



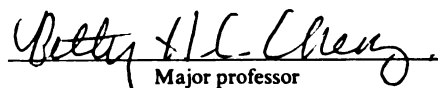
This is to certify that the  
dissertation entitled

Applying Formal Methods to Software Reuse  
presented by

Jun-Jang Jeng

has been accepted towards fulfillment  
of the requirements for

Ph.D. degree in Computer Science

  
Major professor

Date 12/23/93

**LIBRARY**  
**Michigan State**  
**University**

**PLACE IN RETURN BOX** to remove this checkout from your record.  
**TO AVOID FINES** return on or before date due.

DATE DUE	DATE DUE	DATE DUE
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

**MSU is An Affirmative Action/Equal Opportunity Institution**

c:\crl\datedue.pm3-p.1

# **Applying Formal Methods to Software Reuse**

By

*Jun-Jang Jeng*

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Department of Computer Science

1993



# **ABSTRACT**

## **Applying Formal Methods to Software Reuse**

By

*Jun-Jang Jeng*

This dissertation presents an approach, based on formal methods, to the specification, classification, retrieval, and modification of reusable software components. From a set of reusable components that are described by formal specifications, a two-tiered hierarchy of software components is constructed. The hierarchical structure provides a means for representing, storing, browsing, retrieving, and modifying the reusable components; furthermore, the formal specifications provide a means for verifying that a given software component correctly satisfies the current problem. The lower-level hierarchy facilitates the application of logical reasoning techniques for a fine-grained, exact determination of reusable candidates. The higher-level hierarchy provides a coarse-grained determination of reusable candidates. Based upon the framework of the two-tiered component hierarchy, a set of candidate components, which are more general than or analogous to the query specification, can be retrieved from the hierarchy. Two methods are proposed for the modification of candidate components in order to satisfy the query specification. One is to modify the component that is more general than the query specification; the other is to modify a component based on analogy. The graphics-based implementation of the reuse framework is described.

## ACKNOWLEDGMENTS

First of all, I'd like to thank my advisor, Dr. Betty H. C. Cheng for her constant support over the years. Her advice, encouragement, and willingness to allow me so much freedom of direction, provided a wonderful research environment to work.

I am grateful to my committee members, Professors Abdol H. Esfahanian, Jacob M. Plotkin, and John Geske, for their interests in my work, and their helpful suggestions. I am also indebted to Dr. Moon-Jung Chung for his guidance at the early stage of my doctoral research.

My appreciation also extends to my colleagues of Computer Science Department at Michigan State University for providing a great research atmosphere, and my friends for making my life bountiful during these years at East Lansing.

This dissertation would not have been possible without the support from my parents, sisters, and brother. Last but not least, I want to thank my dear wife Shuefung for her continued trust and love to make this dissertation a reality.

# TABLE OF CONTENTS

<b>LIST OF FIGURES</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Need for Software Reuse . . . . .	1
1.2 The Need for Formal Methods . . . . .	3
1.3 Research Contributions . . . . .	5
1.4 Organization of Chapters . . . . .	6
<b>2 Component Reuse Framework</b>	<b>7</b>
2.1 Research Problems/Challenges . . . . .	7
2.1.1 Specification . . . . .	8
2.1.2 Classification . . . . .	8
2.1.3 Retrieval . . . . .	8
2.1.4 Modification . . . . .	9
2.1.5 Other Issues . . . . .	9
2.2 Reuse Framework . . . . .	9
2.3 Reuse Tools . . . . .	12
<b>3 Specification of Reusable Components</b>	<b>15</b>
3.1 Introduction . . . . .	15
3.2 Larch Shared Language . . . . .	17
3.3 Larch Interface Language . . . . .	19
3.4 Component Specification . . . . .	21
3.5 Method Specification . . . . .	24
3.6 Subtypes and Subclass . . . . .	24
3.7 Defining Generality . . . . .	28
<b>4 Construction of Hierarchical Component Library</b>	<b>31</b>
4.1 Lower-Level Hierarchy . . . . .	31
4.1.1 Determining Generality . . . . .	32
4.1.2 Building Lower-Level Hierarchy . . . . .	33

4.2	Higher-Level Hierarchy . . . . .	38
4.2.1	Measure of Similarity between Components . . . . .	39
4.2.2	Hierarchical Clustering . . . . .	43
4.2.3	Hierarchical Clustering Algorithm . . . . .	46
4.3	Implementation . . . . .	47
4.3.1	Browsing Hierarchy . . . . .	47
4.3.2	Implementation of the Construction of Hierarchy . . . . .	49
4.4	Summary . . . . .	53
<b>5</b>	<b>Search and Retrieval of Reusable Components</b>	<b>55</b>
5.1	Hashing Scheme for Software Components . . . . .	55
5.2	Retrieval Algorithm . . . . .	58
5.3	Implementation of Retrieval Process . . . . .	62
<b>6</b>	<b>Modification of Reusable Components Based on Generality</b>	<b>65</b>
6.1	Introduction . . . . .	65
6.2	Predicate Transformer <i>wp</i> . . . . .	66
6.2.1	Simple C++ Statements . . . . .	68
6.2.2	The <i>abort</i> Statement . . . . .	68
6.2.3	The <i>skip</i> Statement . . . . .	69
6.2.4	The <i>assignment</i> ( $y := E$ ) Statement . . . . .	69
6.2.5	The <i>alternative</i> ( <i>IF</i> ) Statement . . . . .	69
6.2.6	The <i>iterative</i> ( <i>DO</i> ) Statement . . . . .	72
6.2.7	The <i>compound</i> ( $S_0; S_1$ ) Statement . . . . .	73
6.3	Modifying a More General Component . . . . .	73
6.4	Modification Process . . . . .	76
6.5	Modification Example . . . . .	78
6.6	Summary . . . . .	84
<b>7</b>	<b>Modification of Reusable Components Based on Analogy</b>	<b>86</b>
7.1	Introduction . . . . .	86
7.2	Analogical Matching . . . . .	89
7.3	Heuristics for the Matching Process . . . . .	90
7.4	Top-Down Matching Approach . . . . .	92
7.5	Matching Algorithm . . . . .	95
7.6	Matching Example . . . . .	99
7.7	Modification Example . . . . .	106
7.8	Summary . . . . .	108

<b>8</b>	<b>Case Study</b>	<b>110</b>
8.1	Introduction . . . . .	110
8.2	Widgets and Larch . . . . .	111
8.3	The Specification for Widgets . . . . .	113
8.3.1	Primitive Widget . . . . .	114
8.3.2	<i>Object</i> and <i>SetResource</i> . . . . .	115
8.3.3	Well-Formed Window . . . . .	119
8.4	Specification of Scrolled Text Editor with Popup Menus . . . . .	120
8.4.1	The X Model . . . . .	121
8.4.2	The LSL Level . . . . .	122
8.4.3	The Interface Level . . . . .	129
8.5	Applying Reuse Processes to Specification Components of Xt/Motif Widgets . . . . .	132
8.5.1	Construction Process . . . . .	134
8.5.2	Retrieval Process . . . . .	138
8.5.3	Modification Process . . . . .	138
8.6	Summary . . . . .	141
<b>9</b>	<b>Related Work</b>	<b>145</b>
9.1	Related Work for Reuse . . . . .	145
9.2	Related Work for Analogy . . . . .	150
9.2.1	Analogy in AI . . . . .	150
9.2.2	Analogy in Software Engineering . . . . .	153
9.3	Related Work of Specifying GUI . . . . .	156
<b>10</b>	<b>Concluding Remarks</b>	<b>158</b>
10.1	Summary . . . . .	158
10.1.1	Construction of Component Hierarchy . . . . .	158
10.1.2	Incorporation of Formal Methods to Software Reuse . . . . .	159
10.1.3	Retrieving Reusable Components from the Hierarchy . . . . .	159
10.1.4	Modifying More General Components . . . . .	160
10.1.5	Modifying Analogous Components . . . . .	160
10.2	Future Work . . . . .	161
<b>A</b>	<b>Analogical Reasoning</b>	<b>163</b>
<b>B</b>	<b>Bottom-Up Matching Approach</b>	<b>167</b>
<b>C</b>	<b>The Specifications for Motif Widgets</b>	<b>169</b>

C.1	The Functionalities of Motif Widgets . . . . .	169
C.2	The LSL Traits of Motif Widgets . . . . .	171
C.3	The LIL Specifications of Widgets . . . . .	179
<b>D</b>	<b>Algorithm to find LGCs</b>	<b>183</b>
<b>E</b>	<b>Program Listing</b>	<b>186</b>
E.1	C++ Implementation for List . . . . .	186
E.2	C++ Implementation for Array . . . . .	188
E.3	C++ Implementation for Stack . . . . .	189
E.4	C++ Implementation for DoubleList . . . . .	190
E.5	A Window with Pulldown Menu (Existing Component) . . . . .	193
E.6	A Window with Popup Menu (Modified Component) . . . . .	196
<b>BIBLIOGRAPHY</b>		<b>200</b>

## LIST OF FIGURES

2.1	Development and Reuse Modules. . . . .	10
2.2	Overview of Component Reuse Module. . . . .	13
2.3	Reuse Tools . . . . .	14
3.1	An Example Trait: <i>Table</i> . . . . .	18
3.2	More Examples of Traits. . . . .	20
3.3	Basic Construct for Component Specifications . . . . .	22
3.4	Component interfaces for <i>intSet</i> . . . . .	23
3.5	Grammar for Component Methods. . . . .	25
3.6	The Borland C++ Subclass Relationship. . . . .	26
3.7	Relations Among the Traits . . . . .	29
3.8	Subsumption Test Algorithm based on $\sqsupseteq_{term}$ . . . . .	30
4.1	Resolution Rule . . . . .	32
4.2	Full Resolution Rule . . . . .	32
4.3	Modified Resolution Rule . . . . .	33
4.4	Using MST to decide the <i>generality</i> relationship between components <i>Comp<sub>A</sub></i> and <i>Comp<sub>B</sub></i> . . . . .	34
4.5	Building the lower-level hierarchy by pair-wise comparison. . . . .	35
4.6	Building lower-level hierarchy by recursive comparison. . . . .	37
4.7	Example of building hierarchy by recursive comparison. . . . .	38
4.8	Example of comparing two SOLs. . . . .	39
4.9	Matrices for components X and Y. . . . .	42
4.10	Similarity matrix $s'(X, Y)$ . . . . .	43
4.11	Refinement of partitions in an agglomerative clustering algorithm . .	45
4.12	Agglomerative Hierarchical Clustering Algorithm . . . . .	46
4.13	Hierarchical Clustering Algorithm . . . . .	48
4.14	Two-tiered hierarchy formed by the subsumption test and the cluster- ing algorithms . . . . .	49
4.15	Sample application of subsumption test algorithm . . . . .	50

1



4.16	Sample application of clustering algorithm . . . . .	51
4.17	Specification of Component <i>Queue</i> . . . . .	52
4.18	Specification of Component <i>DoubleQueue</i> . . . . .	54
5.1	Knuth-Bendix Ordering . . . . .	59
5.2	The Process of Retrieving Software Components. . . . .	60
5.3	The Algorithm for Retrieval by Subsumption Test. . . . .	61
5.4	Sample application of construction and retrieval processes . . . . .	63
6.1	Modifying Implementation Based on Specification Changes. . . . .	75
6.2	The Process of Modifying More a <i>more general method</i> . . . . .	77
6.3	The LCL Specification of <b>List</b> Class. . . . .	79
6.4	The LCL Specification of <b>Array</b> Class. . . . .	81
7.1	Analogical Reuse Modification Process. . . . .	88
7.2	Matching Algorithm . . . . .	96
7.3	Simple example of the matching algorithm. . . . .	100
7.4	Larch specification of <b>Stack</b> class. . . . .	102
7.5	Larch specification of <b>DoubleList</b> class. . . . .	103
7.6	An implementation for matching process. . . . .	105
8.1	The <i>ManagerWindow</i> trait. . . . .	112
8.2	Part of the <i>ScrolledText</i> specification. . . . .	113
8.3	The dependencies of the Window's traits. . . . .	114
8.4	The <i>BasicWindow</i> trait. . . . .	115
8.5	The <i>Primitive</i> trait. . . . .	116
8.6	The <i>Object</i> trait. . . . .	118
8.7	The <i>SetResource</i> trait. . . . .	118
8.8	The <i>WFWidget</i> trait. . . . .	120
8.9	An Example of Scrolled Text Editor with Popup Menus. . . . .	121
8.10	Parent-child relationships between widgets. . . . .	123
8.11	The <b>LtScrolled</b> trait. . . . .	124
8.12	The <b>LtTextField</b> trait. . . . .	126
8.13	The <b>LtText</b> trait. . . . .	127
8.14	The <b>LtMenu</b> trait. . . . .	128
8.15	The <b>LtScrolledText</b> trait. . . . .	129
8.16	The <b>LtScrolledTextMenu</b> trait. . . . .	130
8.17	Part of the Interface Specification of <b>Primitive</b> Widget. . . . .	131
8.18	The Larch Interface Specification of <b>Popup</b> . . . . .	132

8.19	The Larch Interface Specification of <code>Popupmenu</code> . . . . .	133
8.20	The unstructured widget components. . . . .	135
8.21	The lower-level hierarchy of widget components. . . . .	136
8.22	The two-tiered hierarchy of widget components. . . . .	137
8.23	An example of retrieving an exact-matched component. . . . .	139
8.24	An example of retrieving a set of more general components. . . . .	140
8.25	An example of computing matching between two methods. . . . .	142
8.26	The output of the pulldown menu implementation. . . . .	143
8.27	The output of the popup menu implementation. . . . .	143
A.1	Gentner's Model of Analogy . . . . .	164
A.2	PI Model . . . . .	165
A.3	Analogy and Generalization . . . . .	166
D.1	The algorithm to find a set of LGCs. . . . .	184
D.2	Coloring, joining, and collecting a partial ordering set. . . . .	185

# CHAPTER 1

## Introduction

Software reuse has been claimed to be a means for overcoming the software crisis [1, 2, 3, 4]. Some of the major issues that make software reuse difficult are component classification, retrieval, modification, and library maintenance. Research in Software Reuse has great potential to facilitate an improvement in the productivity, the quality, and the reliability of software development. Results from this research serve to enhance the practice of software reuse by using formal methods to provide a means to construct a hierarchical structure of reusable software components, retrieve reusable software components from the constructed hierarchy, and modify retrieved reusable components based on a comparison between existing and user-supplied specifications.

### 1.1 The Need for Software Reuse

The rapid advancement in the hardware industry has provided users and developers with increased and less expensive processing power and storage capacity in a relatively short period of time. This new technology facilitates new applications, thus leading to a growing demand for reliable and functionally complete systems. In today's technology race, software has proven to be the bottleneck in the production

process [5, 6, 7, 8, 9]. Delivered software is often of poor quality and is very difficult to maintain. Software reuse has the potential to increase the quality of developed software. Software reuse has been practiced for years, for example, the libraries of mathematical subroutines, report generators, and many application packages. However, in order to achieve the desired benefits, software reuse must be expended beyond small- or medium-sized software systems. In order to handle the size and complexity of modern software, we are obliged to solve the problem of Very Large Scale Reuse (VLSR). VLSR introduces a new area of research problems centered around the issue of making the software representation sufficiently general to allow reuse over a broad range of target systems, and of automating the reuse process, including the understanding and adaptation of reusable components for a new system.

For more than twenty years, programmers dreamed of developing software system that are engineered like traditional physical systems. They expressed great interest in having a software components catalog from which software parts can be assembled, much as is done with mechanical engineering or electronic engineering. Reusable software component libraries have been suggested as a means for facilitating software reuse [2, 3, 10]. The idea of a software component library was first invented by McIlroy in the NATO Software Engineering Conference of 1968 [11]. McIlroy proposed a library of reusable components and automated techniques for customizing components for different degrees of precision and robustness. A similar view based on software ICs has been proposed by Cox [12].

There is considerable room for transferring general purpose engineering design and management techniques to reusable software component's design and management. Reusable hardware components, such as TTL integrated circuits are universally used in hardware design [4]. In the Texas Instruments' TTL Data Book, a diverse collection of device abstractions are presented to help the users understand and select among candidate components. The analogy is often drawn between TTL components

and off-the-shelf reusable software components. Some researchers believe that finding an analogous diverse collection of expositions for the understanding of reusable software components is the fundamental operational problem that must be solved in the development of any reuse system. However, general purpose reusable components libraries will likely consist of several orders of magnitude more components than the libraries of hardware components. The issue of scale will make large scale reusable component libraries more difficult to understand and use than the hardware analogy.

There are notable differences between software and products from other engineering disciplines, and these differences induce some technical and non-technical issues of reusable software components. Despite some progress in the development of reuse systems in recent years, most software systems are not developed from reusable components.

There are both technical and non-technical impediments to achieving software reuse. The examples of non-technical impediments include management resistance, psychological barriers etc [13]. Serious efforts must be expanded in addressing these two issues in order to achieve success, much as is true for software engineering. However, in this dissertation, only the technical issues will be addressed. The premise is that with a solid foundation for software reuse, the tasks for the managers, designers, programmers, and users will be greatly facilitated in making software reuse a reality.

## 1.2 The Need for Formal Methods

The goal of software reuse is to increase the productivity of programmers and improve the quality of developed software [2]. Software reuse has been impeded by the lack of effective techniques for representing and managing software component libraries. In current industrial settings, software reuse is often based on the availability of the original authors to act as consultants or the existence of relevant and descriptive documen-

tation. Another approach is to exploit information retrieval methods that are based on analyses of natural language documentation. These methods have also been proposed for constructing software libraries [14, 15]. Unfortunately, software components represented by natural language may hinder the retrieval process due to the problems of ambiguity, incompleteness, and inconsistency inherent to natural languages. Neither of these options facilitate the automation of software reuse determination, nor do they provide a means for verifying that a given software component correctly satisfies a new problem specification. The above mentioned problems can be minimized by using formal specifications to represent software components [16, 17, 18, 19] and applying formal methods to the software artifacts that are formally described. Formal methods provide a means for establishing properties by formal reasoning. This formal reasoning is not possible by means of informal methods. Rigorous reasoning guarantees the correctness of some properties via strict argumentation. Formality also allows for showing consequences of a specification. Therefore, implicit contradictions of requirements that are not obvious in the process of software development can be detected in an early stage of software production.

Formal methods enable people to write specifications that have precise meanings. Precision demands deeper insight into the problem domain than informal specifications. Specifications usually become much clearer when expressed in a formal notation and then rephrased into natural language. Thus, formal specifications can help clients comprehend what they want to purchase. Formal specifications can be used to guide the implementation of a system [20, 21]. Because of the precision of the description, the programmers know exactly what needs to be implemented. Formal specifications help clarify what the purpose of each component of a system, therefore increasing the maintainability property. Formal methods also lead to software components that are easier to reuse. A precise and clear description as provided by a formal specification can provide assistance to the user of a software library.

As a consequence of all the advantages mentioned above, formal methods decrease the cost of development. The investment in formal methods in the early stages of production aims to eliminate errors and ambiguity and to discover missing properties in the requirements and specifications. These assets minimize costs imposed during maintenance and implementation.

### 1.3 Research Contributions

This dissertation presents an approach, based on formal methods, to the specification, classification, retrieval, and modification of reusable software components. From a set of reusable components that are described by formal specifications, a two-tiered hierarchy of software components is constructed. The hierarchical structure provides a means for representing, storing, browsing, retrieving, and modifying the reusable components; furthermore, the formal specifications provide a means for verifying that a given software component correctly satisfies the current problem. The lower-level hierarchy is created by a subsumption test algorithm; this level facilitates the application of logical reasoning techniques for a fine-grained, exact determination of reusable candidates. The higher-level hierarchy provides a coarse-grained determination of reusable candidates and is constructed by applying a hierarchical clustering algorithm to the most general components from the lower-level hierarchy. Based upon the framework of the two-tiered component hierarchy, a set of candidate components, which are more general than or analogous to the query specification, can be retrieved from the hierarchy. Two methods are proposed for the modification of candidate components in order to satisfy the query specification. One is to modify the component that is more general than the query specification; the other is to modify an analogous component. All of the above reuse tools are incorporated and integrated within a graphical reuse framework. In addition, we present several examples, includ-

ing a large scale example, to illustrate the applicability of the reuse framework on real problems.

## 1.4 Organization of Chapters

The remainder of the dissertation is organized as follows. Chapter 2 gives the framework of our reuse system for construction, retrieval, and modification of reusable software components. Major research issues associated with component reuse are identified. Chapter 3 describes the formal specification language that we use to specify the reusable software components. Chapter 4 focuses on the task of constructing the two-tiered hierarchy of reusable components. Chapter 5 presents retrieval processes for finding the candidate components from the two-tiered hierarchy. Chapter 6 describes a method to modify those retrieved components that are more general than the query specification. Chapter 7 describes a new approach to modifying retrieved components that are analogous to the problem specification. Chapter 8 presents a moderate-sized example to illustrate the application of our reuse processes. Chapter 9 reviews other projects that are related to our work. Chapter 10 summarizes the dissertation and suggests future investigations.



# CHAPTER 2

## Component Reuse Framework

In order to make software reuse successful, several problems must be resolved including software classification, retrieval, modification, and library maintenance [22, 23]. Software component libraries have been suggested as a means for facilitating software reuse [2, 3, 10, 24]. The potential benefits of software reuse and of large component libraries are obvious but the promise of increased productivity and quality remain elusive because several issues (problems) have not been resolved. In this chapter, we describe the research issues associated with component reuse and give an overview of our component reuse framework. Based on this framework, we have developed several reuse processes to facilitate the reusability of software components. The reuse tools associated with the reuse processes are also briefly described at the end of this chapter.

### 2.1 Research Problems/Challenges

Reuse is a simple idea: attempt to use something for one purpose that was originally intended and used for another purpose. It can be a procedure invoked several times with different parameters. However, it is nontrivial to define the object that we will reuse. A software component may consist of one or more of the following related items:

source code, specifications, requirement, architectures, and test cases, thus indicating that we need to reuse more than code [25]. This dissertation focuses on the reusability of software components based on formal specifications of the software components. In general, to build a reuse system based on software components involves the following techniques: specification, classification, retrieval, and modification. Each of these topics will be described in turn.

### 2.1.1 Specification

Specification describe the behavior of reusable software components. One of the major tasks in developing large reusable component libraries is in providing concise semantic abstractions for components. It is not reasonable to present a large number of source code components to a user and expect the source code to serve as the only description of the component behavior. What is required is an abstraction specification level that describes the behavior of the component in the library in a more succinct form. A higher level than source code should be utilized because we want to emphasize *what* the component does rather than *how* it does for the purpose of reuse.

### 2.1.2 Classification

Classification addresses the process how the system/designer catalogs the reusable components in the software component library. Techniques may include the construction of a component hierarchy and domain analysis. This dissertation focuses on automating the former technique. The latter one has largely been addressed in [26].

### 2.1.3 Retrieval

Retrieval concerns the process how the programmers identify and retrieve a set of reusable components from the software component library that could be unstructured

or hierarchical depending on the scheme of the construction process [27]. A retrieval process should also provide some type of browsing mechanism to assist the user in navigating through the library.

#### 2.1.4 Modification

Modification refers to changes to existing components needed to satisfy new/current problem needs. Direct modification of source code is a difficult task for most reusers of the software components from a component library. Therefore, we are interested in modifying the specifications of retrieved components that are *more general than* or *analogous to* the query specification. A user can modify the source code based on the information of the specification that is required to be changed.

#### 2.1.5 Other Issues

*Decomposition* of the query specification into simpler queries reduces the retrieval complexity and enhances the reusability of the component library. *Integration* of reusable components concerns the framework in which a collection of components can be combined to form a set of reusable components and an executable system. *Synthesis* of the complete reusable component represented formally specified refers to the development of an executable code component from the specification component.

### 2.2 Reuse Framework

Our model for reusing software components divides the conventional life-cycle models into two phases: first, the *component development module* delivers developed components; second, the *component reuse module* supplies reusable components to *component development module* and stores the developed components into the *software components library*. The latter module primarily addresses the reusability of existing

library components and can be decomposed into several processes. Our approach to reusability is based on software models and formal methods. The overview of com-

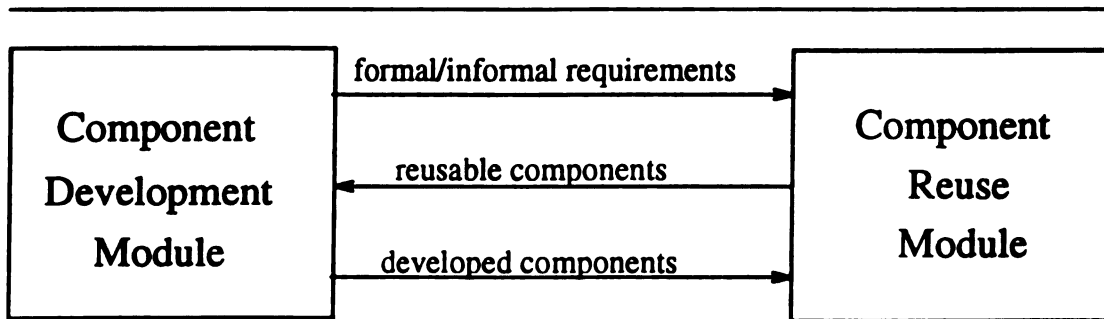


Figure 2.1. Development and Reuse Modules.

---

ponent reuse module is shown in Figure 2.2. Each box represents a different process of the component reuse module.

**Reusable Software Component Library** contains the reusable software components that include the requirements, specification, source code, and other supporting documents. Components must be organized under a classification scheme to facilitate any searching technique. The process for the *Construction of a Hierarchy* supports to build up the hierarchies to classify the components in the library. Automation for component selection will be very essential since a manual search through a library containing thousands of components is not practical. The **Retrieval** process narrows the choice of candidate reusable components by applying automated reasoning and hashing techniques to the specifications of software components in the component library. The **Modification** process modifies the candidate components to suit the required specification. The modification needs to be accomplished at a higher level than the code level since direct editing of source code should be avoided as much as possible. The above four processes (specification, construction, retrieval, and

modification) are specifically addressed in this dissertation. However, other processes are shown in Figure 2.2 in order to give a complete overview of our reuse framework.

The **Component Development Module** is project oriented. Its goal is to deliver the products required by the customer and can be modeled by the traditional software model, e.g., waterfall model. Two kinds of requirements may be given from this module: *informal* or *formal*. Informal requirements need to be converted into formal requirements, represented in the Larch specification language for our system, by two processes, **Requirement Analysis** and **Specification Editor**. We believe the full automation of these two processes are beyond current technology. But some progress has been made in helping users create formal specification from informal requirements [28, 29].

In order to reduce the complexity of the *Retrieval* and *Modification* processes, decomposing the required specification into “smaller” specifications is necessary. But some issues are still not clear, for example, “how to define the size of a specification?”, “how to automate the decomposition of a specification?” and “what is the optimal level of granularity?” etc. The **Integration** process attempts to integrate the modified components to a reusable component. The correctness of components should be preserved after integration. After obtaining a reusable component that fits the requirement from the *Component Development Module*, the process **Synthesis** transforms the reusable specifications into executable code such as C or Pascal based upon the information from the modification process since we do not want to reimplement software that already exists. The classification of reusable components in a library can be based on the application domain: each component hierarchy belongs to a distinct domain. Examples of domains are flight control system, information management software for insurance system, and so on. The purpose of the **Domain Analysis** process is to catalog the reusable software components into different application domains and to ensure that the components within a given hierarchy are in

the same domain.

