

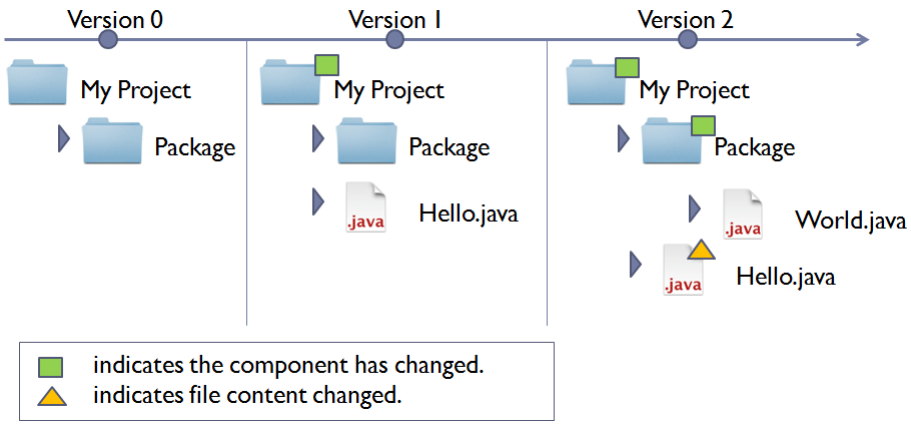


Figure 3.12: Project versioning

each project to evolve independently of other projects. Projects are managed by an object called Project Manager. It tracks which projects are opened and which are closed. A project will only load a component when it is accessed thus avoiding loading the all components of the entire project at once. In addition, it will only load the necessary deltas (chunks) to construct the request version.

A project is versioned as a single object. A project's version applies to all of the components within that project. Any change to any of the components in the project results in a new version for the project. This is called *project versioning*. Figure 3.12 demonstrates the concept of project versioning. The project has undergone three versions. *Version 0* has a folder, **My Project**, which has a folder **Package**; *Version 1*, **Hello.java** is added to **My Project**; *Version 2*, **World.java** is added to **Package** and **Hello.java** has been modified. If we were to retrieve any version of the project, we would have the directory structure of the project and the changes made to those files at that version. For example, if we were retrieved *Version 0*, we would get **My Project** with **Package**. However, if we retrieve *Version 2*, we would get **My Project** with **Package** and **Hello.java** and **Package** has **World.java**. Project can be thought of as a versioned file system that remembers changes made to it. The versions are the snapshots of the project at different points in time. The snapshots are created by the user when they check-in their changes to the project.

The component and project versioning layer also adds some enhancements to the version tree to support meaningful branching, merging and version labels. Now there is the notion of a main development trunk in the version where main development occurs

and branches to support specific changes and experimental changes. For example, a branch can be created to support a bug fix. Unless a check-in is specified as a branch, it is pushed to the main development trunk. When users check-in changes, they can assign tags and descriptions to the version. The extension to the version tree supports merging of versions. A version that is the result of a merge has two parents: source and target. The source versions changes are applied to the target version resulting in the merged version. In addition, graphical visualization of the version trees is provided. Unlike conventional SCM tools, the source version is kept in the system and can later be retrieved. Conventional SCM tools simply throw away the source version and keep only the merged version.

3.5 Product Line Versioning Layer

For our versioning system to support software product line evolution, it needs a versioning model that can model software product lines, allows for independent evolution of shared code and products, manages shared code and product relationships, and supports change propagation. We use a product line model, in which core assets are developed independently from products, core assets can be used in multiple products and changes may propagate from core assets and products. We named the model, *Product Line Versioning Model*. Our model is derived from the generic model described in Chapter 2. It consists of a single core assets project and one or more product projects. These projects are extensions of the project object from the component and project versioning layer. The core assets project is where all the core assets are created and managed. A product project represents a single product. It may share code from the core asset project and it may have product specific code (see Figure 3.13). Because each project has its own version space, it can evolve independently of the core assets and other products. We only allow code from the core assets project to be shared with products and not the other way; although, it would be possible to support this.

3.5.1 Shared Component

Reuse is a central theme of product line engineering, so any versioning system that would support product lines natively should support artifact sharing within the versioning system. Our approach uses shared components to enable reuse. They contain some additional metadata that identifies the project containing the component and the relevant version of the project. In addition, Molhado SPL manages the evolution of shared components. A shared component can refer to objects of any granularity, from a single file to a directory containing files or other directories. Figure 3.13 shows how shared components are used to share core assets in multiple products. The green box represents a shared component and the edges denotes sharing.

The implementation of shared components allows sharing across products but that flexibility may complicate things. We do not want sharing to be spaghetti like. So, we force a sharing policy under which only products may share assets from the core asset project. An asset in the core assets project may be shared by multiple products. Any artifacts in a product that are to be shared by other products must first be moved to the core asset project. The point in time when changes should be moved from products to the core assets depends on policy set by the software developers. Molhado provides the mechanism but does not provide guidance as when changes should propagate and who gets to make those decisions.

Shared components are versioned objects themselves and so they can evolve over time. At different versions, a shared component may refer to a component at a different version or a completely different component. Figure 3.14 demonstrates the idea of versioned shared component. The figure shows the evolution of the shared component **a** in a product and the referenced **a** in the core assets project. At version P_{v1}, the shared component in the product is referring to the component **a** of version C_{v1} in the core assets project. Component **a** continues to evolve in the core assets project resulting in versions C_{v2} and C_{v3}. The product developers decided that they would like to have the new changes made to **a** in the core assets project and now the shared component in the product is made to refer to the latest version of **a**. Thus, in version P_{v2}, shared component **a** now refers to component **a** of version C_{v3} in the core assets project. Note that if we retrieve the product at version P_{v1}, shared component **a** would refer to **a** of version C_{v1} in the core project.

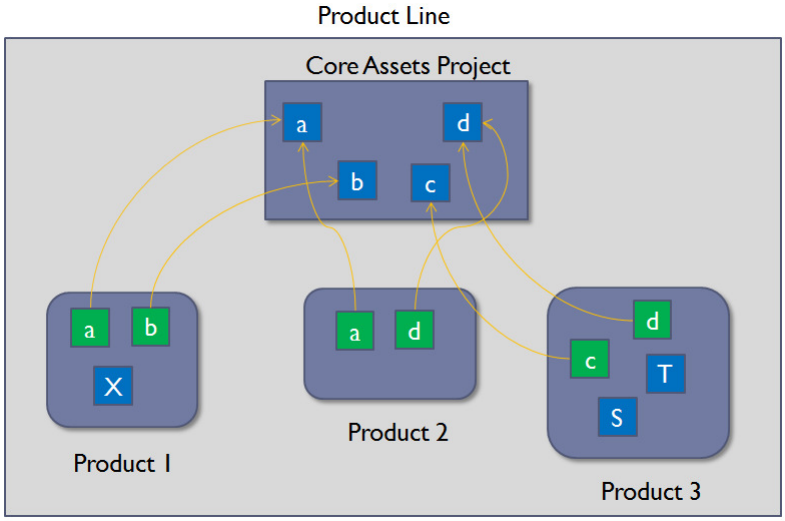


Figure 3.13: Product line versioning model. Green box represents shared component, yellow edge represents sharing, and blue box in products represents product specific code.

In addition, the shared component concept allows for product specific changes to shared components without interfering with changes made to the component in the core project. For example, a shared component may require small changes to make it work in a product or a feature may be introduced into the shared component that is specific to the product. These changes should not be visible to other products mean while the component in the core assets project continuing evolving without interfering with the shared component and its product specific changes.

3.5.2 Change Propagation and supported cases

This section describes the kinds of change propagations that are supported by Molhado SPL. Changes can propagate from the core assets project to products or from products to the core assets project. When changes propagate from core assets project to products, this is referred as *forward change propagation*. An example of forward change propagation is the update of a shared asset in a product with the more recent changes made to the same asset in the core assets project. These changes could be error correction, new content in a file, or a set of files in a document product line or a bug fix or improvement in a software product line. When changes propagate from a product to the core assets project, this is *backward change propagation*. An

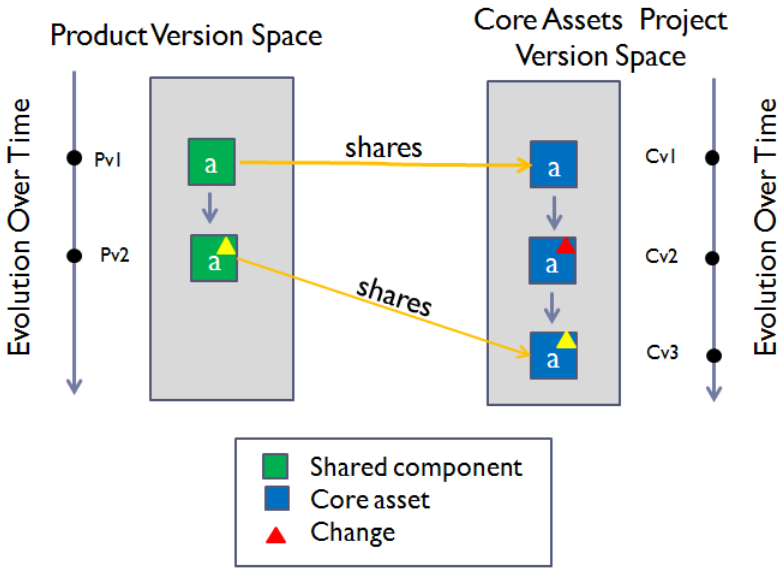


Figure 3.14: ersioned shared component. A shared component may refer to different version or different component at different version of the product.

example of backward change propagation is the propagation of an error correction made in a shared asset in a product to the core assets project so that other products can incorporate the changes to their share assets. As a matter of policy for SPLs, backward change propagations should occur only when the changes being propagated is important to the product line so other product can use.

Table 3.1 describes all possible forms of change propagation that can occur by showing before and after states of a hypothetical asset **a**. Changes to an asset **a** in the core assets project are indicated by $\mathbf{a}'$ and changes to an asset in a product project are indicated **a***. The merged result of the changes of the asset from both core and product is indicated by $\mathbf{a}^{*\prime}$. From Table 3.1, cases one through 4 show forward change propagation while cases 5 to 8 represent backward change propagation. Case 4 and case 8 show the propagation of newly created assets. Our prototype supports all of these forms of change propagation.

The following describes each of the cases in more details:

- Case 1: The product is sharing the asset **a** from the core assets project. Changes have been made to **a** in the core assets project. In this case, the changes made in the core assets project are brought to the shared asset in the product. An example of this case is when a correction made to an asset in the core is useful to

	Before		After	
Case	Core	Product	Core	Product
1	$\mathbf{a}'$	a	$\mathbf{a}'$	a$'$
2	$\mathbf{a}'$	a*	$\mathbf{a}'$	a*$'$
3	$\mathbf{a}'$	a*	$\mathbf{a}'$	a$'$
4	**a**		**a**	a
5	**a**	a*	**a***	a*
6	$\mathbf{a}'$	a*	$\mathbf{a}'$*	a*
7	$\mathbf{a}'$	a*	**a***	a*
8		**a**	**a**	a

Table 3.1: Possible cases of change propagation. Boldface items are concrete, while items in non-bold characters are represented indirectly by shared components.

the product sharing the asset and thus the changes are pushed to the product.

- Case 2: The product is sharing the asset **a** from the core assets project. Changes have been made to the shared asset **a** in the product and to the asset **a** in the core assets project. In this case, the changes from the asset **a** in the core project is merged with the shared asset **a** with the product specific changes. Now **a** of the product has both set of changes. This case might represent a product's independent evolution while bringing correction changes from the core project.

- Case 3: Changes have been made to the asset in both the core assets project and the product project. The author wants to replace the modified asset in the product with the modified asset in the core assets project. After the author performs this action, the asset in the product will be identical to the one in the core project. In this case, the author may find that the product specific changes might not be useful and could be replaced with the changes made in the core project. Perhaps both changes address similar issue.

- Case 4: An asset from the core assets project that had not been shared with the product is now shared with the product. It may be that the asset is needed by the product or found to be useful to the product.

Cases five to case eight are similar to cases one to four but changes are propagated in the opposite direction. There is a semantic difference between case four and case eight. In case 4, a product is made to share a component from the core assets project.

In case eight, a component is moved to the core assets project so that other products can use it.

3.5.3 Implementing Change Propagation

Molhado SPL allows product specific changes to shared components without interfering with the changes made to the referred component in the core project. To support product specific changes to shared core assets, the core assets project create a product specific branch to support the changes in order to avoid interference between the products changes and the changes in the core asset project. When a product developer checks in their product project with changes to a shared core asset, the core assets projects automatic create a branch to support it. Subsequent check-in simply of changes to this shared asset for that particular product creates more versions in the product specific support branch created earlier.

The following is a walk through evolution of a simple product line example that demonstrates how Molhado SPL supports product line evolution and change propagation. Figure 3.15 depicts the main development trunk of the Product A (PA) project, the Core Assets (CA) project, and the Product B (PB) project. The Core Assets project consists of asset **a**, **b**, and **c**. Product A is using **a**, and **b** from the Core Assets project while Product B is using **a**, and **c**. Below each of the box are the version trees for each of the project. In Figure 3.15, the version trees of the three projects consist of trees with one version. In the figure, Product A is at version PA_{v1}, the Core Assets project is at CA_{v1} and Product B is at version PB_{v1}. Note that the version's prefix indicates the name of the project. The green boxes in the products indicate shared components. For example, the **a** component in Product A is a shared component referring to the **a** in the Core Assets project at version CA_{v1}.

In Figure 3.16, changes are made to **a** of Product A, **a** of the Core Assets project, and **c** of Product B. The changes made in both Product A and Product C are to shared assets. To support product specific changes to shared assets, Molhado SPL automatically creates branches (shown as red) to support the product specific changes as indicated in the Core Asset project's version tree. These branches are created when these changes are checked in. PAb_{v1} is created to support the change to **a** in Product A and PBb_{v1} is for Product B's changes to **c**. The main idea is that all changes to

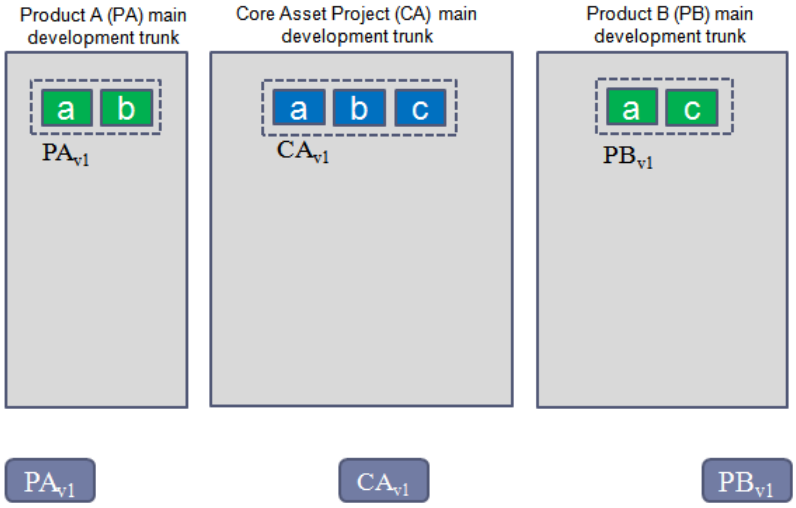


Figure 3.15: Product A is using **a** and **b** and product B is using **a** and **c** from the core asset project. Product A is at version PA_{v1}; product B is at PB_{v1}; and Core Assets project at CA_{v1}.

shared components are stored in the Core Assets project and special branches are created to support product specific changes.

Figure 3.17 shows how change propagation are performed. Product A updates **a** with the changes made in the Core Assets project (forward change propagation). This results in version PA_{v3}. Behind the scenes, Molhado SPL performs a merge of **a** in the Core Assets project with **a** of version PA_{bv1} (product A's support branch for **a**). Now the shared component **a** of Product A is referring to **a** of version PAB_{v2} which has both changes. The change made to **c** in Product B is pushed to the Core Assets project (backward change propagation). In addition, the product specific component **d** is pushed to the Core Assets project so that other products can use it. Molhado SPL performs a merge of the changes made in **c** in Product B of version PB_{v1} with **c** of Core Assets project of version CA_{v2} resulting in CA_{v3}. Now **c** in the Core Assets project has the changes pushed by Product B and the component **d**. When **d** is propagated or pushed to the Core Assets project, **d** is also removed from Product C and a shared component is created in its place that refers to the **d** in the core assets project.

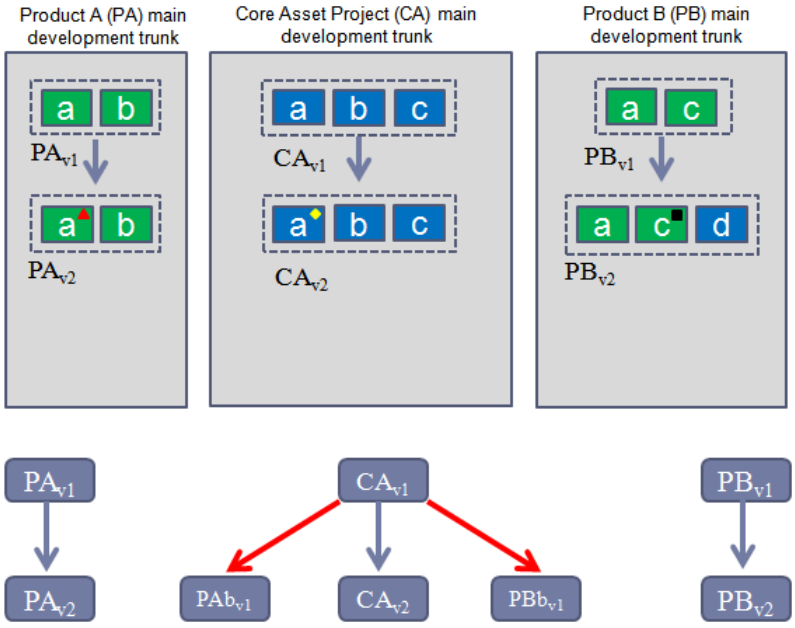


Figure 3.16: Product A has product specific changes to **a** resulting in version PA_{v2}. Core asset has changes to **a** resulting in CA_{v2}. Product B makes product specific changes to **c** and introduces a product specific component **d** (shown in blue) resulting in PB_{v2}.

3.6 User interfaces

Molhado SPL provides a set of user interfaces for working with projects and versioning support. The UI is built on top of the NetBeans Platform [26]. It provides a UI for accessing the different projects and for editing components. Components that are shared are indicated by the duplicate file icons and the *shared* annotations next to the names. Product specific components appear as regular files. Each of the projects loaded is associated with a version. It provides text editors and visual differences between files. Shared component can propagate changes either through push and pull dropdown commands called. The UI also provides version trees for each project. Figure 3.18 is a screen shot of the interface.

3.7 Summary and Discussion

In this chapter, we have described Molhado SPL for supporting product line engineering. It is flexible enough to support different type of projects from document product line

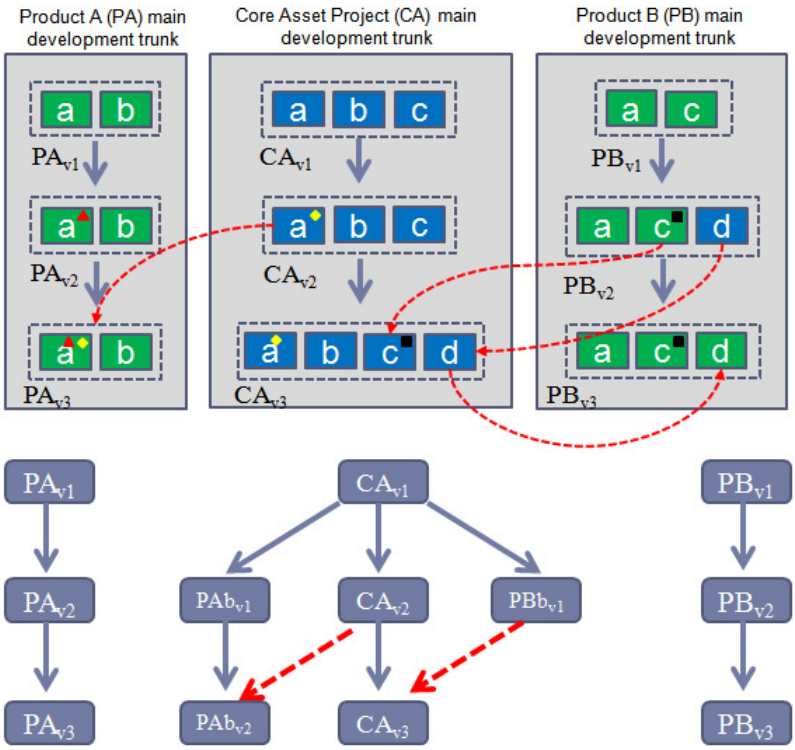


Figure 3.17: Product A updates **a** with changes from the core assets project (forward change propagation). Changes to **c** in product B is pushed to the Core Assets project (backward change propagation). In addition, product specific component **d** is moved to the Core Assets project so other products can use (backward change propagation). As a result, product B now shares **d**. Change propagation is supported by merging (indicated as red) as indicated in the core assets project's version tree.

to software product line. It has a version model for a product line consisting of a single core assets project and multiple product projects where core assets are shared among the products with the use of shared components. Using the shared component data structure and the branching of the core assets project, we are able to support independent development of core assets and products and change propagation between them. Molhado SPL supports eight cases of change propagations.