## 2.3 Reuse Tools

The reuse processes of our system can be realized by the reuse tools shown in Figure 2.3. The **Constructor** performs the construction process with input components from either existing libraries or a group of newly developed components. The **Browser** enables the user browse through the constructed component hierarchy. The **Retriever** retrieves a set of candidate components from the component library based on the user's query and returns them to the user. The user then can either accept the candidate components or submit them to the **Modifier**, thus requesting assistance from the system in the modification of the candidate components. Figure 2.3 only gives the reuse tools in which our investigations address. Other tools such as the *component specification editors*, *component synthesizer*, and *query specification decomposer* are also important in the *Component Reuse Framework*, but they are beyond the scope of this dissertation.

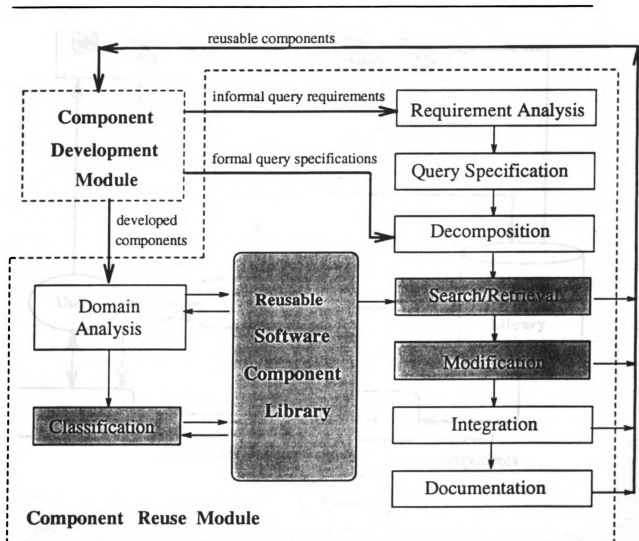


Figure 2.2. Overview of Component Reuse Module.

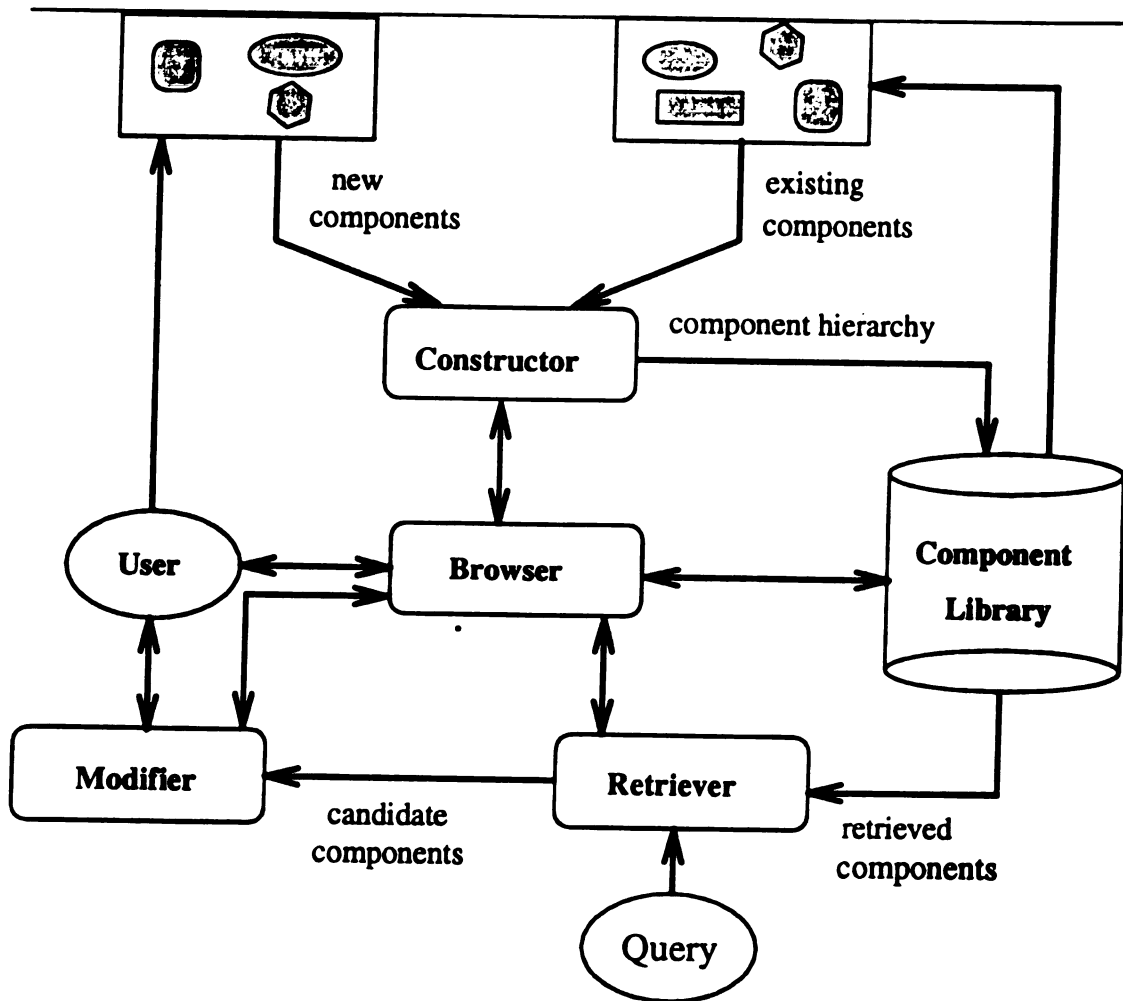


Figure 2.3. Reuse Tools



# CHAPTER 3

## Specification of Reusable Components

### 3.1 Introduction

A *formal specification* is a description that is expressed in a notation whose syntax and semantics are well-defined. Musical compositions are commonly specified in *standard notation*, which is understood by musicians everywhere. Such notations help us to specify things very precisely, completely, and unambiguously.

Most software is made up of procedural and data abstractions, that is, procedures and user-specified and system-defined data structures [30]. Object-oriented analysis can be used to decompose complex software, which involves defining a set of user-specified data abstractions or *abstract data types* (ADTs) [31, 32, 33, 34, 35, 36]. Thus, in order to apply an object-oriented approach to software reuse, we focus on data abstraction, where it is assumed that procedural abstractions are implicitly addressed when discussing the operations that are applicable to the data abstractions. The specification for a software component corresponds to the specification of an ADT and a set of *methods* that operate on that ADT. The specifications of object-oriented (or object-based) languages have been explored by many researchers [37, 38, 39]. There-

fore, we intend to build on what has already been done when choosing a specification language for object-oriented programming languages.

The formal specification language used in this work is *Larch* [39, 40]. *Larch* was designed to specify the properties of ADTs. The advantage gained in using *Larch* is the explicit separation of concerns between state-independent and state-dependent properties. *Larch* provides a *two-tiered* approach to specification. In the first tier, the specifier writes *traits* in the Larch Shared Language (LSL) to assert state-independent properties of a program. Each trait introduces *sorts* and *operators* and defines equality between terms composed of the operators.

In the second tier, the specifier writes *module specification* in a Larch interface language to describe state-dependent effects of a program, such as GCIL [41]. A **requires** clause states each procedure's precondition; a **modifies** clause lists those objects whose values may possibly change; an **ensures** clause gives its postcondition. The assertion language for the pre- and postconditions is drawn from LSL traits. Through **based on** clauses, a Larch interface links to LSL traits by specifying a correspondence between programming-language specific types and LSL sorts. An object has a type and a value that ranges over terms of the corresponding sort.

The Larch Shared Language is presented in Section 3.2. The Larch interface language is described in Section 3.3, where *Larch/C++* is used as an example to demonstrate the concept of the interface language. Section 3.4 and Section 3.5 present the specifications for component and method, respectively. The distinction between *subclass* and *subtype* is discussed in Section 3.6. Finally, the *generality* relationship between a pair of components is defined in Section 3.7.

## 3.2 Larch Shared Language

We use the interface languages to specify reusable program components (e.g. C++ classes). Since each LIL deals with specifying the behavior of components observable in a particular programming language, it provides a mechanism for writing assertions, in terms of pre- and postconditions, about the program states; these states can be translated into predicate calculus formulas that are the specifications used for reusable components in our system [42, 43, 44]. It incorporates programming-language-specific notations for constructs such as side effects, exception handling, and iterators. Several programming languages have been specified by Larch interface languages, such as C [45], Modula-3 [37], and CLU [40]. been chosen as such role

The Larch family of specification languages supports many useful features for designing modular and reusable systems. Many language-independent abstractions are useful in a wide variety of specifications, for example, integers, lists, sets, queues, arrays, relations, mappings, and orders. Larch facilitates the accumulation of open-ended collections of *reusable* specification components in the form of LSL handbooks. Based on the set of reusable specifications, larger reusable specification components can be built. Data and functional abstractions play an important role in the program design of Larch's specifications from which modular programming components are relatively easily constructed. In LSL, all the data types and terms appearing in interface languages are explicitly defined to ensure that the understanding of programmers matches the specifier's. This feature is exploited in the reuse system when determining the candidate reusable components derived from the defined terms in LSL library. The rest of this section presents an overview of the *traits* in the LSL. If the reader is familiar with LSL then the remainder of this section may be skipped.

The *trait* is the basic unit of specification in the LSL. Sometimes the collection of operations will correspond to an ADT. Frequently, however, it defines some type that is

not fully characterized. Figure 3.1 shows an example of a trait *Table* specifying a class of tables that stores values in indexed places. It is similar to conventional algebraic specifications. The part of the specifications following the **introduces** keyword gives a list of operators, each with its signature. Each trait defines a *theory* in typed first-

---

```

Table: trait
  introduces
    new:  $\rightarrow$  Tab
    #  $\in$  # : Ind, Tab  $\rightarrow$  Bool
    lookup: Tab, Ind:  $\rightarrow$  Val
    isEmpty: Tab rightarrow Bool
    size: Tab  $\rightarrow$  Card
  assert  $\forall i, i': \text{Ind}, val : \text{Val}, t : \text{Tab}$ 
    lookup(add(t,i,val),i') == if  $i = i'$  then  $val$  else lookup( $t, i'$ )
     $\neg(i \in \text{new})$ 
     $i \in \text{add}(t, i', val) == i = i' \vee i \in t$ 
    size(new) == 0
    size(add( $t, i, val$ )) == if  $i \in t$  then size( $t$ ) else lookup( $t, i'$ )
    isEmpty( $t$ ) == size( $t$ ) = 0

```

Figure 3.1. An Example Trait: *Table*.

---

order logic with equality. A theory is a set of formulas without free variables. It contains the trait's assertions, the conventional axioms of first-order logic, everything that follows from them and nothing else. Equational theories are useful, but a stronger theory is often needed, for example, when specifying ADTs. The clause **generated** by **asserts** that all values of a sort can be generated by a given list of operators, thus providing a generator induction schema for the sort. This clause is useful for specifying constructors of ADTs. The clause **partitioned by** asserts that all distinct values of a sort can be distinguished by a given list of operators. Terms that are not

distinguishable using any of the partitioning operators of their sort are equal.

For modularity, it is often useful to include a separate trait by reference. This convention makes it easier to reuse pieces of others specifications and handbooks. We might add a clause following the keyword **include** or **import** to some trait. The theory with the including (importing) trait is the theory associated with the union of all of the **introduces** and **asserts** clauses of the trait body and the included traits. Figure 3.2 gives two more trait examples, *Container* and *Member*. A new clause beginning with **assumes** is introduced. The trait *Member* builds on *Container* by assuming it. It constrains the *new* and *insert* operators that it inherits from *Container*, as well as the operators that it introduces,  $\in$  and  $\notin$ . Similarly, the trait *Set* assumes the traits *Container* and *Member*. The theories associated with any trait includes the theories of the traits that is **assumes**, **includes**, or **imports**.

### 3.3 Larch Interface Language

The *Larch Interface Language* (LIL) describes the effects of a program component's operations. In this section, we define the specifications related to a *component* of object-oriented programming languages. Leavens and Choen have designed a preliminary version of Larch/C++ [38]. We use similar specifications to specify program components (classes).

Most notations of Larch/C++ are taken from LCL [45] that is a Larch interface language for the C language. The basic notations from LCL are described as follows. The *states* are mappings from *locs* to *objects*, where *locs* refer to the locations of variables. Each variable identifier names a loc. The major kinds of objects are

- *basic values* are mathematical abstractions defined by LSL.
- *locs* are abstractions of computer memory cells.
- *structs* are collections of locs.

---

**Container (E, C): trait**

**introduces**

**new:**  $\rightarrow C$

**insert:**  $C, E \rightarrow C$

**asserts**

**C generated** by new, insert

**Member (E, C): trait**

**assumes** Container

**introduces**

**#  $\in$  #:**  $E, C \rightarrow \text{Bool}$

**#  $\notin$  #:**  $E, C \rightarrow \text{Bool}$

**asserts**  $\forall c: C, e1, e2: E$

$\neg(e1 \in \text{new})$

$e1 \in \text{insert}(c, e2) == e1 = e2 \vee e1 \in c$

$e1 \notin c == \neg(e1 \in c)$

**implies** converts  $\in, \notin$

**Set (E, C): trait**

**assumes** Container, Member

**introduces**

**#  $\cup$  #:**  $C, C \rightarrow C$

**#  $\cap$  #:**  $C, C \rightarrow C$

**#  $\setminus$  #:**  $C, C \rightarrow C$

**delete:**  $C, E \rightarrow C$

**isempty:**  $C \rightarrow \text{Bool}$

**size:**  $C \rightarrow \text{Card}$

**asserts**  $\forall c: C, e1, e2: E$

**C partitioned** by  $\in$ .

$e1 \in (s1 \cup s2) == e1 \in s1 \vee e1 \in s2$

$e1 \in (s1 \cap s2) == e1 \in s1 \wedge e1 \in s2$

$e1 \in (s1 \setminus s2) == e1 \in s1 \wedge e1 \notin s2$

$e1 \in \text{delete}(s1, e2) == e1 \in s1 \wedge e1 \neq e2$

$\text{isEmpty}(\{\}) = \neg \text{isEmpty}(\text{insert}(s1, e1))$

$\text{size}(\{\}) == 0$

$\text{size}(\text{insert}(s1, e1)) == \text{if } e1 \in s1 \text{ then } \text{size}(s1) \text{ else } \text{succ}(\text{size}(s1))$

$s1 \subseteq s2 == \text{isEmpty}(s1 \setminus s2)$

Figure 3.2. More Examples of Traits.

---

- *unions* are similar to structs, except that their locs overlap.
- *arrays* are bounded vectors of adjacent locs, indexed from 0.
- *pointers* are references to collections of one or more adjacent locs.

The following LCL primitives are available for accessing the initial and final states.

- `''` can be applied to locs, arrays and structs. It is used to extract their values from the final states. If no symbol is applied then the extracted values are from initial states.
- `*` is used to dereference a pointer, producing its locs with offset 0.
- `->` is a syntactic shorthand to deference a pointer to a struct and then select one of its members.
- `[i]` is used to index an array, producing a loc.
- `[]` is applied to a pointer to cast it into an array.

Each identifier's type defines the kind of objects which may be in any state. Each LSL value has a unique *sort*. There is a mapping from LCL abstract types to LSL sorts. Each abstract data types defined in LCL is based on an LSL sort. LCL specifications are written using types and values. The properties of these values are defined in LSL, using operators on the sorts on which those types are based. The mapping between LCL (or Larch/C++) ADTs and LSL sorts enhances the *reusability* of specifications and verifications.

### 3.4 Component Specification

The component construct is used as the vehicle for encapsulation and data hiding. Objects are defined in terms of components. A component is a user-defined type. A

7

(

t

k

t.

b

pi

th

ca.

ing

anc

spe

cons



---

```
component component_name_identifier
{
    inherit: component_name_identifier*
    private:
        members (data and methods)
    protected:
        members (data and methods)
    public:
        members (data and methods)
};
```

Figure 3.3. Basic Construct for Component Specifications

---

component description defines the characteristics of all objects declared to be in the same component. The component construct also provides the basic unit of reusability. It binds an underlying data type with a set of public, protected, and private methods (functions) that allow the underlying data type to be manipulated by passing messages to the object. The skeleton of a component specification is shown in Figure 3.4. The key word **inherit** indicates that the current component inherits the properties from the components of the following components. The specification of inheritance will be discussed shortly. The specifications in the **private** section define the behavior of private methods in this component. Similarly, **protected** and **public** sections specify the corresponding methods in this component.

The objects declared to be a part of a given component may communicate with its callers by returning a result, by accessing objects accessible to the caller, or by modifying such objects. The specification of each method in an object can be comprehended and used without reference to the specifications of other methods. An example of specifying several methods in the component **intSet** is shown in Figure 3.4, which consists of a function prototype followed by a body specified in terms of pre- and

---

```

component intSet based on S from Set (Int for E)
{
public:
    insert()
        ensures new(return)  $\wedge$  return' = {}
    insert(int e)
        modifies (this)
        requires this' = add(this,e)

    delete(int e)
        modifies (this)
        ensures this' = rem(this,e)
}

```

Figure 3.4. Component interfaces for intSet.

---

postconditions. The **requires** clause describes restrictions on the arguments, thereby defining how the caller may use the method. We interpret an omitted **requires** clause as equivalent to “**requires true**.” The **ensures** clause places constraints on its behavior when it is called properly. They relate two states, the state when the method is called, which we call *preCondition*, and the state when it terminates, which we call *postCondition*. A **requires** only refers to the values in the *preCondition*. An **ensures** clause may also refer to values in the *postCondition*. As in C++, *this* denotes the object at which a component operation is invoked. The reserved symbol **return** is used as an implicit result to denote the object returned as a result of executing an operation. An omitted **modifies** clause is equivalent to the assertion **modifies nothing**, meaning no objects are allowed to change in value. A **new(object\_list)** predicate that appears in a constructor’s postcondition asserts that fresh storage is allocated for each object listed in *object\_list*. The values specified in the members

of **private** segment are not allowed to be changed by the callers. It says that the method must not change the value of any objects visible to the callers in the **private** segment.

### 3.5 Method Specification

The expressions used in *method* of components are based on first-order predicate logic. Figure 3.5 gives the grammar of the specification of expressions in methods. In this grammar, symbols expressed in the Roman font represent terminals, italicized symbols represent terminals, bold-faced symbols denote keywords, the Kleene star (\*) denotes zero or more repetitions of the preceding unit, and parentheses ('()') indicate groupings. The symbol "::" separates an identifier from a description of the *value* denoted by the identifier, and the symbol ":" separates identifier declarations from a description of the *type* associated with the identifier. The operators obey the following decreasing precedence order: negation ( $\neg$ ), conjunction ( $\wedge$ ), disjunction ( $\vee$ ), implication ( $\Rightarrow$ ), and if and only if ( $\Leftrightarrow$ ). Primitive types, e.g. *Bool*, *Int*, and *Real*, are pre-defined in Larch traits and can be referenced by the users.

### 3.6 Subtypes and Subclass

The features that most distinguish object-oriented programming languages from those only supporting data abstraction are *message passing* and *inheritance*. In this dissertation, only inheritance specification will be addressed since specification for message passing is beyond the scope of this dissertation. A useful abstraction in an object-oriented programming language method is the use of supertypes as abstractions of their subtypes [46]. Because of the complex inheritance mechanism of C++, the subtyping relationship is abandoned in Larch/C++ [38]. Since it is an interesting issue

---

```

method method_name((variable: type_name)*)
    requires: expression
    modifies: expression
    ensures: expression
expression = true
    | false
    | ( expression )
    |  $\neg$  expression
    | expression  $\wedge$  expression
    | expression  $\vee$  expression
    | expression  $\Rightarrow$  expression
    | expression  $\Leftrightarrow$  expression
    | (  $\forall$  variable : type :: expression )
    | (  $\exists$  variable : type :: expression )
    | predicate_name [( term ( , term )*)]
    | term def expression

term = variable
    | function_name [( term ( , term )*) ]

```

Figure 3.5. Grammar for Component Methods.

---

when we attempt to define specifications of classes in an object-oriented programming language, This section presents a discussion on the topic of subtyping [38, 47].

A type, that is, an ADT, is a behavioral notion and may be implemented by many different classes. A class is a program module that implements an ADT. A subtype is also an ADT, each of whose objects behave in a way similar to some objects of its supertypes. A subclass is an implementation that is derived by inheritance from its superclass. In contrast, a subtype represents a behavioral relationship. When a subtype is derived from some supertype, the object's behavior with this subtype can be verified by the objects with its supertype instead of reverifying this subtype, whereas a subclass relationship is a purely implementation relationship. The *generality* relationship in our system [42, 44] is similar to the *supertype-subtype* relationship.

The C++ type system does not distinguish completely between subtypes and sub-

classes. Each class name is the name of a type for type checking purposes. The only way to declare *S* as a subtype of *T* is to declare class *S* as a public subclass of *T*. *S* and *T* may have an implementation relationship but have no subtype relationship. Conversely, if *S* and *T* have a subtype relationship then they must have a subclass relationship. Because the designers of Larch/C++ tried to make Larch/C++ match C++, there is no **type** keyword in the class specification of Larch/C++. Therefore, *S* and *T* have a subclass relationship in C++ but they may not have the subtype relationship at the specification level. For example, in the Borland C++ library, class *Set* is a subclass of class *HashTable* (See Figure 3.6) but we would not intuitively consider *HashTable* to be a supertype of *Set* at the specification level. In the C++ language, the subclass relationship is implementation-specific and cannot represent the true supertype-subtype relationship.

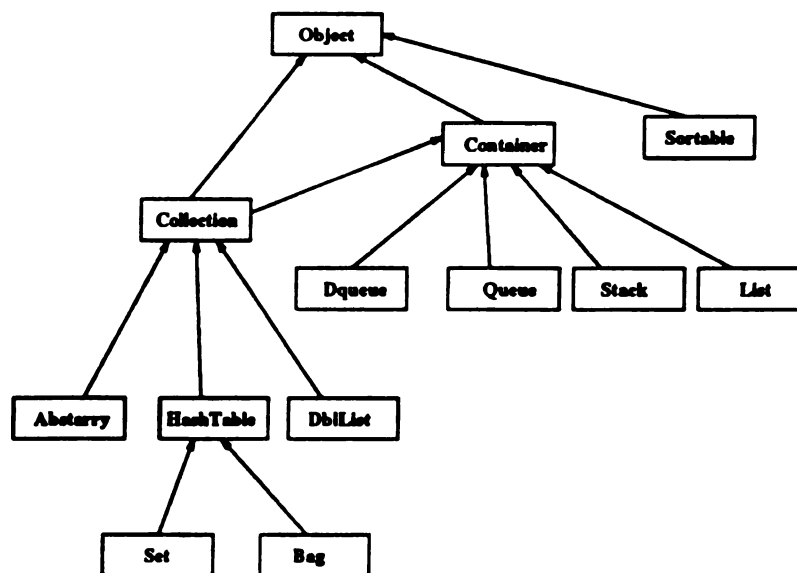


Figure 3.6. The Borland C++ Subclass Relationship.

---

clas

inte

def

by

the

The

her

Th

re

sea

cla

era

ove

spe

tio

mo

is

is

cor

If

the

po

of

sa

There are many ways to interpret what inheritance of specifications means for a class specification. The goal of the semantics of inheritance should be to define the interface of a class that implements the specification and to describe the effect of each defined message sent to objects of that type or its subtype. This goal can be satisfied by *overloading* the trait functions to interpret the inherited member functions, but this kind of overloading is only meaningful for subtypes that are public subclasses. The other goal is to construct a specification for a class that uses no specification inheritance, thus reducing the specification problem to an already specified component. This goal can be achieved by in-line expansion of the inherited specification, and then reinterpret the trait function in the specification by the used traits of the subclass.

Regarding the specifications of the semantics of multiple inheritance, current research has not completely solved this problem so far. We say *conflict* happens when a class inherits two methods with the same name from different classes. There are several ways to eliminate the conflicts. The user is forced to give an explicit specification, overriding all other specifications. Or the user may choose the most “representative” specification. Finally, we can use some scheme of composing the inherited specifications, for example, taking the disjunction of the preconditions, the intersection of the modifies clauses, or the conjunction of postconditions. It is not clear what approach is appropriate for reusing the inherited specifications. Currently, the third alternative is used as the semantics of multiple inheritance.

As for the definition of *subclass*, two classes A and B are used to explain the concept of subclass. Ideally, a subclass should be designed to implement a subtype. If *r* is an operation of A and B is a subclass of A, then the precondition of *r* in the specification of B may be no stronger than the precondition of *r* in A, and the postcondition of *r* in the specification of B must be no weaker than the postcondition of *r* in A. This rule ensures that the implementation of an operation in subclass B satisfies the specification of that operation in the superclass A. However, this case

de  
su  
ca  
re

3

Ch  
sif  
rev  
*B*  
des  
ma

All  
the  
har  
dev  
stru  
wit

De  
trai  
the  
with

I  
of th  
libra



does not hold in C++ because, in C++, an operation in subclass B may not exist in the superclass A. Therefore, the subtype relationship is changed to a new relationship called *generality* that is defined in next section in order to match the C++ subclass relationship.

### 3.7 Defining Generality

Chapter 4 will describe the construction of the lower-level hierarchy that serves to classify a set of software components according to the subsumption relationship between reusable components. In simple terms, component *A* is said to subsume component *B* if *A* is more *general* than *B*, denoted by  $A \sqsubseteq_{comp} B$ . A new resolution rule is described that increases the range of candidates, as compared to the number of exact matches, that can be retrieved using automated reasoning techniques.

We use LSL traits as the basic reusable units in specifying program components. All the terms (including predicates and functions) are assumed to be well defined in the Larch trait library. Therefore, reusing traits drawn from a generally accessible handbook will serve to standardize the notation. A Larch trait browser [48] is being developed to facilitate the user's search for suitable reusable components or the construction of a query specification. Figure 3.7 shows the relations among some traits with respect to a new relation  $\sqsubseteq_{term}$  which is defined as follows:

**Definition 3.1** *Let the relationships of assumption, inclusion or importation among traits be called the relationships of inheritance. Assume  $S$  and  $T$  are two traits in the Larch library:  $S \sqsubseteq_{term} T$  if  $S$  has the reflexive and transitive closure of inheritance with respect to  $T$ .*

In Figure 3.7, a trait *inherits* the properties of its parent traits. From the definition of the relation  $\sqsubseteq_{term}$ , a partial ordering is imposed upon the basic traits of the LSL library. This partial ordering can be applied to the subsumption test algorithm,

in t

bet

trai

orde

(

met

pone

$f$  is

post(

dition

condi

Chap

erality

An

classes

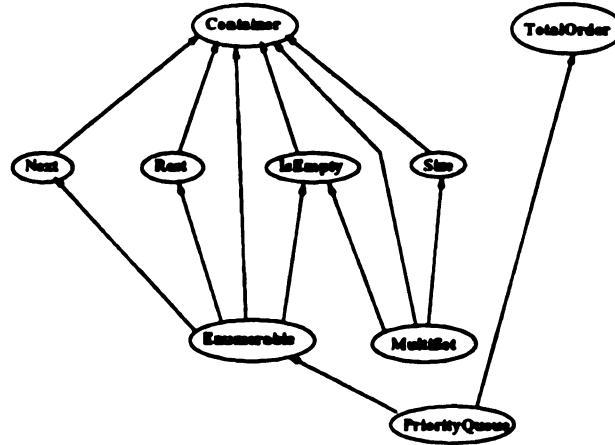


Figure 3.7. Relations Among the Traits

in the construction and retrieval processes, to determine the *generality* relationship between every pair of specification components that are built from the set of LSL traits. Figure 3.8 shows a modified subsumption test algorithm based on the partial ordering  $\sqsupseteq_{term}$ .

Component  $A$  is *more general* than component  $B$  ( $A \sqsupseteq_{comp} B$ ) if, for every method (operation)  $f$  in component  $A$ , there exists at least one method  $f'$  in component  $B$  such that  $f$  is more general than  $f'$ , denoted by  $f \sqsupseteq_{method} f'$ . Method  $f$  is said to be more general than another method  $f'$  if  $pre(f') \sqsupseteq_{clause} pre(f)$  and  $post(f) \sqsupseteq_{clause} post(f')$ , where  $pre(f)$  and  $post(f)$  represent the pre- and postconditions of  $f$ .  $pre(f') \sqsupseteq_{clause} pre(f)$  ( $post(f) \sqsupseteq_{clause} post(f')$ ) means that the precondition(postcondition) of  $f(f')$  *subsumes* the precondition(postcondition) of  $f'(f)$ . Chapter 4 gives an algorithm that builds the lower-level hierarchy based on the *generality* relationship between components ( $\sqsupseteq_{comp}$ ).

An ADT, is a behavioral notion and may be implemented by many different classes. A class is a program module that implements an ADT. A subtype is also

T  
di

an

A s

A s

some

to t

subc

the s

super

to th

---

The input is the set of expressions  $W$ . The output is either an empty set or the empty clause, represented as  $\square$ .

1. Let  $W = \{\neg L_1\sigma, \dots, \neg L_m\sigma\}$ .
2. Set  $k = 0$  and  $Z^0 = \{A\}$ , where  $A$  is the original set of clauses.
3. If  $Z^k$  contains  $\square$ , terminate;  $A$  subsumes  $B$ , denoted as  $A \sqsupseteq_{\text{clause}} B$ .  
 Otherwise  
 let  $Z^{k+1} = \{\text{Resolvents of } C_1 \text{ and } C_2 \text{ w.r.t. } \sqsupseteq_{\text{term}} \mid C_1 \in Z^k \text{ and } C_2 \in W\}$ .
4. If  $Z^{k+1}$  is empty, terminate;  $A$  does not subsume  $B$ . Otherwise, set  $k = k + 1$  and go to (3).

Figure 3.8. Subsumption Test Algorithm based on  $\sqsupseteq_{\text{term}}$ .

---

an ADT, each of whose objects behave in a way similar to objects of its supertypes. A subclass is an implementation that is derived by inheritance from its superclass. A subtype represents a behavioral relationship. When a subtype is derived from some supertype, the object's behavior with this subtype can be verified according to the objects with its supertype instead of reverifying this subtype. In contrast, a subclass relationship is a purely implementation relationship. In the C++ language, the subclass relationship is implementation-specific and cannot represent the true supertype-subtype relationship. The *generality* relationship in our system is similar to the *supertype-subtype* relationship.

C

C

C

This

orga

scrib

cons

and

ment

comp

4.1

The o

of reu

search

plicabi

fine-gr

to the

# CHAPTER 4

## Construction of Hierarchical Component Library

This chapter presents an approach, based on formal methods, to the classification and organization of reusable software components. From a set of reusable components described by formal specifications, a two-tiered hierarchy of software components is constructed. The formal specifications represent software that has been implemented and verified for correctness. The hierarchical organization of the software component specifications provides a means for storing, browsing, and retrieving reusable components that are amenable to automation.

### 4.1 Lower-Level Hierarchy

The objective of the construction process is to construct a hierarchical organization of reusable components that will provide a fast means for browsing, retrieving, and searching of software components exploiting the automated reasoning techniques applicable to formal specifications. The lower-level hierarchy provides a means for a fine-grained, precise determination of reuse, where logical reasoning can be applied to the specifications.

4.

Ch

rel

by

res

---

an

to

resc

---

the a

wher



### 4.1.1 Determining Generality

Chang and Lee's subsumption test algorithm [49] is used to decide the *subsumption* relationship between clauses, that is, whether clause  $A$  subsumes clause  $B$ , denoted by  $A \sqsupseteq_{\text{clause}} B$ . We exploit the traditional resolution strategy [50] to compute the resolvents of two clauses, say  $C_1$  and  $C_2$  (Figure 4.1). The full resolution rule involves

---


$$\begin{array}{l} C_1: L, K_1, \dots, K_n \\ C_2: \neg L, M_1, \dots, M_m \\ \hline \text{resolvent: } K_1, \dots, K_n, M_1, \dots, M_m. \end{array}$$

Figure 4.1. Resolution Rule

an instantiation of the formulas by a *substitution*  $\sigma$ , which is a mapping of variables to terms. Application of this substitution must result in the same atoms for both resolution literals [51]. In Figure 4.2, if  $\sigma$  is the most general substitution that makes

---


$$\begin{array}{l} \text{clause 1: } L, K_1, \dots, K_n \\ \text{clause 2: } \neg L', M_1, \dots, M_m \quad L\sigma = L'\sigma \\ \hline \text{resolvent: } K_1\sigma, \dots, K_n\sigma, M_1\sigma, \dots, M_m\sigma. \end{array}$$

Figure 4.2. Full Resolution Rule

the atoms  $L$  and  $L'$  equal, then it is the *most general unifier* for the two expressions, where the resolvent is obtained by instantiating the remaining variables of clauses 1

and 2

$L$  and

user

the re

$-L'$  c

cong

deriva

resolu

of two

(MST

compe

The M

metho

(Comp

4.1.2

Based

of com

and 2. Atom  $L$  is said to be *congruent* to atom  $L'$ , denoted by  $L \simeq L'$ , when both  $L$  and  $L'$  are in an equivalence class partition *eq\_class* that may be defined by the user or the system (see Section 4.2.1 for further details). Following the approach of the resolution rule, if a *congruity* relationship exists between  $L$  and  $L'$ , then  $L$  and  $\neg L'$  can be eliminated in order to obtain a *c-resolvent*, a resolvent with respect to the congruity relationship. As a result, a modified resolution rule given in Figure 4.3 is derived, where  $\sigma$  is a *substitution* that maps variables to terms. Using the modified

---


$$\begin{array}{l}
 C_1: L, K_1, \dots, K_n \\
 C_2: \neg L', M_1, \dots, M_m \quad L\sigma \simeq L'\sigma \\
 \hline
 \text{resolvent: } K_1\sigma, \dots, K_n\sigma, M_1\sigma, \dots, M_m\sigma.
 \end{array}$$

Figure 4.3. Modified Resolution Rule

---

resolution rule, the subsumption test algorithm [49] is modified to find the *c-resolvent* of two clauses  $C_1$  and  $C_2$  rather than their resolvent. The modified subsumption test (MST) algorithm can be applied to every pair of methods of the two components being compared in order to determine the *generality* relationship between two components. The MST algorithm between two sets of methods is shown in Figure 4.4, where *methods<sub>A</sub>* (*methods<sub>B</sub>*) is the set containing the methods of the component *Comp<sub>A</sub>* (*Comp<sub>B</sub>*). The cardinality of *methods<sub>A</sub>* (*methods<sub>B</sub>*) is  $m$  ( $n$ ).

#### 4.1.2 Building Lower-Level Hierarchy

Based on Algorithm 1, the *generality* relationship can be determined between any pair of components in order to build the lower-level hierarchy. The straightforward ap-

6  
1

Fig  
Co

pro

bet

How

orde

and

to co

impr

---

**Algorithm 1** *More\_General\_Component*

**Input:** Two sets  $methods_A = \{A_1, A_2, \dots, A_m\}$  and  $methods_B = \{B_1, B_2, \dots, B_n\}$ .

**Output:** The generality relationship between components  $Comp_A$  and  $Comp_B$ .

**Procedure:**

```

begin
  find  $\leftarrow$  true;
  while  $methods_A \neq \{\}$  and  $find = true$  do
    select some  $A_i \in methods_A$ ;
     $methods_A \leftarrow methods_A \setminus A_i$ ;
     $set_B \leftarrow methods_B$ ;
    find  $\leftarrow false$ ;
    while  $set_B \neq \{\}$  and  $find = false$  do
      select some  $B_j \in set_B$ ;
       $set_B \leftarrow set_B \setminus B_j$ ;
      if  $A_i \sqsupseteq_{method} B_j$ 
      then find  $\leftarrow true$ ;
    endwhile;
  endwhile;
  if find = false
  then return(" $\neg(Comp_A \sqsupseteq_{comp} Comp_B)$ ");
  else return(" $Comp_A \sqsupseteq_{comp} Comp_B$ ");
end.

```

Figure 4.4. Using MST to decide the *generality* relationship between components  $Comp_A$  and  $Comp_B$ .

---

proach is to construct the lower-level hierarchy by performing a pair-wise comparison between all components. The pair-wise comparison algorithm is shown in Figure 4.5. However, the transitivity property of the *generality* relationship can be exploited in order to reduce the execution time of building the lower-level hierarchy. If  $A \sqsupseteq B$  and  $B \sqsupseteq C$  then the relation  $A \sqsupseteq C$  is automatically established without having to compare components  $A$  and  $C$ . A few definitions are given before presenting the improved algorithm. For some *set of lattices* (SOL)  $\Psi$ , the set of top nodes in  $\Psi$

---

Alg

Inp

Out

Pro

e

R

---

---

**Algorithm 2** *Pairwise\_Comparison*

**Input:** *A set of components*  $SET = \{C_1, C_2, \dots, C_n\}$ .

**Output:** *A hierarchy of components based on the generality relationship.*

**Procedure:**

```

begin
  while  $SET \neq \{\}$ 
    select some component  $C_i \in SET$ ;
     $SET \leftarrow SET \setminus C_i$ ;
     $set \leftarrow SET$ ;
    while  $set \neq \{\}$ 
      select some  $C_j \in set$ ;
       $set \leftarrow set \setminus C_j$ ;
      if  $C_i \supseteq_{comp} C_j$ 
        /* More_General_Component algorithm will be used to
           compare  $C_i$  and  $C_j$  */
        then make  $C_i$  a parent of  $C_j$ 
      else if  $C_j \supseteq_{comp} C_i$ 
        then make  $C_j$  a parent of  $C_i$ 
      endif
    endwhile;
  endwhile;
end.
```

---

Figure 4.5. Building the lower-level hierarchy by pair-wise comparison.

---

is

a

no

as

Fo

th

de

3

3

Th

alle

the

amp

is sh

*Rec*

ted l

cont.

plyin

gener

proac

then

top n

dants

 $\alpha$  and

the ne

SOL. 1



is denoted by  $Top(\Psi)$  and the set of bottom nodes by  $Bottom(\Psi)$ . If node  $\alpha$  has no parent nodes in the SOL  $\Psi$ , then  $\alpha \in Top(\Psi)$ . Similarly, if  $\alpha$  has no children nodes in the SOL  $\Psi$ , then  $\alpha \in Bottom(\Psi)$ . The internal nodes in  $\Psi$  are defined as  $Internal(\Psi) = \Psi \setminus (Top(\Psi) \cup Bottom(\Psi))$ , where ' $\setminus$ ' represents set subtraction. For some node  $\alpha \in \Psi$ , the set of parent nodes of  $\alpha$  is denoted by  $parent(\alpha)$  and the set of children nodes by  $child(\alpha)$ . The set of the descendants of  $\alpha$ , denoted by  $descendant(\alpha)$ , is defined as follows:

$$\beta \in descendant(\alpha) \Leftrightarrow ((\beta \in child(\alpha)) \vee (\exists \gamma : \gamma \in child(\alpha) : \beta \in descendant(\gamma)))$$

The set of the ancestors of  $\alpha$ , denoted by  $ancestor(\alpha)$ , has a similar definition. A parallel algorithm to build the lower-level hierarchy based on recursive comparisons and the generality relationship is given in Figure 4.6. A pictorial representation of an example construction of the lower-level hierarchy by procedure *Recursive\_Comparison* is shown in Figure 4.7, where dashed lines represent the application of the procedure *Recursive\_Comparison*, solid lines represent the *generality* relationship, and the dotted lines encapsulate SOLs. Initially, the example contains eight SOLs and each SOL contains only one component. These eight SOLs are merged into one SOL after applying the two procedures *Compare* and *Merge*. *Compare*( $\Psi_i, \Psi_j$ ) determines the generality relationship between nodes in SOLs  $\Psi_i$  and  $\Psi_j$  by using a recursive approach. For example, if some node  $\alpha$  is more general than some top node  $\beta$  of  $\Psi$ , then it is not necessary to compare  $\alpha$  with the descendants of  $\beta$ . However, if some top node  $\beta$  is more general than  $\alpha$  then the comparison between  $\alpha$  and the descendants of  $\beta$  is required. The same reasoning can be applied to the comparison between  $\alpha$  and the bottom nodes of the SOL  $\Psi$ . The procedure *Merge*( $\Psi_i, \Psi_j$ ) “connects” the newly generated *generality* relationship between SOLs  $\Psi_i$  and  $\Psi_j$  to form a new SOL. *Recursive\_Comparison* can be implemented as a parallel algorithm since the

Al

Imp

Our

Pro

comp

and  $L$

Ap

illustr

node

a pare

in *des*

with *t*

in *desc*

recursiv

---

**Algorithm 3** *Recursive\_Comparison*

**Input:** A set  $\{\Psi_0, \Psi_1, \dots, \Psi_{n-1}\}$ , where  $\Psi_i$  represents a set of lattices and assume  $n = 2^m$ .

Initially,  $\Psi_i = \{C_i\}$  where  $C_i$  is a component.

**Output:**  $\Psi_0$  contains a hierarchy of components based on the generality relationship.

**Procedure:**

```

begin
  for  $i := 0$  to  $m-1$  do
     $d \leftarrow 2^i$ ;
    do all  $\Psi_k$  where  $0 \leq k \leq 2^m - 1$  /* Parallel execution of all iterations */
      if  $k \bmod 2^{i+1} = 0$ 
        then
          Compare( $\Psi_k, \Psi_{k+d}$ );
           $\Psi_k \leftarrow \text{Merge}(\Psi_k, \Psi_{k+d})$ ;
        endif;
      end_do_all;
    endfor;
  end.

```

---

Figure 4.6. Building lower-level hierarchy by recursive comparison.

---

comparisons between the SOLs are independent of each other. Only the nodes in *Top* and *Bottom* sets are compared in the procedure *Compare*.

Applying algorithm *Compare*( $SOL_A, SOL_B$ ) to two SOLs  $SOL_A$  and  $SOL_B$  is illustrated in Figure 4.8. For discussion purposes, attention is focused on the top node  $E$  in  $SOL_A$  and the bottom node  $F$  in  $SOL_B$ . If  $F \sqsupseteq E$ , then make node  $F$  a parent of node  $E$  since all nodes in  $ancestor(F) \cup \{F\}$  must subsume the nodes in  $descendant(E) \cup \{E\}$ . However, if  $E \sqsupseteq F$ , then node  $E$  needs to be compared with the nodes in  $ancestor(F)$  and node  $F$  needs to be compared with the nodes in  $descendant(E)$  in order to obtain complete *generality* relationships. Using the recursive method to build the lower-level hierarchy may reduce the computational

time c  
elimin

## 4.2

After a  
ters in  
compon  
compon

---

of the

C

includ

of clu

and t

gener

level

and  $X$

genera

cluster

#### 4.2.1

In this

similar

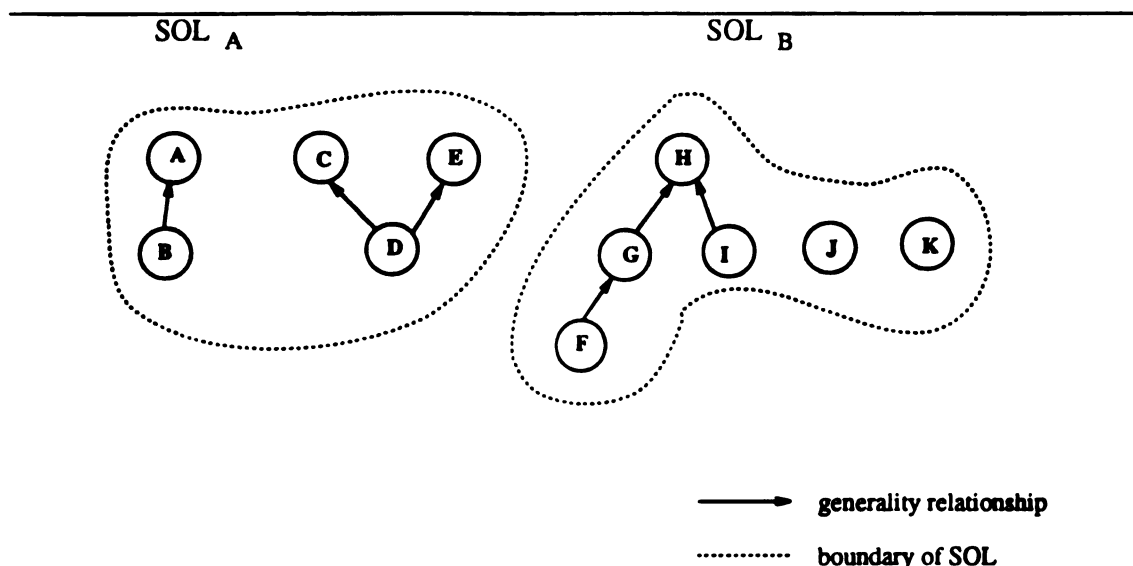


Figure 4.8. Example of comparing two SOLs.

---

of the lattices in *ASG*.

Classification by clustering techniques has been used in many areas of research, including information retrieval and image processing [53]. Typically, the objective of clustering is to form a set of clusters such that the intercluster similarity is low, and the intracluster similarity is high. Applying a clustering algorithm to the most general components of the lower-level hierarchy leads to the generation of the higher-level hierarchy of the component library. The similarity between two components  $X$  and  $X'$ , denoted by  $s(X, X')$ , is used as the basic criterion to determine clusters. In general, the criterion used to evaluate similarity determines the shape of the resultant clusters.

#### 4.2.1 Measure of Similarity between Components

In this section, a simple evaluation method for computing similarity is given. The similarity between a pair of components,  $X$  and  $Y$ , is denoted by  $s(X, Y)$ . Similarity

is s

In a

form

rega

cius

I

then

The o

where

The

order a

represe

inequal

associat

greater

The equ

The



is symmetric, thus for any two components,  $X$  and  $Y$ :

$$s(X, Y) = s(Y, X).$$

In addition, similarity  $s$  is said to be normalized if  $0 \leq s(X, Y) \leq 1$ . Each predicate formula in the library expressed in DNF represents a software component and is regarded as one of the input objects that are to be classified by the hierarchical clustering algorithm.

If  $s(X, Y)$  represents the similarity between two software components  $X$  and  $Y$ , then it is assumed that  $X$  and  $Y$  are of the following forms.

$$X = x_1 \vee x_2 \vee \dots \vee x_m \text{ and}$$

$$Y = y_1 \vee y_2 \vee \dots \vee y_n.$$

The disjuncts  $x_i$  and  $y_i$  are defined in terms of conjuncts, that is,

$$x_i = p_{i_1} \wedge p_{i_2} \wedge \dots \wedge p_{i_{u_i}}, \quad 1 \leq i \leq m \quad \text{and}$$

$$y_i = q_{i_1} \wedge q_{i_2} \wedge \dots \wedge q_{i_{v_i}}, \quad 1 \leq i \leq n,$$

where  $u_i$  and  $v_i$  are the number of conjuncts within disjunct  $x_i$  and  $y_i$ , respectively.

The disjuncts of each object are ordered from left to right in a nondecreasing order according to the number of conjuncts in each disjunct. Given that  $u_i$  and  $v_i$  represent the number of conjuncts for disjuncts  $x_i$  and  $y_i$ , respectively, the following inequalities are true:  $u_{i-1} \leq u_i$  and  $v_{i-1} \leq v_i$ , for all  $i$ . Moreover, each conjunct  $p_{i_k}$  is associated with an equivalence class **eq\_class**. For example, if  $p_{i_k} = \text{greater}(a, b)$  and *greater* is in the equivalence class for *comparison*, then **eq\_class**( $p_{i_k}$ ) = *comparison*.

The equivalence classes may be specified by the users or be system-defined.

The number of equivalence classes in a software component library is assumed

to be  
const  
X ha  
), R  
disj

Simil  
matr

From  
struct

where  
The si  
is calc  
equiva  
on syn

to be a known value, say  $T$ . Using the above definitions, a matrix  $X_{m \times (T+1)}$  is constructed for every component  $X$ . The matrix  $X_{m \times (T+1)}$  derived from component  $X$  has  $m$  rows and  $T+1$  columns.  $X(i, j)$  represents the entry in row  $i$  and column  $j$ . Row  $i$  represents the  $i^{th}$  disjunct of  $X$  as follows, where there are  $u_i$  conjuncts in disjunct  $x_i$ .

$$\begin{aligned} X(i, 0) &= u_i, \quad 0 \leq i \leq m - 1 \text{ and} \\ X(i, j) &= l, \quad x_i \text{ has } l \text{ terms in eq\_class } j. \end{aligned}$$

Similarly, for component  $Y$  containing  $v_i$  conjuncts in disjunct  $y_i$ , the corresponding matrix is defined by

$$\begin{aligned} Y(i, 0) &= v_i, \quad 0 \leq i \leq n - 1 \text{ and} \\ Y(i, j) &= l, \quad y_i \text{ has } l \text{ terms in eq\_class } j. \end{aligned}$$

From the derived matrices  $X_{m \times (T+1)}$  and  $Y_{n \times (T+1)}$ , the *similarity matrix*  $s'_{m \times n}$  is constructed. The following expression defines  $s'_{m \times n}$ .

$$\begin{aligned} \text{for all } i, j \quad &\text{if } X(i, 0) = Y(j, 0) \\ &\text{then } s'(i, j) = \frac{\sum_{t=1}^T 2 * \min(X(i, t), Y(j, t))}{\sum_{t=1}^T (X(i, t) + Y(j, t))} \\ &\text{else } s'(i, j) = 0 \end{aligned} \tag{4.1}$$

where  $s'(i, j)$  is the similarity of the  $i^{th}$  disjunct of  $X$  and the  $j^{th}$  disjunct of  $Y$ . The similarity between two conjunctive expressions from two software components is calculated according to the minimum number of common occurrences of a given equivalence class. Since the results from the clustering process are purely based on syntactic similarities, only the disjuncts with the same number of conjuncts are

S  
t  
s

H  
is

E  
w

C

to

cl

co

re

to

be

the

	0	1	2	3	4	5
1	2	1	1	0	0	0
2	3	0	1	2	0	0
3	3	0	0	1	1	1

(a) Matrix for X

	0	1	2	3	4	5
1	1	0	0	1	0	0
2	2	0	1	1	0	0
3	3	0	0	2	0	1
4	3	0	1	0	0	2

(b) Matrix for Y

Figure 4.9. Matrices for components X and Y.

selected for comparison. The semantic similarities are used in the construction of the lower-level hierarchy. Assume  $N$  is the number of nonzero entries in  $s'_{m \times n}$ . The similarity between software components  $X$  and  $Y$  is calculated as follows:

$$s(X, Y) = \frac{\sum_{i=1}^m \sum_{j=1}^n s'(i, j)}{N}. \quad (4.2)$$

Here,  $s(X, Y)$  is a normalized similarity since  $0 \leq s(X, Y) \leq 1$ . The following example is presented for clarification purposes.

**Example 4.1** Suppose the similarity of two components  $X$  and  $Y$  is to be computed, where both specifications are in DNF. Let  $X = (C_1 \wedge C_2) \vee (C_2 \wedge C_3 \wedge C_3) \vee (C_3 \wedge C_4 \wedge C_5)$  and  $Y = (C_3) \vee (C_2 \wedge C_3) \vee (C_3 \wedge C_3 \wedge C_5) \vee (C_2 \wedge C_5 \wedge C_5)$ , where  $C_i$  refers to the term that corresponds to the  $i^{\text{th}}$  equivalence class. There are 5 equivalence classes in this case, so  $T = 5$ .  $X$  has 3 disjuncts and  $Y$  has 4 disjuncts. The corresponding matrices for  $X$  and  $Y$  are shown in Figure 4.9a and 4.9b, where the vertical axis represents the disjuncts in each component and the horizontal axis refers to the equivalence classes. From Formula (4.1), the similarity matrix  $s'(X, Y)$  can be computed yielding results shown in Figure 4.9c, where the vertical axis represents the disjuncts in component  $X$  and the horizontal axis refers to the disjuncts in the  $Y$

co

ob

w

A

Th

l

an

fre

un

4.

Imp

ues

.X

{G

and

---

	1	2	3	4
1	0	2/4	0	0
2	0	0	4/6	2/6
3	0	0	4/6	3/6

Figure 4.10. Similarity matrix  $s'(X, Y)$

---

*component. From Formula (4.2), the similarity  $s(X, Y) = \frac{2/4+4/6+2/6+4/6+2/6}{5} = \frac{1}{2}$  is obtained. This value is used as input to the clustering algorithm when determining which software components should be merged into one cluster.*

A similarity value automatically determines the distance between two components. That is, the distance between two components  $X$  and  $Y$  may be defined as  $d(X, Y) = 1 - s(X, Y)$ . The distance between two components within a cluster should be small and distance between two clusters should be large. However, a distance (computed from similarity) does not necessarily satisfy the triangle inequality, and therefore is unable to form a metric space [14].

### 4.2.2 Hierarchical Clustering

Input to a clustering algorithm is a set of components and the similarity values between each pair of components. A finite set of components is denoted by  $X = \{x_1, x_2, \dots, x_n\}$ . Output from the clustering algorithm is a partition  $\Gamma = \{G_1, G_2, \dots, G_N\}$ , where  $G_k, k = 1, \dots, N$  is a subset of  $X$  such that

$$G_1 \cup G_2 \cup \dots \cup G_N = X, \quad \forall l, k, l \neq k, G_l \cap G_k = \emptyset, \quad (4.3)$$

and  $G_1, G_2, \dots, G_N$  are the clusters of  $\Gamma$ .

The relationship between the partition of clusters generated from the intermediate stages of refinement, denoted by  $\Gamma^i$ ,  $i = 1, \dots, K$ , is expressed as follows:

$$\Gamma^i = \{G_1^i, \dots, G_{N_i}^i\}, \quad \Gamma^j = \{G_1^j, \dots, G_{N_j}^j\}, \quad i = 1, \dots, K, \quad i < j < K + 1, \quad (4.4)$$

where for all  $l$ ,  $N_l \geq N$ , and  $N$  is the final number of partitions.  $\Gamma^j$  is a refinement of  $\Gamma^i$ ,  $i < j$ , that is, for any member subset  $G_k^i \in \Gamma^i$ , there exists  $G_l^j \in \Gamma^j$  such that  $G_k^i \subseteq G_l^j$ . Such groups formed by intermediate partitions yield a hierarchy of clusters. A method for generating such a hierarchy is termed *hierarchical clustering* [52].

In general, hierarchical clustering algorithms are divided into two categories: *divisive* algorithms and *agglomerative* algorithms. A divisive algorithm starts with the set  $X$  and divides it into a partition  $\Gamma^K = \{G_1^K, \dots, G_{N_K}^K\}$ , then each cluster  $G_i^K$  is subdivided to form a finer partition  $\Gamma^{K-1}$ , and so on. An agglomerative algorithm initially regards each component as a single cluster:  $\Gamma^1 = \{\{x_1\}, \{x_2\}, \dots, \{x_n\}\}$ . The clusters are merged into a coarser partition  $\Gamma^2$ , and the merging process continues until the trivial partition  $\Gamma^K = \{X\}$  is obtained. Thus an agglomerative clustering algorithm generates a sequence of partitions  $\Gamma^1 \rightarrow \Gamma^2 \rightarrow \dots \rightarrow \Gamma^K$  that is ordered from a finer partition to a coarser one. This algorithm can be stopped at any partition  $\Gamma^l$ ,  $1 \leq l \leq K$ , if the maximum value of computed similarities is below a specified threshold or if the number of clusters generated for a partition is equal to a user-specified or system-defined value.