Older SCM systems such as RCS [27] and CVS [6] do not support code sharing. More recent SCM systems such as Mercurial [28], Subversion [20], GIT [29] and Bazaar [30] supports sharing of repositories, which are closer to the idea of sharing component. They do not address the software product line evolution problem. Because of their support of sharing of projects, it may be possible to adapt one of the systems to support product line and this could be future research topic.

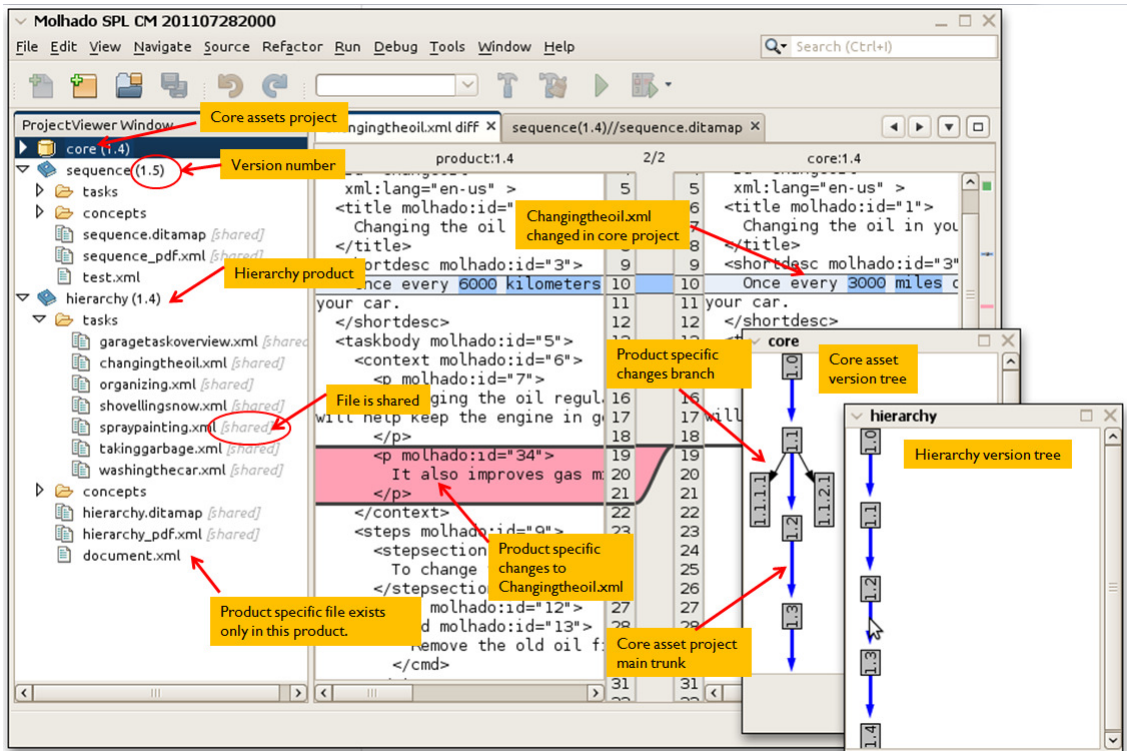


Figure 3.18: User interfaces

Chapter 4

Three-way XML Document Merging

In a collaborative environment, team members are often working in parallel. They may be modifying the same files in isolation and may be unaware of each other's changes. Different team members may be modifying different parts of the same document or they may be modifying the same part of the document with *conflicting* changes such as modifying the same line or updating the same attribute value of a particular XML element. Although, it is possible for the team to integrate these changes manually, it would be a painful process especially when changes are made to a large number of files. Thus, for collaborative work, it is crucial to have tools to automate the process of integrating or *merging* changes made by multiple team members into one unified document, and to help detect and resolve conflicting changes during the merge process. For text files, there exist tools such as the GNU *diff3* [31] for merging two documents derived from a common base document. This process is called *three-way merging* because it requires three documents: the original or *base* document and the two modified or *derived* documents. Software configuration management (SCM) tools such as CVS [6] and Subversion [20] make heavy use of three-way merging to support collaborative editing of program source code.

There exist both commercial and research tools to perform three-way merging of XML documents [32, 33] . These tools rely on hash values of nodes and node content for node matching and can miss nodes in XML documents that have undergone large transformations. To do a complete merge, these tools build three if not four trees in

memory even when changes are trivial. These tools can merge changes to a sizable single file in a few seconds. However, in big projects, document sets are large and the time to merge many documents could be substantial and irritating. Furthermore, the tools lack useful conflict resolution interfaces, which adds considerably to the effort required for practical document merging.

Our approach to three-way merging starts with the use of unique IDs so that nodes that have undergone substantial transformations can still be matched with their original version. The unique IDs also help to better distinguish move operations and to improve conflict detection. To reduce memory usage, we use a versioned tree data structure so that only one full document tree must be created in memory, along with enough tree deltas to represent any changes. The versioned tree supports a change history that allows it to merge only those nodes that have changed while ignoring all others.

The outline of this chapter is as follows. The semantics of XML that are relevant to merging are described in Section 4.1. Section 4.2 gives a brief introduction to three-way tree merging. We describe how versioned XML documents are mapped to versioned tree structure in Section 4.3. Section 4.4 and 4.5 give the definitions of the longest common subsequence algorithm and the diff3 merge algorithm. Section 4.6 describes the merge and conflict rules as well as the handling of false conflicts, node mapping and the merge algorithm. Finally, Section 4.7 presents an evaluation of the algorithm, while Section 4.8 discusses related work and the conclusion and future work are described in Section 4.9.

4.1 XML Tree Semantics

XML (Extensible Markup Language) [34] is a markup language that is used for many purposes ranging from representation of data in remote procedure calls to configuration files such as Ant build scripts, to human oriented documents found in publications, documentation, and marketing materials. Some examples of XML representations include the document representations found in OpenOffice, Microsoft Office, and DocBook, plus various Internet languages such as xHTML, SVG and many more.

The XML language definition specifies that the order of elements is significant, but that the order of attributes in an element's start tag is not significant. So, XML defines

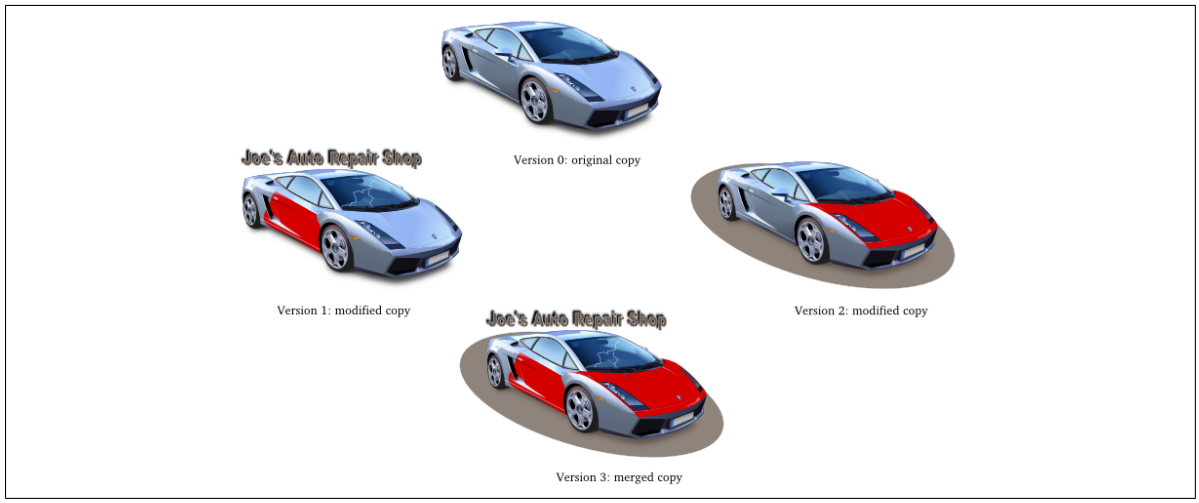


Figure 4.1: A three-way merge example for SVG documents.

ordered trees of elements with *unordered attributes* on each element. The ordered tree semantic is obviously important for text documents such as OpenOffice XML, and it also matches the standard semantics of the painter's algorithm for two-dimensional graphics documents, as in SVG. However, many XML document types have unordered semantics for at least some elements of their trees. For example, when defining linear gradients in SVG, the order of the `stop` elements is not important.

Although one could build a specialized XML merge tool for a specific XML document type, this chapter addresses general XML documents and assumes that element order is important. This influences how the algorithm detects conflicts, especially insertion and move conflicts. However, in accordance with standard XML semantics, we ignore the order of attributes and treats all permutations of attributes as equivalent. The proposed algorithm also assumes that there are unique identifiers for all nodes that exist in the base document. This includes those elements in the modified documents that also exist in the base document, even if they have been modified themselves. We use the attribute `mouid` to store the unique ID of an element.

4.2 Three-way Tree Merging

Three-way merging of simple text documents is well understood by software developers who use SCM tools such as CVS and Subversion. These tools use GNU *diff3* [31] to perform three-way differencing and merging of text files, which treats lines of text

as fundamental units of work that are arranged in a linear sequence in a file. This simple textual differencing does not provide a satisfying way to model changes to XML documents because it fails to adequately reflect the tree structure in those documents.

The problem of three-way tree merging can be defined as follows. Suppose t_0, t_1 and t_2 are ordered document trees where t_1 and t_2 are each derived from t_0 by separate sets of changes. t_0 is the *common base* or *common ancestor* of t_1 and t_2 in the revision tree. The problem of merging t_1 and t_2 is to find t_3 such that t_3 can be derived by a combination of the changes made to t_0 to produce t_1 and the changes made to t_0 to produce t_2. In general, the problem of merging trees involves matching the nodes between the base tree and changed trees, identifying the changes made and finding a merge of the two sets of changes.

Figure 4.1 is an example of a three-way merge of SVG documents. The top image (Version 0) is the original document. It has been copied and shared with two other users so that each one can add something to the original image. In Version 1, the user added the text ‘‘Joe's Auto Repair Shop”, painted the side door, removed the right mirror, and drew a crack on the windshield to represent a damaged car. The second user modified his copy to create a shadow below the car and painted the hood. The third file (Version 3) is the merge result produced by the proposed algorithm.

4.3 Versioned Tree Data Model

In this section we describe the versioned data structure and how it is used to model XML tree and its change detection mechanism. We use versioned data structures from the Fluid project [35].

4.3.1 Versioned Data Structure

The Fluid project's [35] goal is to develop tools for Java program transformation. As part of the project, a low level versioning system was developed using Fluid's internal representation (IR). Fluid's central representations are nodes, slots, attributes and versions.

- *Nodes* are loci of identity and contain no other information.

- *Versions* are points in tree-structured discrete time. They are arranged into a tree called the *version tree* where the root is the *initial* or first version and parent versions represent older states than their children.

- *Slots* are locations that can store information including references to other slots and nodes. *Versioned slots* are specialized slots that can store different information at different versions.

- *Attributes* are names that map nodes to slots. Given a node and an attribute, we can obtain the slot value assigned to that node. This models the idea that nodes have attributes.

Nodes, attributes and slots then can be thought of as a table where the rows are the nodes, the columns are the attributes and the cells are the slots that store the values. With the addition of versions, the table becomes a three dimensional table where the third dimension is version. With versioned slots, given a node, an attribute, and a version, the value at that version is retrieved. This structure models the notion that a node's attributes have different values at different versions.

The API to the Fluid IR provides a change-and-commit model for managing versions. When a new version of a document is created, it is represented by an empty table of nodes, attributes and slots that is the *current version*. This table is filled in with information and at some point the result is *committed* as an initial version, v_0. Once committed, v_0 never changes again in any way. Further editing changes to the ''current version'' are permitted, but those changes are part of a new, unnamed version. Eventually, those changes may be committed as v_1. Once more than one version has been committed into the system, it becomes possible to set the ''current version'' to be any such version and to start making a new version derived from it.

4.3.2 Change History

Changes to a node are recorded by a listener object of type *ChangeRecord*. This occurs during parsing of derived documents as described in Section 4.6.4. Using a *ChangeRecord* object, a given node n can be queried whether it has changed in v_2 relative to v_1. *ChangeRecord* objects are coarse-grained; they know that n has changed but not the nature of the change. The change could be an update to one

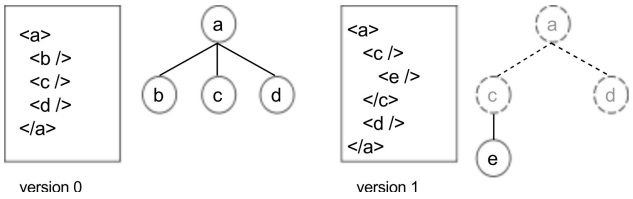


Figure 4.2: Representing XML documents as versioned trees.

or more attributes or an addition or deletion of one or more children. For example, if the value of an attribute a of node n is updated, n is marked as changed from the previous version, but *ChangeRecord* says nothing about a being changed other than that node n has changed from previous version. A node that has a child added or removed is marked as changed, but the child that was added or removed is not marked as changed. Using a *ChangeRecord* we can obtain a list of nodes that have been changed from a given version. This gives us the notion of change history and is used during the merge process to ignore nodes that have not changed.

4.3.3 XML Documents as Versioned Trees

More complex structures such as containers and trees are formed using nodes, attributes and slots. A container is a collection of slots and can act as a linked list or as a fixed size array. A node in a versioned tree has a *children* attribute and a *parent* attribute. The parent attribute maps to a slot containing the reference of the parent node. The children attribute maps to a container containing the references of the child nodes. To represent an XML element, a node is given a *tagname* attribute, and an *attributes* attribute that points to a collection of name-value pairs which hold the name and value of an XML attribute as specified in an XML element's start tag. Notice that we are now talking about two kinds of attributes (Fluid and XML) at the same time. This is confusing, but appears unavoidable. For text nodes in the XML tree, there is one additional Fluid attribute *text* that holds the text or character data. We ignore comments and processing instructions although they could easily be represented.

Mapping the Fluid attributes to versioned slots allows us to represent trees at different versions. Figure 4.2 shows the modeling of two XML documents where the second is a modified copy of the first as versioned trees. For each version, the XML source is shown on the left, while the corresponding versioned tree is shown on the

right. Nodes with solid circles are nodes that were created in that version. Their Fluid attribute values can be accessed by descendant versions in the version tree but not by ancestor versions. For example, nodes *a, b, c* and *d* in version 0 were created in version 0. Since version 1 is a child of version 0, it shares the nodes from version 0 which are shown with dashed circles. In version 1, only node *e* was created while node *b* was deleted. We refer to the representation of these changes as the *delta* between version 0 and version 1. Note that the tree at version 0 and version 1 is really the same tree in memory but depending on which version has been chosen as the current version, traversal of the tree will give the tree structure for that version.

4.4 Longest Common Subsequence

A sequence Y is a subsequence of the sequence X if all of Y's elements are in the sequence X in the same order that they appear in X. A subsequence is different from a substring in that a substring is a sequence which is continguous. For example, given the sequence $X = \langle a, b, c, d, e, f \rangle$ is a sequence, then $Y = \langle B, E, F \rangle$ is a subsequence of X with a length of three and $Z = \langle B, D, E, F \rangle$ is another subsequence of X of length four. An empty sequence is also a subsequence of X. A common subsequence of two sequences is a sequence which is a subsequence to both sequences. For example, $X = \langle A, B, C, D, E, F \rangle$ and $Y = \langle B, H, C, E, F, I \rangle$, $Z = \langle B, C, E \rangle$ is a common subsequence of X and Y. $W = \langle B, C, E, F \rangle$ is also a common subsequence of X and Y. Z is not the longest common subsequence, but W is. The *longest common subsequence* (LCS) problem is to find the maximum subsequence of two or more sequences. For the rest of the discussion, we only deal with subsequences of any two sequences. The problems of finding the LCS can be solved in polynomial time using dynamic programming.

4.5 Diff3 Merging Algorithm

The *diff3* algorithm, which is the basis in the unix *diff3* utility command, is considered the standard algorithm for differencing and merging of text files. We will refer to *diff3* as both the algorithm and the unix *diff3* command. *diff3* operates on three files with one of the files being the *common ancestor* in the revision tree of the other two files.

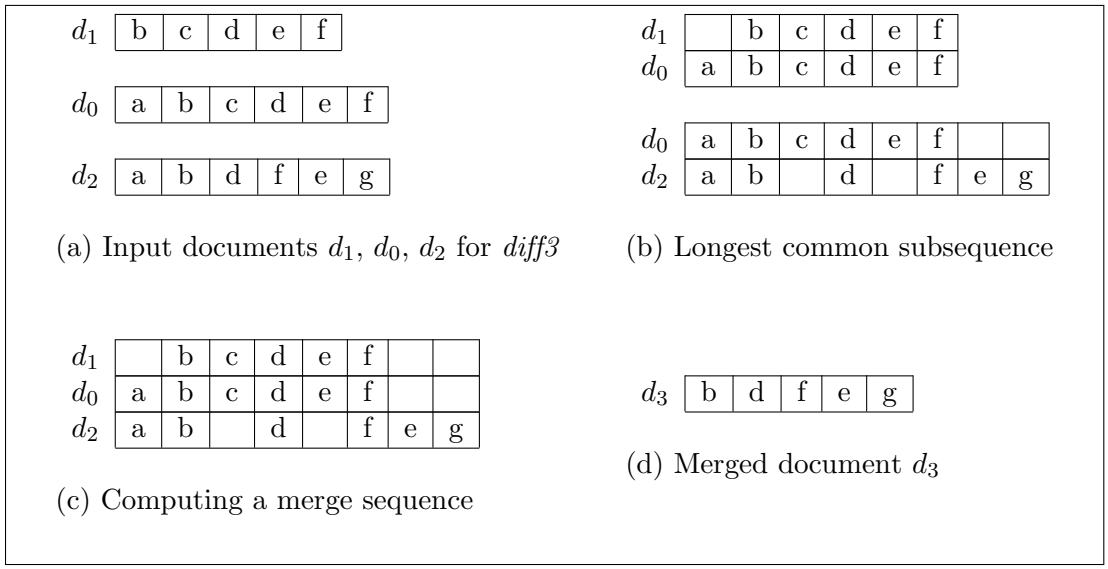


Figure 4.3: *diff3 algorithm*

Suppose two collaborators need to modify the same text file. One approach is one waits for the other to finish modifying the file before the other can start modifying the file. The second approach is for each to copy the original file and modify the file independently. When they are finished modifying their own copy and they would like to integrate their changes, they would take the original document, and the two documents with the new changes and feed them into *diff3* which in turn produces a new document with their changes incorporated and marked any conflict in the result file. *diff3* will use the information from the original document in order to merge the two derived documents. *diff3* computes the *longest common subsequences* (LCS) of the original document and the document modified by the first user, and the original document with the document created by the second user. Then *diff3* uses the LCSs to compute the merge. *diff3* supports two kind of operations: insertion and deletion. A move is seen as an insertion followed by a deletion or a deletion followed by an insertion. Here's example of *diff3* on merge: the content document d_0 has `[a b c d e f]` , d_1 has `[b c d e f]`, and d_2 has `[a b d f e g]` as shown in Figure 4.3 (a). Document d_0 is the original document or the base document as indicated. Document d_1 and d_2 are modified copies of d_0. d_1 has `a` removed while d_2 added `g`, removed `c`, and swapped `e` and `f`. Suppose d_3 is the document produced by merging d_1 and d_2 using d_0 as the base document, then the content of d_3 becomes `[b d f e g]`. Figure 4.3

describes the *diff3* process. Figure 4.3(a) shows the three document inputs, (b) *diff3* computes the longest common subsequence for d_1 and d_0 and d_0 and $d2$, (c) computes the merge sequence. The merge sequence is computed by a simple rule: if x is in d_0 and d_1 but not in d_2, x will be in the merged document; if x is in d_1, d_0, and d_2 then x is in the merge document; if x is in either d_1 or d_2 but not both, and x is not in d_0, then x is in the merge document; if x is in d_1, and d_2, but not d_0, then x can be in the merge document depending on whether it is considered a conflict or not. The algorithm of *diff3* is formally described by Khanna, et al [36].

4.6 Proposed XML Merge Algorithm

This section describes the merge algorithm. It starts out by describing rules for merging and conflict detection. Then it describes how elements in derived XML documents are matched to elements that already existed in the base document. Then once these preliminaries are complete, the algorithm is presented in detail.

Throughout this section, we will use t_0 to denote the base document tree, t_1 and t_2 as the two derived document trees, and t_3 as the merged document tree. v_k refers to the version for which t_k is the document tree. We also use t' to denote either t_1 or t_2, n to denote an XML element, and n' to denote an updated version of n. The assertion $n \in t_k$ says that element n is a part of tree t_k in version v_k.

4.6.1 Merge Rules

We model the changes between versions of an XML document as operations on the base document t_0. Operations that can occur are addition, deletion, update and move. An *addition* occurs when a new element is created and added to t' which did not exist in t_0. The *deletion* of an element removes the subtree of which that element is the root. An *update* operation occurs when an element in t' sees changes in one or more XML attributes, in its tagname, or in its sequence of child elements. An element is *moved* when its parent is changed or its position in the sequence of children of the same parent is changed. Notice that the movement of elements is marked as two operations: an update of the parent and a move of the child.

The following are the merge rules. They are written under the assumption that

there are no conflicts. The rules describe the operations that will be applied to t_0 in order to produce t_3.

- *addition*: if n is added to t', then n also appears in t_3.
- *deletion*: if n is deleted in t', then n is deleted in t_3.
- *update*: if n is updated to n' in t' then n' replaces n in t_3.
- *update*: if n is updated in both t_1 and t_2 and the changes are disjoint, then all changes are made in t_3.
- *move*: if n is moved in t', the move also occurs in t_3.

4.6.2 Conflict Detection Rules

A *conflict* occurs when changes to the same element occur in both t_1 and t_2. For example, if n has an attribute value changed in t_1 but n is removed in t_2, there would be a conflict. The following list specifies the cases that cause conflicts. In this list, we let n_1 and n_2 to refer to respective elements in t_1 and t_2 and the elements are different. We use n to refer to a single element that is relevant in both trees.

- n_1 is added or moved to a location in t_1 and n_2 is added or moved to the same location in t_2. This is a conflict because the correct order of the two elements cannot be determined.
- n is moved in both t_1 and t_2, but its new location in those trees is not the same.
- n is updated or moved in t_1 and n is deleted in t_2 or vice versa.
- attribute a of n is updated with different values in both trees.
- n is deleted in t_1 and n or one of its descendants are updated or moved in t_2.

4.6.3 False Conflict Handling

The following cases are combinations of changes that we believe should not be considered conflicts because we want to avoid overloading users with false conflicts. An alternative model would be to have conflict levels similar to the error/warning distinction in many compilers.

- attribute a of n is updated with the same content in both t_1 and t_2.
- n is deleted from both t_1 and t_2.
- n is moved to same location in both t_1 and t_2.

4.6.4 Node Matching

In our system, node matching and change detection occur at parse time for the three documents. Our approach relies on versioning services provided by the Fluid IR. Note that we give every element in the base document a `mouid` attribute containing a unique ID string.

The base document is first read in and a versioned tree t_0 is created for it and committed as the representation of version v_0. A hash table is created mapping UIDs to elements in t_0.

Next, the parser reads in the the first modified document in order to construct a delta representing t_1. Each element is examined to see if it has a `mouid` attribute. If an element does, then that element is compared to the corresponding element in t_0. If the element has changed, then the corresponding operations are added to the delta for t_1. Otherwise, no action is taken. Note that we ignore the permutation of attributes so that elements having different permutations of attributes in their start tags will not trigger a change event. If an element read in for t_1 does not have a UID, then it is a new element and an element representing it is added to the delta for t_1. When the entire document has been processed, the delta for t_1 is committed as the representation of a new version v_1.

The system then turns to parsing the second modified XML document that will be t_2. The current version is set to be v_0 and parsing proceeds exactly as with the first document, except that change representation are added to a delta for t_2, which is committed as the representation of v_2.

Our approach requires UIDs be placed on the elements of the base document before it is shared. Stamping UIDs on the base document has a linear runtime relative to the numbers of elements. Any element in the derived documents that does not have a UID must be a new node introduced by the editor. If the editor stamps new elements with UIDs then new elements in both derived documents must not have duplicate

UIDs. Since most editors do not assign UIDs to XML elements, we propose that before a document is shared, it should first be stamped with UIDs. We also assume that any editors used will preserve UIDs during editing but do not need to stamp new elements with UIDs. We have tested our approach with the Inkscape [37] and GLIPS SVG editor [38] and they both preserve our UIDs. We also propose that to better support merging, tools should stamp XML elements with UIDs when first creating a file. Our current approach is to stamp the file using our tool, and then share the file. The file can then be edited by tools that do not strip out the UIDs.

Other XML merging tools do element matching by computing hash values of each element from all three files and then matching elements that have the same hash values or whose hash values meet a certain threshold for a closeness approximation. These approaches require creating full document trees for all three documents before elements can be matched. Furthermore, our UID-based approach is able to accurately represent radical changes to elements that make matching difficult under the hash-based model. An important restriction of the UID-based approach is that it can't merge base and derived documents that lack UIDs. Thus, our approach will not work with XML documents produced by arbitrary tools. We suggest that the performance and power advantages of the UID-based approach argue for the addition of UID support to XML editing environments and are engaged in research to show that scaleable approaches exist for doing so.

4.6.5 The Algorithm

Once t_0, t_1 and t_2 have been constructed from the documents by the parser, the merge process can begin as described by algorithm 1. Before the merge, t_3 is created as a new revision or a *branch* of t_2, which places t_3 in a child version of the version of t_2. t_3 and t_2 are identical initially. To merge t_1 and t_2, we look for nodes in t_1 that are marked as changed from t_0 by querying the *ChangeRecord* for t_1. For each changed element n in t_1, we get its children in t_0, t_1 and t_2. An LCS (Longest Common Subsequence) based on node IDs is computed for the sequence of children of n in t_0 and t_1 and another LCS for the sequence of children of n in t_0 and t_2. We then use a node sequence *diff3* algorithm to compute a new sequence of children for the merged node n in t_3. (Khanna *et al.* [36] give a formal presentation of the *diff3* algorithm.) Element n's

children in t_3 are then manipulated by algorithm 2 such that n's children will have the same sequence as the merge sequence. Lastly, the attribute values for n in t_1 and t_2 are then merged, which is a much simpler process, since order of attributes is irrelevant.

```
input  : t0, t1, t2: document trees
output : t3: merged document tree

t3 ← branchOf(t2);
changeList ← getChangedNodes(t1, t0);
foreach n ∈ changeList do
    children0 ← getChildren(n, t0);
    children1 ← getChildren(n, t1);
    children2 ← getChildren(n, t2);
    lcs1 ← computeLCS(children0, children1);
    lcs2 ← computeLCS(children0, children2);
    mergeSeq ← computeMergeSeq(lcs1, lcs2);
    mergeAttributes(n, t0, t1, t3);
    mergeChildren(mergeSeq, n, t0, t1, t2, t3);
end
return t3 ;
```

Algorithm 1: three-way merge

Algorithm 2 describes the process of merging the children sequence from n in t_1 and t_2. Elements that are in the merge sequence, but not in the children sequence of n in t_3, are elements that have been added in t_1. To distinguish a move from an add, we check to see if the element that appears in the merge sequence already exists in t_0 and t_3. If it does not, then it is a new element in t_1 and we simply add it to t_3. We then copy the attribute values of n from t_1 because t_3 can't access attribute values of n because t_3's version is not a decedent of the version which t_1 is in. Elements that are not in the merge sequence, but are children of n in t_3 are elements that have been removed in the merge. So, these elements are removed from t_3. Once all the changed elements' children have been merged, t_3 is the tree that merges the changes from both t_1 and t_2.

The runtime and space complexity of the algorithm are quadratic. In the worst case, in which the tree is one level deep and there are N nodes, then the runtime is $O(N-1)^2$. Space complexity is also $O(N-1)^2$ in the worst case since we use a quadratic LCS algorithm. The versioned tree data structure reduces memory usage

```
input  : mergeSeq, n, t0, t1, t2
output : n ∈ t3 with its children sequence matching mergeSeq

removeChildren(n, t3);
foreach c ∈ changeSeq do
    if c ∉ t3 ∧ ∉ t0 then
        c ← copyNodeContent(c, t1, t3);
    else
        parent ← getParent(c, t3);
        removeChild(parent, c, t3);
    end
    addChild(n, c, t3);
end
```

Algorithm 2: Merge children

when there are elements in the modified documents that are also in the base document. When there are no nodes in common between all three documents, our approach will build three complete trees, but this seems to be a rare and degenerate case.

The tree data structures throughout the merge algorithm look like those in Figure 4.4. Throughout the entire process from parsing all the way to the end of the merge, there is just one full tree and some delta nodes that are new in t_1 and t_2. There is no need to create any new node for t_3 as all of its nodes are from t_1 and t_2. Note that node f in t_3 has a dashed circle but a solid label and a solid edge linking it to the tree. This denotes that the Fluid IR node for the element itself is shared between t_3 and t_1, but since t_3 is not a child version of t_1, it can not access the Fluid attribute values of f in t_1. Hence, the Fluid attribute values must be copied into the delta for t_3 where they appear as newly added information.

4.6.6 Implementation

The proposed three-way XML tool is implemented in Java and its conflict resolution interface is implemented using Java Swing. The tree and node conflict visualization make use of the Netbeans `Node`, `Explorer` and `Action` API taken from the Netbeans Platform [39]. The document parser was implemented using the SAX (Simple API for XML) API.

The tool is command-line based but it provides a graphical interface for resolving

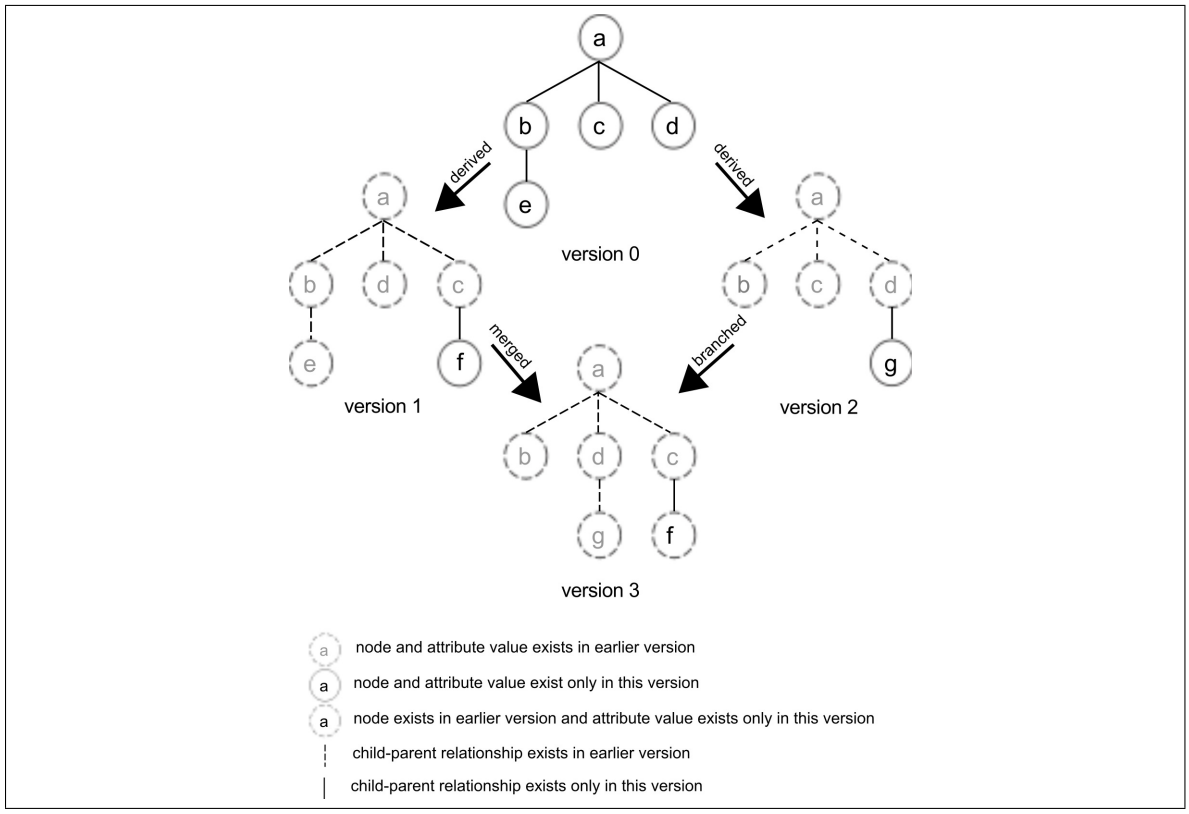


Figure 4.4: Three-way merged of versioned tree.

conflicts as shown in Figure 4.5. Existing XML three-way merge tools, such as 3dm [33], generate only log files with cryptic messages, which makes it difficult for user to locate conflicts in large and complex XML files. When the merge tool detects one or more conflicts, it displays the conflict resolution interface and expands those elements that are in conflict. As the user places the cursor over a conflicting element in the interface, a caption is displayed that specifies the type of conflict: attribute update conflict, insertion/move location conflict, or deletion conflict. For attribute conflicts, the user can select a node and edit the conflicting attributes using the attribute editor. The attribute editor displays only the conflicting attributes of the selected elements. The interface allows the user to move and delete elements and to update an element's name in order to resolve conflicts.

Figure 4.5 displays the conflicts that resulted from merging two Inkscape [37] SVG documents derived from the same document. The *svg* element conflict is due to the filename attribute because Inkscape stores the filenames in the SVG document

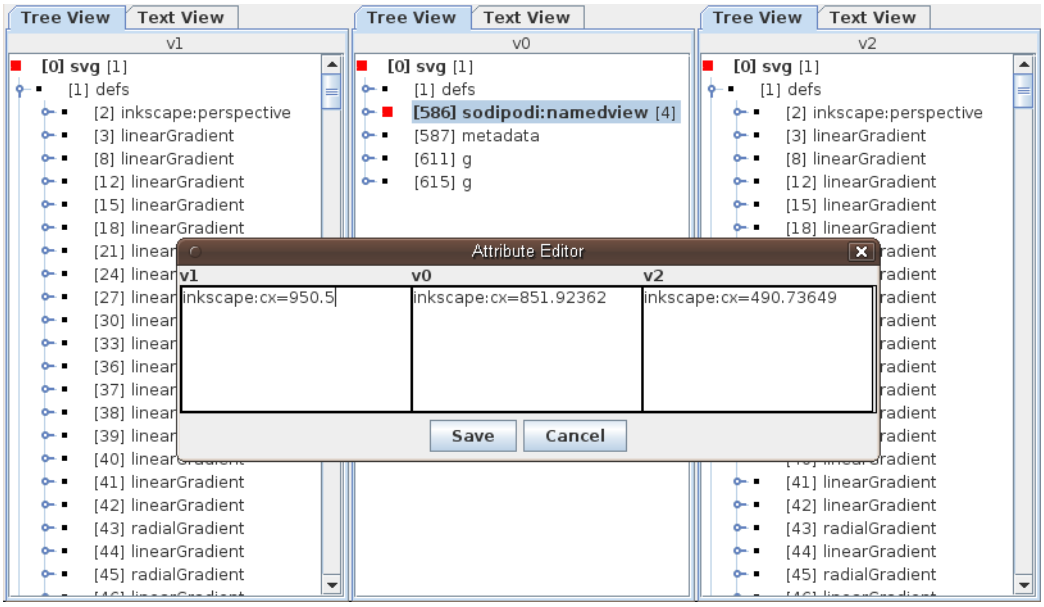


Figure 4.5: Graphical conflict resolution interface. Conflicting elements are marked with a large square in front of the element name. The number displayed after a conflicting element helps the user identify the conflicting elements in the other trees. The number in front of the element is the node ID and allows users to match elements in different trees especially when the element name is not meaningful and there are many elements with that name.

itself and these two files were saved with different names. Inkscape also saves the window's coordinates in the SVG document, which results in merge conflict in *sodipodi:namedview* element. The editor displays an attribute editor for the conflicting node *sodipodi:namedview*. The conflict here is the attribute *inkscape:cx* which represents the x coordinate of the window that was saved. Not shown in the figure is the conflict between v_1 and v_2 due to adding two gradient nodes to the same location. Hence, the interface expands both trees in v_1 and v_2 to show the conflicts but not v_0 since v_0 does not have any of these nodes.

The interface also provides a text view of each of the tree documents. The user may choose to edit the text rather than using the tree view editor. Unlike some source code merging software, the current implementation does not provide color markup to highlight conflicts.

4.7 Performance Evaluation

In this section we present an experiment to evaluate the performance of the our merge tool in terms of speed, memory usage and scalability relative to existing research and commercial three-way XML merge tools. For three-way merge tools, we are only aware of *3dm* [33] and *deltaxml* [32]. For this experiment, we used the trial version of *deltaxml*, which is limited to documents with ten thousand elements. We also include *xcc* [40] in this experiment, which is a two-way diff tool and can apply its delta to XML files that the delta was not originally created for. This feature allows *xcc* to provide a limited form of three-way merging and in many cases xcc produced incorrect merges. Still, the performance of xcc should act as a good point of comparison since it does less work than a three-way merge tool.

The questions the experiment tries to answers are: a) whether the use of the versioned tree structure reduces overall memory usage, b) whether the use of the versioned tree structure speeds up merging, c) how the algorithm scales with increases in both the number of elements and the number of changes to the document.

4.7.1 Test data and Hardware Configuration

To test the performance of our proposed algorithm, we created a set of five automatically generated XML base documents and created derived versions based on random editing changes.

Five base documents were generated at sizes of 2000, 4000, 6000, 8000, and 10000 elements. Every element had both a unique name and a unique identifier attribute, plus four attributes `a0` to `a3` whose values were random strings of digits.

Then, for each base document, we created sets of editing changes at six different sizes: 0, 20, 40, 60, 80, and 100. We divided these sets into equal-sized halves and applied them to the base documents to create two derived versions for each base document. The zero-change size was a degenerate case where the so-called derived documents were identical to the base document. Each set of modifications was divided into updates, deletions, additions, and moves where deletions, insertions, and moves were allotted 20% of the modifications and the remaining 40% of modifications were updates to attribute values. The nodes and attributes selected to be modified were chosen randomly using a uniform distribution among nodes and attributes, but we

also ensured, by a combination of automatic and manual means, that there were no conflicts between the change sets for any tool. The total numbers of files created was 55 including the base versions.

The experiment was conducted on a Lenovo Thinkpad X61 with 2.2 Ghz Core 2 Duo processor and 4 GB of RAM, running Ubuntu 10.04 Beta 1 AMD 64 with a Solid State Disk (approximately 100MB/Sec read). The code was built and run on OpenJDK 1.6 as shipped with Ubuntu 10.04. All the tools ran with default settings (`-Xmx` variable was not set).

We use `System.currentTimeMillis()` as a way to determine the amount of time a segment of code executes and `Runtime.totalMemory()` and `Runtime.getFreeMemory()` to determine the amount of memory currently used in the Java Virtual machine. We modified *3dm*'s code to print execution time and memory usage. For *deltaxml*, we used the `PipelinedSynchronizer` API of the *deltaxml* package to do the merge. Since we do not have the source code, we can only measure the entire execution from parsing to saving the merge result. This prevents us from determine how fast *deltaxml* parses XML files and writes merged result to disk compare to *3dm*, and *xcc*.