In most agglomerative algorithms, only one pair of clusters is merged at a time. Hence if  $\Gamma^i = \{G_1^i, \dots, G_{N_i}^i\}$  and  $\Gamma^{i+1} = \{G_1^{i+1}, \dots, G_{N_{i+1}}^{i+1}\}$ , then  $N_{i+1} = N_i - 1$ . That is,  $N_i = n - i + 1$ ,  $i = 1, \dots, n$  and  $\Gamma^1 = \{\{x_1\}, \{x_2\}, \dots, \{x_n\}\}$ ,  $\Gamma^n = \{X\}$ . Figure 4.11 gives a pictorial representation of the refinement process. Similarity between clusters is used as the criterion for the selection of a pair of clusters in  $\Gamma^i$  that are to be merged. A pair of clusters  $(G_p, G_q)$  is selected to be merged if it has the



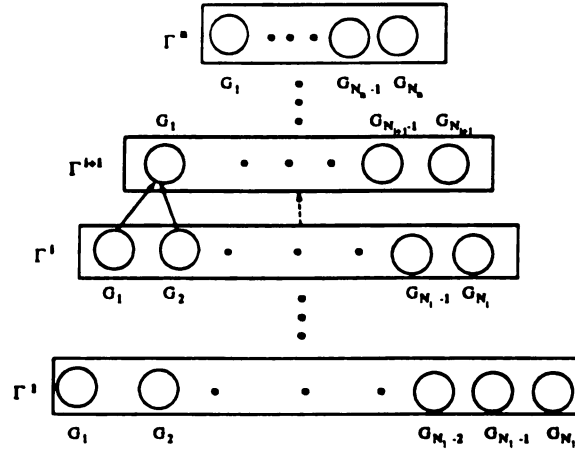


Figure 4.11. Refinement of partitions in an agglomerative clustering algorithm

maximum value of similarity among all pairs of clusters. Let the current partition be  $\Gamma = \{G_1, \dots, G_N\}$ . The similarity value between two clusters is the maximum value of all similarities calculated between disjuncts from the respective components. Formally, the *sim* relationship is expressed as

$$\text{sim}(G_p, G_q) = \max_{G_p, G_q \in \Gamma_{n-N}} \left( \max_{X \in G_p, Y \in G_q} s(X, Y) \right).$$

An agglomerative procedure is given in Figure 4.12. The similarities between the new cluster and other clusters are computed as follows: if  $G_p$  and  $G_q$  are merged into a new cluster  $G_r$ , then

$$\forall i, \text{sim}(G_r, G_i) = \min(\text{sim}(G_p, G_i), \text{sim}(G_q, G_i)).$$

1  
:  
l  
w  
o  
a  
tr  
a  
b  
ar  
of

---

**Algorithm 4** *Agglomerative Clustering*

**Input:** *A set of disjoint lattices.*

**Output:** *A unified cluster.*

**Procedure:**

1. *Let each root of a tree or the top element of a lattice of the ASG be an initial cluster consisting of the single element.*
2. *Find the pair of clusters that has the maximum value of similarity among all pairs of clusters.*
3. *Merge the pair of clusters found in step 2 into a new cluster.*
4. *If there is only one cluster remaining, then stop. Otherwise, update similarity values between clusters; go back to step 2.*

Figure 4.12. Agglomerative Hierarchical Clustering Algorithm

---

### 4.2.3 Hierarchical Clustering Algorithm

The hierarchical clustering algorithm used is similar to Kruskal's algorithm for finding a minimal spanning tree [54], which always chooses an edge with the least weight in the construction of the spanning tree. In this case, weights are replaced by similarity values for software components and the maximal weight rather than the least weight is sought. After applying the algorithm, a tree-like hierarchical clustering is obtained. Figure 4.13 contains the detailed description of the hierarchical clustering algorithm where  $X$  is the set of predicate components,  $s(X_i, X_j)$  is the similarity between components  $X_i$  and  $X_j$ , and  $sim(G_k, G_l)$  is the similarity between clusters  $G_k$  and  $G_l$ . The algorithm begins by creating a cluster for each software component to be classified, that is, the most general components found in the lower-level hierarchy, and the first partition contains all of the initial clusters. Next, a pairwise calculation of similarity between the clusters is made. Based on the similarity values, two clus-

ters yielding the greatest value are selected to be merged. After the two clusters are merged, the similarity values between clusters is updated, thus defining the partition for the next iteration of the clustering algorithm. The user may specify an upper bound on the number of iterations (refinements) or stop the clustering algorithm while viewing the clustering process. This flexibility allows the user to incorporate background experience in order to determine when further refinements will fail to yield substantial changes between partitions. The final hierarchically organized library could be of the form given in Figure 4.14, where filled nodes, termed *real nodes*, represent software components and unfilled nodes are newly generated nodes created by the hierarchical clustering algorithm, called *meta-nodes*. A meta-node acts as a container for the software components from it which it was derived. Dashed lines represent relationships formed by the MST algorithm and the solid lines are formed by the hierarchical clustering algorithm representing similarity relationships.

## 4.3 Implementation

In order to facilitate the user's involvement in determining reuse based on formal specifications, the construction of the software component hierarchy has been implemented in the framework of a graphical browser. This section discusses the implementation framework and describes an example construction of a hierarchy from a set of software component specifications.

### 4.3.1 Browsing Hierarchy

A prototype system for constructing the hierarchical library has been implemented in the Quintus ProWindows language \*, a dialect of Prolog that supports the object-oriented organization of graphical elements. There are several advantages to using

---

\*A product of Quintus Computer Systems, Inc.

---

**Algorithm 5** *Hierarchical Agglomerative Algorithm*

**Input:** The set  $X = \{x_1, x_2, \dots, x_n\}$  and the similarities  $s(x_i, x_j)$ ,  $1 \leq i, j \leq n$ .

**Output:** one or more clusters.

**Procedure:**

**begin**

$N = n$ ;

**for**  $i = 1, \dots, N$  **do**

$G_i = \{x_i\}$

**endfor**;

$\Gamma_1 = \{G_1, G_2, \dots, G_N\}$ ;

$Limit = 1$ ;

**for**  $1 \leq i, j \leq N, i \neq j$  **do**

$sim(G_i, G_j) = s(x_i, x_j)$

**endfor**; */\* Initialization \*/*

*/\* If there is more than one cluster then iterate, otherwise stop. \*/*

**while**  $(N > Limit)$  **do**

$N = N - 1$ ;

*/\* Select the pair of clusters to be merged \*/*

      find a pair of clusters  $G_p$  and  $G_q$  such that

$$\begin{aligned} sim(G_p, G_q) &= \max_{G_i, G_j \in \Gamma_{n-N}, i \neq j} sim(G_i, G_j) \\ &= \max_{G_i, G_j \in \Gamma_{n-N}, i \neq j} \max_{x \in G_i, y \in G_j} s(x, y); \end{aligned}$$

$G_r = G_p \cup G_q$ ;

$\Gamma_{n-N+1} = (\Gamma_{n-N} - \{G_p, G_q\}) \cup \{G_r\}$

*/\* Update the similarity values \*/*

**for all**  $G_i \in \Gamma_{n-N+1}, G_i \neq G_r$ , **do**

$$calculate\ sim(G_r, G_i) = \max_{x \in G_r, y \in G_i} s(x, y)$$

**endfor**;

$Limit = query\_user\_for\_number\_of\_clusters$ ;

*/\* Query user for a limit on the number of generated clusters \*/*

**endwhile**;

**return**  $\Gamma_{n-N+1}$

**end.**

Figure 4.13. Hierarchical Clustering Algorithm

---

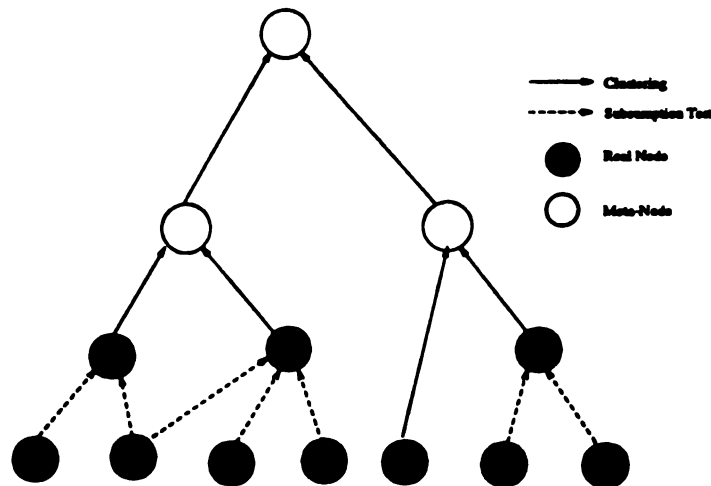


Figure 4.14. Two-tiered hierarchy formed by the subsumption test and the clustering algorithms

Prolog, and specifically ProWindows, as the implementation language. First, Prolog's declarative properties facilitate the handling of first-order predicate logic specifications and the application of automated reasoning capabilities. Second, Prolog's procedural properties facilitate the implementation of backtracking algorithms such as those used in the search for reusable components. Third, the support for graphics in ProWindows using Prolog predicates facilitates a homogeneous implementation of a system for handling the construction, searching, and browsing of a software component library. Specifically, ProWindows provides direct support for high-level features of the user interface such as dialog boxes, scrolling menus of sorted items, and term editing windows.

### 4.3.2 Implementation of the Construction of Hierarchy

Figures 4.15 and 4.16 show screen dumps of sample applications of the subsumption test algorithm and the hierarchical clustering algorithm, respectively, to a

set of software components. Originally, there were fifteen components in the library specified in the format described in Chapter 3. A larger example will be described in Chapter 8. Selecting the *Clustering* option shown in the pop-up menu in Figure 4.15 will execute the subsumption test algorithm, which causes the four subwindows shown in Figure 4.15 to be displayed, where each window contains one cluster. In the example, component *DoubleQueue* is a *child* of

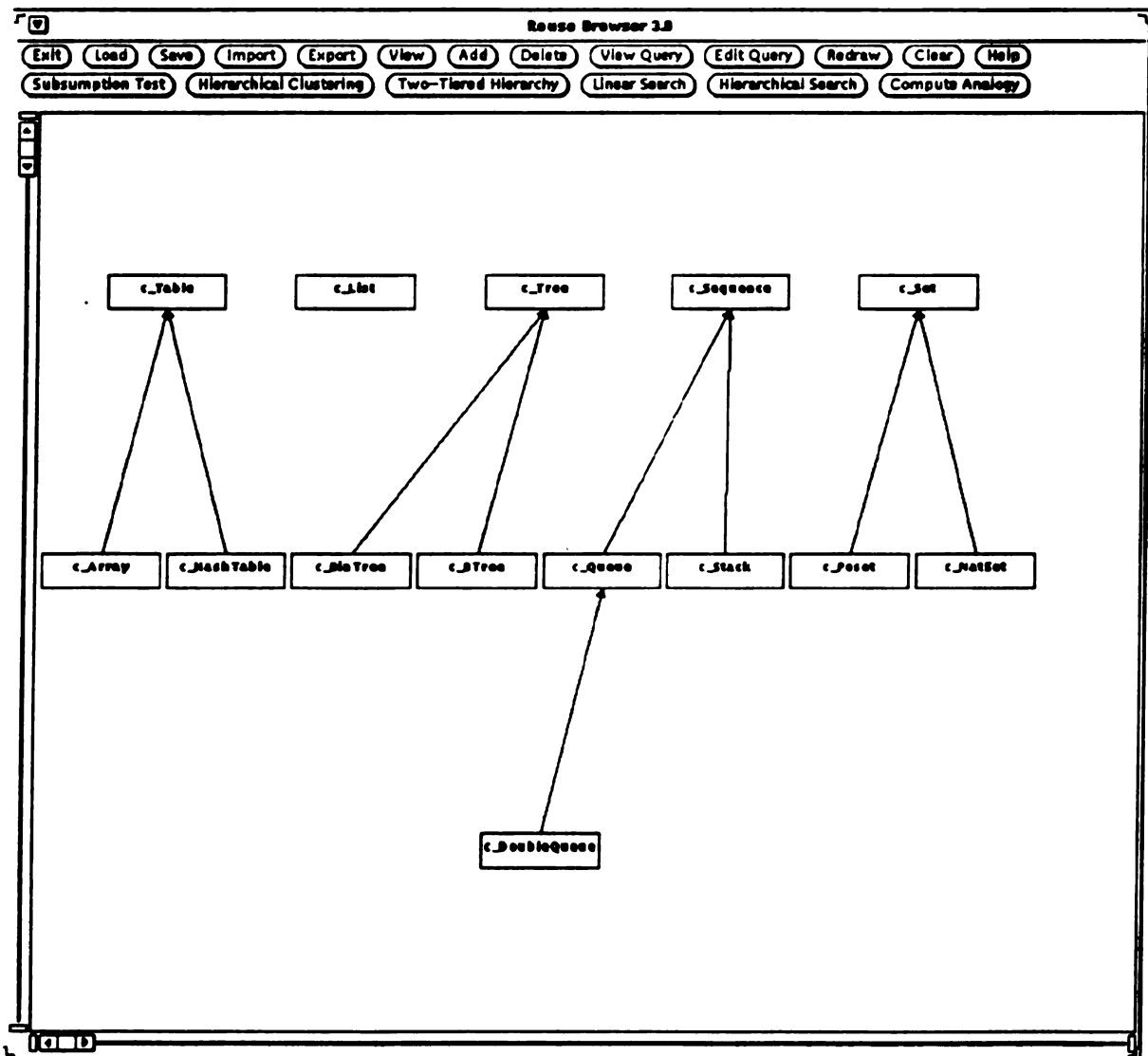


Figure 4.15. Sample application of subsumption test algorithm





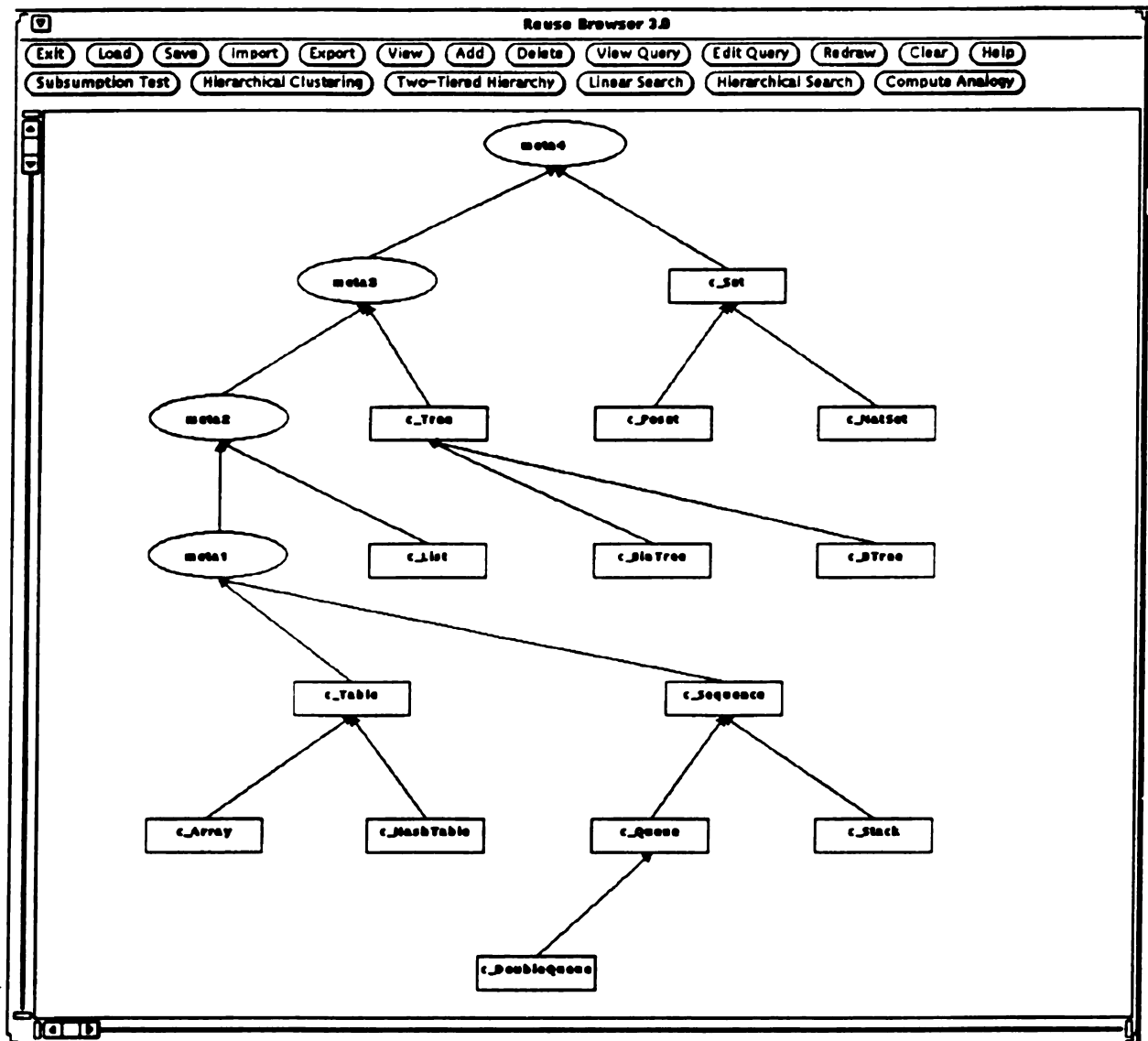


Figure 4.16. Sample application of clustering algorithm

the component *Queue*, where *DoubleQueue* and *Queue* represent the classes (or types) *DoubleQueue* and *Queue*, respectively. Figure 4.17 gives the specification of *Queue*, which contains five methods: *Queue*, *~Queue*, *Queue::appendAtEnd*, and *Queue::clear*. The first two methods *Queue* and *~Queue* represent the constructor and destructor of the component *Queue*, respectively. *Queue::appendAtEnd* appends an element to *Queue*. *Queue::clear* removes all objects that *Queue* con-

tains. Similarly, the specification of *DoubleQueue* is given in Figure 4.18, which

---

```

component Queue
abstract type Queue;
public
  void Queue(Queue *queue) {
    modifies *queue;
    ensures (*queue)' = NULL_Queue;
  }
  void ~Queue(Queue *queue) {
    modifies *queue;
    ensures trashed(*queue);
  }
  void Queue::appendAtEnd(Queue *queue, Object& element) {
    requires ¬full(queue);
    modifies *queue;
    ensures last(queue') == element ∧
      butLast(queue') == butLast(queue) ∧
  }
  void Queue::clear(Queue *queue) {
    modifies *queue;
    ensures isEmpty(*queue)
  }

```

Figure 4.17. Specification of Component *Queue*.

---

contains six methods: *DoubleQueue*, *~DoubleQueue*, *DoubleQueue::appendAtEnd*, *DoubleQueue::appendAtHead* and *DoubleQueue::clear*. For data types *DoubleQueue* and *Queue*, (*Queue::appendAtEnd*  $\sqsubseteq_{\text{method}}$  *DoubleQueue::appendAtEnd*), and (*Queue::clear*  $\sqsubseteq_{\text{method}}$  *DoubleQueue::clear*). Therefore, *Queue* is more general than *DoubleQueue* as a whole, and *Queue* is displayed as a parent of *DoubleQueue* in the browser graphically.

After determining the *generality* relationships between components, the clustering

re  
tw  
en  
co  
in  
b  
A  
4

algorithm is applied to the four groups, displayed in Figure 4.15, in order to create a connected hierarchy in the library. At this stage, only the roots of the trees from the output of the MST algorithm are chosen as input to the clustering algorithm. In the example, components *Table* and *Sequence* are merged and the hierarchical clustering algorithm yields a meta-node *meta1* for them. Figure 4.16 shows the results of the clustering algorithm, where the components *meta1*, *meta2*, and *meta3* are represented by newly-created meta-nodes. Finally, a two-tiered hierarchy of software components is constructed, shown in Figure 4.16, that can be used in the retrieval process. Upon completion of the construction process, the user may choose to rename meta-nodes (e.g. *meta1* and *meta2*) to more descriptive names.

## 4.4 Summary

A classification scheme of software components expressed in Larch specifications has been presented in this chapter. We have also described algorithms for implementing this scheme. The classification algorithms, implemented in Prolog, are able to construct a two-tiered hierarchical library from formal specifications. Thus, the hierarchy can help users store, browse, and search existing reusable components. The two-tiered hierarchy of reusable components provides a framework for the following reuse processes: retrieval and modification.

---

```

component DoubleQueue
abstract type DoubleQueue;
public
  void DoubleQueue(DoubleQueue *deque) {
    modifies *deque;
    ensures (*deque)' = NULL_DoubleQueue;
  }
  void ~DoubleQueue(DoubleQueue *deque) {
    modifies *deque;
    ensures trashed(*deque);
  }
  void DoubleQueue::appendAtEnd(DoubleQueue *deque, Object& element) {
    requires ¬full(deque);
    modifies *deque;
    ensures last(deque') == element ∧
      butLast(deque') == butLast(deque) ∧
  }
  void DoubleQueue::appendAtHead(DoubleQueue *deque, Object& element) {
    requires ¬full(deque);
    modifies *deque;
    ensures first(deque') == element ∧
      butFirst(deque') == butFirst(deque) ∧
  }
  void DoubleQueue::clear(DoubleQueue *deque) {
    modifies *deque;
    ensures isEmpty(*deque)
  }

```

Figure 4.18. Specification of Component *DoubleQueue*.

---

# CHAPTER 5

## Search and Retrieval of Reusable Components

In the previous chapter, we proposed a model to classify the reusable components by forming a two-tiered hierarchy of software components that are specified in terms of the Larch specification language. Based upon this framework, we address the issue of identification and retrieval of reusable components in this chapter. We present a hashing scheme to provide an initial indication of components for a query. A retrieval algorithm is described in detail in Section 5.2. The output of the retrieval algorithm can be used as input to the modification process that examines and modifies the extracted components for reusability.

### 5.1 Hashing Scheme for Software Components

In this section, the method used to retrieve software components is described. A hashing function  $HF: Component \rightarrow R$  is defined, which maps a component to a unique real number used in the retrieval process to reduce the search space of reusable components.

The following auxiliary definitions are necessary for the hashing function defini-

tion. For a given component  $C$  that is specified by the predicate  $P$ , the *closure* of  $P$  has the following definition.

**Definition 5.1** For some predicate  $P$ , let  $\text{closure}(P)$  denote the closure of  $P$ , a set of predicates and symbols. Let  $\text{pred\_func}$  be a predicate name or a function name.  $\text{closure}(P)$  is defined as follows:

- $P \in \text{closure}(P)$ .
- if  $Q = \text{pred\_func}(arg_1, arg_2, \dots, arg_n)$  and  $Q \in \text{closure}(P)$ , then  $arg_1 \in \text{closure}(P)$ ,  $arg_2 \in \text{closure}(P)$ , ..., and  $arg_n \in \text{closure}(P)$ .

The *level* of a term  $T$  with respect to predicate  $P$  is defined as follows:

**Definition 5.2** Let  $\text{pred\_func}$  be a predicate name or a function name. The level of some term  $T$  upon predicate  $P$  is denoted as  $\text{level}(P, T)$  and

- If  $T \notin \text{closure}(P)$  then  $\text{level}(P, T) = 0$ .
- If  $T \in \text{closure}(P)$  and  $T$  is a variable or a constant, then  $\text{level}(P, T) = 1$ .
- If  $T \in \text{closure}(P)$  and  $T$  is a predicate or a function and  $T = \text{pred\_func}(arg_1, arg_2, \dots, arg_n)$ , then  $\text{level}(P, T) = \max\{\text{level}(P, arg_1), \text{level}(P, arg_2), \dots, \text{level}(P, arg_n)\} + 1$ .

For a predicate  $P$ , the set of variables  $X_i$ ,  $i > 0$ , contains the variables in  $\text{closure}(P)$  with the  $i^{\text{th}}$  type, where it is assumed that all possible data types can be referenced by an index and the indices range from 1 to the total number of data types defined. Let  $X = \bigcup_{i>0} X_i$  and  $\Sigma = \text{closure}(P) \setminus X$  (set subtraction). The *arity* of the element  $f$  of  $\Sigma$  is denoted by  $\alpha(f)$ . If  $a, t \in \text{closure}(P)$  and  $\text{level}(t) \geq \text{level}(a)$ , then  $n(a, t)$  is used to represent the number of occurrences of  $a$  in  $t$ .

A *partial ordering* on a set of terms is an irreflexive transitive binary relation. If  $\succ$  is a partial ordering, then the symbol  $\succeq$  is used to denote  $a \succ b$  or  $a = b$ . Now a partial ordering  $\succ_\Sigma$  is chosen for  $\Sigma$ , and a non-negative number, or weight,  $w(f)$  is assigned to each element  $f \in \Sigma$ , and a positive weight  $w_i$  is assigned to each variable of type  $i$  subject to the following conditions:

- If  $\alpha(f) = 0$  then  $w(f) \succeq w_i$  for all  $i$ .
- If  $\alpha(f) \geq 1$  and  $w(f) = 0$  then  $f \succ_{\Sigma} g$  for all elements  $g$ .

For each such choice of weights and partial ordering, the Knuth Bendix Ordering (KBO) [55] is defined on the terms, which is denoted by  $\succ$  ( $\succ_{\Sigma}, w$ ) or just  $\succ$ . For this ordering, a weight is assigned to each term as follows. For any term  $t$ , let

$$w(t) = \sum_i \sum_{x \in X_i} n(x, t) * w_i + \sum_{g \in \Sigma} n(g, t) * w(g).$$

Thus, for example, the weight of a variable of type  $i$  is  $w_i$ , and the weight of a term is the sum of the weights of all symbols appearing in it. Suppose some component  $C$  is represented by a first-order predicate  $P$ . In order to incorporate the structural information of  $P$  into the computation of the weight of  $P$ , the levels of the terms (Definition 5.2) in  $P$  are determined, where the value of the level is used to redefine the weight of the predicate  $P$ . For this case, a *term* can be a variable, a constant, a predicate, or a function. Now the weight of the predicate  $P$  is defined as follows:

$$w(P) = \sum_i \sum_{x \in X_i} n(x, P) * w_i + \sum_{j=1}^{level(P,P)} \sum_{level(P,g)=j \wedge g \in \Sigma} n(g, P) * w(g) * \alpha_j$$

where  $\alpha_j$  denotes the input parameter, which emphasizes how the *level* of the terms in  $P$  influences the *weight* of  $P$ , that is, part of the structural information of  $P$  is incorporated in the calculation of  $w(P)$ .

Since the software components in the framework are expressed in first-order logic, which contains a relatively small set of different types of symbols, it is simple to determine their weights. However, it is possible that two terms may have the same weight, thereby allowing two components to be mapped to the same value. In order to resolve this conflict, each component  $C$  is labeled with an *index*,  $ind(C) = (W, r)$  containing a pair of real numbers, where  $W$  denotes the weight of the component and



$r$  denotes the rank of this component with respect to the other components having the same weight. The rank can be deterministically decided by the Knuth-Bendix Ordering described in Figure 5.1.

**Definition 5.3** *For some component  $C$  with weight  $W$ , where no other component has the same weight then  $\text{ind}(C) = (W, 1)$ . If there exist  $k$  other components with weight  $W$  and all of them are greater than  $C$  according to the KBO, then  $\text{ind}(C) = (W, k + 1)$ .*

Similarly, the partial ordering for an *index* is defined as follows:

**Definition 5.4** *For any two components  $C$  and  $C'$ ,  $\text{ind}(C) = (w, i)$  and  $\text{ind}(C') = (w', i')$ . We say  $\text{ind}(C) \succ \text{ind}(C')$  if either  $w \succ w'$  or  $w = w'$  and  $i < i'$ .*

Finally, the hashing function  $HF$  is defined as follows.

**Definition 5.5** *For some component  $C$  with index  $\text{ind}(C) = (w, i)$ , the hashing function is defined as  $HF(C) = w * \phi + i$ , where  $\phi$  is a parameter that can be set to a value that is either pre-defined by the system or set by the user.*

## 5.2 Retrieval Algorithm

The previous sections explained how software components are indexed, that is, hashed to real numbers by the hashing function  $HF$ . We also need to be able to retrieve the components that satisfy the requests if one exists and to help the user select the most similar components via the two-tiered hierarchy. In order to retrieve the candidates for a given query, we devise the algorithm in Figure 5.2. The hashing scheme in steps 1 and 2 has been illustrated in Section 5.1. The following explains how the last two stages are supported in our approach. According to the two-tiered hierarchy of software components described in Chapter 4, there are two kinds of

---

If  $s$  and  $t$  are two terms, then  $s \succ t$  if and only if  $n(x, s) \succeq n(x, t)$  for all variables  $x \in X$  and either

1.  $w(s) \succ w(t)$ , or
2.  $w(s) = w(t)$   
 and either
  - (a)  $s = f^n(x)$  and  $t = x$  for some  $n \geq 1$ , or
  - (b)  $s = f(s_1, \dots, s_{\alpha(f)})$  and  $t = f(t_1, \dots, t_{\alpha(f)})$  and  $f \succ g$ ,  
 ( $\succ$  represents well-founded ordering) or
  - (c)  $s = f(s_1, \dots, s_{\alpha(f)})$  and  $t = f(t_1, \dots, t_{\alpha(f)})$  and  $s_1 = t_1, \dots, s_{k-1} = t_{k-1}$   
 and  $s_k \succ t_k$  for some  $k$  with  $1 \leq k \leq \alpha(f)$ .

Figure 5.1. Knuth-Bendix Ordering

---

nodes in the hierarchy. The first kind of nodes, called *real* nodes, are the original components, located in the nodes of the lower-level hierarchy. The second kind of nodes, called *meta* nodes, are those generated by the clustering algorithm and located in the higher-level hierarchy. In step 2 of the algorithm in Figure 5.2, the components retrieved are always located in the real nodes since only real nodes are mapped to real numbers by the hashing function  $HF$ .

The subsumption relationship is the major structuring mechanism of the lower-level hierarchy. For the real nodes, a parent node subsumes any one of its child nodes, that is a parent node is more general than its child node. This property can be exploited in the retrieval process. Whenever a component is retrieved, the components located in its child nodes can also be suggested as reusable candidates by traversing the component hierarchy. However, since we emphasize the retrieval of large scale software components, we need to automate the retrieval process as much as possible. Figure 5.3 describes the retrieval algorithm that applies the subsumption test given a query specification and a set of candidate components. The output of

---

**Algorithm 6** *Retrieval Process*

1. The query specification  $Q$  is hashed to some number  $HF(Q)$ .
2. Given a fixed value  $\delta$ , a set of components  $\Omega$  is returned to the user and

$$(\forall C : C \in \Omega : |HF(C) - HF(Q)| \leq \delta),$$

*that is, the difference between the hashing function value of each retrieved component and the hashing function value for the current problem is less than some value  $\delta$ .*

3. Find the nearest common ancestor of all nodes in  $\Omega$ , that is, the retrieved nodes.
4. Depending on the type of ancestor node, one of the following steps should be followed:

**Case 1:** *If the nearest common ancestor is a meta-node, then compute the similarity values for all specifications  $C'_i$  contained in the node with the current specification and  $\Omega' = \{C'_i\}$ .*

**Case 2:** *If the nearest common ancestor is a real node  $R$ , then  $\Omega' = \{R\}$ .*

5. For all  $C \in \Omega'$ , apply subsumption test algorithm to  $C$  and  $Q$  to find the components that subsume or are subsumed by  $Q$ . If there are such components, then return them to the user otherwise go to step 6.
6. Using background information, the user may choose to browse through the hierarchy of the software components in order to retrieve the components most suitable for the new specification.

Figure 5.2. The Process of Retrieving Software Components.

---

---

**Algorithm 7** *Retrieval by Subsumption Test*

**Input:** A set of candidate components  $\Omega$  and Query Specification  $Q$ .

**Output:** Three sets of components  $\Delta_{general}$ ,  $\Delta_{specific}$ , and  $\Delta_{reference}$ .

**Procedure:**

**begin**

$\Delta_{general} = \Delta_{specific} = \Delta_{reference} \leftarrow \emptyset$ .

**for each real node**  $C$  **let**  $mark(C) \leftarrow false$ ; **endfor**;

**while**  $\Omega \neq \{\}$

**for each component**  $C \in \Omega$

$mark(C) \leftarrow true$ ;

$\Omega \leftarrow \Omega \setminus \{C\}$ ;

**switch**  $(Q, C)$ :

**Case**  $(Q \supseteq C \text{ and } C \supseteq Q)$ :  $return(C)$ ;

**Case**  $(Q \not\supseteq C \text{ and } C \not\supseteq Q)$ :

**for each component**  $C' \in parent(C)$

**if**  $C'$  **is a real node and**  $mark(C') = false$

**then**  $\Omega \leftarrow \Omega \cup \{C'\}$

**else if**  $C'$  **is a meta-node then**  $\Delta_{reference} \leftarrow \Delta_{reference} \cup \{C'\}$

**endfor**;

**Case**  $(Q \supseteq C)$ :

$\Delta_{specific} \leftarrow \Delta_{specific} \cup \{C\}$ ;

**for each component**  $C' \in parent(C)$

**if**  $mark(C') = false$  **then**  $\Omega \leftarrow \Omega \cup \{C'\}$

**endfor**;

**Case**  $(C \supseteq Q)$ :

$\Delta_{general} \leftarrow \Delta_{general} \cup \{C\}$ ;

**for each component**  $C' \in children(C)$

**if**  $mark(C') = false$  **then**  $\Omega \leftarrow \Omega \cup \{C'\}$

**endfor**;

**endswitch**;

**endfor**;

**endwhile**;

**return**  $\Delta_{general}$ ,  $\Delta_{specific}$ , and  $\Delta_{reference}$ ;

**end.**

---

Figure 5.3. The Algorithm for Retrieval by Subsumption Test.

this algorithm is a set of updated candidate components that are more similar to the query requirements. For two predicates  $A$  and  $B$ ,  $A \sqsupseteq B$  means that  $A$  subsumes  $B$ . For some node  $A$ , in our two-tiered hierarchy,  $child(A)$  denotes the set of its child nodes (components) and  $parent(A)$  denotes the set of the parent nodes of  $A$ . The input to this algorithm is the query specification  $Q$  and a set of candidate components  $\Omega$  derived from hashing scheme  $HC$ . The output from this algorithm contains three sets of components:  $\Delta_{general}$ ,  $\Delta_{specific}$ , and  $\Delta_{reference}$ .  $\Delta_{general}$  includes those components that subsume  $Q$  and  $\Delta_{specific}$  includes those components that are subsumed by  $Q$ .  $\Delta_{reference}$  holds a group of meta nodes produced from the retrieval algorithm. Therefore, in our approach the user may obtain two sets of components,  $\Delta_{general}$  and  $\Delta_{specific}$ , that have a subsumption relationship to the user's query specification. Otherwise, the user may communicate interactively with the system's browser to conduct the browsing when  $\Delta_{general}$  and  $\Delta_{specific}$  return no reusable components. Typically, the user starts from the nodes contained in the set  $\Delta_{reference}$  that contains a group of meta nodes as references. Single meta node represents at least two real nodes. Consequently, all meta nodes of  $\Delta_{reference}$  can be replaced by a number of real nodes ranked by their similarity with the user's query specification. Ranking is achieved by comparing analogous functionalities shared by the new and reusable components. The computation of similarity is described in further detail in Section 4.2.1.

### 5.3 Implementation of Retrieval Process

Figure 5.4 contains an example application of the *construction* and *retrieval* processes. A group of reusable components are classified to form a two-tiered hierarchy. The lower-level hierarchy generated by the subsumption test algorithm represents the *generality* relationships among the components, where the parent component is *more general* than the child component. The elliptic nodes are nodes newly created by the

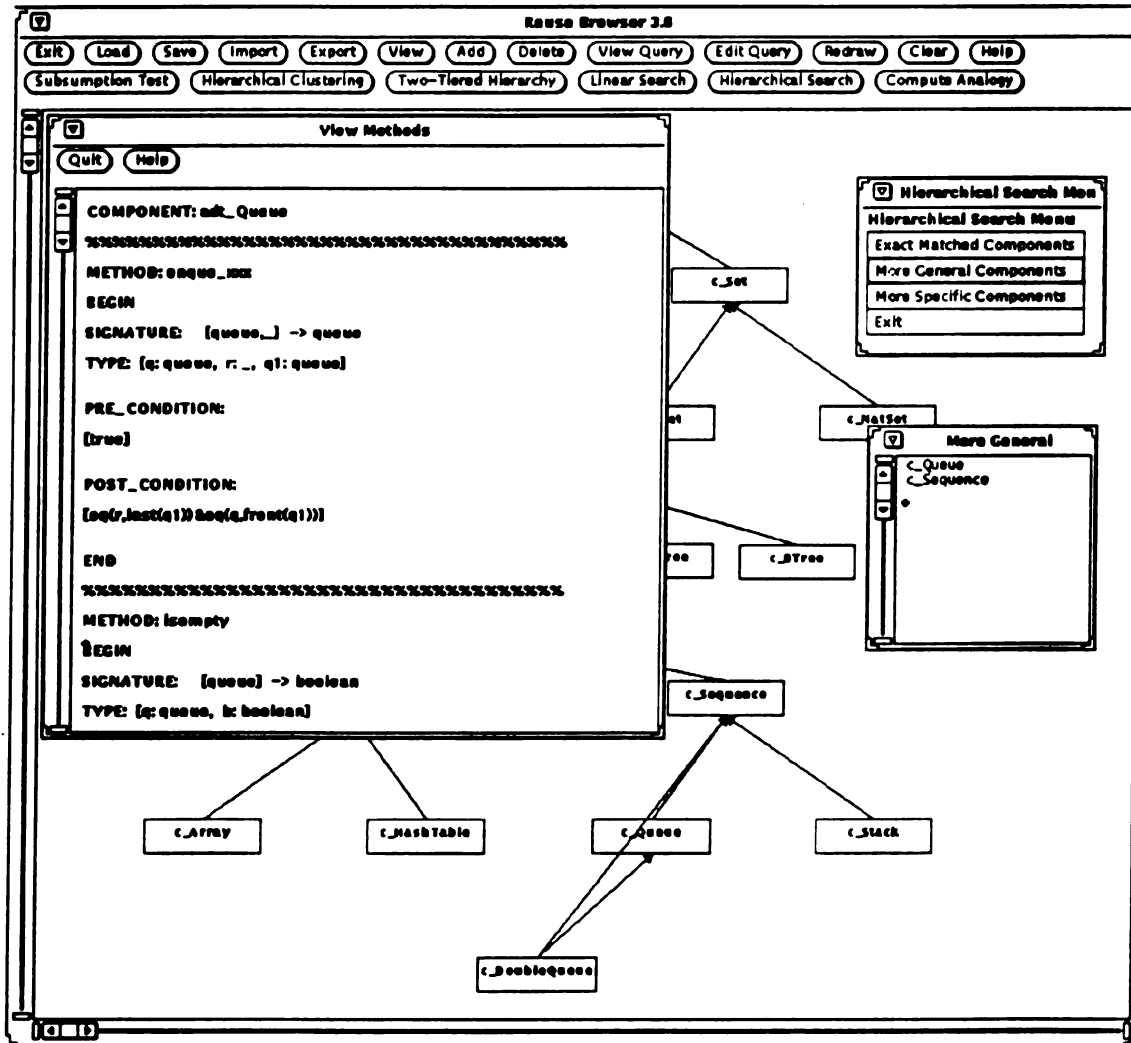


Figure 5.4. Sample application of construction and retrieval processes

clustering algorithm to generate the higher-level hierarchy. On the right hand side, a retrieval option is selected from the main menu to execute the retrieval process. In this example, the user wants to find those components that are more general than the query specification *adt\_queue*, which is partially displayed on the left window of Figure 5.4. Below the search menu, a window displays the result obtained by the retrieval process. The result, in this case, is a set of components that are more general than the query specification *adt\_queue*.

# CHAPTER 6

## Modification of Reusable Components Based on Generality

### 6.1 Introduction

In an attempt to perform software reuse, it is often the case that a given reuse candidate closely matches the needs of a query specification. A challenging problem is to determine what modifications are necessary to an existing specification in order to make it satisfy the query specification. Accordingly, the software implementation of the existing specification must have corresponding changes. Modification differs from transformation in that correctness with respect to the original specification is not necessarily preserved. What we want is for the resultant program to be correct with respect to the transformed specification. Correctness-preserving transformations and specification-changing modifications are thus complementary. The objective of program modification is to obtain a program that satisfies its input-output specification along with the specification for a new program. Comparison of two specifications may suggest a transformation of the given program that will bring it in line with the new specification. Even if the transformed program does not fulfill the goal, it may serve as a basis for constructing the desired program.



One of our reuse processes, modification, will be described in this chapter. This process is based on our previous work in the construction and retrieval processes. We will first present the specification of a C++ class in terms of an existing specification language Larch. Section 6.2 illustrates how to define the expanded *weakest precondition* semantics of the C++ language. Section 6.4 discusses the process of modifying a component that is more general than a query specification. Section 6.6 summarizes this chapter.

## 6.2 Predicate Transformer $wp$

There are several approaches to describe the semantics of a given program. One approach is to regard the final state in which statement  $S$  terminates as a function of the initial state in which  $S$  has been started [56, 57]. For a given postcondition  $R$  and a statement  $S$ , the *weakest precondition*  $wp(S, R)$  describes the set of states in which statement  $S$  can begin execution and terminate with  $R$  true and the *weakest liberal precondition*  $wlp(S, R)$  is the weakest precondition under which  $S$  is guaranteed to establish the postcondition  $R$  if the computation terminates. Generally,  $wlp(S, R)$  is concerned with the partial correctness of  $S$  and  $wp(S, R)$  is concerned with the total correctness of  $S$ . Since  $wp$  is more useful in program design, in this chapter, we only use  $wp$  for the purpose of transforming predicates.

The following properties are from Gries' book [56] except that the "Law of the Excluded Miracle" is omitted, since in this law  $wp(S, false)$  holds precisely in those initial states for which no computation under control of  $S$  exists. The "Law" erected a considerable barrier for the conception of those initial states in which starting  $S$  would be simply "inappropriate" or "impossible".

**Law of Monotonicity:**

$$\text{if } Q \rightarrow R \text{ then } wp(S, Q) \rightarrow wp(S, R)$$

For two predicates  $Q$  and  $R$ , if the relation  $Q \rightarrow R$  holds then the relation  $wp(S, Q) \rightarrow wp(S, R)$  also holds.

**Distributivity of Conjunction:**

$$wp(S, Q) \wedge wp(S, R) = wp(S, Q \wedge R)$$

Given a conjunction of two  $wp$ s for one statement  $S$  with the two postconditions  $Q$  and  $R$ , the conjunctive predicate can be converted into the  $wp$  of  $S$  with respect to  $Q \wedge R$ .

**Distributivity of Disjunction:**

$$wp(S, Q) \vee wp(S, R) \rightarrow wp(S, Q \vee R)$$

A disjunction of two  $wp$ 's of a statement  $S$  with respect to different postcondition,  $Q$  and  $R$ , implies the  $wp$  of  $S$  with respect to the disjunction of the two postconditions.

The syntax of a programming language defines the set of all programs that are representable in it. In this chapter, the semantics of a programming language has to define the semantics of each representable program, that is, it has to define the predicate transformers  $wp$  and  $wlp$  of  $S$ . The definition of the semantics of a programming language can be simplified to a definition of the functions  $wp$  and  $wlp$ , which are functions from a representable program to predicate transformers. The domain, the program, being recursively defined by the grammar,  $wlp$  and  $wp$  need to be defined recursively over the grammar. Similarly, only the  $wp$  is defined in this chapter and its semantics in Gries' pseudo-language and C (C++) is summarized here.

### 6.2.1 Simple C++ Statements

The following is a list of the allowed simple statements in Borland C++ [58].

1. Expression statements: *assignment* and *function call*.
2. Alternative statements: *if-else*, *switch*, *case*, and *default*.
3. Iterative statements: *while*, *do while*, and *for*.
4. Others: *break*, *continue*, *goto*, and *return*.

A *compound statement* consists of a series of statements. A null statement performs no operation and is the same as the *skip* statement. We will describe the *wp* semantics of Gries' primitive programming structures (statements), i.e. *abort*, *skip*, *assignment*, *alternative*, *iterative*, and *compound* statements [56, 21]. For each statement, we attempt to define the *wp* semantics of corresponding C (C++) statements. The *wp* semantics of recursive functions is not described here because it is tedious and difficult to use [59].

### 6.2.2 The *abort* Statement

For a given postcondition  $R$ , the abort command has the *wp*:

$$wp(\text{abort}, R) \equiv \text{false}.$$

The operational interpretation of *abort* is that for all initial states its execution fails to terminate. One of the corresponding C (C++) statements of *abort* is an infinite loop of the form

```
x = 99;
while(x < 100){
    ...
    x--;
    ...
}
```

### 6.2.3 The *skip* Statement

The operational interpretation of *skip* is that its execution, which is guaranteed to terminate, leaves the values of all variables unchanged. Its  $wp(S, R)$  is an identity function as follows

$$wp(skip, R) \equiv R.$$

### 6.2.4 The *assignment* ( $y := E$ ) Statement

$$wp("y := E", R) \equiv (y \geq_{po} E) \Delta domain(E) \Delta R_E^y,$$

where the predicate  $(y \geq_{po} E)$  is *true* when  $y$  is greater than  $E$  in the partial ordering of symbols, and the predicate  $domain(E)$  is *true* when  $E$  is defined in the domain of the current state. If both of the predicates are *true*, then all the occurrences of  $y$  are replaced with  $E$  in  $R$ . The statement  $y := E$  is known as “the assignment statement”. Its operational interpretation is that its execution, which is guaranteed to terminate, leaves the values of all variables, except  $y$ , unchanged, whereas the final value of  $y$  equals the initial value of  $E$ . In order to apply  $wp$  to C++ programming language, the expression  $E$  can be generalized. For example, if  $S \equiv y++$  then  $S$  can be transformed into  $y := y + 1$ , hence  $E \equiv y + 1$ . However, generalization sacrifices the elegance of predicate transformers.

### 6.2.5 The *alternative* (*IF*) Statement

The alternative statement is traditionally written as a list of guarded commands surrounded by the parenthesis pair “if-fi”. A fat bar “ $\square$ ” separates the guarded commands in the list. A typical alternative statement is of the form:

$$\begin{array}{l}
\textit{if } B_1 \rightarrow S_1 \\
\Box B_2 \rightarrow S_2 \\
\vdots \\
\Box B_n \rightarrow S_n \\
\textit{fi.}
\end{array}$$

With the abbreviation  $BB$  given by

$$BB \equiv (\exists i :: B_i)$$

where ‘ $::$ ’ indicates the range of  $i$  is not needed for the discussion. The semantics of the alternative statement  $S$  satisfies

$$wp(S, R) \equiv BB \wedge (\forall i :: B_i \rightarrow wp(S_i, R)),$$

It is straightforward to convert/transform an alternative statement in C++ into the above general form. The *if-else* statement of the form

```

if (B1)
    S1;
else if (B2)
    S2;
else S3;

```

is converted to

$$\begin{array}{l}
\textit{if } B_1 \rightarrow S_1 \\
\Box (\neg B_1 \wedge B_2) \rightarrow S_2 \\
\Box (\neg B_1 \wedge \neg B_2) \rightarrow S_3 \\
\textit{fi.}
\end{array}$$

In C++, the test condition is considered *true* if it is nonzero and *false* otherwise. This means that even negative values are considered *true* but, in Gries’ primitive programming language (GPPG), the test condition must be a Boolean expression. For the purpose of brevity, we omit the information of type checking when we define the *wp* semantics. For the *switch* statement of the form

```

switch(Expr) {
  case B1:
    S1;
    break;
  case B2:
    S2;
    break;
  case B3:
    S3;
}

```

is converted to

```

if (Expr = B1) → S1;
□ (Expr = B2 → S2;
□ (Expr = B3) → S3;
fi.

```

This type of statement is designed to facilitate the handling of multivalued branching. The expression *Expr* in C++ must be reducible or convertible to type `int`. The constants B1, B2, and so on, must all be integers or must be unambiguously convertible to integers. When a *break* expression is encountered in processing a `case` or `default` statement, the switch block is exited. In GPPG, if any of the guarded command statements are executed successfully then the alternative structure is said to have executed properly. The *wp* semantics of *if-fi* in GPPG can not describe the `default` statement in C++. Let us consider another `switch` statement of the form

```

switch(Expr) {
  case B1:
    S1;
    break;
  case B2:
    S2;
  default:
    S3;
    break;
}

```

which is the same as

```

switch(Expr) {
  case B1:
    S1;
    break;
  case B2:
    S2;
  case DEF:  \\ if (Expr != B1 /\ Expr != B2)
    S3;      \\ then Expr = DEF
    break;
}

```

Its corresponding GPPG structure is

```

if (Expr = B1) → S1;
□ (Expr = B2) → S2;
□ (Expr = DEF) → S3;
fi

```

### 6.2.6 The *iterative (DO)* Statement

The iterative command is of the form

```

do B1 → S1
□ B2 → S2
:
□ Bn → Sn
od

```

The *wp* of the iterative command with respect to a postcondition *R* is implied by the predicate

$$wp(do\text{-}od, R) \leftarrow (\exists P :: P \wedge \neg BB \rightarrow R \wedge (P \wedge B_i \rightarrow wp(S_i, P)))$$

where *P* is an invariant of the *do-od* command and *BB* is the disjunction of the guards. The **while** statement is one of the major constructs for iteration in C++. The general format is

```

while(condition)
  statement;

```

which can be converted to the GPPG iteration statement of the form

*do* condition  $\rightarrow$  statement  
*od.*

The most common iteration construct is the **for** statement. Here is the general form:

**for**(init\_statement; condition; control\_statement)  
    loop\_statement;

The **for** statement can be converted to the iterative GPPG statement of the form

*do* first\_iteration  $\rightarrow$  init\_statement  
    □ condition  $\rightarrow$  loop\_statement; control\_statement  
*od*

where the predicate **first\_iteration** is true when the program is in the first iteration of the loop.

### 6.2.7 The *compound* ( $S_0; S_1$ ) Statement

The expression  $S_0; S_1$  indicates that the statement  $S_1$  is executed after  $S_0$ . The *wp* of  $S_0$  is dependent on the *wp* of  $S_1$  with respect to  $R$ . The *wp* semantics is

$$wp(S_0; S_1, R) \equiv wp(S_0, wp(S_1, R))$$

## 6.3 Modifying a More General Component

The modification process is part of our reuse system that applies formal methods to the reuse of existing software. In this section, the problem definition, algorithm, and example of program modification based on specification changes are presented.

After the retrieval process, if an exact-match component has been found then we return it directly to the users for reuse. However, it is unlikely that we can retrieve the components that are exactly the same as the input specification every time. A reuse



system supporting ones this approach will provide a minimal amount of assistance for reusing existing software components, since generated specification components are limited to at most combinations of existing software components. In order to overcome this weakness, a mechanism for modifying a retrieved component to satisfy the query specification is needed. In this section, we present the process of transforming some retrieved component that is *more general* than the query specification with respect to the  $\sqsubseteq_{comp}$  relationship (See Chapter 4 for details regarding this relationship).

The modification of software components can be accomplished by direct editing of the source code. However, editing source code by intuition may invalidate the correctness of the original components and force the developer or user to work at a low level of abstraction. The effort of verification and modification of source code components offsets a significant amount of the effort in component reuse.

Our approach is to modify the reusable component at the specification level instead of the code level. Once the necessary changes the old specification needed to satisfy the query specification have been determined, the information needed to modify the specification is applied to some program synthesizer (or programmers) to generate the required program. The degree of modification can be achieved through the use of automated reasoning techniques since our components are represented by first order logic. Using our approach reduces the drawbacks noted above for modification via direct source editing. Use of automated reasoning and a program synthesizer preserve the correctness of software components for all possible query specifications. Thus the user will be less concerned about the validation of a reused component. Also, given the behavior of what a software component does, it is possible to use and modify a software component without having to understand the low level implementation details in the component. Since fully automated modification is not possible now, semi-automatic modification is an alternative to expedite the whole process.

A *query specification* is a set of requirements that have a form similar to the

component specifications. We define that some existing specification *Old\_Spec* is *reusable* for the query specification *Query\_Spec* if they satisfy the following conditions:

- $Old\_Spec.pre \rightarrow Query\_Spec.pre$
- $Query\_Spec.post \rightarrow Old\_Spec.post$

In terms of logical reasoning, the above two conditions can be regarded as that the preconditions of *Query\_Spec* are a superset of *Old\_Spec* preconditions and the postconditions of *Query\_Spec* are a subset of *Old\_Spec* postconditions.

The problem of modifying an implemented module based on the specification changes is given in Figure 6.1. This diagram illustrates how the *generality* relationship can be incorporated into the modification process. The *refinement* relationship defines the relationship between the specification and implementation modules. Therefore, this diagram can be applied to any level of program abstraction only if the *generality* and *refinement* relationships hold. What we want to find is a mapping that transforms the old implementation module to a new implementation module that satisfies the query specification.

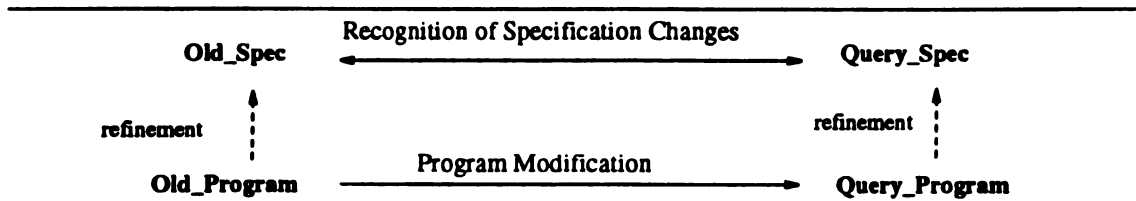


Figure 6.1. Modifying Implementation Based on Specification Changes.

---

## 6.4 Modification Process

Three sets of candidate components are output from the retrieval process for some query specification [42] and they are  $\Delta_{general}$ ,  $\Delta_{specific}$ , and  $\Delta_{reference}$ . The set  $\Delta_{general}$  ( $\Delta_{specific}$ ) contains the candidate components that are more *general* (*specific*) than the query specification. The set  $\Delta_{reference}$  is used as a reference set for browsing the component hierarchy so we regard it as a reusable set. Since the implementation of a more specific component satisfies the query specification, it can be reused directly. Only the *generality* relationship is used to facilitate the modification in Figure 6.1. More helpful relationships will be added to our modification process, for example, modifying the *analogous* components [60].