Note that we use *total time* to mean the total execution time including parsing, merging, and writing the result to disk. When we say *actual merge time* we refer only to the execution of the merge algorithm excluding the parse time and the time spent saving the merge output. For this discussion, our implementation is simply called *molhado*.

4.7.2 Results

First, we compare the execution time and memory usage of molhado, xcc, 3dm and deltaxml as the number of changes increases. The number of elements for this test is fixed at 10,000 while the changes are increased by increments of 20. Figure 4.6 compares the different tools' total execution time with increasing changes. The charts shows that molhado and xcc have the lowest total execution time. 3dm's execution time increases rapidly and linearly with an increase in changes while all other tools' execution times increase by a very small amount. As shown in Figure 4.7, molhado and xcc show an almost constant level of memory usage relative to the number of changes. 3dm and deltaxml show high variability of memory usage with results suggesting that

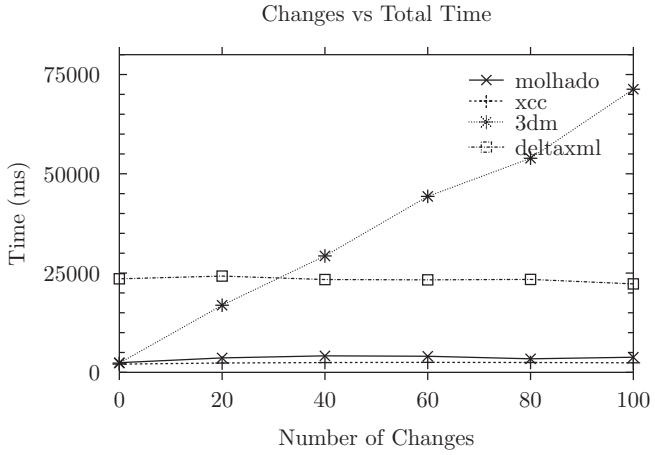


Figure 4.6: Total execution time as changes increase.

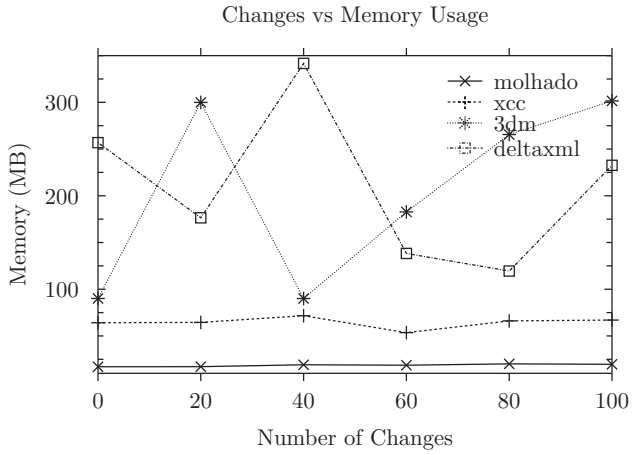


Figure 4.7: Memory usage as changes increase.

memory usage is increasing for 3dm at higher levels. Molhado uses the least memory among all the tools which appears to confirm that the versioned tree data structure reduces memory requirements.

To test whether document size affected total time and memory usage, we fixed the number of changes at 100 while the number of elements increased from 2000 to 10000 in steps of 2000. The results are plotted in Figure 4.8 and Figure 4.9. In this test, molhado and xcc show nearly identical and low total processing times. They also have low memory usage, though there is suggestion that the memory usage of xcc is growing somewhat faster than that of molhado. In constrast, both deltaxml and 3dm appear significantly affected by the number of elements. In total time, deltaxml

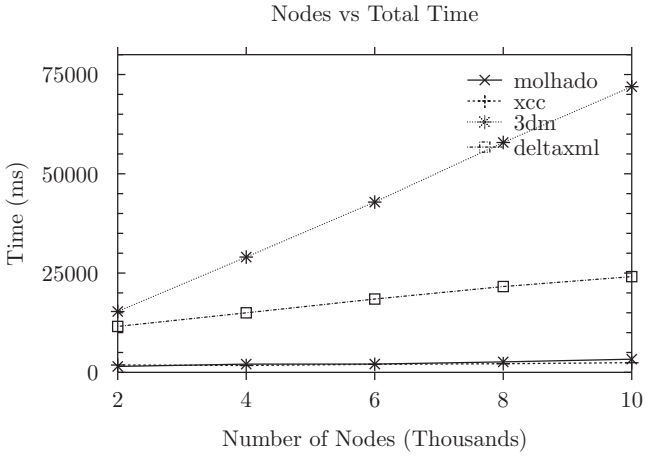


Figure 4.8: Total execution time as elements increase.

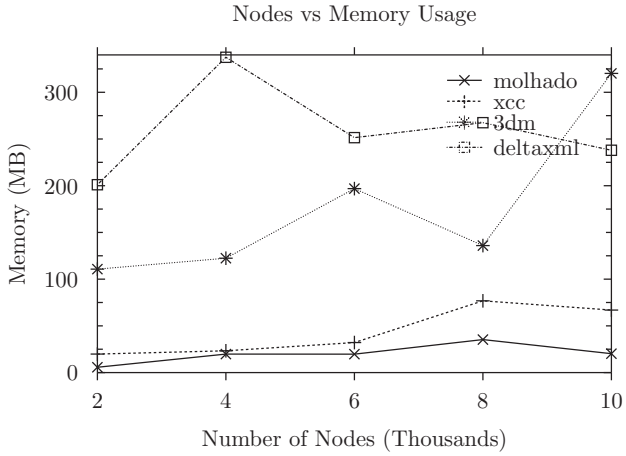


Figure 4.9: Memory usage as elements increase.

appears to have a substantial startup cost in total time, with a modest linear increase as the number of elements increases. 3dm's total time shows a dramatic linear increase with the number of elements. 3dm merges using all elements regardless of whether they have changed, thus increasing merge time linearly. Both 3dm and deltaxml have high memory usage relative to molhado and xcc.

Figure 4.10 and Figure 4.11 plot just the execution of the merge algorithm excluding the time to parse and writing of merge result to disk. Deltaxml was not included because we do not have its source code and were unable to exclude its parse and writing time. In this comparison, 3dm is affected by both increased in changes and the number of elements in linear manner. Although it is hard to see how the number of

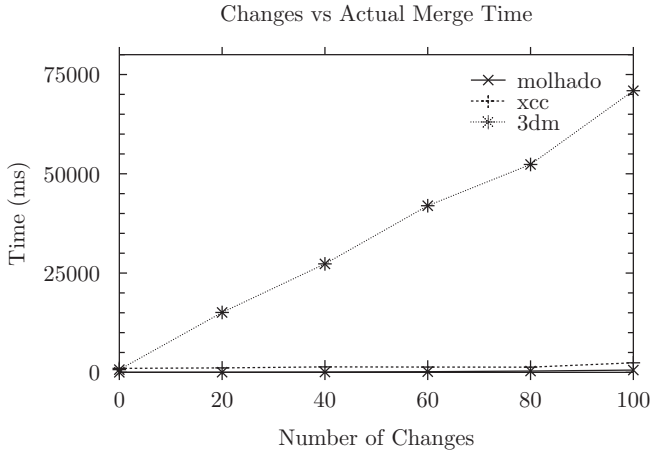


Figure 4.10: Merge time as changes increase.

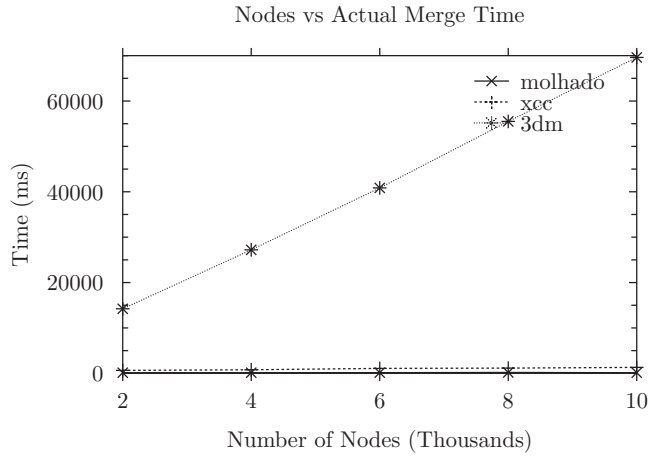


Figure 4.11: Merge time as elements increase.

changes affect molhado in Figure 4.10, the execution growth rate of molhado against changes is shown in detail in Figure 4.12 which shows quadratic scaling in the number of changes. This is probably due to the LCS algorithm we used which has a worst case complexity of $O(n^2)$. Figure 4.13 shows the merge time vs the increase in the number of elements. There is less than 100 ms increase in total time as the number of elements increases from 2,000 to 10,000. This small increase is probably for the traversal of the elements by the *ChangeRecord* data structure, as it takes more time to traverse the larger tree looking for changed elements.

Figure 4.14 shows the parse time for each tool against the number of nodes. The graph shows that molhado has the slowest parser. This is probably because building

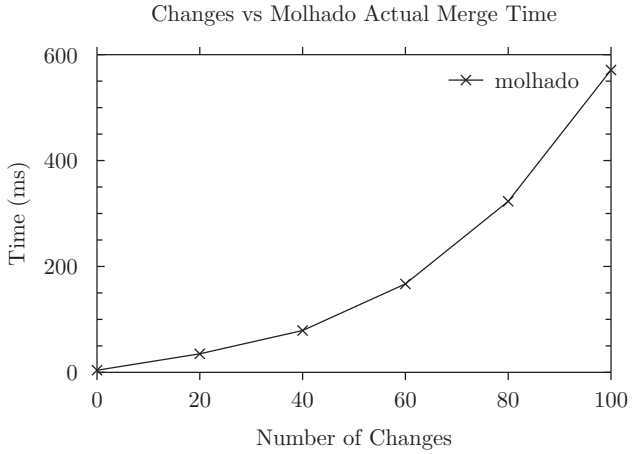


Figure 4.12: Merge time as changes increase.

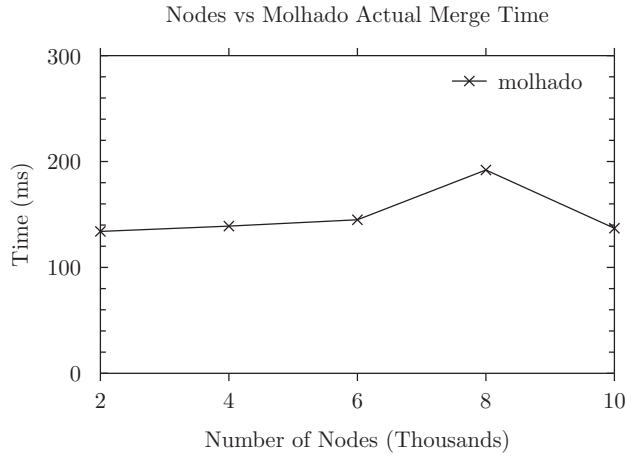


Figure 4.13: Merge time as elements increase.

the versioned tree data structure is more expensive than building a simple DOM tree. Also, during the parsing process, elements in derived trees are marked as changed or not. This requires code that checks whether elements in derived trees are present in t_0 and whether they have changed. Notice that this matching task happens during the ‘‘parsing” process in molhado, but is part of the merging step in the other tools. As Figure 4.8 shows, the large parse time of molhado is dominated by its fast merge time, making it faster than the other tools in total execution time.

On a side note, under formal XML semantics, the permutations of attributes in a start tag should not have significance. Some XML pretty print tools rearrange the attributes as documents are saved. 3dm reports rearrangement of attributes as a

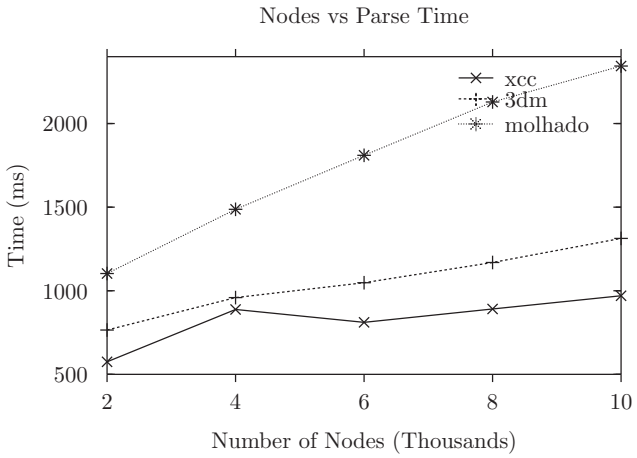


Figure 4.14: Parse time as elements increase.

conflict, stating that the element has been updated, even though the two versions are semantically equivalent. 3dm's performance suffers drastically with permutation of attributes. For example, we performed a simple permutation of the attributes of all elements in the 2000-element file and this resulted in a 16 minute (960 second) merge time for 3dm while the same files without the change in attribute order could be merged in 1.3 seconds. In contrast, molhado performed the same tasks in 1.15 and 1.05 seconds.

Overall, this experiment shows that the molhado merge algorithm is faster than that of the available tools and that the versioned tree data structure reduces memory usage. We also note that while xcc approaches molhado's performance in both time and memory, it is a more limited tool that does not perform a true three-way merge.

4.8 Related Work

Mens [41] gives a survey on the state of software merging. The algorithm for *diff2* line based content is described in detailed by Myers [42]. Other researchers [43, 44, 45, 46, 33] describe two-way structural differencing algorithms which make use of two documents without a base document. Their node matching techniques are based on node values and their hash values. Tools for three-way differencing and merging of XML documents also exist [33] and [32].

The GNU *diff* utility computes the differences between any two text files using

the longest common subsequence (LCS) algorithm [42].

Chawathe et al. [44] describe an algorithm that does not assume the use of unique IDs, but makes use of them if they are available. Cobena et al. [45] implemented a 2-way XML document differencing tool that uses an ID attribute for node matching. It also make uses of a node signature, which is a hash value for the node and its content. Wang et al. [47] proposed X-Diff which uses standard tree-to-tree correction techniques. X-Diff treats XML documents as unordered trees, which is not suitable for most document applications. Al-Ekram et al. [43] described diffX which matches nodes using node types and node name. Two nodes are equal if both their types and labels are equal. Lanham et al. [46] described an algorithm called vdiff. It makes use of the node's unique ID and hash values for matching. Lindholm et al. [48] described an XML differencing algorithm that works on a sequence of token encoding XML documents rather than document trees. It computes a match list from the input token sequences using a sequence alignment algorithm and hash values.

DeltaXML [32], a commercial tool, supports three-way merge of XML documents. Node matching is done using node ID and longest common subsequence alignment at each level of the input trees. By default, the order of children within a node is important, but it could be ignored. DeltaXML supports the addition, deletion and update operations.

Lindholm et al. [33] presented a three-way merge algorithm for XML documents that performs tree to tree mapping. An XML document is encoded as a set of content and parent-child-successor (PCS) relations. A set of changes are consistent if the changes in the set are unambiguous. The set of changes are combined into a ‘‘raw" merge and then inconsistencies are removed by iterating over the ‘‘raw" merge to create the change set for the merge. It supports the addition, deletion, update, and move operations. Nodes are required to match to at most one node. The algorithm is attractive in its simplicity, but it lacks the ability to deal with multiple moves in either modified tree. When there are multiple moves that are within the same parent, the change list will be inconsistent. The algorithm simply ignores the position. In our approach, the positions of the merged children are computed by using *diff3* which places the children in the correct order, and if it cannot determine the position of children, it is marked as a conflict.

Rönnau et al. [40] describes a differencing and merging algorithm for XML documents using context fingerprints, a sequence of hash values of all nodes within certain distance from the edit operation, to help resolve the location of operations during merging. The algorithm supports the add, delete and update operations. The hash values are computed using the node element name and its sorted attributes. This means that a node which has its element name changed will have a different hash value, hence appearing to be a different node. Using node identity, our approach knows the node is the same even if its element name and its attribute values change. Instead of performing a delete and insert operation, it does an update. In Rönnau's approach, finding the node for the merge operation is heuristic where in our approach, we know on what node the operation must occur because of node identity and *diff3*. The complexity of Ronnau's approach results from trying to support merging of documents the difference has not been computed, thus allowing it to perform a limited three-way merge.

4.9 Discussion and Future Work

We have described an approach to three-way XML document merging using a versioned tree data structure, change detection and node identity. We showed that it is fast and uses less memory than other three-way XML merge tools. Unlike other approaches, which create all elements for three or four trees our approach creates one full tree t_0, delta nodes for t_1 and t_2 and no nodes for t_3. With change detection, the algorithm only merges changed nodes, reducing the number of LCS computations. The evaluation experiment shows that the algorithm's performance is largely unaffected by the number of nodes. The algorithm's runtime is affected most by the number of changes and this appears mainly due to the LCS algorithm.

Future work on this algorithm and its implementation should include adjusting the algorithm to gain further improvement in both speed and memory usage. The current implementation uses the simple standard LCS algorithm. Replacing it with the LCS algorithm described by Myers [42], which has a linear space complexity, could further reduce memory usage. Merge performance can further be improved by reducing the numbers of nodes to be merged. Choosing the derived tree (t_1 or t_2) that has the most changes and deriving t_3 from it could well reduce run time. Since the algorithm only

visits nodes changed between t_0 and the one derived tree that is not the parent of t_3, there would be less nodes to visit. This is only an advantage when the change sets of the two trees are different by a large degree. Refining change detection so that it can distinguish the kind of change (changes to attributes vs. changes to the child list) might reduce the number of elements for which it is necessary to compute the children merge sequence. Other things to consider include the use of XML schema to improve correctness of merging, incorporating the use of hash values for elements when there exist no UIDs in all three documents, and allowing the user to specify whether the order of attributes should be preserved. Finally, the evaluation experiment could be expanded to a larger set of simulated documents so that any doubt about the relevance of the performance results could be eliminated.

Chapter 5

Version-aware XML Documents

5.1 Introduction

A conventional document, such as a user manual, a technical report or a software requirement document, will go through multiple revisions. Creating such a document may require a collaboration among multiple persons who will create and edit content in parallel. While the document is evolving, authors may want to see how and why it has changed and they may want revert back to a previous revision if the current revision is beyond repair. The ability to capture the evolution of a document is also important to organizations such as pharmaceutical and aerospace companies that must stay compliant with government regulations where traceability or auditability of documents is important. Also, the ability to retrieve arbitrary revisions frees authors to experiment with the current revision without fear of losing it.

A naive approach to version control is to use a manual process of copying and naming files to indicate revision or version. This solves the problem of version identification but requires the overhead of managing the files and manually selecting the correct files from among many. Collaboration is still possible and merging may be done by hand or by 3-way merge tools such as diff3 if the files are text-based. When multiple authors want to have access to the change history, files for every revision must be copied and shared. When there are too many files and revisions, the overhead and confusion can be overwhelming.

A more attractive approach would be to use conventional version control tools such RCS [27], CVS [6], Subversion [20], Git [29], or Mercurial [28]. These tools

often require access to a central repository or a shared file system where the version data is stored to support both versioning and collaboration. This may require setting up the infrastructure needed, and in organizations that must be in compliance with government rules, gaining approval could be cumbersome. Newer version control systems such as Git [29] and Mercurial [28] use a distributed repository and only require network access when authors want to publish their changes or retrieve changes made by others. These tools are excellent for collaboration in large groups where there's a centralized repository and access to a network. But non-technical authors who do not have the skills or the will to learn the tools are unlikely to adopt them. Some cloud storage systems offer versioning and collaboration support for documents stored but that makes the author dependent on the particular system [49, 50].

We propose an approach called *version-aware documents* that does not require a repository or network to collaborate and makes version control as seamless as possible. Currently our approach is limited to documents that can be represented as XML.

5.2 Background and Related Work

XML (Extensible Markup Language) [34] is a markup language that is used for many purposes ranging from representation of data in databases and remote procedure calls, to configuration files such as Ant build scripts, to human oriented documents found in publications, documentation, and marketing materials. An XML document can be composed from multiple *markup vocabularies* from different domains which may have the same element and attribute names. XML *namespaces* can be used to expand names in order to prevent name collisions between vocabularies with overlapping element and attribute names. The XML Security standards [51] provide support for maintaining document data confidentiality and integrity. Our approach, version-aware documents, makes use of namespaces and XML signature from the XML Security standards for version management and integrity checking. The use of namespaces allows version data to co-exist peacefully with the primary content of an XML document while XML signature is used to prevent users from altering the version data.

Conventional version control tools [6, 29, 28, 20] require the user to maintain a repository either locally or remotely and rely on the diff3 merge tool to support merging. Some researchers are interested in representing *multiple structures* (or variants) of a

document [52] in an XML model but not the versions of each variant. If we view a variant as a version, then we have a set of versions but not a tree of versions. Some researchers use XML to encode data with a time dimension [53, 54, 55, 56] known as *temporal XML* to achieve a time based retrieval functionality found in *temporal databases* [57, 58], where the emphasis is on the ability to query the data for a given time. These approaches extend XML and introduce query techniques and tools over the XML model.

There are several XML differencing techniques [59, 60, 61, 62, 63] which can be used as basis for XML version control. Several approaches to XML document versioning use structural differencing [64, 65, 66, 67, 68]. These versioning approaches store the document and the change information (the *delta*) separately on the file system or in a database. Vion-Dury proposed encoding XML deltas and version history as a standalone XML file [69].

Microsoft Office and OpenOfffice have limited versioning capabilities. Microsoft Word's change tracking system is essentially version control with only two revisions: previous and current. OpenOffice documents support linear versioning. In OpenOffice, each version is stored in full form (ZIP archive) in the current version's ZIP archive. Both Microsoft Office and OpenOffice support collaboration by user accepting or rejecting new changes but lack advance collaboration support such as merging and branching.

5.3 Implementation

Our *version-aware XML documents* are created via three simple extensions to an application's native XML data format using vocabularies and namespace from our versioning XML framework. The namespace in our framework is simply refer to as *molhado*. The extensions do not change the document's semantics and thus the document can be rendered and edited by an editor as if it were a native document. For example, an extended SVG document can be read, rendered, and edited as if it were a normal SVG document. The extensions are:

- New document elements representing reverse deltas;
- An XML signature on the version data to ensure its integrity; and

- The addition of an identifier attribute to each element, which eases identification of changes.

The resulting extended document continues to be valid XML because we use the XML namespace mechanism to declare the purpose of the extensions and avoid name clashes. XML namespaces are designed specifically to support extensions of this type and XML parsers are expected to tolerate such extensions. This means that an editor that does not know about versioning, but uses a namespace-compliant parser, can still open and edit the file in the same way as any normal file that it understands.

As mentioned before, our version-aware documents use reverse deltas (edit operations needed to compute a previous version from the current version) to encode the version history of a document. With reverse deltas, the latest version is stored in full form, while all other versions are stored as deltas. Figure 5.2 is an example of an extended Inkscape SVG document. Each version is encoded as an XML element, and each has a parents attribute which points to the previous version elements. The sub-elements of the versions represent edit operations. Possible edit operations are attribute value update, node child sequence update, node deletion, node addition, and node name update. The latest version is the complete document content ignoring the version data. A version other than the latest version is retrieved by applying a chain of deltas to the latest version. An editor, whether version-aware or not, can modify the document directly as if it were a normal document. An editor without version support can simply ignore the version data and renders the document normally, though the system works best with editors that preserve the delta elements and UID attributes that have been added to the base document representation.

To support branching and merging, each version is labelled with a 32 hex character unique ID and each version has zero or more parents. The unique IDs are generated using the Java Unique ID Generator (JUG) library [70]. The IDs are generated using the system clock and the host MAC address to prevent possible collisions when versions are created in different machines.

Branching occurs when two copied and modified documents are merged. A merge creates branches composed of new changes from both documents. Note that this is a 3-way merge because they have a common ancestor version. The branches start from the common version and join at the merged version forming a diamond. The branches contain all changes made by both users and both changes can be retrieved. Figure 5.1

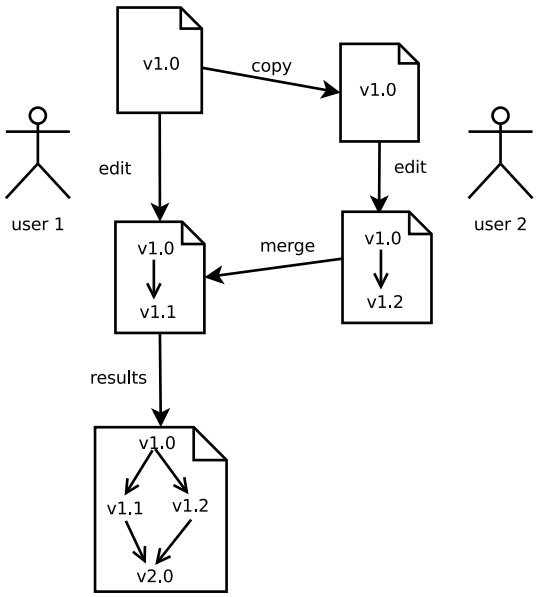


Figure 5.1: Version-aware XML document collaboration workflow example.

illustrates this process.

Version-aware documents contain the entire document history without requiring users to interact with a version repository. To collaborate, an author simply gives a copy of his working file to another author. This could be done in any manner, such as via email or using a flash drive. Once the other author has finished his work, he gives the first author his copy of the file. The first author can then perform a 3-way merge. Merge rules and conflicts detection are described in [71]. Our versioning framework can be supported by native application to make versioning more seamless or by third party tools to enable an application to support versioning. We show in the next section how an editing application can be ‘‘wrapped’’ by a version-aware application so that an end user can remain mostly unaware of how the versioning system works.

5.4 Evaluation

To demonstrate that our approach is feasible, we implemented the XML versioning framework for XML documents and the tools to support the framework. We tested them by Inkscape [37], a popular open-source SVG editor and its native SVG file format. We sought to add versioning support to Inkscape without modifying Inkscape in any way. In order to capture the delta each time Inkscape changes the document,

Figure 5.2: This version-aware Inkscape document contains three revisions. Box 1 represents the entire version-aware document with molhado namespace defined in box 2. Box 3 contains the version data (box 4) and the version data signature (box 5). Box 6 contains the Inkscape document content which represents the latest revision.

we created a Java program called *vinkscape* that is called to open an Inkscape SVG file, instead of calling Inkscape directly. When *vinkscape* is called, it starts Inkscape with the file specified by the user. Once Inkscape has modified the file and exited, *vinkscape* computes the delta between the modified content and the previous content and updates the version data within the Inkscape file. Note that the rendering and editing are done with Inkscape with the versioning data intact. The delta is then signed to ensure the integrity of the version data. Any version can be retrieved and viewed by calling *vinkscape ¡file¿ -rev ¡revision¿*. If the revision argument is not provided by the user, *vinkscape* will load the latest version. Figure 5.3 demonstrates *vinkscape*. The version-aware SVG document *inkscape.logo.svg* has three versions. Each of the Inkscape window shows each of the versions contained in *inkscape.logo.svg*.

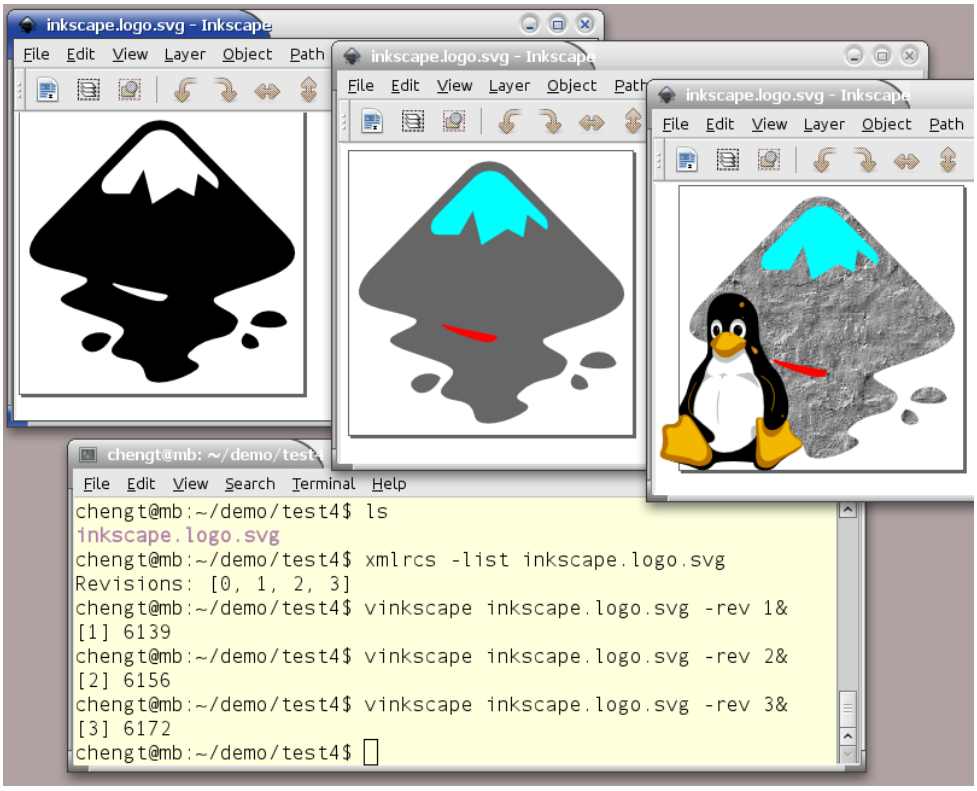


Figure 5.3: vinkscape: retrieval of revision 1, 2 and 3 from a version-aware Inkscape SVG document.

The *vinkscape* command is executed in the terminal window which in turn loads the selected version in Inkscape.

5.5 Discussion and Future Work

We have described an approach, *version-aware XML documents*, in which the version data is stored within the working document. This introduces the notion that versioning is part of the document and collaborative editing does not have to revolve around shared storage or a centralized repository. Our framework can be used to extend a document type to incorporate versioning. We have shown that this is feasible by extending Inkscape SVG files.

There are two ways to use the versioning framework. One is by modifying an application to support the versioning framework. A second is to create external tools to support an application that is not version-aware by capturing the changes as

demonstrated in the Inkscape example.

Currently, our approach do not work with Microsoft Office documents and OpenOffice documents because the applications throw away extension made to document when read. Although their document formats are open, they are less open in term being extensible by third party tools the way Inkscape documents can be extended. We suspect that both OpenOffice and Microsoft Office only use XML for serialization and use in memory data models that are not flexible to support extension to the documents while Inkscape use XML as both native document format and in memory data model. We consider Inkscape and its document format to be truely open.

For future work, we will include features allowing the user to delete versions, delete branches, and clear the version history. To optimize version retrieval performance, we plan to use full document snapshots to represent certain versions, instead of using deltas.

Chapter 6

Feature Model Editing and Debugging

6.1 Introduction

Software product line engineering (SPLE) is a methodology for developing a family of software products in a particular domain by systematic reuse of shared code in order to improve product quality, and reduce development time and cost [72, 73]. A *feature* is a characteristic, quality or functionality of a product. Thus, it is more natural to describe a product in terms of its features than in terms of lower-level implementation and design. A *feature model* is used to describe a product family in terms of common and variant features as well as any *relationships* or *constraints* among those features. A feature model should represent all possible products within the product family.

Feature models are used in all phases of SPLE: to define product line scope, to create requirements and documentation, to drive or guide the process of configuring a product, and in some cases, to generate product code. Because feature models are described using a formal language, automated analysis can be performed to determine the number of possible products or to detect model inconsistency, dead features, or false optional features [74].

A feature model is *inconsistent* when multiple constraints in the model are violated simultaneously in such a way that no product can be represented. When a model becomes inconsistent, the user must debug the model to find and correct the errors. Current tools for feature models offer little or no support for debugging. They provide

little or no explanation of the causes of inconsistency or of any changes that could be made to resolve the inconsistency [74]. These tools can tell the user that the model is inconsistent, but the problem and solution are left to the user to find, which can be tedious in a large feature model.

It would be easy to blame a problem on the recently added constraint that triggered the inconsistency, but it may be that the newly added constraint is correct and has exposed previously unnoticed bugs. Determining this would require the user to trace all the constraints involved in the problem back to the root feature. In a large feature model consisting of hundreds or thousands of features, this would be a daunting task. It would be better to point out to the user the constraints that contribute to the inconsistency [74] in a meaningful way. Showing exactly those constraints that are involved in the conflict would allow the user to more easily find the bug and correct it.

Tool support is critical to the adoption of SPLE. Hence, we propose a visual editor to better support feature model editing and debugging. The tool detects all errors that cause a model to be inconsistent and provides hints to the user as to what the problems are, where they are, and what the possible fixes are. It does all of this visually within the feature model editor in a non-intrusive manner similar to how modern IDEs hint errors and solutions without distracting the user. Our approach impacts anyone using feature models to develop product line software.

6.2 Background

6.2.1 Feature Model

A *feature* is a product's characteristic, quality or functionality. There are two types of features: *solitary feature* and *grouped feature*. Solitary feature is a concrete feature while a group feature represents a concept. Group features consists of *or relation* and *alternative group feature*. Or group feature represents a feature that is a combination of one or more features. The alternative grouped feature represents a feature that can be of several variants. Features are organized in a feature model as a tree-like structure. A feature has a parent feature except for the root feature and zero or more child features. The relationships between parents and children, called *structural*

constraints, define commonality and variability of the parent feature and ultimately the variability of the product line. The structural constraints are:

- *Mandatory*: If the parent is in a product, then the child is also in the product and vice versa.
- *Optional*: The child may or may not be in a product regardless of whether the parent is or is not in the product.
- *Alternative*: If the parent is in a product, then exactly one of its children is also in the product.
- *Or*: If the parent is in a product, then at least one of its children is also in the product.

When the parent-child constraints are not sufficient, *integrity constraints* are used to define constraints between features that are not parent and child. They are:

- *Implies*: if a implies b, then whenever a is in a product, b must also be in the product.
- *Excludes*: if a and b exclude one another, then they can never exist together in any product.

Figure 6.1 is a simple feature model of a family of cars rendered in our editor. A Mandatory constraint is denoted by an edge with a filled small circle. For example, the edge between car and body. An Optional constraint is rendered as an edge with an empty circle. Alternative and Or constraints are rendered as edges with a hollow half circle and edges with a filled half circle, respectively. Implies constraints are rendered as unidirectional dashed edges while Excludes constraints are represented as bidirectional dashed edges, as shown in Figure 6.2. According to the model in Figure 6.1, a car must have a body, an automatic or a manual transmission, an engine which could be electric or gasoline or both and may or may not have cruise control.

Figure 6.2 illustrates how different problems with feature models are shown in the editor, which currently uses color to highlight different problems. Figure 6.2(a) depicts an ‘‘inconsistent model’’ because P mandates both A and B but A and B exclude one another; thus, no product can be derived. Figure 6.2(b) depicts a ‘‘false optional

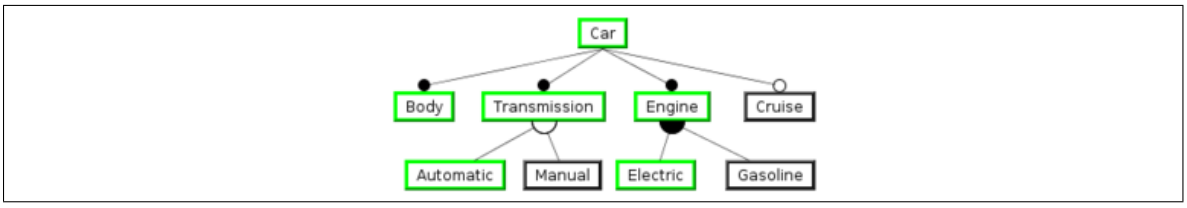


Figure 6.1: A feature model example of a family of cars. Feature highlighted as green indicate one possible car.

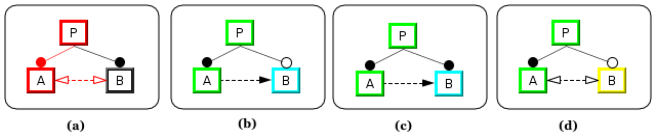


Figure 6.2: Problems and warnings in feature model: (a) an inconsistent model, (b) B is a false optional feature, (c) the implication of B is redundant, (d) B is a dead feature.

feature'' warning, which is given when an implies constraint effectively mandates a feature that the model labels optional. An ''implies redundancy'' warning occurs when an implies constraint is not needed due to other constraints, as shown in Figure 6.2(c). Finally, Figure 6.2(d) shows a ''dead feature'', where an optional feature is excluded by another constraint. Although, the latter three warnings do not create an inconsistent model, they may imply bad design and thus the user should be warned.

6.2.2 Current Support for Editing and Debugging

Currently, there are less than a dozen feature model editors and they each provide a different graphical interface. Most editors provide an interface that allows users to create feature diagrams visually, where features can be organized into a tree-structured hierarchy. Others provide flat tree editing, which is less useful for visualizing the feature model but is still superior to textual editing. Most of these tools are able to tell if a model is inconsistent and some are able to highlight what integrity constraints are violated [75, 76]. Most editors are able to detect dead features and false optional features, which are usually easy to spot and possibly correct if an editor simply highlights the problem features [75]. A more difficult debugging task is correcting an inconsistent model, for which current tools provide little or no support.

Current tools make use of propositional logic, description logic, and ad hoc methods [77] including XML Schema as a way to validate feature models [78]. To determine if a feature model is consistent or not, it is converted to a propositional formula [75, 79, 80] or description logic [81, 82] and fed to a SAT solver or a description logic reasoner. If it is not satisfiable, then the model is inconsistent.

In general, the error reporting interfaces of current tools are weak. For example, when a new constraint is added to FeatureIDE [75] that make the feature model inconsistent, the editor will highlight the text of the newly added rule. However, the conflicting constraints are not highlighted and any highlighting of previously-added inconsistent constraints will be lost. Description logic systems can generate compiler-like textual error messages, but their error messages are in terms of description logic, rather than in terms of the feature model diagram that the user has been working with [82].

6.3 Approach

Our approach uses propositional logic but exploits the idea of minimum unsatisfiable cores [83] and the algorithm QuickExplain [84] to identify the conflicting relations and features, and then proposes possible solutions.

We adopt a similar approach to that of the Eclipse plugin dependency management system [85] for detecting errors in plugin dependencies and providing meaningful explanations to the user. This system has been shown to be scalable even when faced with 10000 installable units [85]. The dependency management technology in Eclipse is also used in Maven3 and the repository manager Nexus [85]. In addition, a feature model is a relatively easy problem for a SAT solver [79].

6.3.1 Feature Model to CNF Formula

To determine if a feature model is consistent or not, we convert a feature model to a CNF boolean formula, for which an off the shelf SAT solver can be used to determine the formula's satisfiability. To generate the formula, a depth first traversal is performed. For every feature and its children, CNF clauses are generated to encode the constraints of the feature, its children, and its integrity constraints. All the clauses

are joined together by $\wedge$ boolean operators. The root feature is encoded as a single literal clause. These are the mappings of features and their constraints to CNF clauses:

- r is root: (r)
- c **mandatory** child of p : $(\neg p \vee c) \wedge (\neg c \vee p)$
- c **optional** child of p: $(\neg c \vee P)$
- $c_1, c_2, \cdots, c_n$ **or** children of p: $(\neg p \vee c_1 \vee c_2 \vee \cdots \vee c_n) \wedge$
  $(\neg c_1 \vee p) \wedge (\neg c_2 \vee p) \wedge \cdots \wedge (\neg c_n \vee p)$
- $c_1, c_2, \cdots, c_n$ **alternative** children of p:
  $(c_1 \vee c_2 \vee \cdots c_n \vee \neg p) \wedge (\neg c_1 \vee \neg c_2) \wedge \cdots \wedge$
  $(\neg c_1 \vee \neg c_n) \wedge (\neg c_1 \vee p) \wedge (\neg c_2 \vee \neg c_3) \wedge \cdots \wedge$
  $(\neg c_2 \vee \neg c_n) \wedge (\neg c_2 \vee p) \wedge (\neg c_{n-1} \vee \neg c_n) \wedge$
  $(\neg c_{n-1} \vee p) \wedge (\neg c_n \vee c)$
- a **implies** b: $(\neg a \vee b)$
- a **excludes** b: $(\neg a \vee \neg b)$

If a SAT solver evaluates the CNF formula to be satisfied, then the model is consistent, and inconsistent otherwise. Editors that use propositional formulas for inconsistency checking stop here with their debugging support and leave the user to identify errors and find solutions. Although, our approach uses propositional formula, we are interested in finding the clauses that are responsible for the inconsistency and this is where we are able to provide visual debugging.