A process for modifying a more general component is presented in Figure 6.2. From the definition of the *generality* relationship, component  $A$  is more general than component  $B$ , i.e.,  $A \sqsupseteq_{comp} B$ , only if all methods of  $A$  are more general than the subset of the methods in  $B$ . Therefore, some methods in  $B$  may have no relationship with any method of  $A$ , even if component  $A$  is more general than component  $B$ . Hence, the modification process shown in Figure 6.2 can only give a partial implementation of a query specification. For the method  $M_{old}$  in an old component that is more general than some method in a query component (specification), we attempt to find the weakest precondition of its implementation with respect to the postcondition of the subsumed query method  $M_{query}$ . The method for finding the *wp* for a C++ language construct is illustrated in Section 6.2. Then construct a conjunctive expression from the found weakest precondition with the precondition of  $M_{old}$  that is a new precondition for the old implementation. Finally, input the new precondition, the old implementation and the new query postcondition to some program synthesizer, either a semi-automated program synthesis system or a programmer, in order to obtain a new implementation satisfying the query specification.

---

**Algorithm 8** *Modifying a More General Component*

**Input:** Two components  $SPEC_{old}$ ,  $SPEC_{query}$ , where the relationship  $(SPEC_{old} \sqsupseteq_{comp} SPEC_{query})$  holds.

**Output:** Partial implementation of  $SPEC_{query}$ .

```

begin
  for every method  $M_{old}$  of  $SPEC_{old}$ 
    for every method  $M_{query}$  of  $SPEC_{query}$ 
      if  $(M_{old} \sqsupseteq_{method} M_{query})$ 
        then
          Let  $PROG_{old}$  be the implementation of  $M_{old}$ .
          Find  $WP \equiv wp(PROG_{old}, M_{query}.post)$ .
          Let  $NEW.pre \leftarrow M_{old}.pre \wedge WP$ .
          Let  $PROG_{query} \leftarrow synthesizer(NEW.pre, PROG_{old}, M_{query}.post)$ .
          Assign  $PROG_{query}$  as the implementation of  $M_{query}$ .
        endif
      endfor;
    endfor;
  end.

```

Figure 6.2. The Process of Modifying More a more general method.

---

Program synthesis will not be discussed here since it involves some techniques that are beyond the scope of this report but one example will be shown shortly. In the modification process, the relationships among components and methods have already been established in the retrieval process so that the complexity of the modification process only arises from the calculation of weakest preconditions and the process of program synthesis. The former uses polynomial time in terms of the size of the query specification plus the length of the old implementation, i.e., source code. It is difficult to measure the complexity of program synthesis, however, we believe the complexity is highly reduced since the needed changes at specification level are determined before the source code is edited.

## 6.5 Modification Example

An example of the modification problem shown in Figure 6.1 is given in this section. An *old* component, the specification of `List` class is given in Figure 6.3. `Object` in the interface represents any kind of mathematical numbers, e.g., real number, characters, strings, and so on. This component defines five methods and all of them will be implemented in the `public` segment of a C++ class. The methods `List` and `~List` are the *constructors* and *destructors* of class `List`, respectively. By convention, an object of a class is initiated by the constructor before any other use. A client of an object should call the destructor when it knows that the object will never be referenced again. The clause *ensures trashed(\*s)* says that upon return from the destructor nothing can be assumed about the storage pointed to by `s` in the precondition state. A good implementation of a destructor will free storage that is no longer needed, although the specification does not require it. The method `List::insert` adds an `Object` element to the tail of a `List`. The method `List::detach` deletes an `Object` element from `List`. The method `List::isA` reports the user that this object belongs to `List` class. Lists that are not aliased to any object visible in the precondition. Thus the `Lists` that they return can be modified without affecting the values of other list. One way of implementing this is to allocate new storage. The operators  $\cup$  and  $\cap$  are assumed to be defined in LSL trait *Set*.

C++ currently has no standard classes or hierarchies other than those used with streams. C++ compiler vendors used the NIH classes [61] to test their own compiler. The *Borland* company decided to include a variation of the NIH classes with *Borland C++*, which they called the *Borland C++ container class library*. The library contains classes that are organized into three logical groups: simple classes, container classes, and iterator classes. Simple classes are classes that cannot be iterated upon such as class `String`. In contrast, the container classes can contain a whole series of other

---

```

component List
abstract type List;
public
    void List(List *list) {
        modifies *list;
        ensures (*list)' = NullList;
    }
    void ~List(List *list) {
        modifies *list;
        ensures trashed(*list);
    }
    void List::add(List *list, Object& e) {
        modifies *list;
        ensures element(list',tail(list')) == e ^
            length(list') == length(list) + 1;
    }
    bool List::detach(List *list, Object& e) {
        requires e ∈ (*list);
        modifies *list;
        ensures in(list',e) == FALSE;
    }
    string List::isA() {
        ensures result = "list class";
    }

```

Figure 6.3. The LCL Specification of List Class.

---

objects, which in turn can be containers themselves. Examples of containers are classes of **Collection**, **Stack**, **Bag**, and **Set**.

The container class library has several classes that are capable of storing multiple objects. In order to handle the objects inside these containers, we need a way to iterate through them. Borland C++ chose to do this with special iterator classes, which allow us to iterate through the objects in a container. The implementation of **List** class is given in Appendix E.1 (except for the destructor). **List** inherits some operations from the **Container** class.

A *query specification*, the specification of **Array** class, given in Figure 6.4. This component defines six methods: **Array**, **~Array**, **Array::addEnd**, **Array::addAt**, **Array::clear**, and **\*Array::isA**. The method **Array::addAt** has no corresponding method specified in the specification of **List** class. Similarly, the operations '∈' and 'delete' are assumed to be defined in the LSL traits library. Let us consider the two classes **List** and **Array**. From the definition of *generality*,  $\text{List} \sqsubseteq_{comp} \text{Array}$  since the following relationships hold:

```
List  $\sqsubseteq_{method}$  Array
~List  $\sqsubseteq_{method}$  ~Array
List::add  $\sqsubseteq_{method}$  Array::addEnd
List::detach  $\sqsubseteq_{method}$  Array::clear
List::isA  $\sqsubseteq_{method}$  Array::isA
```

For the purpose of conciseness, we only consider the relationship between two methods - **List::add** and **Array::addEnd**. Following the procedure of modification process for more general component, we first find the weakest precondition of the statements in the method **List::add** with respect to the postcondition of **Array::addEnd**. The statements in the method **List::add** are

---

```

component Array
abstract type Array;
public
    void Array(Array *array) {
        modifies *array;
        ensures (*array)' = NULL_Array;
    }
    void ~Array(Array *array) {
        modifies *array;
        ensures trashed(*array);
    }
    void Array::addEnd(Array *array, Object& element) {
        modifies *array;
        ensures array'(maxIndex(array')) == element ^
            size(array') == size(array) + 1 ^
            maxIndex(array') == maxIndex(array) + 1;
    }
    void Array::addAt(Array *array, Object& element, int atIndex) {
        modifies *array;
        ensures array'(atIndex) == element ^
            size(array') == size(array) + 1;
    }
    bool Array::clear(Array *array, int Index) {
        modifies *array;
        ensures array'(Index) == NULL;
    }
    string Array::isA() {
        ensures result = "array class";
    }

```

Figure 6.4. The LCL Specification of Array Class.

---



```

void List::add( Object& newElement )
{
    ListElement *newElement =
        new ListElement( &newElement );
    newElement->next = head;
    head = newElement;
    itemsInContainer++;
}

```

The corresponding GPPG structure of the body statements is

```

List_Prog:
{
    newElement->next := head;
    head := newElement;
    length(list) := length(list) + 1;
}

```

The postcondition of List::add in Figure 6.3 is

```

List_Post:
{
    element(list',head(list')) == newElement /\
        length(list') == length(list) + 1
}

```

The postcondition of Array::addEnd in Figure 6.4 is

```

Array_Post:
{
    element(array',maxIndex(array')) == newElement /\
    size(array') == size(array) + 1 /\
    maxIndex(array') == maxIndex(array) + 1
}

```

The relationship

$$(List :: add) \sqsubseteq_{method} (Array :: addEnd)$$

holds because the following relationships hold:

```

list \sqsubseteq_{term} array
head(list) \sqsubseteq_{term} maxIndex(array)
length \sqsubseteq_{term} size

```

The relationship  $(list \sqsubseteq_{term} array)$  refers to the *generality* relationship defined in the LSL level. The second and the third relationships hold since the corresponding

terms are defined in the same equivalence classes, for example, **length** and **size** are both in the class referring to the cardinality of the objects in some ADT. Replacing the terms in **List\_Prog** by the more specific terms, we have

```

Array_Prog_1:
{
    newElement->next := array[maxIndex(array)];
    array[maxIndex(array)] := newElement;
    size(array) := size(array) + 1;
}

```

The first statement is unexecutable since the assignment command is applied to two objects with different types, so we discard this statement momentarily and have the following candidate program for **Array::addEnd**:

```

Array_Prog_2:
{
    array[maxIndex(array)] := newElement;
    size(array) := size(array) + 1;
}

```

Applying predicate transformer *wp* to **Array\_Prog\_2** with respect to **Array\_Post**, we have

$$\begin{aligned}
 &wp(\text{Array\_Prog\_2}, \text{Array\_Post}) \\
 &= (\text{newElement} == \text{newElement}) \wedge \\
 &\quad (\text{size}(\text{array}) + 1 == \text{size}(\text{array}) + 1) \wedge \\
 &\quad (\text{maxIndex}(\text{array}') == \text{maxIndex}(\text{array}) + 1) \\
 &= \text{TRUE} \wedge \text{TRUE} \wedge (\text{maxIndex}(\text{array}') == \text{maxIndex}(\text{array}) + 1) \\
 &= (\text{maxIndex}(\text{array}') == \text{maxIndex}(\text{array}) + 1)
 \end{aligned}$$

Since the precondition of **List::add** is **TRUE**, the *NEW.pre* of the modified program will be

$$\text{maxIndex}(\text{array}') == \text{maxIndex}(\text{array}) + 1.$$

Based on the information of the above weakest precondition, we know that the value of *maxIndex* of some array needs to be incremented by one first before **Array\_Prog\_2**

executes to compute the correct result. Therefore, we have the following modified GPPG statements that satisfy *Array\_Post*:

```

Array_Prog_3:
{
    maxIndex(array) = maxIndex(array) + 1;
    array[maxIndex(array)] := newElement;
    size(array) := size(array) + 1;
}

```

And we convert *Array\_Progr\_3* into C++ program and have the implementation for *Array::addEnd*:

```

void Array::addEnd( Object& newElement )
{
    while( theArray[ whereToAdd ] != ZERO &&
           whereToAdd <= upperbound )
    {
        whereToAdd++;
    }
    if( whereToAdd > upperbound )
    {
        reallocate( whereToAdd - lowerbound + 1 );
    }
    theArray[ whereToAdd - lowerbound ] = &newElement;
    whereToAdd++;
    itemsInContainer++;
}

```

Other parts of the implementation of *Array* class are illustrated in Appendix E.2. Clearly, modifying the source code of a reusable class such as *List* is not fully automated yet. However, the modification process can be greatly facilitated given the information from the specification changes as described in the above example.

## 6.6 Summary

This chapter showed how C++ statements can be generalized to GPPG and defined the *wp* semantics for simple GPPG constructs. More definitions will be required for the *wp*

semantics like function and recursion in the future. We also described the modification process as applied to modify the source code whose specification is more general than the query specification. A general issue of modifying a reusable program component from the specification changes is the incompatibility of *subclass* definition in the application language and the *generality* definition in the specification level. The subclass definitions in C++ are so implementation specific such that they cannot be used in the specification level. In the next chapter, we will generalize our modification process to other types of specification revision problems, i.e., using *analogy* to determine program modification [60].

# CHAPTER 7

## Modification of Reusable Components Based on Analogy

This chapter presents an approach, based on formal methods, to the modification of reusable software components. From the framework of a two-tiered hierarchy of reusable software components, the candidate components that are *analogous* to the query specification are retrieved from the hierarchy. An analogous retrieved component is compared to the query specification to determine what changes need to be applied to the corresponding program component in order to make it satisfy the query specification.

### 7.1 Introduction

Analogy is often used in everyday situations to assist in decision making. The main objective of analogy is to make use of past experiences to solve similar problems. Analogical reasoning has long been recognized as an important tool to overcome the search complexity of finding solutions to novel problems or inducing generalized knowledge from experience. Analogy presents a basic and challenging question: when are two specifications (problem representations), for some purpose, alike? [62]. This

section outlines the scope of analogical reasoning techniques that we are using to enhance our software reuse system [43, 42].

We regard the modification process as a problem solving process, where Figure 7.1 contains a framework for modifying components based on the analogy of formal specifications and is similar to the problem of modifying an implemented program presented in Chapter 6. The problems in this process are the specifications that represent reusable software components and the solutions become the executable implementations of the corresponding specifications. The objective of software development in our context is to “solve” specification *Query\_Spec* which is referred to as a *query specification*. *Old\_Spec* is referred to as a *candidate specification* whose implementation *Old\_Program* is known. A *match* that is found between the two specifications represents the similarity of the two specifications. Based on this match, the analogy is used to guide the modification process, i.e., the software developer. Figure 7.1 shows how the *match* can be incorporated into the modification process. The *refinement* relationship defines the relationship between the specification and implementation modules.

If a “good” analogical match between the candidate and query specifications can be found, then the effort required to develop the appropriate implementation, *Query\_Program*, will be reduced. We call this approach to modify existing program based on an analogical match between two specifications, the *Analogical Reuse Modification Process* (ARMP). The problem of finding promising candidate specifications for some query specification from large knowledge bases of known and implemented specifications is referred to as the *base filtering problem* [63]. In Chapter 5, a retrieval scheme based on the similarities among reusable components finds a set of candidate specifications that are *similar* to the query specification. Our retrieval process augments the ARMP model in the form of a pre-processing phase/stage.

The idea that programs should be constructed by a series of transformations has

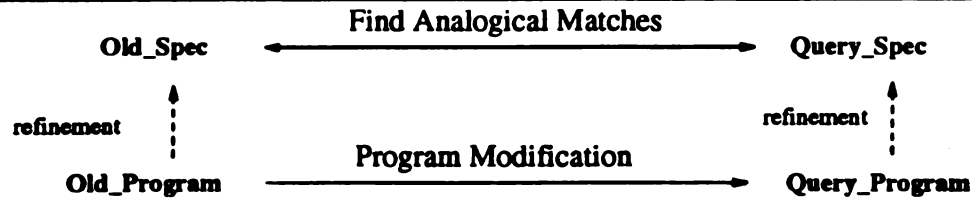


Figure 7.1. Analogical Reuse Modification Process.

---

been widely promoted. Modification is different from traditional program transformation because a program transformation is typically correctness preserving with respect to the original specification, but the program modification approach needs a program that satisfies its input-output specification along with the specification for a new program. Program modification process can be divided into the following phases:

- *Match*: discover an analogy between two specifications.
- *Verification*: check the validity of the proposed modification.
- *Replacement*: apply the proposed modification to the program.
- *Recoding*: rewrite any unexecutable statements.
- *Synthesis*: synthesize new segments.
- *Optimization*: make the new program optimized, e.g., more efficient.

We emphasize the matching process in the ARMP model, i.e., the first phase of the modification process because we try to find an effective way to reduce the effort of modification by providing a set of “useful” matches to the programmers. Dershowitz [64] presents an appealing approach to program construction by modification. His method reflects the observation that programmers only devoted a limited amount of time and effort to create code for a given specification from scratch. Programmers often apply their knowledge about earlier programs to the development of

similar problems. Our work focuses on augmenting Dershowitz's methods in order to make it amenable to automatic applications.

In attempt to analyze and design programs to perform the analogical matching process. We found it useful to consider the phenomenon of analogical reasoning as a whole. Both the brief accounts of *analogy in human reasoning* and the difference between *analogy* and *generalization* are described in Appendix A.

## 7.2 Analogical Matching

An analogical match is defined to be a group of pairings between symbols in terms of *candidate* and *query* specifications. It consists of the following form:  $(symbol_1, symbol_2, position_1, position_2)$  where  $symbol_1$  is located at  $position_1$  of term  $term_1$ , and similarly for  $symbol_2$ ,  $position_2$ , and  $term_2$ . The two symbols in any associated pair in their corresponding positions are called the match. For instance, for the two terms  $max(a, array(b))$  and  $min(queue(c), a)$ . An analogical matching process may generate the following match:

$$\langle (max, min, [], []), (a, a, [1], [2]), (array, queue, [2], [1]), (b, c, [2, 1], [1, 1]) \rangle.$$

The above example exhibits a bijective mapping between terms  $max(a, array(b))$  and  $min(queue(c), a)$ . However, as far as the flexibility of a match is concerned, some required features of match are needed to enhance the power of the matcher. For example, (1) predicates, function, and constant symbols may be matched with different predicate, function, and constant symbols, respectively; (2) the arguments for predicates and functions that are matched may be permuted by the matching process. Since the argument order of functions and predicates is often arbitrary, it is obviously unreasonable to insist that matches preserve argument order; (3) matches may contain inconsistencies where a symbol on one side is matched with two distinct



symbols on the other; (4) finally, symbols and subterms may be left unmatched in an analogy.

According to Hesse's theory [65], analogy is most useful when the domain of query specification is not well understood. Applying his theory to the software reuse problem, if the relationships between the query and candidate specifications are not well known, e.g. no *generality* relationship [43], then we apply the analogical matching process because we are uncertain as to whether we can reuse the old programs to satisfy the new specification. Furthermore, it seems that the nature of analogy is empirical, that is, only after analogy has been applied to a query specification can we determine whether a specific match is good or bad. There is no formal theory or rule that rigorously describes the process for generating a reliable analogical match. Therefore, most analogical matching algorithms use *heuristics* to direct searches for good analogical matches. A given heuristic captures system-defined criteria as to what constitutes a reasonable analogy. It is believed that the notion of analogical heuristics is a useful tool in building analogical reasoning systems. A top-down matching process is described in this chapter and a bottom-up approach can be found in Appendix B.

### 7.3 Heuristics for the Matching Process

Since the search for analogical matches can potentially be combinatorially explosive, using an exhaustive search mechanism is certainly not feasible. Consider the following two expressions from propositional logic:

$$\{ a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c), a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \}.$$

A matching is an association between the two expressions; i.e., a subset of the Cartesian product of the sets of symbol occurrences in the expressions. In this example, each expression contains 13 symbols, so the Cartesian product contains  $13 \times 13 = 169$

symbols, and hence has  $2^{169}$  subsets. Obviously, some heuristics are definitely needed to prune the search space.

When a heuristic is used in analogy systems, issue is to determine what kind of information should the system have to enhance the applicability of the heuristics, that is, what contextual knowledge should be included in the heuristics. A powerful automated analogy system should be able to operate without much contextual knowledge. However, if the context is very important, a “pure” syntactic analogy system may fail on some intuitively straightforward examples. Therefore, an ideal analogy system should allow some contextual knowledge to be supplied interactively by the users or domain experts.

Identical associations are often believed to make good analogues, similarly, the matchings containing high proportions of identical associations make good analogies. We call this approach the *identical symbols heuristic*. While it is obvious that most interesting analogies involve a significant proportion of non-identical associations. However, the identical symbols heuristic is valuable in finding similar specifications (problems). This heuristic has already been incorporated in the construction and retrieval processes in our system. In the analogical matching process, we also use this heuristic since the candidates to be matched are retrieved from a hierarchy by the retrieval process (See Chapter 5).

Another promising analogical approach is to consider matchings that respect the structure of the terms. We call this approach a *homomorphism heuristic*. A homomorphism is a mapping that obeys the structure of the entities to which it relates. Most analogy systems use some form of structural mapping. For example, Gentner [63] introduces the notion of *systematicity* of a structural mapping; this property refers to the extent to which matched objects are mutually constrained by identical relations.

For some heuristic criteria preferring matchings between items (predicates, function symbols, or atoms) of the same or similar semantic type according to an equiv-

alence class partition of symbols, we call this approach the *semantic type heuristic*, which requires some kind of type hierarchy or other means to make a similarity judgement. In Chapter 3, we used the Larch LSL trait hierarchy as the semantic type hierarchy to determine the generality relations between two terms. Since the semantics of an LSL trait hierarchy is well-defined, an LSL trait hierarchy can be used as the basis for a heuristic to determine the similarity between any two logical entities. In Chapter 4, the construction process accesses the *semantic class* (or *equivalence class*) for each of the predicates and functions in the pre- and postconditions that are clustered into a unified hierarchy. For example, the predicate *bigger*( $a, b$ ) denoting that atom  $a$  is bigger than atom  $b$ , belongs to the semantic class *comparison*( $atom_1, atom_2$ ). The intended meaning is that *bigger* is a comparison operator applied to two atoms. Similarly, the predicate *smaller*( $a, b$ ) belongs to the same semantic class. In all the analogies considered in ARMP, predicates are only mapped to predicates, and propositional connectives are mapped only to propositional connectives. All functions and constants are converted to predicates. Thus there are no functions and constants.

## 7.4 Top-Down Matching Approach

In the context of recognizing reusable candidate specifications for solving query specifications, the analogical matching process should have a flexible notion of analogy-based match rather than imposing a design bias towards any single heuristic. It should be easily tailorable to any particular domain-specific heuristic (knowledge). The heuristics mentioned in Section 7.3 should be refined to precisely defined values. That is, some numerical metric should be invented to measure the usefulness of a given analogy. Once a precise definition for the goodness of an analogy match is given, the analogy problem can be regarded as an optimization problem. For an optimization problem, finding a global optimum is not feasible at most of the time

therefore we might reasonably want to generate a set of local optimal analogies.

Since predicate logic is used to express the behavior of software components, the object language of our analogy system is based upon first-order logic. A *semantic type heuristic* suggests the match of two terms but the pairings are not limited to the terms in the same semantic group. We explore how to include the properties of associativity and commutativity in predicate logic in the analogy matching process. For example, it is unsatisfactory that the analogical matches from a set of paired atoms is based on an arbitrary preservation of argument order.

The analogical matching can be regarded as a recursive problem solving process. The initial problem is to match two terms expressed in predicate logic. As the matching process continues, new subproblems are produced and recursively solved. No more subproblems will be produced in two cases: (1) one of the subproblem's terms is a constant or variable; (2) no new analogical pairing is applicable for this subproblem.

The generation of subproblems from the matching process can be classified into two kinds of branches: *or-branch* and *and-branch*. When we want to match terms containing a commutative operator, the set of derived subproblems may suggest more than one way of solving the problem. Therefore, this case applies, the current problem should *branch* into a set of new subproblems, each generating a new group of pairings. For example, consider the following problem:

$$\{ f_1(x_1, y_1) \wedge f_2(x_2, y_2), g_1(a_1, b_1) \wedge g_2(a_2, b_2) \}.$$

Since  $\wedge$  is a commutative operator, the matching process may produce two sets of matches:

$$( (\wedge, \wedge, [], []), (f_1, g_1, [1], [1]), (f_2, g_2, [2], [2]) )$$

and

$$( (\wedge, \wedge, [], []), (f_1, g_2, [1], [2]), (f_2, g_1, [2], [1]) ).$$

Respectively, the matching process generates the following two sets of subproblems:

$$\{ \{(x_1, y_1), (a_1, b_1)\}, \{(x_2, y_2), (a_2, b_2)\} \}$$

and

$$\{ \{(x_1, y_1), (a_2, b_2)\}, \{(x_2, y_2), (a_1, b_1)\} \}.$$

Thus, the current state of matching between a pair of terms involves a set of partial matches and two sets of *or-branch* subproblems. We only need one of the *or-branch* subproblems to be solved in order to proceed to later stages of the matching process. The *or-branch* subproblems are generated by permuting the order of arguments to obtain new sets of argument mappings.

In contrast, if the match process tries to match two terms where one or both of them are predicates, then the problem is branched into two subproblems, each yielding one set of pairings. For the purpose of illustration, we change the operator  $\wedge$  of the previous example to an equality predicate as follows:

$$\{ f_1(x_1, y_1) > f_2(x_2, y_2), g_1(a_1, b_1) \geq g_2(a_2, b_2) \}$$

where  $>$  and  $\geq$  are not commutative operators. Hence, only one match is generated:

$$\langle (>, \geq, [], []), (f_1, g_1, [1], [1]), (f_2, g_2, [2], [2]) \rangle$$

And the matching process generates the following two sets of subproblems:

$$\{(x_1, y_1), (a_1, b_1)\}$$

and

$$\{(x_2, y_2), (a_2, b_2)\}.$$

such that the current problem is split into two new sets of subproblems, with each represented as an *and-branch* together with a partial mapping. The newly generated matches from these two subproblems should not conflict with each other, that is, no inconsistent pairings will be generated. We will define *inconsistency* shortly.

From the above examples, we may conclude that *and-branch* subproblems are generated whenever the argument pairing within an identical argument mapping is

performed; *or-branch* subproblems are generated whenever the matching process encounters commutative terms and attempts to perform argument pairings of permuted argument mappings of the terms. The terms in the latter case include ordinary predicative connectives, for example,  $\wedge$  and  $\vee$ .

**Definition 7.1** *Conflict.* Some pairing  $\sigma_1 = (x, y, P_x, P_y)$  has a **conflict** with an existing mapping  $\Phi$  iff (1) there is some pairing  $\sigma_2 \in \Phi$ , and  $\sigma_2$  consists of either  $x$  or  $y$  but not both; or (2) there exists some pairing  $\sigma_2 = (x, y, Q_x, Q_y) \in \Phi$  and both  $\sigma_1$  and  $\sigma_2$  are argument pairings within the same predicates or functions, but either  $P_x \neq Q_x$  or  $P_y \neq Q_y$ .

**Definition 7.2** *Consistency.* Some pairing  $\sigma$  is consistent with some existing mapping  $\Phi$  if  $\sigma$  has no conflicts with  $\Phi$ .

If the matching algorithm is restricted to the preservation of argument order, then the second requirement of Definition 7.1 will never be applicable in the matching process.

## 7.5 Matching Algorithm

Given a matching subproblem, consisting of a pair of specifications  $\alpha$  and  $\beta$  represented in first-order logic, and an existing mapping  $\Phi_{old}$ , the matching algorithm attempts to find a new consistent mapping  $\Phi_{new}$  and returns it to the user. We assume all variables of  $\alpha$  and  $\beta$  have been either skolemized or universally quantified. The matching algorithm is given in Figure 7.2. This algorithm is based on the matching process approach presented in Section 7.4.

Several heuristics are exploited in Algorithm 9. This algorithm uses a top-down scheme to compare two input terms. The functor symbols of two input terms should be matched before argument pairing tests are performed, hence a homomorphism heuristic is *partially* incorporated. It is denoted as a *partial* homomorphism because commutative argument pairings are allowed in this algorithm. If some term is a commutative operator, then several versions of the term with permuted arguments are created to generate the *or-branch* subproblems, which is case 3 in Algorithm 9.