6.3.2 Supporting Debugging

Error explanation is key in helping user to understand and fix errors. To support error explanation, we need to find the minimum number of violated constraints [74] and the least number of modifications needed to correct the violation [85]. This is a problem of finding the minimum unsatisfiable cores (MUS) [83], the smallest set of clauses that makes the formula unsatisfiable, such that a removal of any clause from the set causes the formula to be satisfiable.

The MUS gives us one error that contributes to the inconsistency of a feature model. But it is not useful to indicate only one error when there are many. Rather, the user should be able to review all errors in order to plan which ones to fix and in what order. To detect all the errors in the model, we need to remove one of the clauses that represents an integrity constraint that caused the conflict without changing the feature model tree and then find the next MUS to determine the next conflict. We repeat this until the formula is satisfiable, by which point all errors have been detected. This entire process is described in Algorithm 3.

```
input  : model: feature model
output : conflicts: conflicts

formula ← toCNF(model);
while ¬ isSatisfiable(formula) do
    mus ← getMUS(formula);
    foreach clause ∈ mus do
        conflicts ← conflicts + clause;
        if isExcludesRelation(clause) then
            formula ← formula − clause;
        end
    end
end
return conflicts ;
```

Algorithm 3: Extracting conflicts in a model

Once the editor has collected all the possible errors, the conflict visualization communicates as much information as possible, telling the user which constraints and features are involved in the conflict by tracing all conflicting constraints visually to the root feature. Any constraint in this trace is a solution; the conflict can be removed by simply removing or changing conflicting constraints. If a user adds a constraint and causes the model to be inconsistent but the constraint is correct, then there must have been a bug in the model prior to adding the constraint. An example of a bug is an optional constraint created as a mandatory constraint. The constraints trace lets the user visually inspect the constraints involved in the conflict to determine which is the bug.

Figure 6.3 shows two errors and three warnings in the model within our editor.

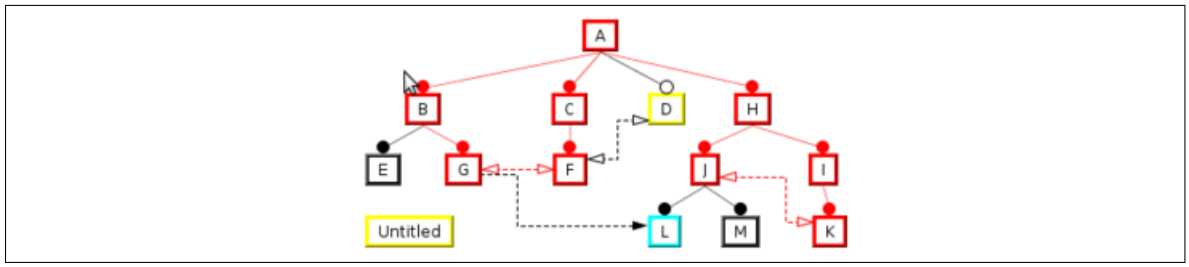


Figure 6.3: Editor: highlighting errors and warning in the model.

Errors are highlighted in red. One error is *A-B-G-F-C-A* and another is *A-H-J-K-I-H* and both are caused by Excludes constraints but any highlighted constraint is a possible bug and possible solution. The warnings including *Untitled*, *D*, and *L*. *Untitled* is a dead feature because it has no parent and would never be in any product. *D* is excluded by *F* and so *D* could never be any product. The Implies constraint between *G* and *L* is not needed because *L* is included in all products.

In the first error, changing any of the Mandatory constraints highlighted to an Optional constraint would remove the first error. Alternatively, the Excludes constraint between *G* and *F* could be removed. Removing the second error follows the same idea. Only someone with domain knowledge can decide what action to take, just as fixing a program bug requires semantic knowledge of the problem.

6.4 Implementation

The visualization of the feature model and constraints is implemented using NetBeans Visual Library [86]. The Visual Library provides the basic widgets and edges to render an underlying model. In our case, we customized the node widgets, anchors, anchor shapes, and edge connections. The underlying models of our data structure are both a graph and a tree. A graph models the entire feature model including constraints, while the tree data structure models the parent-child hierarchy of features. We take this approach because in the future, we want to provide a tree rendering of the model as an alternative editing mode and in configuration of products. The graph model allows more flexibility in having every edge labeled. The SAT solver we use is Sat4j [87], an open source library of SAT solvers which aims to allow Java programmers to access cross-platform SAT-based solvers. It is used by many projects such as FeatureIDE [75]

and in Eclipse's plugin management system [85].

6.5 Discussion and future work

We have presented a visual feature model editor that hints errors in non-intrusive way and provides a natural debugging and editing work flow. It uses the concept of minimal unsatisfiable cores to provide the constraints and features that are involved in a model's inconsistency to provide the minimal change needed to fix errors visually. Other tools only go so far as to report that the feature model is inconsistent. They do not provide help on how to fix the model and do not report all the errors a model may have, nor do they help the user spot bugs that are far from the constraints which have been causing the issue of inconsistency. Better tool support for feature model editing and debugging will help domain engineers and application engineers create and use feature model and thus encourage the adoption of feature model and SPLE as a whole.

We plan to evaluate our prototype in terms of scalability and usability on realistic feature models, to extend the tool to support cardinality-based feature models editing and debugging and product configuration, to support evolution of feature models, and to support collaborative editing. Visual editing of complex boolean constraints other than implies and excludes constraints is in progress.

Chapter 7

Proof of Concept Evaluation

Molhado SPL is a research prototype designed to solve the evolution problem in software product line. Its versioning model is not tied to a particular type of product line, which means that, in addition to software product lines, it can support document product lines as well. This chapter describes the proof of concept evaluation of Molhado SPL in supporting software and document product line evolution. The main goal is to show that the system supports the eight cases of change propagation described in Chapter 3. First, we evaluate the system with a document product line created using DITA [88] and then we evaluated it with a software product line called the Graph Product Line [89]. This evaluation does not address issues such as scalability and usability, which are left for future work. If Molhado SPL can support the following criteria, we claim that it supports software product line evolution.

- Importing a product line into Molhado SPL,
- Supporting product derivation,
- Supporting independent evolution of products, core assets, and shared assets, and
- Supporting the eight cases of change propagation described in Chapter 3.

One of the difficulties in evaluating Molhado SPL is the lack of examples of document or software product lines we can use. We had difficulty in finding any open source software or document projects that are built using the product line paradigm. There are projects written in C and C++ that can be compiled for different operating

systems or hardware using conditional compilation but they are implemented as single product projects. For example, the Linux kernel is implemented as one big project but it contains conditional compilations in the code to allow it be built for for different hardware. Molhado SPL assumes a different product line structure where the core assets and the products are separate projects. We are able to find two small examples, which we use in this evaluation. The first one is a document product line created using DITA [88]. It comes with the DITA Open Toolkit software [88]. The second one is a software product line called the Graph Product Line [89]. It has been used by other researchers to evaluate their techniques of software product line engineering.

7.1 DITA

The Darwin Information Typing Architecture (DITA) [88] is an XML-based, topic oriented architecture for authoring and publishing in a variety of forms. DITA is intended to help technical writers create system documentation for different devices (paper, PC, smartphone) and different purposes (traning, reference, marketing). DITA allows users to create reusable topics which can then be used to create multiple deliverables (such as PDF, online help, Web pages, etc) for different purposes and products as shown in Figure 7.2. DITA content is created as small topic items. DITA borrows many elements from HTML such title element, body, pagrapgraph, table, and list elements. Topics are assembled to create different deliverables using specification files called DITA maps, in which a user specifies the references of the topic files as shown in Figure 7.1. A DITA map can also specify how topics link together. Each DITA map is intented to define a different deliverable and maps have different topics but they typically share many common topics. By default, DITA has three basic topic types which are a specialization of the generic topic type:

- A *task* topic is used to describe the steps to perform a task and the expected results.
- A *concept* topic is used to describe the necessary concepts and definitions to understand a task.
- A *reference* topic contains factual information that is useful for performing a task.

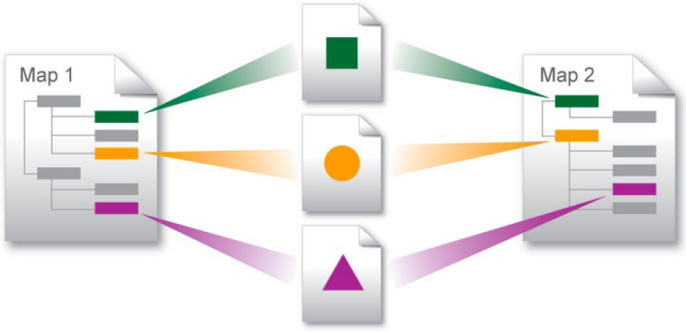


Figure 7.1: Topics are assembled in DITA maps to create different deliverables. There are two DITA maps sharing topics [3].

Contents of one topic can be reused in anothother topic through the use of conref attribute. A DITA topic can include non-DITA documents such as images, videos, words, PDF, and others through the use of references. In addition, DITA also provides conditional processing that supports filtering and name substitution. DITA can be extended by the process of specialization to create new topic types and new attributes specific for an organization, or industry. For example, there is a specilization of DITA for the seminconductor industry.

One can view a DITA project as a product line (Figure 7.2). The topic files are the reusable core assets. They are reused to generate different products targeting different audiences and different hardware readers (ebook, PDA, mobile phone, computers, and prints). Thus, a DITA project can serve as a good testbed for Molhado SPL to test software product line configuration management concepts, especially the sharing of core assets among products and change propagation between the core assets and products. DITA was attractive to us because the standard is well-described and because the XML representation is easy to work with.

The DITA sample used in this evaluation came with the DITA Open Toolkit, which is a open-source tool for transforming DITA documents and maps into multiple document output formats. It uses Ant scripts to generate the documents. The sample used in this evaluation contains 15 concept topics, and 7 task topics related to tasks that often are done in a garage such as changing the oil of a car. It has two DITA maps for generating documents in different layout and two Ant scripts for generating PDF outputs. Figure 7.3 shows a PDF generated for the sample.

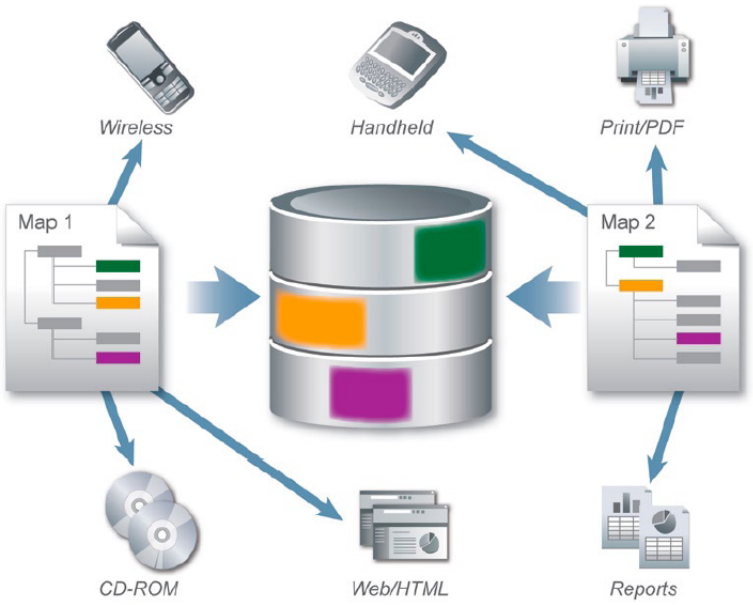


Figure 7.2: DITA topics and maps are used to create multiple deliverables from common topics [3].

To test Molhado SPL, we added support for DITA projects and their automated product derivation. The core assets of a DITA product line consist of files representing concept, task, and reference topics, Ant build script files, and DITA map files. First we import the sample DITA project that came with the DITA Open Toolkit into Molhado SPL. This creates the core assets project assets but no products. To derive a product, a user right-clicks on a DITA map and selects the popup command for product derivation. Molhado SPL then selects the core assets referenced by the DITA map for the product. This feature is implemented into Molhado SPL to recognize DITA map files, and to process the links in maps, and links in files referenced by the map. This allows Molhado SPL to automatically create a product project, select the files needed by the DITA map and share them with the product project. The user can optionally choose an Ant build script from the core assets or write their own. Once the user has a versioned product project, the user can add product specific content, modified shared content, and create new Ant scripts. To evaluate product derivation, we created two products: sequence and hierarchy as shown in Figure 7.3.

To test the different change propagation cases, we made changes to the necessary topic files and performed change propagation. Once a change propagation is performed, we checked to see if the result matched our expectation. We did this for each of the

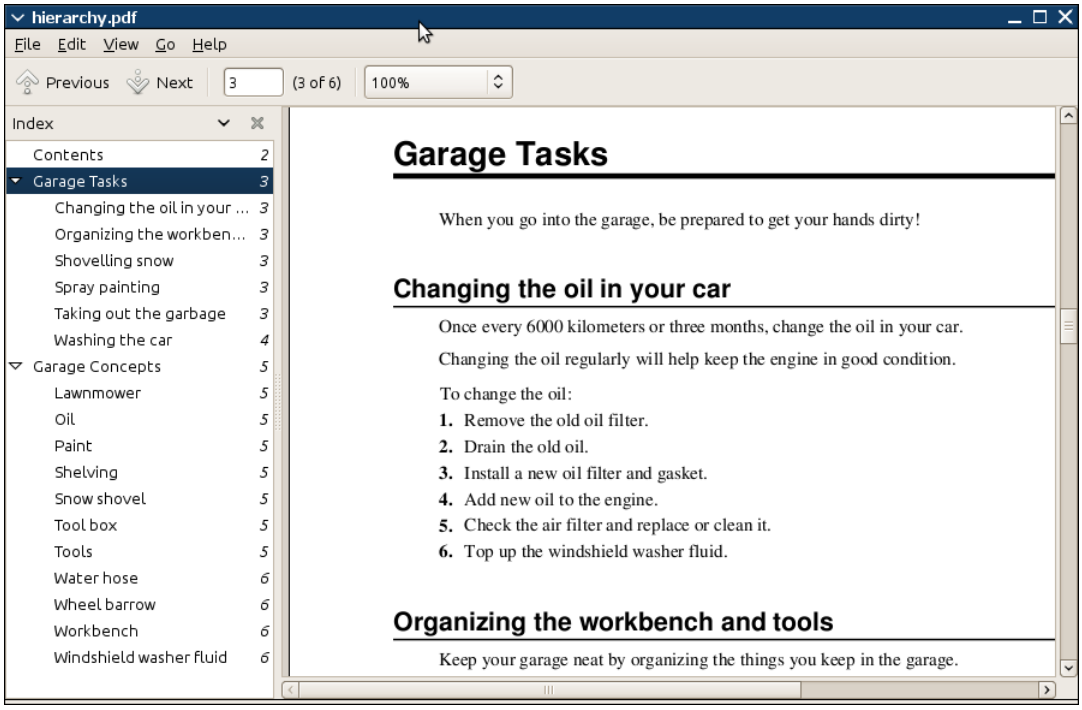


Figure 7.3: An output generated from the DITA sample using DITA Open Toolkit.

eight cases of change propagation:

In case 1, the changes in the core assets project is propagated to shared components in the product. To test this, we made changes to a core asset, ‘‘changingtheoil.xml” by adding a line of text, and committed the changes. Then we selected a shared component in the product, ‘‘hierarchy”, that refers to ‘‘changingtheoil.xml” in the core asset and performed a forward change propagation. Once the changes resulted from the change propagation were committed, we made sure that the new text was shown in the shared file ‘‘changingtheoil.xml” in the product ‘‘hierarchy”. Although case 1 looks like a merge, it is simply an update in the shared component to point to the core asset version that has the new changes.

In case 2, the shared asset in a product is modified and the asset referred to by the shared asset is also modified in the core assets project and the changes from the core asset is merged with shared asset resulting in a three way merge. To test this case, we modified the asset in the core assets project and the shared asset in a product independently. We did this by updating a text line in ‘‘changingtheoil.xml” in the core assets project and adding a text line in the shared asset of ‘‘changingtheoil.xml.” In

our evaluation, we did not create changes that would cause conflicts. We committed those changes. We then selected the shared core asset that has been changed and performed a forward change propagation and then committed the changes and verified that the changes the shared core asset have both changes. In this case, Molhado SPL performed a 3-way merge and the shared component then points to the version resulted from the merge.

In case 3, the situation is the same as in case 2, but when we performed the forward change propagation, we ignore the changes made in the shared asset. The mechanism used in change propagation in this case is identical to case 1 except that the shared asset has been changed but the changes are irrelevant. The forward change propagation updated the shared asset to point to the latest version of the referred asset in the core assets project. We made changes to a shared asset ‘‘shovelingthesnow.xml” and the asset referred to by this shared asset. We right clicked on the shared asset and chose ‘‘pull change replace.” The result only showed the changes from the core assets project.

In case 4, a component from the core assets project is being shared with a product. To test if this feature works, we selected an asset in the core asset and shared it with a product by selecting and dragging the component to the product. This case is also performed when a product is derived in which multiple components are being shared. When a product is derived, core assets are shared in a product. Without the support for this case, product derivation would not be supported.

Case 5-8 are identical to case 1 to 4 but in the opposite direction. Thus, we exclude their discussion. Case 8 is different from case 4 semantically. Case 8 makes a product specific asset into a core asset. In this case, we create ‘‘document.xml” and right clicked and selected popup command option, ‘‘Make core asset.” The file was moved into the core asset project and ‘‘document.xml” in the product became a shared asset with the shared icon and the shared annotation.

Note that all of these propagation mechanisms were tested during implementation of Molhado SPL to make sure they do what they are supposed to do. But getting them to perform correctly using an example further proves that Molhado SPL does support them. The screen shot of a working session of Molhado SPL with a DITA project is shown in Figure 7.4.

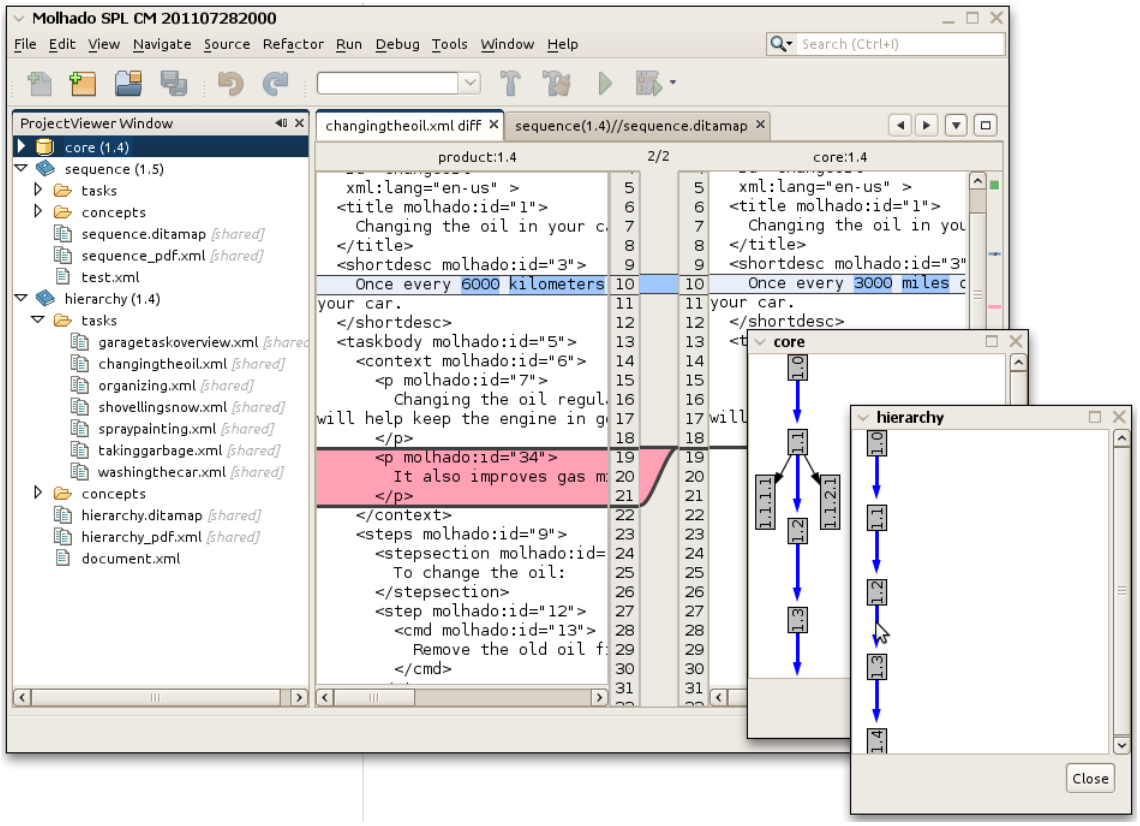


Figure 7.4: Screen shot of a DITA document product line in Molhado SPL.