---

**Algorithm 9**  $Match(\alpha, \beta, \Phi_{old}, \Phi_{new})$

**Input:** Two terms  $\alpha, \beta$ , and a current partial mapping  $\Phi_{old}$ .

**Output:** A new partial mapping  $\Phi_{new}$ .

**Procedure:**

```

begin
  switch( $\alpha, \beta$ )
    case 1: one of  $\alpha$  and  $\beta$  is a constant or variable
      if consistent( $(\alpha, \beta), \Phi_{old}$ ) then
         $\Phi_{new} \leftarrow \Phi_{old} \cup \{(\alpha, \beta)\};$ 
      else  $\Phi_{new} \leftarrow \Phi_{old};$ 
      return;
    case 2:  $\alpha = f(x_1, x_2, \dots, x_n)$  and  $\beta = g(y_1, y_2, \dots, y_n)$ ,
      either  $f$  or  $g$  is not commutative % and-branch subproblems
      if consistent( $(f, g), \Phi_{old}$ ) then
         $\Phi_0 \leftarrow \Phi_{old} \cup \{(f, g)\};$ 
        for all  $i$  do
           $Match(x_i, y_i, \Phi_{i-1}, \Phi_i);$ 
        end_for;
         $\Phi_{new} \leftarrow \Phi_n;$ 
      else  $\Phi_{new} \leftarrow \Phi_{old};$ 
      return;
    case 3:  $\alpha = f(x_1, x_2)$  and  $\beta = g(y_1, y_2)$ ,
      both  $f$  and  $g$  are commutative % or-branch subproblems
      if consistent( $(f, g), \Phi_{old}$ ) then
         $\Phi_0 \leftarrow \Phi_{old} \cup \{(f, g)\};$ 
         $\Psi_\alpha \leftarrow permutation(x_1, x_2)$ 
         $\Psi_\beta \leftarrow permutation(y_1, y_2)$ 
         $\Psi \leftarrow argument\_mapping(\Psi_\alpha, \Psi_\beta)$ 
        for all  $[\alpha\_arg\_list_i, \beta\_arg\_list_i] \in \Psi$  do in parallel
          for all  $x_k \in \alpha\_arg\_list_i$  and  $y_k \in \beta\_arg\_list_i \in \Psi$  do
             $Match(x_k, y_k, \Phi_{k-1}, \Phi_k);$ 
          end_for;
           $\Omega_i \leftarrow \Phi_n$ 
        end_for;
         $\Phi_{new} \leftarrow evaluate(\Omega_1, \Omega_2, \dots, \Omega_{card(\Psi)});$ 
      else  $\Phi_{new} \leftarrow \Phi_{old};$ 
      return;
    case 4:  $\alpha = f(x_1, x_2, \dots, x_m)$  and  $\beta = g(y_1, y_2, \dots, y_n)$ ,
       $m \neq n$  % different number of arguments
      if  $((\alpha_1, \beta_1) = transform(\alpha, \beta))$  then
         $Match(\alpha_1, \beta_1, \Phi_{old}, \Phi_{new});$ 
      return;

```

end.

Figure 7.2. Matching Algorithm

Otherwise, *and-branch* subproblems for case 2 are generated for each pair of permuted arguments of the input terms.

The definition of predicate *consistency* is given in Definition 7.2. Therefore, implicitly, we exclude the possibility of *mismatched analogy* that allows a mapping that is not one-to-one, even though it may be a useful analogy in the real world. If mismatches are allowed in the analogy system, then a considerably large amount for domain knowledge would need to be encoded in the system's knowledge base.

Case 4 in Algorithm 9 deals with the condition when the arguments of two input terms have different sizes ( $m \neq n$ ). In this case, we need some transformation rules to “rephrase” the input terms to make their arguments have the same cardinality. The transformation rules require domain knowledge. For example, suppose we want to match  $\sqrt{a}$  and  $\frac{a}{d}$ , since the functions *square-root* and *division* have different numbers of arguments, they need to be transformed. If the system knows the rule  $\sqrt{a} = \frac{\sqrt{a}}{1}$ , then a match can be easily found:

$$\langle (/ , / , [], []), (\sqrt{a}, c, [1], [1]), (1, d, [2], [2]) \rangle.$$

If the system only knows the rule  $\sqrt{a} = \sqrt{a} \times 1$ , then we have the following match:

$$\langle (/ , \times , [], []), (\sqrt{a}, c, [1], [1]), (1, d, [2], [2]) \rangle.$$

Our algorithm provides a framework for a domain-independent matching process but the domain knowledge is tailorable to more specific types of information.

The complexity of this algorithm increased when each operator is commutative.

Suppose  $P$  is a predicate, then we define

$$\max\_level(P) = \max_{T \in \text{closure}(P)} \{level(P, T)\}$$

where  $\text{closure}(P)$  and  $level(P, T)$  have been defined in Chapter 5. From the top-down matching algorithm, two terms  $A$ ,  $A \in \text{closure}(\alpha)$ , and  $B$ ,  $B \in \text{closure}(\beta)$ , are to be



matched if and only if

$$\text{max\_level}(\alpha) - \text{level}(\alpha, A) = \text{max\_level}(\beta) - \text{level}(\beta, B)$$

For each pair of commutative operators, the matching process generates two subproblems. Therefore, it is trivial to determine the number of the levels in the search tree since it is bounded from above by  $\min\{\text{max\_level}(\alpha), \text{max\_level}(\beta)\}$ . If  $\text{length}(P)$  represents the number of symbols in the predicate  $P$ , then this algorithm's upper bound is

$$\min\{\text{max\_level}(\alpha), \text{max\_level}(\beta)\} \times 2^{\min\{\text{max\_level}(\alpha), \text{max\_level}(\beta)\}}.$$

The function *permutation()* generates all possible permuted argument lists for a commutative operator. In Algorithm 9, *permutation()* always generates two sets of permuted arguments because, currently, the operators that are able to generate *or-branch* subproblems are commutative and they consist of only two arguments. Once *associativity* is incorporated into the case of generating an *or-branch*, then the number of the sets of permuted arguments becomes exponential with respect to the lengths of input terms and the matching process encounters the possibility of combinatorial explosion.

The function *argument\_mapping()* creates bindings between a pair of selected argument lists that are the output of the function *permutation()*. In our approach, the corresponding arguments of two terms should be defined over the same semantic class, i.e., equivalence class (semantic type) heuristic.

The function *evaluate()* chooses the *best* mapping from a set of mappings that are output from a set of *or-branch* subproblems. Certainly, *evaluate()* needs some assessment scheme that determines the degree of reusability for some analogical mapping.

The assessment scheme should be able to select the most promising mapping from a set of candidate mappings to be reused, and it needs much programming and domain knowledge. The assessment scheme is not formally addressed in this dissertation, but a simple assessment scheme is given in the matching example of Section 7.6.

## 7.6 Matching Example

A simple example given in Figure 7.3 is used to illustrate the matching algorithm presented in the previous section. The initial problem with input terms is formulated as:

$$\{ x \geq \max(a, \text{array}(b)), y \leq \min(\text{queue}(c), d) \}.$$

And we use the procedure call

$$\text{Match}(x \geq \max(a, \text{array}(b)), y \leq \min(\text{queue}(c), d), \{ \}, \Phi_{\text{new}})$$

to initiate the matching process.

Figure 7.3 shows the processing tree of the matching procedure. Each node represents a matching problem and the root is the input problem being considered. We define the *and-node* (*or-node*) to be a node generating *and-branch* (*or-branch*) subproblems. A *mapping-node* is a node that generates a pair of matched terms, i.e., *match*. Every *mapping-node* has only one child node. For example, matching the operators ' $\geq$ ' and ' $\leq$ ' creates *and-branch* subproblems since the comparison operators are not commutative. In contrast, the match of the operators *max*() and *min*() produces *or-branch* subproblems because *max* and *min* are commutative. The matching algorithm is recursively applied to the subproblems until either an empty set is generated or an inconsistency occurs. In this example, the only *or-node* generates the following two subproblems:

$$\{ \{(a, \text{queue}(c)), (\text{array}(b), d)\}, \{(a, d), (\text{queue}(c), \text{array}(b))\} \}.$$

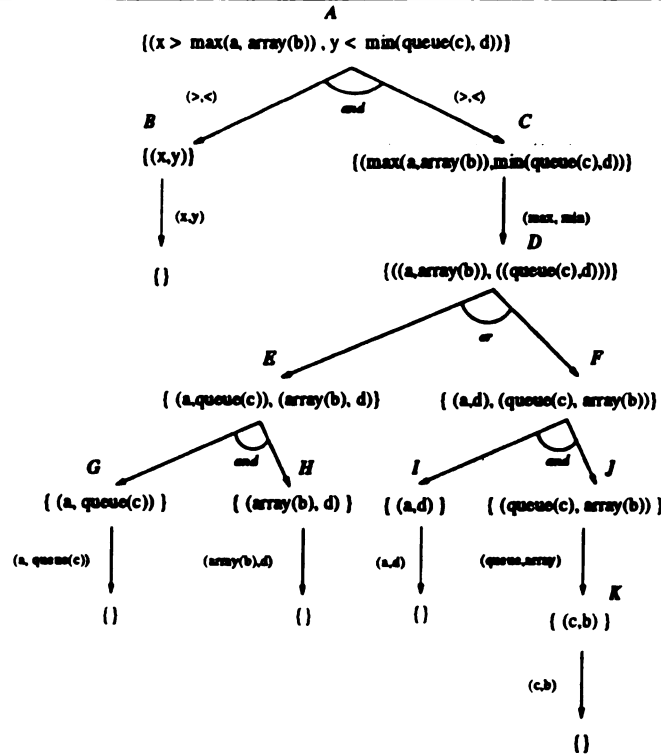


Figure 7.3. Simple example of the matching algorithm.

where one of them is chosen to be returned to the parent node based on some assessment scheme. Currently, we apply a simple assessment scheme where we assign the weights 1, 2, and 3 to constants, variables and operators, respectively. The weight of an *and-node* is the sum of the absolute values of its children's weights. The weight of an *or-node* is the minimum of the weights of its children. The weight of a *mapping-node* is the difference of the matched terms' weights plus the weight of its child. The empty node has weight 0. The *or-node* chooses the set of matched pairs from the children with minimum weight and passes them to its parent node. In this example, the weights of nodes D and E are -1 and 1, respectively, so the weight of node B is 2. The weight of node C is 0 since both its children have weight 0. Therefore, node A will return the matched pairs from node C to its parent node.

As shown in Figure 7.3, there is no inconsistency in this example. Moreover, we assume that the arguments are defined over the same semantic domain. Since an or-branch is created, we should have two sets of matched matches returned from this process. From the processing tree, we can easily recognize two sets of matches:

$$\langle (\geq, \leq, [], []), (x, y, [1], [1]), (max.min, [2], [2]), \\ (a, queue(c), [2, 1], [2, 1]), (array(b), d, [2, 2], [2, 2]) \rangle$$

and

$$\langle (\geq, \leq, [], []), (x, y, [1], [1]), (max, min, [2], [2]), \\ (array, queue, [2, 2], [2, 1]), (a, d, [2, 1], [2, 2]), (b, c, [2, 2, 1], [2, 1, 1]) \rangle;$$

where the latter set of matches is chosen according to our assessment scheme. A more advanced assessment scheme is necessary to evaluate the matches and to give the match information to the users. The assessment scheme can be an explanation-based or tutoring system that contains a large amount of programming knowledge. The final “best” match is located in the set of final partial mappings  $\Phi_{new}$ .

The rest of this section presents another example to show the applicability of the matching algorithm. An *existing* component, the Larch specification of the **Stack** class is given in Figure 7.4. This component defines three methods and all of them are implemented in the public segment of a C++ class. The methods **Stack** and **~Stack** are the *constructor* and *destructor* of class **Stack**, respectively. The method **Stack::push (Stack::pop)** adds (deletes) an **Object** element to (from) **Stack**. **Stack::top** returns the topmost element of **Stack**. The C++ implementations for the methods of **Stack** class are given in Appendix E.3.

A *query specification*, the specification of **DoubleList** class, is given in Figure 7.5. In addition to constructor and destructor, this component defines four methods: **DoubleList::addAtHead**, **DoubleList::addAtTail**, **DoubleList::detachAtHead**, and **DoubleList::detachAtTail**. For the purpose of conciseness, we only consider

---

```

component Stack
abstract type Stack;
public
    void Stack(Stack *stack) {
        modifies *stack;
        ensures (*stack)' = Null_Stack;
    }
    void ~Stack(Stack *stack) {
        modifies *stack;
        ensures trashed(*stack);
    }
    void Stack::push(Stack *stack, Object& newElement) {
        requires ¬full(*stack);
        modifies *stack;
        ensures top(*stack',newElement) ∧
            size(stack') == size(stack) + 1;
    }
    bool Stack::detach(Stack *stack, Object& topElement) {
        requires ¬empty(*stack);
        modifies *stack;
        ensures top(*stack, topElement) ∧
            size(stack') == size(stack) - 1;
    }
    Object& Stack::topElement() {
        requires ¬empty(*stack);
        ensures result = topElement ∧ top(*stack,topElement);
    }

```

---

Figure 7.4. Larch specification of Stack class.

---

```

component DoubleList
abstract type DoubleList;
public
    void DoubleList(DoubleList *dbllist) {
        modifies *dbllist;
        ensures (*dbllist)' = NULL_DoubleList;
    }
    void ~DoubleList(DoubleList *dbllist) {
        modifies *dbllist;
        ensures trashed(*dbllist);
    }
    void DoubleList::addAtHead(DoubleList *dbllist, Object& newElement) {
        modifies *dbllist;
        ensures head(*dbllist',newElement) ^
            length(dbllist') == length(dbllist) + 1;
    }
    void DoubleList::addAtTail(DoubleList *dbllist, Object& newElement) {
        modifies *dbllist;
        ensures tail(*dbllist',newElement) ^
            length(dbllist') == length(dbllist) + 1;
    }
    void DoubleList::detachAtHead(DoubleList *dbllist, Object& element) {
        requires head(*dbllist,element)
        modifies *dbllist;
        ensures ¬head(*dbllist',element) ^
            (element ∉ *dbllist') ^
            length(dbllist') == length(dbllist) - 1;
    }
    void DoubleList::detachAtTail(DoubleList *dbllist, Object& element) {
        requires tail(*dbllist,element)
        modifies *dbllist;
        ensures ¬tail(*dbllist',element) ^
            (element ∉ *dbllist') ^
            length(dbllist') == length(dbllist) - 1;
    }

```

Figure 7.5. Larch specification of DoubleList class.

---

the relationship between two methods - `Stack::push` and `DoubleList::addAtTail`. In order to find an analogous existing component based on query specification `DoubleList`, we apply our matching algorithm to the methods of `DoubleList`. Figure 7.6 shows the results of the application of the matching algorithm to the method `DoubleList::addAtTail` and the method `Stack::push`.

A prototype system for facilitating software reuse has been implemented in the Quintus ProWindows language,\* a dialect of Prolog that supports the object-oriented organization of graphical elements. Our system provides the functions of constructing the hierarchical library [42], retrieving the existing components that have a generality relationship with the query component [44], and assisting users in the modification of more general and analogous existing components to satisfy the query specification [66]. The left part of Figure 7.6 displays the two-tiered hierarchy of a group of components described by formal specifications. The **Candidate Analogies** window displays the matches that are the results of the matching algorithm. The matches are helpful in terms of modifying the existing components for reuse because the users may discover inherent similarities between two components that have no logical relationships that can be found by automated reasoning. Given these candidate matches, the user can reuse or redesign the query component. In this example, the system suggests several matches that may be useful in the modification process. The result suggests that, in order to satisfy the query specification, the input object should be changed from `stack` to `dbllist` and the new element should be added at the tail of `dbllist` instead of the top of `stack`. The C++ implementation for the method `DoubleList::addAtTail` and other methods of `DoubleList` class is shown in Appendix E.4.

The results from the matching process potentially facilitate the modification process by providing the necessary information of specification changes as described in the above examples. A modification example based on analogy will be illustrated in

---

\* A product of Quintus Computer Systems, Inc.

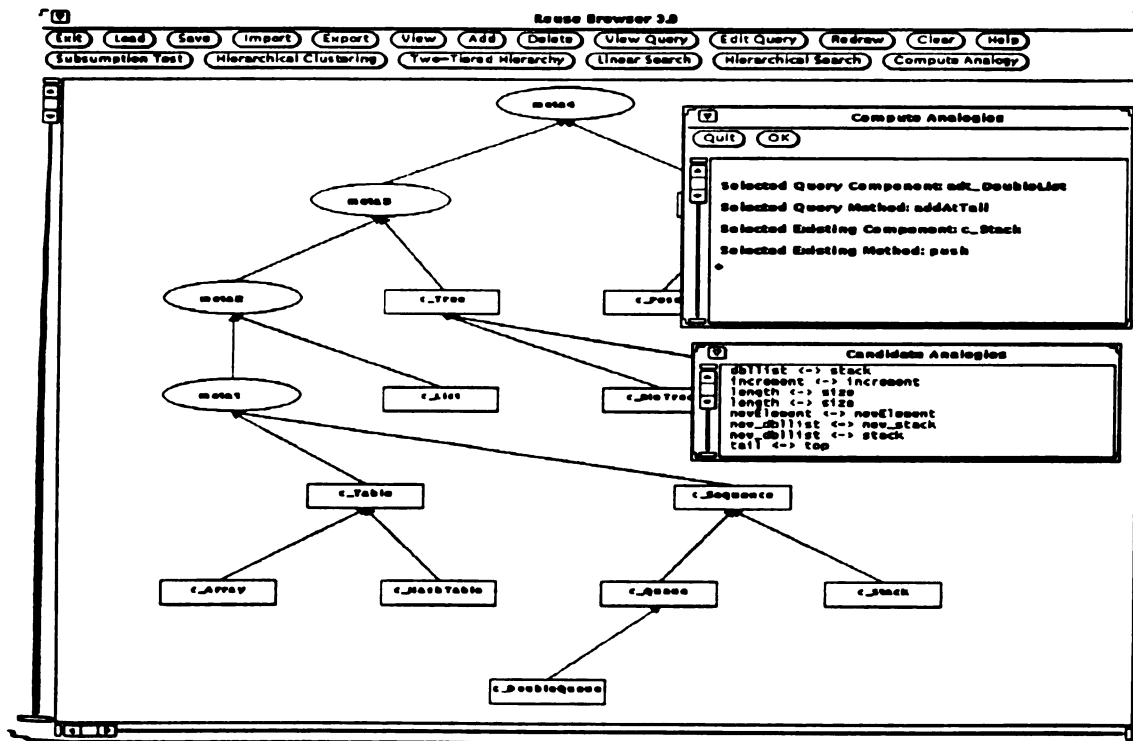


Figure 7.6. An implementation for matching process.



detail in next section. Another challenge is to provide the candidates for the match process, i.e., how to solve the *base filtering problem* mentioned in Section 7.1. Based upon the *similarity* defined in the construction and retrieval processes, the system returns a set of *similar* components (to the query specification) to the users as the candidate specifications for the matching process.

## 7.7 Modification Example

Program modification is a combination of analogy, transformation, synthesis, and verification. In this section, we give an example of program modification based on analogy. We can see how the matching process plays an important role in the modification process. Consider the following specification of a square root program:

$$\begin{aligned} \{Q_1 : a \geq 0 \wedge e > 0 \wedge r \in Real\} \\ \{R_1 : |\sqrt{a} - r| < e\} \end{aligned}$$

where  $Q_1$  and  $R_1$  are the pre- and postconditions of the square root program, respectively. We are given two numbers  $a$  and  $e$  and the desired result is an approximation  $r$ , which is a real number, to the square root of  $a$  with a tolerance value  $e$ . Assume we have an existing program that performs real division as follows:

```

{ Q2 : 0 ≤ c < d ∧ e > 0 }
begin
  s := 1; q := 0;
  while s > e do
    s := s/2;
    if d * (q+s) ≤ c then q := q + s fi
  od.
end.
{ R2 : | c/d - q | < e }
```

Then we apply the matching process to the pair of postconditions of these two Programs, i.e.,  $R_1$  and  $R_2$ :  $\{ (|\sqrt{a} - r| < e), (|c/d - q| < e) \}$ . If the system has the transformation rule:  $\sqrt{a} = \frac{\sqrt{a}}{1}$ , then a match can be found as follows:

$((/, /, [], []), (\sqrt{a}, c, [1, 1, 1], [1, 1, 1]), (d, 1, [1, 1, 2], [1, 1, 2]))(r, q, [1, 2], [1, 2]), (e, e, [2], [2]))$

The match is then applied to the real division program and it becomes

```

begin
  s := 1; r := 0;
  while s > e do
    s := s/2;
    if 1 * (r + s) ≤ √a then r := r + s fi
  od.
end.

```

However,  $1 * (r + s) \leq \sqrt{a}$  contains  $\sqrt{a}$  which is to be implemented so we need to rewrite the statement

$$1 * (r + s) \leq \sqrt{a}$$

by

$$(r + s)^2 \leq a$$

that preserves the semantics but eliminates  $\sqrt{a}$  from the program. Squaring, addition, and comparison are regarded as elementary operations. We obtain the following program:

```

{ Q1 : a ≥ 0 ∧ e > 0 ∧ r ∈ Real }
begin
  s := 1; r := 0;
  while s > e do
    s := s/2;
    if (r + s)2 ≤ a then r := r + s fi
  od.
end.
{ R1 : | √a - r | < e }

```

In this algorithm, the result  $r$  falls in the range  $[0, s]$ , i.e.,  $0 \leq r < s$ . However,  $0 \leq r \leq \sum_1^n (1/2)^n < \sum_1^\infty (1/2)^n = 1$  so  $0 \leq r < 1$ . Once  $a > (1 + e)^2$ , this program will never find the desired answer. This problem can be solved by replacing the initialization command  $s := 1$  by  $s := a + 1$  because the square root of  $a$  is bounded

from above by  $a + 1$ . Consequently, the desired program becomes

```

{  $Q_1 : a \geq 0 \wedge e > 0 \wedge r \in Real$  }
begin
  s := a+1; r := 0;
  while s > e do
    s := s/2;
    if  $(r + s)^2 \leq a$  then r := r + s fi
  od.
end.
{  $R_1 : |\sqrt{a} - r| < e$  }

```

Despite the simplicity of this example, the potential benefits of program by modification is apparent. In the example of this section, the programmer can save some programming effort by reusing the modified program instead of having to program everything from scratch. Once the analogical match is found, the programmer has to develop only those parts of the program that cannot be reused from the old one, which hopefully is much less than to generate the entire program.

## 7.8 Summary

This chapter presents the framework and an approach to applying analogical matching process to reusing software components that are described by formal specifications. Section 7.5 presents a top-down matching algorithm that is consistent, equivalence-class based, and partial homomorphism-preserving. This algorithm allows the commutative operator to be matched with another commutative operator, thus increasing the power of the matching process, i.e., the possibility of finding more analogies. In Section 7.7, an example shows the applicability of the matching process to program modification that assists the user in developing programs that make use of existing software.

The specification of the query and candidate specifications may be given in a form that obscures any analogical matching. Thus, we would like to express the spec-

ification in some equivalent form that makes their similarity more pronounced, for example, the transformational process *transform()* changes the form of some specification but preserves its meaning. This problem is in general difficult to solve. Two functions in the matching algorithm that are not presented in detail are *evaluation()* and *argument\_mapping*. The latter can be implemented by sorting arguments and binding the arguments with same or *similar* semantic types. The challenge is how to allow *associativity* to be incorporated into the matching algorithm and to avoid the combinatory explosion. The evaluation function is more complex in the sense of a large amount of domain and programming knowledge required to make a reasonable assessment and return the best mapping to users.

# CHAPTER 8

## Case Study

### 8.1 Introduction

This chapter describes the application of our reuse framework on a large-scale example. More specifically, the problem domain is the construction of graphical user interfaces based on existing graphical software components. The Larch specification language is used to specify the X window's widgets, where we include a discussion of the issues that we encountered in the process of specifying widgets. We also include a discussion of the strengths and weaknesses of our reuse processes [42, 43, 44, 60, 67, 68].

Several toolkits such as Xt [69], Motif [70, 71] and XView [72] have been built to enhance the development of graphical user interface software. However, these toolkits can only be applied at the implementation level and not at the specification level. Therefore, some common problems in software engineering, such as completeness and consistency, may arise during the life cycle of a software system that includes the graphical user interface developed by these toolkits [73].

We begin with a brief description of X window's widget set/library and Larch in Section 8.2. In Section 8.3, we present a sketch of the specification for window widgets. The full specification is in Appendix C.2 and C.3. Section 8.4 gives an example of specifying a scrolled text editor with popup menus. Section 8.5 shows the

application of the reuse process to a set of formally specified window widgets. Finally, Section 8.6 summarizes this chapter.

## 8.2 Widgets and Larch

This section gives a brief introduction to Motif widgets and their Larch specifications. Detailed descriptions of Motif widgets can be found in Motif programming manuals [70, 71, 74]. Complete Larch specification languages (LSL and Larch interface languages) can be found in [37, 40, 45, 75, 76].

The Motif widget set contains many components, including scrollbars, menus, buttons, etc., and they can be combined to create user interfaces. The Motif widget set is designed to encourage the programmer to follow the Motif Style Guide, which is based on the behavior of Microsoft's Presentation Manager [70]. We can divide the widget classes provided by Motif into several categories based on the general functionality they offer. For example, some widgets display information, while others allow the user to select from a set of choices. Still others allow other widgets to be grouped together in various combinations. Appendix C.1 lists the developed Motif widgets of X window systems.

As mentioned in Chapter 3, the Larch Shared Language (LSL) describes state-independent properties of a program. For example, the *ManagerWindow* trait (Figure 8.1) introduces the sort *ManagerWin* and the operators *move\_win* and *resize\_win*; three equations constrain the meanings of *move\_win* and *resize\_win*, respectively. Each trait name starts with "Lt" represents "Larch Trait".

We write the *module specification* in a Larch interface language to describe state-dependent effects of a program. A **requires** clause states each procedure's precondition; a **modifies** clause lists those objects whose values may possibly change; an **ensures** clause, given the postcondition. The assertion language for the pre- and post-

---

```

LtManagerWindow(ManagerWin): trait
  includes LtCompositeWindow

  ManagerWin tuple of pos: Coord,
                      size: Size,
                      label: String

  introduces
    move_win: ManagerWin → ManagerWin
    resize_win: ManagerWin → ManagerWin
  asserts
    ∀ win: ManagerWin
      move_win(win).pos == coord_map(win.pos)
      move_win(win).size == win.size
      move_win(win).label == win.label
      resize_win(win).label == coord_map(win.pos)
      resize_win(win).label == length_map(win.size)
      resize_win(win).label == win.label

```

Figure 8.1. The *ManagerWindow* trait.

---

conditions is drawn from LSL traits. Through based on clauses, a Larch interface links to LSL traits by specifying a correspondence between programming language-specific types and LSL sorts. An object has a type and a value that ranges over terms of the corresponding sort. Part of the interface specification for the Scrollable Text widget (Figure 8.2) defines the type *ScrolledText*, which is based on the *Scrollable* sort and *Text* sort, introduced in the *ScrollableState* and *TextState* traits, respectively. The *scroll\_win* procedure's precondition requires that the window selected is exactly the window that is to be scrolled, where *window\_to\_O*(*win*) is a coercion operator. The postcondition states that the display value of the window is updated (as defined by the *set\_display* operator whose meaning is obtained from *ScrollableState*) and that all windows are unselected. In a postcondition, an undecorated formal *e*, stands for the initial value of the object; *e'* stands for the final value. The *modifies* clause states that *scroll\_win* may change only the display of the selected window.

---

```

type ScrolledText based on Scrollable from Scrollable Window
                                Text from Text Window
:
operation scroll_win(w: Window, view: View)
    requires e.selected_win = {window_to_O(win)}
    modifies (e_win)
    ensures win' = set_scrolled_display(win, view)  $\wedge$  e'.selected_win = {}
:

```

Figure 8.2. Part of the *ScrolledText* specification.

---

### 8.3 The Specification for Widgets

Several software systems have been formally specified by the Larch specification language. For example, one of the earliest case studies of formally specifying “real” systems with the language Larch involves Avalon/C++ objects [77]. Also Larch has been used to specify visual editor Miró [78, 79, 80]. The behavior of concurrent systems are formally described by a Larch interface language GCIL [81, 41]. A *copying collector* for garbage collection has also been specified in Larch [82].

In this section, we specify the Motif widgets in Larch. The specification itself is composed of two parts, one in the Larch Shared Language (LSL) which is used to specify general Motif widgets, and one in the Larch/C (LCL) which uses the LSL traits to specify the specific C implementation of the widgets. Figure 8.3 illustrates the hierarchical structure of LSL traits, where a trait inherits all the properties of its parent trait. Each oval corresponds to a trait, and an arrow indicates that one trait includes another.