7.2 The Graph Product Line

The Graph Product Line (GPL) [90] is a family of graph applications created by Don Batory at the University of Texas. Batory considers it a standard problem for evaluating product line methodologies. It is small but possesses the formal qualities of a software product line and thus makes a good test bed for Molhado SPL. Graph application can be created using GPL with different features and making use of different search algorithms. A graph is either directed or undirected and its edges can be weighted or unweighted. A graph application can have at most one search algorithm which is either breadth-first search or depth-first search. In addition, it can also have one or more algorithms from the following selections:

- Vertex numbering (Number): assigns a unique number to each vertex in a graph, based on a graph search algorithm.
- Connected components (Connected): finds the connected components in an

undirected graph.

- Strongly connected components (StronglyConnected): finds strongly connected components in a directed graph.
- Cycle checking (Cycle): determines whether the graph has cycles.
- Minimum spanning tree (MST): finds a minimum spanning tree (MST).
- Single-source shortest path (Shortest): finds the shortest paths from one vertex to all other vertices.

The GPL can be described using a feature model as shown in Figure 7.5. Because of lack of space in the figure, we excluded strongly connected components from the feature model. The feature model describes the possible products that can be created and the constraints that govern the possible products. The features are arranged in a tree where the root represents the product and the children represent product features. In the feature model, a box represents a concrete feature such as ‘‘Directed” or an abstract feature such as ‘‘Graph Type”. An abstract feature represents a selection of possible features. Feature with a filled circle on top is required in every product if all of its ancestors features are also required. Features with an empty circle are optional. For example, in every graph product, it must have a graph type, which is either directed or undirected. It may or may include have a search algorithm. If it includes a search algorithm, it can only include one: either breadth first search or depth first search. The dashed edges represent constraints. In this feature model, we only have requires constraints but it is possible to have excludes constraints where two features cannot exist in the same product. An example of a requires constraint is the constraint between shortest path feature and directed feature. The selection of shortest path requires the graph type to be directed so that selecting undirected would not be legal. Figure 7.6 shows one possible product, in which the selected features are colored green or (light grey in black and white print out).

The GPL example used for evaluating Molhado SPL was implemented by Michael Haufe, an undergraduate CS computer science student at UW-Milwaukee. He implemented GPL using just plain Java features as inheritance, Java containers and generics to implements the GPL. Others implemented it using AspectJ, other modified Java, and code mixing using XML. The goal of Molhado SPL is to support SPL without

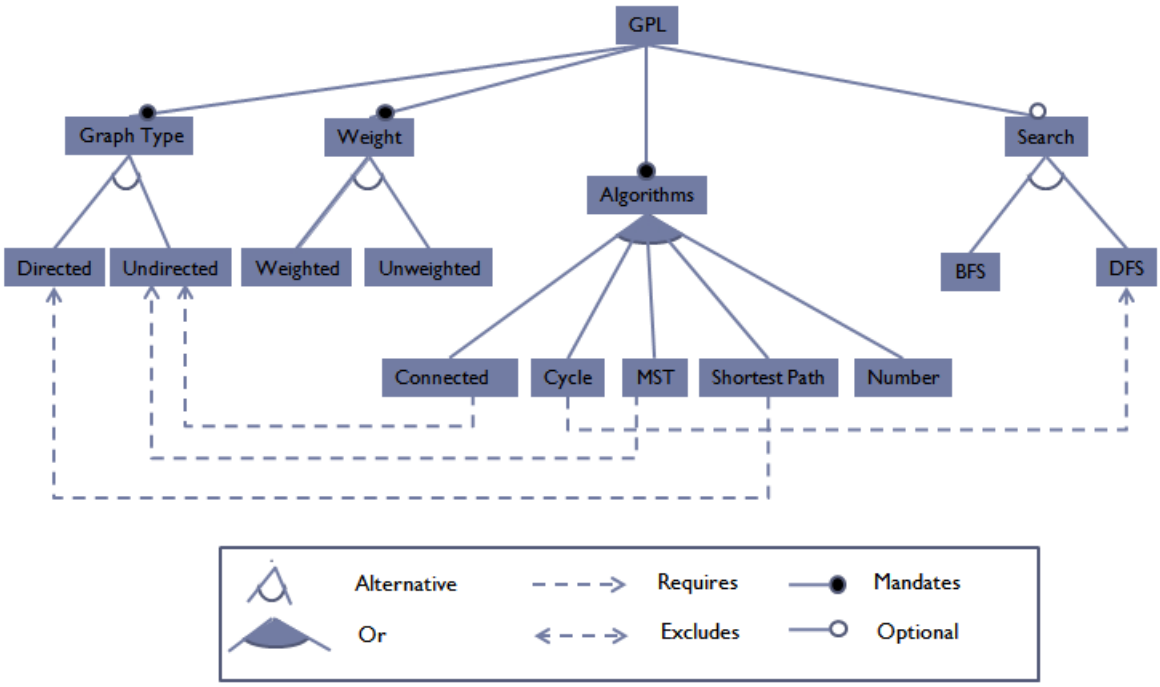


Figure 7.5: The feature model of the Graph Product Line.

introducing changes to the programming language. We did not fully implement the GPL. Instead, we only implemented enough classes to test different type of graphs and different search algorithms.

To evaluate Molhado SPL, we imported the source code implementing the GPL into Molhado SPL to manage its evolution. We then created a feature model in Molhado SPL that describes the possible products. In the feature model, we specified which files are responsible for the feature. A product is derived when a user selects the necessary features. Molhado SPL automatically shares the files that are needed for that product. Once a product is derived, it can undergo independent changes such as adding new classes or changing shared files. The evaluation for the eight cases is the same as in the document product line so the discussion is not included. Figure 7.7 shows the GPL with two products and the feature model.

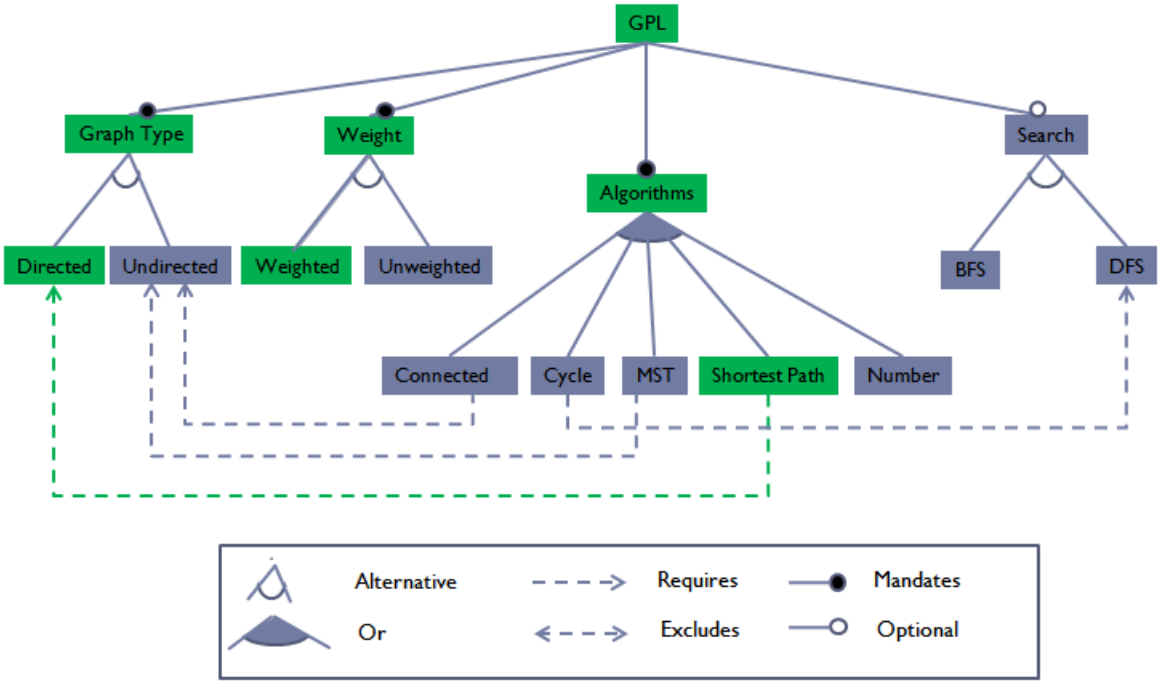


Figure 7.6: One possible product instance of the Graph Product Line.

7.3 Summary and Discussion

In this chapter, we described a proof of concept evaluation of Molhado SPL using a document product line (DITA) and a software product line (GPL). Although Molhado SPL supports both types of product lines, product derivation had to be implemented separately for each type of product line. For example, to support automatic product derivation for DITA document product line, Molhado SPL has to know about DITA maps and on how to extract references to topics in a map so that the topics can be shared with a product derived using that map. To support automatic product derivation for the software product line, we added support for feature models to Molhado SPL and using the feature model created specifically for that product line to derive products. A user can still perform product derivation without automatic product derivation support by creating an empty product and selecting the necessary core assets from the core assets project and sharing them with the product. Manual product derivation requires that the user has the knowledge of the product line to perform the derivation. In contrast with automatic product derivation, the user has help from the product line system, for example, DITA maps already specify what files

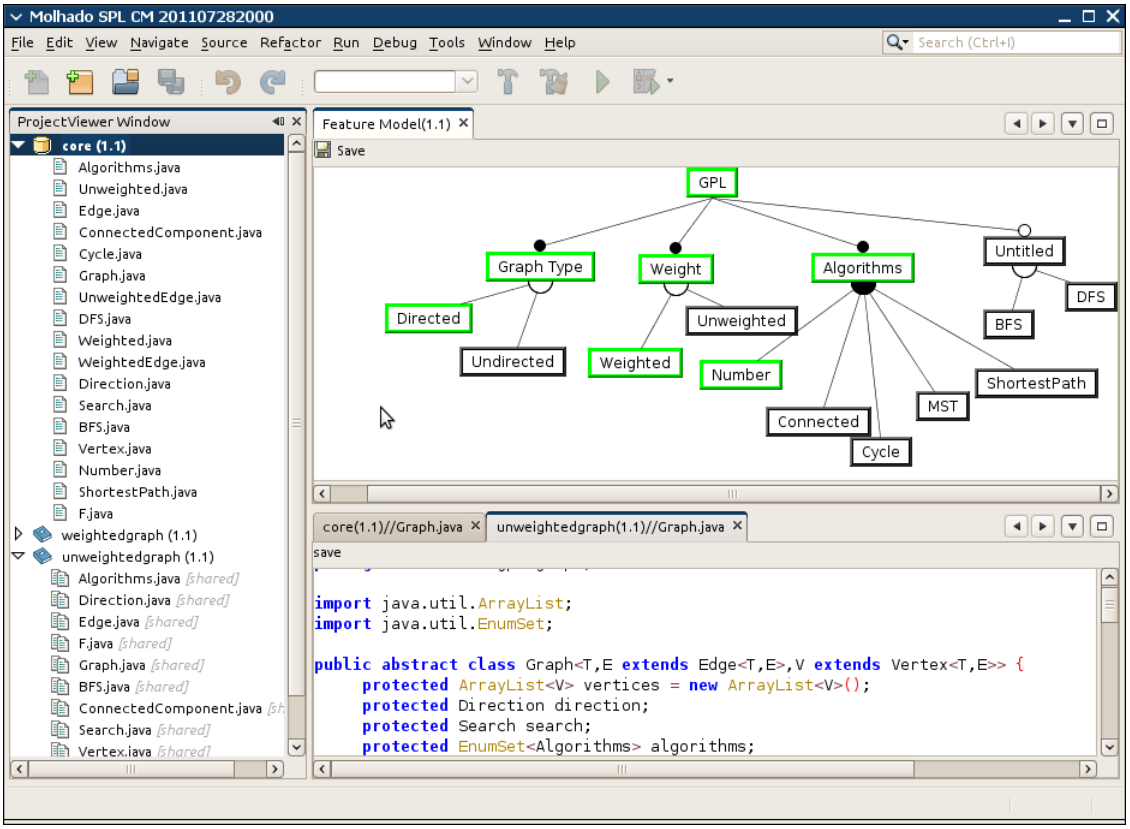


Figure 7.7: Screen shot of the Graph Product Line in Molhado SPL.

go into a product and the feature model tells the user what features are allowed and not allowed. Automatic product derivation helps prevent the user from deriving a broken product. Once a product is derived, developers can modify or add product specific changes to shared assets or add product specific assets thus allowing the product to evolve independent from the core asset project.

We evaluated if Molhado SPL supports product line by importing existing product lines. Independent evolution of core assets and project assets are demonstrated by making independent changes to core assets and products and checking-in their changes. Independent evolution of core assets project and the product projects is supported at the component and project versioning level because each project has its own version space. We also evaluated each of the change propagation cases by making changes and performing change propagation for each of the cases and making sure that each propagation operation completed correctly. We have demonstrated that Molhado SPL's versioning model is flexible by supporting different types of product

line. Because Molhado SPL is an early research prototype, we are only interested that it does what it is supposed to do. Thus, we lack performance, usability and scalability evaluation which are important and would be good topics for future research.

Chapter 8

Conclusion

In this dissertation, a configuration management system called Molhado SPL is proposed to support the evolution of software product lines. Currently, there are no software configuration management (SCM) tools that directly support software product line evolution. Conventional SCM tools are designed to support single product development. The use of conventional SCM tools forces developers to treat a software product line as a single software project either by introducing new programming language constructs or through the use of conditional compilation. Molhado SPL addresses the evolution problem at the configuration level instead of at the code level. We studied the type of operations needed to support software product line and identified eight cases of change propagation. We proposed a versioning model that handled all of the change propagation cases.

8.1 Approach

Molhado SPL supports independent evolution of core assets and products, the sharing of code and the tracking relationships of products and shared code, and the eight cases of change propagation. The Molhado SPL consists of four layers with each layer providing different type of services. At the heart of Molhado SPL are the versioning model, component object, shared component object, and project objects that allow for independent evolution of products and shared artifacts, for sharing, and for supporting change propagations. Further, they allow product specific changes to shared code without interfering with the core asset that is shared. Products can also introduce

product specific asset that only exist in that product.

The low-level versioning layer provides support for versioning, storage and loading of data structures. The component and project versioning layer allows for representation of files and software and document projects. It supports merging and adds enhancements to the version tree. The product line versioning layer provides the versioning model to support product lines and the various mechanisms for change propagation. To support change propagation, we implemented an XML merge algorithm that is shown to be fast and efficient. The algorithm not only merges data structures within Molhado SPL but also merge XML files as standalone software. To support software product derivation and modeling of software product lines, we implemented a feature model editor with debugging support. We also implemented the version-aware XML framework to support the recording of version history in XML documents. This serves two purposes. One is that documents can be edited outside of the repository and can later be brought back into the repository with their change histories. Another purpose is to support collaborative editing of documents in parallel without the need for a versioning repository.

We evaluated Molhado SPL with two product lines: a document product line and a graph software product line. We showed that Molhado SPL supports independent evolution of products and core assets and all eight change propagation cases. We did not evaluate Molhado SPL in terms of scalability or usability. This could be future work. In addition, we may explore the option of using existing configuration management tools such as Subversion, GIT and Mercurial as the low-level versioning layer instead of the current Fluid system. These systems have already handled the scalability issues and have network support.

8.2 Contributions

The contributions made in this dissertation are as follows.

- Molhado SPL is the first prototype to solve the evolution problem at the configuration management level instead of at the source code and programming language level. In doing so, we created a versioning model with code sharing capability, which is unlike any versioning model of conventional SCM systems.

We contribute concepts and terminology on change propagation for software product line.

- The second contribution is a 3-way XML merge algorithm that exploits versioned data structures and unique ID to increase speed and to reduce memory usage.
- The third contribution is a visual feature model editor with debugging. Many have attempted to support feature model debugging using different logic reasoning but they lack user interface to support debugging. We proposed a user interface that hints errors in the feature model editor similar to an IDE.
- Our last contribution is the version-aware XML document framework which embeds versioning history inside XML documents without breaking them or changing their semantics. In the past, version histories are stored externally in files or database which means a file and its history can't be exchange easily between different users.

8.3 Future Work

There are many areas for possible future work. Molhado SPL is incomplete as it is only a research prototype to test out ideas. To be completely usable and to support multiple users, it will need to be implemented as a client-server system. At the moment, it supports only a single user at a time. It could be implemented using a centralized server approach like CVS and Subversion, or using a distributed approach, like GIT and Mercurial. Currently, Molhado SPL does not use snapshots to improve version retrieval and that could be a feature in the future version. The version-aware XML document framework needs further evaluation especially with realistic documents such as Microsoft Office or Open Office documents. Because version-aware XML documents require every node to have a unique ID, there must be a way to introduce unique ID into Office documents. Currently, Molhado SPL has not exploited version-aware XML document as a way to export documents from the versioning system and later import them back into the versioning system. Supporting an extended feature model would be a goal for Molhado SPL's feature editing and debugging. An extended model replaces the And, Or, and Alternative operations with cardinality-based operations.

Bibliography

[1] C. W. Krueger, ‘‘Variation management for software production lines,’’ in *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, (London, UK), pp. 37--48, Springer-Verlag, 2002.

[2] L. Yu and S. Ramaswamy, ‘‘A configuration management model for software product line,’’ *INFOCOMP Journal of Computer Science*, vol. 5, no. 4, pp. 1--8, 2006.

[3] ‘‘Dita maturity model,’’ 2012.

[4] V. Sugumaran, S. Park, and K. C. Kang, ‘‘Introduction,’’ *Commun. ACM*, vol. 49, no. 12, pp. 28--32, 2006.

[5] M. Staples and D. Hill, ‘‘Experiences adopting software product line development without a product line architecture,’’ in *APSEC ’04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC’04)*, (Washington, DC, USA), pp. 176--183, IEEE Computer Society, 2004.

[6] T. Morse, ‘‘CVS,’’ *Linux Journal*, vol. 1996, no. 21es, p. 3, 1996.

[7] S. Deelstra, M. Sinnema, and J. Bosch, ‘‘Product derivation in software product families: a case study,’’ *J. Syst. Softw.*, vol. 74, no. 2, pp. 173--194, 2005.

[8] A. van Deursen, M. de Jonge, and T. Kuipers, ‘‘Feature-based product line instantiation using source-level packages,’’ 2002.

[9] R. C. van Ommering, ‘‘Configuration management in component based product populations,’’ in *SCM*, pp. 16--23, 2001.

[10] R. van Ommering, ''Building product populations with software components,'' in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, (New York, NY, USA), pp. 255--265, ACM Press, 2002.

[11] ''Global software product line and inifinity diversity.''

[12] L. N. Paul Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rev ed., 2001.

[13] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, September 2005.

[14] C. Krueger, *Easing the Transition to Software Mass Customization*. Springer, 2002.

[15] R. Conradi and B. Westfechtel, ''Version models for software configuration management,'' *ACM Comput. Surv.*, vol. 30, no. 2, pp. 232--282, 1998.

[16] P. H. Feiler, ''Configuration management models in commercial environments,'' Tech. Rep. CMU/SEI-91-TR-7 ESD-9-TR-7, Carnegie Mellon Software Engineering Institute, Mar. 1991.

[17] S. Dart, ''Concepts in configuration management systems,'' in *Proceedings of the third International Software Configuration Management Workshop*, ACM Press, 1991.

[18] W. A. Babich, *Software configuration management: coordination for team productivity*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[19] ''Cvs-concurrent versions system.''

[20] ''Subversion.tigris.org.'' http://subversion.tigris.org/.

[21] M. Rochkind, ''The source code control system,'' *IEEE Transactions on Software Engineering*, vol. 1, no. 4, pp. 364--370, 1975.

[22] M. Staples, ''Change control for product line software engineering,'' in *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*

(APSEC'04), (Washington, DC, USA), pp. 572--573, IEEE Computer Society, 2004.

[23] J. van Gurp and C. Prehofer, ''Version management tools as a basis for integrating product derivation and software product families,'' in *Proceedings of the Workshop on Variability Management - Working with Variability Mechanisms at SPLC 2006* (P. Clements and D. Muthing, eds.), no. 152.06/E, pp. 48--58, Oct. 2006.

[24] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson, ''Feature-oriented domain analysis (foda) feasibility study,'' tech. rep., 1990.

[25] H. Spencer and G. Collyer, ''ifdef considered harmful, or portability experience with c news,'' 1992.

[26] ''Welcome to NetBeans.'' http://www.netbeans.org.

[27] W. F. Tichy, ''RCS - A system for version control,'' *Software - Practice and Experience*, vol. 15, no. 7, pp. 637--654, 1985.

[28] ''Mercurial SCM.'' http://mercurial.selenic.com.

[29] ''Git - fast version control system.'' http://git-scm.com.

[30] ''Bazaar versioning system.'' http://bazaar.conical.com.

[31] ''Diffutils homepage.''

[32] R. L. Fontaine, ''Merging xml files: A new approach providing intelligent merge of xml data sets,'' in *In Proceedings of XML Europe 2002*, 2002.

[33] T. Lindholm, ''A three-way merge for XML documents,'' in *Proceedings of the 4th ACM symposium on Document Engineering*, pp. 1--10, ACM Press, 2004.

[34] ''W3C XML.'' http://www.w3c.org/XML.

[35] J. Boyland, A. Greenhouse, and W. L. Scherlis, ''The Fluid IR: An internal representation for a software engineering environment.'' http://www.fluid.cs.cmu.edu.

[36] S. Khanna, K. Kunal, and B. C. Pierce, ‘‘A formal investigation of diff3,’’ in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)* (Arvind and Prasad, eds.), Dec. 2007.

[37] ‘‘Inkscape. draw freely..’’ http://inkscape.org.

[38] ‘‘Glips graffiti editor.’’ http://glipssvgeditor.sourceforge.net/.

[39] ‘‘Netbeans platform.’’

[40] S. Rönnau, C. Pauli, and U. M. Borghoff, ‘‘Merging changes in xml documents using reliable context fingerprints,’’ in *DocEng '08: Proceeding of the eighth ACM symposium on Document engineering*, (New York, NY, USA), pp. 52--61, ACM, 2008.

[41] T. Mens, ‘‘A state-of-the-art survey on software merging,’’ *Software Engineering, IEEE Transactions on*, vol. 28, pp. 449--462, May 2002.

[42] E. W. Myers, ‘‘An o(nd) difference algorithm and its variations,’’ *Algorithmica*, vol. 1, pp. 251--266, 1986.

[43] R. Al-Ekram, A. Adma, and O. Baysal, ‘‘diffX: an algorithm to detect changes in multi-version XML documents.,’’ in *CASCON* (J. R. Cordy, A. W. Kark, and D. A. Stewart, eds.), pp. 1--11, IBM, 2005.

[44] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, ‘‘Change detection in hierarchically structured information,’’ in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 493--504, ACM, 1996.

[45] G. Cobena, S. Abiteboul, and A. Marian, ‘‘Detecting changes in xml documents,’’ *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 41--52, 2002.

[46] M. Lanham, A. Kang, J. Hammer, A. Helal, and J. Wilson, ‘‘Format-independent change detection and propoagation in support of mobile computing,’’ in *In Proceedings of the XVII Symposium on Databases (SBBD 2002*, pp. 27--41, 2002.

[47] Y. Wang, D. DeWitt, and J. Cai, ‘‘X-diff: A fast change detection algorithm for xml documents,’’ 2003.

[48] T. Lindholm, J. Kangasharju, and S. Tarkoma, ‘‘Fast and simple XML tree differencing by sequence alignment,’’ in *DocEng ’06: Proceedings of the 2006 ACM symposium on Document engineering*, (New York, NY, USA), pp. 75--84, ACM, 2006.

[49] ‘‘Dropbox - online backup, file synch, and sharing made easy.’’ http://www.dropbox.com.

[50] ‘‘Google docs.’’ http://www.google.com/google-d-s/documents.

[51] ‘‘XML security standard.’’ http://www.w3.org/standards/xml/security.

[52] K. Djemal, C. Soule-Dupuy, and N. Valles-Parlangeau, ‘‘Management of document multistructurality: Case of document versions,’’ in *Research Challenges in Information Science, 2009. RCIS 2009. Third International Conference on*, pp. 325--332, april 2009.

[53] N. Fousteris, Y. Stavrakas, and M. Gergatsoulis, ‘‘Multidimensional xpath,’’ in *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, iiWAS ’08, (New York, NY, USA), pp. 162--169, ACM, 2008.

[54] A. O. Mendelzon, F. Rizzolo, and A. Vaisman, ‘‘Indexing temporal xml documents,’’ in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB ’04, pp. 216--227, VLDB Endowment, 2004.

[55] F. Zhang, X. Wang, and S. Ma, ‘‘Temporal xml indexing based on suffix tree,’’ in *Software Engineering Research, Management and Applications, 2009. SERA ’09. 7th ACIS International Conference on*, pp. 140--144, dec. 2009.

[56] Y. Zhang, X. Wang, and Y. Zhang, ‘‘A labeling scheme for temporal xml,’’ in *Web Information Systems and Mining, 2009. WISM 2009. International Conference on*, pp. 277--279, nov. 2009.

[57] G. Ozsoyoglu and R. T. Snodgrass, ''Temporal and real-time databases: A survey,'' *IEEE Trans. on Knowl. and Data Eng.*, vol. 7, pp. 513--532, August 1995.

[58] R. T. Snodgrass, *The TSQL2 Temporal Query Language*. Norwell, MA, USA: Kluwer Academic Publishers, 1995.

[59] R. Al-Ekram, A. Adma, and O. Baysal, ''diffx: an algorithm to detect changes in multi-version xml documents,'' in *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '05, pp. 1--11, IBM Press, 2005.

[60] K.-H. Lee, Y.-C. Choy, and S.-B. Cho, ''An efficient algorithm to compute differences between structured documents,'' *Knowledge and Data Engineering, IEEE Transactions on*, vol. 16, pp. 965--979, aug. 2004.

[61] T. Lindholm, J. Kangasharju, and S. Tarkoma, ''Fast and simple xml tree differencing by sequence alignment,'' in *Proceedings of the 2006 ACM symposium on Document engineering*, DocEng '06, (New York, NY, USA), pp. 75--84, ACM, 2006.

[62] A. Marian, ''Detecting changes in xml documents,'' in *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, (Washington, DC, USA), p. 41, IEEE Computer Society, 2002.

[63] S. Rönnau, G. Philipp, and U. M. Borghoff, ''Efficient change control of xml documents,'' in *Proceedings of the 9th ACM symposium on Document engineering*, DocEng '09, (New York, NY, USA), pp. 3--12, ACM, 2009.

[64] S. Y. Chien, V. J. Tsotras, and C. Zaniolo, ''Xml document versioning,'' *SIGMOD Rec.*, vol. 30, pp. 46--53, September 2001.

[65] S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang, ''Storing and querying multiversion xml documents using durable node numbers,'' in *Web Information Systems Engineering, 2001. Proceedings of the Second International Conference on*, vol. 1, pp. 232--241, dec. 2001.

[66] L. A. Rosado, A. P. Marquez, and M. S. Sanchez, ‘‘A data model for versioned xml documents using xquery,’’ in *Digital Information Management, 2008. ICDIM 2008. Third International Conference on*, pp. 931--933, nov. 2008.

[67] Z. Vagena, M. M. Moro, and V. J. Tsotras, ‘‘Supporting branched versions on xml documents,’’ in *Research Issues on Data Engineering: Web Services for e-Commerce and e-Government Applications, 2004. Proceedings. 14th International Workshop on*, pp. 137--144, march 2004.

[68] R. K. Wong and N. Lam, ‘‘Managing and querying multi-version xml data with update logging,’’ in *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng ’02, (New York, NY, USA), pp. 74--81, ACM, 2002.

[69] J.-Y. Vion-Dury, ‘‘Stand-alone encoding of document history(or one step beyond XML diff),’’ in *Proceedings of Balisage: The Markup Conference 2010*, vol. 5, 2010.

[70] ‘‘Java uuid generator (JUG) home page.’’ http://jug.safehaus.org.

[71] C. Thao and E. V. Munson, ‘‘Using versioned tree data structure, change detection and node identity for three-way xml merging,’’ in *Proceedings of the 10th ACM symposium on Document engineering*, DocEng ’10, (New York, NY, USA), pp. 77--86, ACM, 2010.

[72] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[73] D. Weiss and R. C. T. Lai, *Software Product Line Engineering*. Addison-Wesley, 1999.

[74] D. Batory, D. Benavides, and A. Ruiz-Cortes, ‘‘Automated analysis of feature models: challenges ahead,’’ *Commun. ACM*, vol. 49, pp. 45--47, December 2006.

[75] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, ‘‘FeatureIDE: A tool framework for feature-oriented software development,’’ in *Proceedings of ICSE ’09*, (Washington, DC, USA), pp. 611--614, IEEE Computer Society, 2009.

[76] D. Beuche, ‘‘Modeling and building software product lines with pure::variants,’’ in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC ’11, (New York, NY, USA), pp. 46--1, ACM, 2011.

[77] D. Benavides, S. Segura, and A. Ruiz-Cortés, ‘‘Automated analysis of feature models 20 years later: A literature review,’’ *Inf. Syst.*, vol. 35, pp. 615--636, September 2010.

[78] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger, ‘‘XML-based feature modelling,’’ in *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004*, pp. 5--9, Springer-Verlag, 2004.

[79] M. Mendonca, M. Branco, and D. Cowan, ‘‘S.P.L.O.T.: software product lines online tools,’’ in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA ’09, (New York, NY, USA), pp. 761--762, ACM, 2009.

[80] M. Antkiewicz and K. Czarnecki, ‘‘FeaturePlugin: feature modeling plug-in for eclipse,’’ in *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, eclipse ’04, (New York, NY, USA), pp. 67--72, ACM, 2004.

[81] M. Noorian, A. Ensan, E. Bagheri, H. Boley, and Y. Biletskiy, ‘‘Feature model debugging based on description logic reasoning,’’ in *The 17th International Conference on Distributed Multimedia Systems (DMS 2011)*, KSI, 2011.

[82] H. H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan, ‘‘Verifying feature models using OWL,’’ *Web Semant.*, vol. 5, pp. 117--129, June 2007.

[83] I. Lynce and J. P. Marques-Silva, ‘‘On computing minimum unsatisfiable cores,’’ in *International Symposium on Theory and Applications of Satisfiability Testing*, pp. 305--310, May 2004.

[84] U. Junker, ‘‘QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems,’’ in *Proceedings of AAAI ’04*, pp. 167--172, AAAI Press, 2004.

[85] D. Le Berre and P. Rapicault, ‘‘Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution,’’ in *Proceedings of the 1st international*

workshop on Open component ecosystems, IWOCE '09, (New York, NY, USA), pp. 21--30, ACM, 2009.

[86] ''Netbeans visual library 2.0,'' 2011.

[87] D. Le Berre and A. Parrain, ''The Sat4j library 2.2, system description,'' *System*, vol. 7, no. June 2009, pp. 59--64, 2010.

[88] ''DITA OASIS Standard.'' http://dita.xml.org/standard/.

[89] R. E. Lopez-Herrejon and D. S. Batory, ''A standard problem for evaluating product-line methodologies,'' in *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, (London, UK, UK), pp. 10--24, Springer-Verlag, 2001.

[90] C. Liya and H. Zhang, ''Evaluating product line technologies: A graph product line case study,'' in *Proceedings of the Second Australian Undergraduate Students' Computing Conference* (G. Abraham and B. I. P. Rubinstein, eds.), (Melbourne, Victoria, Australia), pp. 32--39, Australian Undergraduate Students' Computing Conference, December 2004.

VITA

Title of Dissertation

A Configuration Management System for Software Product Lines

Full Name

Cheng Thao

Place and Date of Birth

Xiengkhouan, Laos

Colleges and Universities,	Years attended and degrees
University of Wisconsin-Milwaukee	Engineering, Ph.D., 10
University of Wisconsin-Milwaukee	Computer Science, M.S., 4
University of Wisconsin-Green Bay	Human Biology, B.S., 5

Publications

1. **C. Thao** ‘‘Managing Evolution of Software Product Line,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)* (To appear)
2. **C. Thao** and E. V. Munson. ‘‘Version-Aware XML Documents,” in *Proceedings of the 11th ACM Symposium on Document Engineering (DocEng'11)*, New York, NY, USA, 2011. ACM.

3. **C. Thao**. ‘‘Managing Evolution of Software Product Line” in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)* , IEEE Computer Society Press, 2012 (to appear)
4. **C. Thao** and E. V. Munson. ‘‘Version-Aware XML Documents,” in *Proceedings of the 11th ACM Symposium on Document Engineering (DocEng'11)*, New York, NY, USA, 2011. ACM.
5. **C. Thao** and E. V. Munson. ‘‘Flexible Support for Managing Evolving Software Product Lines,” in *Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering (PLEASE'11)*, ACM, New York, NY, USA, 2011, pages 60-64.
6. **C. Thao** and E. V. Munson. ‘‘Using Versioned Tree Data Structure, Change Detection and Node Identity for Three-Way XML Merging,” in *Proceedings of the 10th ACM Symposium on Document Engineering, DocEng'10*, pages 77-86, New York, NY, USA, 2010. (won best paper award, acceptance rate 31%)
7. **C. Thao**, E. V. Munson, and T. Nguyen, ‘‘Software Configuration Management for Product Line Derivation in Software Product Families,” in *Proceedings of the 15th IEEE Conference on Engineering of Computer Based Systems (ECBS 2008)*, Belfast, UK, April 2008, pages 265-276
8. H. Jain, **C. Thao**, and H. Zhao. ‘‘Enhancing Electronic Medical Record Retrieval through Semantic Query Expansion,” *Journal of Information Systems and e-Business Management,* 2010
9. **C. Thao,** , H Jain, and H. Zhao (2008). ‘‘Semantic Query Expansion for Effective EMR Retrieval,” in *Proceedings of the Second China Summer Workshop on Information Management (CSWIM 2008)* (pp. 6-11). Kunming, China (refereed, academic audience, formal paper).
10. K. E. Khan, A. Santos, **C. Thao**, J. J. Rock, P. G. Nagy, K. C. Ehlers: ‘‘A Presentation System for Just-in-time Learning in Radiology.” *J. Digital Imaging* 20(1): 6-16 (2007)
11. M. Wu, P. M. Rhyner, **C. Thao**, L. Kraniak, . C Cronk, K. Cruise: ‘‘A Tablet-PC Application for the Individual Family Service Plan (IFSP)”. *J. Medical Systems* 31(6): 537-541 (2007)

12. **C. Thao**, and M. Wu , ‘‘A Hand-held Application for Individual Family Service Plan (IFSP),” in *Proceeding of American Medical Informatics Association Annual Symposium (AMIA)*, 2006

13. M. Wu, **C. Thao**, X. Mu, E. V. Munson: ‘‘A fisheye viewer for microarray-based gene expression data.” *BMC Bioinformatics* 7: 452 (2006)

14. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**, ‘‘Infrastructures for Development of Object-Oriented Configuration Management Services”, in *Proceedings of the 27th ACM/IEEE International Conference on Software Engineering (ICSE 2005)*

15. **C. Thao** and E. V. Munson. ‘‘A Relevance Model for Web Image Search,” Web Document Analysis II, *Proceedings of the Second International Workshop on Web Document Analysis*, Edinburgh, August 3, 2003. Pattern Recognition and Image Analysis (PRImA) Group, 2003, page 57-60.

16. C. Kahn, and **C. Thao**. ‘‘GoldMiner: a radiology image search engine.” American Journal of Radiology (AJR)2007; 188:1475-1478

17. T. N. Nguyen, E. V. Munson, J. T. Boyland and **C. Thao**; , ”Multi-level configuration management with fine-grained logical units,” in 31st EUROMICRO Conference on Software Engineering and Advanced Applications, pages 248- 255, Sept. 2005. (Acceptance rate: 45%)

18. T. N. Nguyen, E. V. Munson, and **C. Thao**. ‘‘Managing the evolution of Web-based applications with WebSCM,” in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005. ICSM’05, pages 577-586, Sept. 2005. (Acceptance rate: 31%)

19. T. N. Nguyen, E. V. Munson, and **C. Thao**. ‘‘Object-oriented configuration management technology can improve software architectural traceability, in *Third ACIS International Conference on Software Engineering Research, Management and Applications*, Aug. 11-13, 2005, pages 86-93. (acceptance rate: 48%)

20. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**, ‘‘Infrastructures for Development of Object-Oriented Configuration Management Services’’, in *Proceedings of the 27th ACM/IEEE International Conference on Software*

Engineering (ICSE 2005), May 15-21, 2005, St. Louis, Missouri, USA, IEEE Computer Society Press, pages 215224, 2005. (Acceptance rate: 12%)

21. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**. ‘‘Structure-oriented product versioning,” in *International Conference on Information Technology: Coding and Computing*, 2005. Vol. 2, pages 455-460, April 2005. (Acceptance rate: 50%)

22. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**. ‘‘Configuration management for designs of software systems,” in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, 2005, pages 236-243, April 2005.

23. T. N. Nguyen, **C. Thao** and E. V. Munson, ‘‘On Product Versioning for Hypertexts”, in *Proceedings of the 12th ACM International Workshop on Software Configuration Management* (SCM-12), a workshop associated with the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2005), September 59, 2005, Lisbon, Portugal, pages 112132, ACM Press, 2005.

24. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**. ‘‘Flexible Fine-grained Version Control for Software Documents”, in *Proceedings of the 11th IEEE Asia-Pacific Software Engineering Conference* (APSEC 2004), November 30 December 3, 2004, Busan, Korea, pp. 212-219, IEEE Computer Society Press, 2004. (Acceptance rate: 39%)

25. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**, ‘‘Architectural Software Configuration Management in Molhado”, in *Proceedings of the 20th IEEE International Conference on Software Maintenance* (ICSM 2004), September 11-17, 2004, Chicago, Illinois, USA, pp. 296-306, IEEE Computer Society Press, 2004. (Acceptance rate: 28%)

26. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**, ‘‘The Molhado hypertext versioning system”, in Proceedings of the 15th ACM International Conference on Hypertext and Hypermedia (Hypertext 2004), August 9-13, 2004, Santa Cruz, California, USA, pp. 185-195, ACM Press, 2004. (Acceptance rate: 23%)

27. T. N. Nguyen, E. V. Munson, and **C. Thao**, ‘‘Software Traceability via Versioned Hypermedia”, in Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE 2004), June 20-24, 2004, Alberta, Canada, pp. 288-295, Knowledge Systems Institute Publisher, 2004. (Acceptance rate: 33%)
28. T. N. Nguyen, E. V. Munson, and **C. Thao**, ‘‘Fine-grained, structured Configuration Management for Web projects”, Proceedings of the 13th International World Wide Web Conference (WWW 2004), May 17-25, 2004, New York City, New York, USA, pp. 433-443, ACM Press, 2004. (Acceptance rate: 13%)
29. T. N. Nguyen, E. V. Munson, J. T. Boyland, and **C. Thao**, ‘‘Molhado: Object-Oriented Architectural Software Configuration Management”, Formal Research Demonstration at the 20th IEEE International Conference on Software Maintenance (ICSM 2004), September 11-17, 2004, Chicago, Illinois, USA, pp. 510, IEEE Computer Society Press , 2004. (Acceptance rate: 28%)
30. J. Dabrowski, **C. Thao**, and E. V. Munson. ‘‘Image Search and Information Visualization Research at UW-Milwaukees Multimedia Software Laboratory,” in Proceedings of the 2001 CADIP Symposium, Baltimore, Maryland, October 2001.

Major Department

Computer Science

Minor

Math