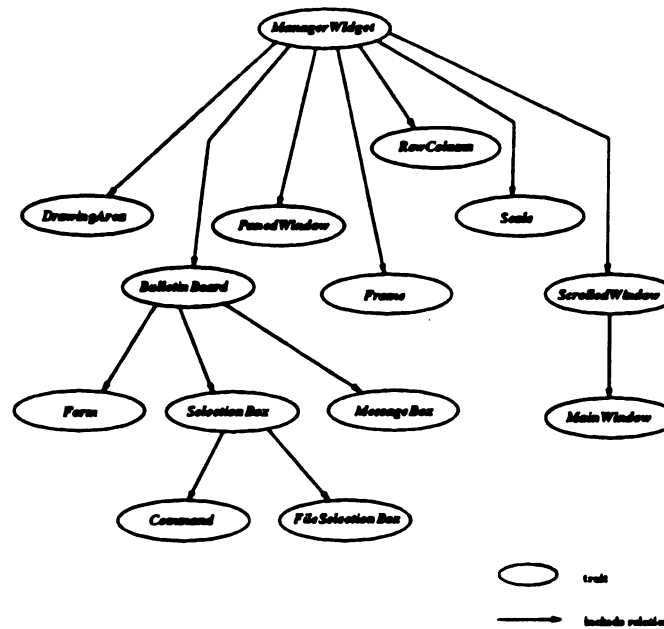


Figure 8.3. The dependencies of the Window's traits.

### 8.3.1 Primitive Widget

Figure 8.4 shows a trait that defines a window as an object composed of content and clipping regions, foreground and background colors, and a window identifier. The member symbol ' $\in$ ' is qualified by a signature in the last line of the trait because it is overloaded, and it is necessary to indicate ' $\in$ ' is converted. Figure 8.5 shows the trait *Primitive* that defines a primitive widget used in the definitions of most widgets. The *Set* trait is defined in the Larch Handbook; the renaming of sort identifier in the first *Set* trait gives the sort *WidgetSet* for sets of items of sort *Widget* and all other operators. The operator that generates a widget is *create\_win*.

We define each of the remaining operators in the trait with equations in the **asserts** clause. The operator *create*(*w*) creates the widget *w*, and *quit*(*w*) destroys the widget *w*. The operator *open*(*w*) opens current widget *w*, and *close*(*w*) closes

---

```

LtBasicWindow: trait
  assumes LtCoordinate
  includes LtRegion, LtView, Displayable(Window)
  asserts Window tuple of pos: Coord,
                                size: Size,
                                cont, clip: Region,
                                front, back: View,
                                id: WId
  forall w: Window, cd: Coord
    cd ∈ w == cd ∈ w.clip
    w[cd] == if cd ∈ w.cont w.front else w.back
  implies converts [-], ∈: Coord, Window → Bool

```

Figure 8.4. The *BasicWindow* trait.

---

current widget  $w$ . The operator  $move(w, \delta)$  moves the widget  $w$  in the screen by  $\delta$ . The operator  $resize(w, \gamma)$  resizes the widget  $w$  to the size  $\gamma$ . The observer operator  $size(w)$  reports the size of the widget  $w$  and  $pos(w)$  the position of the widget  $w$ . The operator  $overlap(w_1, w_2)$  determines if widgets  $w_1$  and  $w_2$  overlap. The operator  $id(w)$  returns the identifier of  $w$ . The operator  $parent(w)$  returns the parent widget of  $w$ , and  $child(w)$  the child widgets of  $w$ . The observer operator  $is\_child(w_1, w_2)$  decides if  $w_1$  is a child widget of  $w_2$ . The function  $coor\_map$  is a function that transforms some widget's coordinates from its original position to the destination. Similarly, the function  $length\_map$  is a function that transforms some widget's geometry and size. A new sort *Widget* is introduced and will be inherited by all widget traits that are defined in Appendix C.2.

### 8.3.2 Object and SetResource

The *Object* trait defines two new sorts that are useful in manipulating objects in a trait regardless of what kind of widgets they are; the *SetResource* trait introduces

---

```

LtPrimitive: trait
  includes LtBasicWindow, LtView, Set(Widget, WidgetSet)
  Widget tuple of pos: Coord
                    size: Size
                    id: Real

  introduces
    create:  $\rightarrow$  Widget
    destroy:  $\rightarrow$  Widget
    open: Widget  $\rightarrow$  View
    quit: Widget  $\rightarrow$  View
    selected: Widget  $\rightarrow$  Bool
    move: Widget, Coord  $\rightarrow$  Widget
    size: Widget  $\rightarrow$  Size % observers
    set_size: Widget, Size  $\rightarrow$  Widget
    pos: Widget  $\rightarrow$  Coord
    set_pos: Widget, Coord  $\rightarrow$  Widget
    overlap: Widget, Widget  $\rightarrow$  Bool
    id: Widget  $\rightarrow$  Integer
    parent: Widget  $\rightarrow$  Widget
    child: Widget, Real  $\rightarrow$  WidgetSet
    is_child: Widget, Widget  $\rightarrow$  Bool

  asserts
     $\forall w$ : Widget
      move(w).pos == coord_map(w.pos)
      move(w).size == w.size
      move(w).id == w.id
      resize(w).id == coord_map(w.pos)
      resize(w).id == length_map(w.size)
      resize(w).id == w.id
    w generated by create
    w partitioned by id

```

---

Figure 8.5. The *Primitive* trait.

sorts and operators to change an arbitrary resource's attribute in a widget. Since many of the widget operators are essentially the same, we would like to operate on *objects* or a set of objects rather than having separate operators for different widgets in window systems. For example, selecting a displayable widget does not depend on what kind of widget is selected, so would be appropriate to have a single operator to select a widget. The *Object* trait (Figure 8.6) introduces the new sorts *Obj*, a union of all widget types, and *ObjSet*, a set of objects [80]. The union of shorthand provides coercion operators between the union set and its component sorts. So the union declaration:

*Obj* union of *wgt*: *Widget*

produces operators with the following signatures:

*WGT\_O*: *WidgetSet*  $\rightarrow$  *Obj*  
*\_.WGT\_TYPE*: *Obj*  $\rightarrow$  *WidgetSet*  
*TAG*: *Obj*  $\rightarrow$  *Obj.tag*

The operator *WGT\_O* coerces a set of widgets to an object, *.WGT* coerces an object back into a set of widgets, and *TAG* is used to determine what kind of widgets the object contains. The *Object* trait also introduces operators to manipulate sets of objects. The operator *objects* returns the set of all objects in a window; *widgets* extracts the set of widgets from a set of objects. The operator *toggle\_in* adds the specified object to a set of objects if it is not in it, otherwise it deletes the object. Moreover, the coercion function is defined *recursively*, that is, a set of objects can also be coerced into an object.

The *SetResource* trait (Figure 8.7) contains the specification for the *set\_attr* operator, which takes an object *Object*, field name, and a value, and returns a new object that is the same as *Object* except that it has a new value for its field *Attr*.

---

```

LtObject(Obj): trait
  includes LtBasicWindow, LtPrimitiveWidget, Set(Obj, ObjSet)

  Obj union of widget: Widget

  introduces
    objects: Window → ObjSet
    widgets: ObjSet → WidgetSet

  asserts
    ∀ w: Widget, ws: WidgetSet, a: Attr,
      as: AttrSet, obj: Obj, os: ObjSet, win: Window

    objects(create_window) == {}
    objects(insert_widget(win, w)) ==
      insert(objects(win), {w}.wgt_type(w))

    widgets({}) == {}
    widgets(insert(os, obj)) ==
      if TAG(obj) = win_type(obj)
      then insert(widgets(os), obj.wgt_type(obj))
      else widgets(os)

    implies converts objects, widgets: ObjSet → WidgetSet

```

Figure 8.6. The *Object* trait.

---

```

LtSetResource(Attr, Val): trait
  includes Set(Attr for E, AttrSet for C), LtObject(Ob for Obj)
  introduces
    valid_attr: Ob, Attr → Bool
    valid_value: Attr, Val → Bool
    get_value: Ob, Attr → Val
    set_attr: Ob, Attr, Val → Ob
  asserts
    get_value(set_attr (Ob, Attr, Val), Attr') ==
      if equal(Attr, Attr')
      then Val
      else get_value(Ob, Attr')

```

Figure 8.7. The *SetResource* trait.

### 8.3.3 Well-Formed Window

Before introducing well-formedness, we define some relationships between widgets. For two widgets  $w_1$  and  $w_2$ , if  $w_1$  contains  $w_2$  then  $w_1$  is the parent of  $w_2$  and  $w_2$  is a child widget of  $w_1$ . We define  $w_2$  to be a descendant of  $w_1$  if either (1)  $w_2$  is a child of  $w_1$  or, (2) there is another widget  $w_3$ , where  $w_3$  is a child of  $w_1$  and  $w_2$  is a descendant of  $w_3$ . The relationship *ancestor* is defined similarly to *descendant*. With *LtPrimitive* trait, we have introduced the widget sort, *Widget*, and the basic operators. One well-formedness condition for widgets is that for any two widgets, say  $w_1$  and  $w_2$ , if  $w_2$  is a descendant of  $w_1$ , then  $w_1$  must not be a descendant of  $w_2$ . Another constraint is that the geometry, size, and location of a descendant is determined by its parent widget. If a widget is destroyed then all of its *child* widgets are destroyed, too. The widgets with the above constraints are called *well-formed* widgets. If all widgets of a window are well-formed, then the window is a *well-formed* window.

The well-formed Widget trait in Figure 8.8 introduces operators that define well-formedness properties and the new well-formed version of the operators that create and modify a widget. In many cases, the result of a well-formed operator differs from the result of its non-well-formed counterpart. For example, deleting just a widget may violate well-formedness, since it may result in “dangling” widgets that have no parent. Hence, *delete\_wf\_widget* must delete all *child* widgets before deleting the widget. Thus, we introduce an additional operator, *delete\_children* to handle the deletion of a set of *child* widgets of a widget. The operator *delete\_objs* returns a widget that is the result of deleting a set of objects in a well-formed manner. The operator *extract\_wf* returns a widget  $w$  that is a maximally well-formed subset of a set of objects, i.e., no ancestor of  $w$  is well-formed. The result of *extract\_wf(os)* is a widget that contains all the objects of  $os$  except the dangling widgets. The mean-

ing of the operators *child*, *parent*, *descendant*, and *ancestor* are straight forward. The operator *is\_well\_formed*(*w*) checks if *w* is a well-formed widget. The result of *managed\_by*(*w*<sub>1</sub>, *w*<sub>2</sub>) will show whether the resources of *w*<sub>1</sub> is managed by *w*<sub>2</sub>. The formal assertion of well-formedness is defined in Figure 8.8.

---

```

LtWFWidget(WFWidget): trait
includes LtPrimitive, Set(Widget, WidgetSet), LtSetResource, Object
introduces
  ancestor: Widget, Widget → Bool
  descendant: Widget, Widget → Bool
  is_well_formed: Widget → Bool
  managed_by: Widget, Widget → Bool
  extract_wf: ObjSet → WidgetSet
  delete_wf_widget: Widget, Widget → Widget
  delete_objs: Widget, ObjSet → Widget
  delete_children: Widget → Widget
asserts for all w, w': WFWidget
  is_well_formed(w) ==
    (managed_by(w, parent(w)) ∧
      (for all w'::descendant(w', w) ⇒ ¬ ancestor(w', w)))
  parent(w, w') == child(w', w)
  ancestor(w, w') == descendant(w', w)
  for all w' ∈ extract_wf(w) ⇒ is_well_formed(w')

```

Figure 8.8. The WFWidget trait.

---

## 8.4 Specification of Scrolled Text Editor with Popup Menus

Given the Motif Widget Model, we now build the formal specification for a text editor with scroll bars and popup menus, which is shown in Figure 8.9. We begin by establishing a model of the text editor state at the trait level. Since the text

editor widget with the popup menu has *child* widgets *ScrolledText* and *PopupMenu*, we also need to specify their traits in advance. The interface level specification then introduces the editor operations defined in terms of changes to the state. Much of the lower-level detail is assumed, e.g. mapping a mouse to keyboard actions and how text interacts with words. Many of the details can be found in X window's manuals [83, 84].

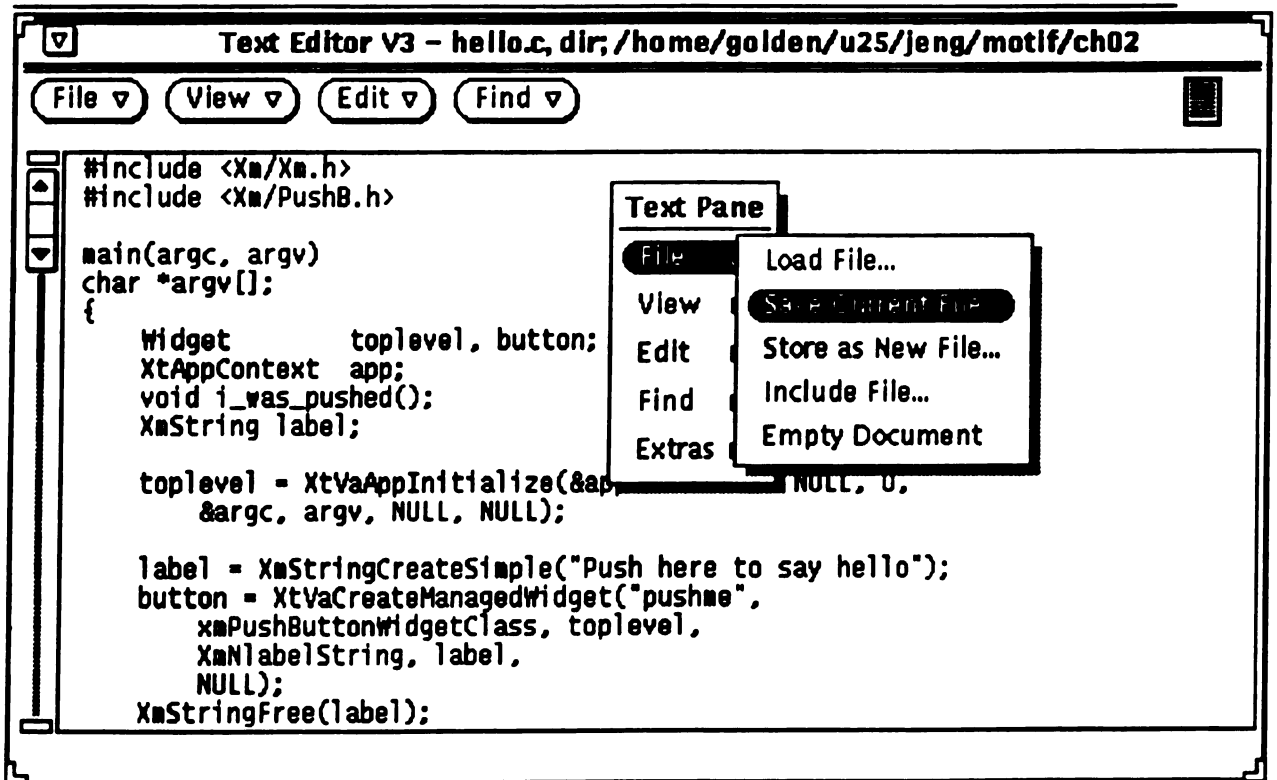


Figure 8.9. An Example of Scrolled Text Editor with Popup Menus.

#### 8.4.1 The X Model

The basic interface of this editor is straightforward. The main part of this editor is the editing area, where the user actually edits the document. On the top of the



editing area, there are several popup menus that allow the user to specify operations, e.g., load a new file. The user can also click in the editing area to popup the same menu as those on the top of the editor. Along the right side of the editing window, there is a scroll bar that allows the user to view the text vertically if the text is longer than the display area.

The scrolled text editor with a popup menu contains a number of different managers and primitive widgets, but appears to the user as a single, conceptually focused user-interface object. The parent-child relationships among the widgets in this editor can be graphically illustrated using the tree-structure shown in Figure 8.10. A widget contains a set of child widgets. for example, the `MenuBar` widget at the top of the editor contains four `PullDownMenu` widgets with labels *File*, *View*, *Edit*, and *Find*, respectively. Each `PullDownMenu` widget contains a `CompoundString` widget for labeling and a `CascadeButton` widget which also contains a set of `PushButton` widgets. Due to limited space, not all widgets used in Figure 8.9 are shown, e.g., only the *child* widgets of `MenuItem File` are displayed in Figure 8.10.

## 8.4.2 The LSL Level

The main purpose of formal specification is to help the user grasp the concept of the system easily instead of becoming immersed in implementation details. For example, a user is not concerned with how a managed widget is mapped to the pixel level (intensity and location) on the screen. For another example, in the domain of Motif widgets, there is not a specific widget designed for a scrolled text editor. A user can create a scrolled text editor by using the text editor widget (`XmText`) and scrolled window widget `XmScrolledWindow`). However, at the specification level, we are free to design such a specification component in order to enhance the understandability of the system. The specification of the scrolled text editor with popup menus consists of several widget traits. The interface specifications of those objects are described in

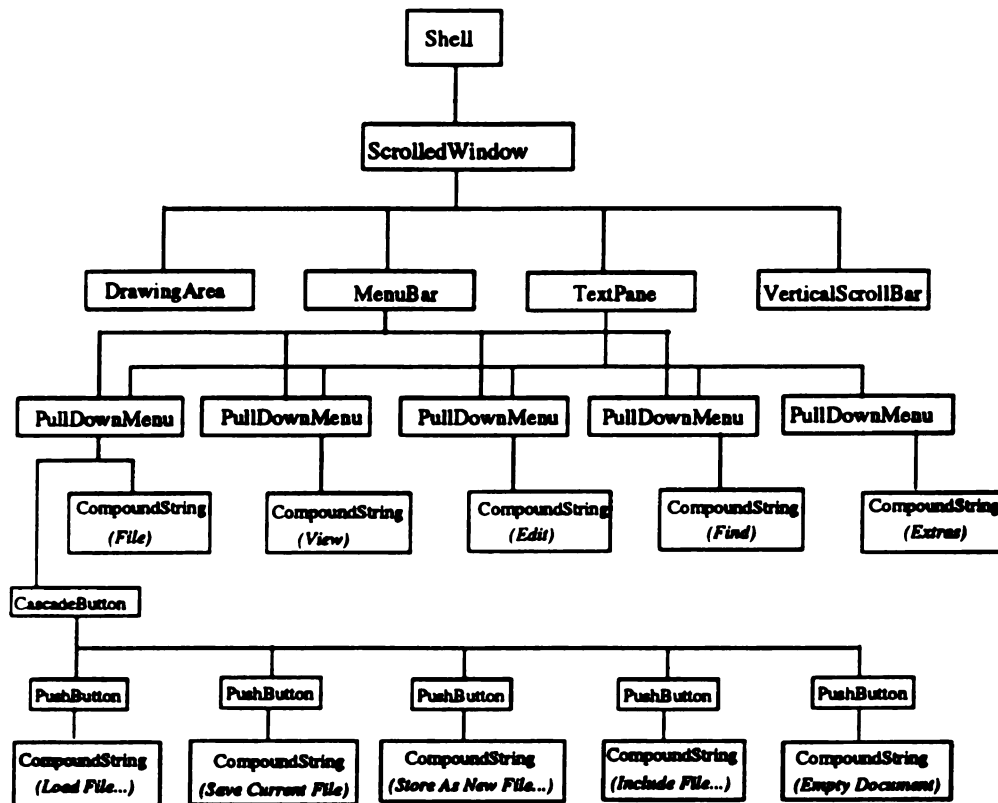


Figure 8.10. Parent-child relationships between widgets.

the following section. Before describing the trait of the scrolled text editor with a popup menu, we specify five constituent traits: `LtScrolled`, `LtTextField`, `LtText`, `LtScrolledText`, and `LtMenu`.

Figure 8.11 gives the trait `LtScrolled` that is the Larch Shared specification of the `ScrolledWindow` widget `XmScrolledWindow`. `ScrolledWindow` provides a scrollable view of data that may not be visible all at once. ScrollBars allow a user to scroll the visible part of the window through the large display. A `ScrolledWindow` widget can be created so that it scrolls automatically without application intervention or so that an application provides support for all scrolling functions. Our `ScrolledWindow` trait `LtScrolled` only gives necessary operators for a `ScrolledWindow`, and other features

such as “scroll up by the size of the viewing window” are omitted.

---

```

LtScrolled: trait
  includes LtView, LtManager, LtPrimitive
  Scrolled tuple of type: {vertical, horizontal},
                                policy: { automatic, app_defined },
  introduces
    create: → Scrolled
    destroy: Scrolled →
    scroll_up: Scrolled → Scrolled
    scroll_down: Scrolled → Scrolled
    scroll_left: Scrolled → Scrolled
    scroll_right: Scrolled → Scrolled
  asserts
    scroll_up(create).cont == Empty
    scroll_down(create).cont == Empty
    scroll_left(create).cont == Empty
    scroll_right(create).cont == Empty
    scroll_up(scroll_down).cont == scroll_down(scroll_up).cont
    scroll_left(scroll_right).cont == scroll_right(scroll_left).cont

```

Figure 8.11. The LtScrolled trait.

---

A **TextField** widget provides a single-line text editor that has a subset of the **Text** widget that is a text editor. We put the specification of **TextField** first, since it is easier to specify a full text editor if we have ideas about how to specify a single-line text editor. The line editor allows movement of a cursor on a line of text and provides a means to change symbols on the line. A user of this line editor will be offered the following facilities [85]:

1. Generating an empty line and placing the cursor at the initial position
2. Inserting a symbol at the cursor position, and moving the cursor to the right of the new symbol.
3. Deleting a symbol to the right of the cursor position.

4. Moving the cursor one position to the left, provided that it is not already at the beginning of the line.
5. Moving the cursor one position to the right, provided it is not already at the end of the line.
6. Moving the cursor to the beginning of the line.
7. Moving the cursor to the end of the line.
8. Clearing the whole line.

The cursor will be placed between symbols, so it divides a line into two parts: what is to the left of it and what is to the right of it. This observation proposes to specify a line together with its cursor as a pair of strings, where the cursor is viewed as taking a position between two strings, and the line is viewed to be the concatenation of two strings. The following specification given Figure 8.12 is based on this idea.

A Text widget (`XmText`) provides a text editor that allows text to be inserted, modified, deleted, and selected. The operations of Text subsume those of single-line editor. We may generalize the operators used in `LtTextField` to design the new trait `LtText` for the Text widget. A document (file) is regarded as a *stream* in UNIX system. Borrowing the concept of a file from UNIX\*, we introduce a new abstract data type *stream* that is a list of symbols. Similarly, we specify a text together with its cursor as a pair of streams, where the cursor is viewed as taking a position between two streams, and the text is viewed as the concatenation of two streams. Figure 8.13 gives the `LtText` trait. The difference between a text and a line is that the former has a two-dimensional layout and a user can move the cursor around the screen in both horizontal and vertical directions. Therefore, we introduce the operator *line* to indicate at which line the cursor is currently located. Other parts of `LtText` are analogous to those of `LtTextField`.

Menus are basic interactive components used in the design of a graphical user interface. A Menu consists of a set of items and allows the user to choose one of

---

\*UNIX is a trademark of AT& T.

---

```

LtTextField(TF): trait
  includes LtView, String, LtCharacter, LtPrimitive
  introduces
    cursor: String, String → TF
    create: → TF
    insert: TF → TF
    delete: TF → TF
    move_left: TF → TF
    move_right: TF → TF
    first: TF → Char
    last: TF → Char
    clear: → TF
  asserts forall c: Char; s,s1,s2: String; tf: TF
    insert(c,cursor(s1,s2)) == cursor(append(s1,c),s2)
    delete(cursor(s1,Empty)) == cursor(s1,Empty)
    delete(cursor(s1,append(c,s2))) == cursor(s1,s2)
    move_left(cursor(Empty,s)) == cursor(Empty,s)
    move_left(cursor(append(s1,c),s2)) == cursor(s1,append(c,s2))
    move_right(cursor(s,Empty)) == cursor(s,Empty)
    move_right(s1,cursor(append(c,s2))) == cursor(append(s1,c),s2)
    first(cursor(s1,s2)) == cursor(Empty,concat(s1,s2))
    last(cursor(s1,s2)) == cursor(concat(s1,s2),Empty)
    clear(tf) == cursor(Empty,Empty)

```

Figure 8.12. The `LtTextField` trait.

---

the items to perform the desired event. In Figure 8.9, the menubar consists of four items: *File*, *View*, *Edit*, and *Find*. The popup menu contains the same items plus one more item, *Extras*. Each item of the menubar and popup menu is a pulldown menu that displays another set of items to be selected. In the domain of Motif, `MenuBar`, `PopupMenu`, and `PulldownMenu` are all instances of the `RowColumn` widget with different types of `XmRowColumnType` resource. However, at the specification level, we should not restrict the menu systems to one implementation such as Motif.

The specification of menu should be abstract enough so that it can describe how a menu behaves without regards to the underlying implementation details. Figure 8.14

---

```

LtText(Text): trait
  includes LtTextField, LtView, LtLine (LN for Line_Number)
    LtCharacter(CN for Character_Number), LtPrimitive
  introduces
    cursor: LN, String, String  $\rightarrow$  Text
    create:  $\rightarrow$  Text
    insert: Text, Stream  $\rightarrow$  Text
    delete: Text, Stream  $\rightarrow$  Text
    move_left: Text  $\rightarrow$  Text
    move_right: Text  $\rightarrow$  Text
    move_up: Text  $\rightarrow$  Text
    move_down: Text  $\rightarrow$  Text
    first: Text  $\rightarrow$  LN
    last: Text  $\rightarrow$  LN
    line:  $\rightarrow$  LN
    contents: Text, LN  $\rightarrow$  Stream
    clear: Text  $\rightarrow$ 
  asserts forall c: Char; cn:CN; s,s1,s2: Stream; n, n1, n2: LN; t: Text
    first(create) == Empty
    last(create) == Empty
    first(cursor(s1,s2)) == cursor(Empty,concat(s1,s2))
    last(cursor(s1,s2)) == cursor(concat(s1,s2),Empty)
    insert(c,cursor(s1,s2)) == cursor(append(s1,c),s2)
    delete(cursor(s1,Empty)) == cursor(s1,Empty)
    delete(cursor(s1,append(c,s2))) == cursor(s1,s2)
    move_left(cursor(Empty,s)) == cursor(Empty,s)
    move_left(cursor(append(s1,c),s2)) == cursor(s1,append(c,s2))
    move_right(cursor(s,Empty)) == cursor(s,Empty)
    move_right(s1,cursor(append(c,s2))) == cursor(append(s1,c),s2)
    line(move_up((cursor(s1,s2)))) ==
      if equal(line(cursor(s1,s2)),1)
      then 1 else line(cursor(s1,s2)) - 1
    line(move_down((cursor(s1,s2)))) ==
      if equal(line(cursor(s1,s2)),line.max)
      then line.max else line(cursor(s1,s2)) + 1
    clear(t) == cursor(Empty,Empty)

```

---

Figure 8.13. The LtText trait.

---

```

LtMenu(Menu): trait
  includes LtPrimitive, LtButton
  Menu tuple of b1: Button,
                 b2: Button,
                 ...
                 bn: Button
  introduces
    open: → Menu
    close: Menu →
    display: Menu → Bool
    submenu: Button → Menu
    selected: Button → Bool
    associated: Button, Event → Bool
    fire: Event → Bool
    act: State, Event → State
  asserts forall b, bi, bj ∈ ButtonSet; e ∈ EventSet
    if selected(bi) ∧ i ≠ j then ¬ selected(bj)
    if selected(b) ∧ associated(b,e) then fire(e)
    if selected(b) ∧ associated(b,DisplaySubmenu)
      then display(submenu(b))

```

Figure 8.14. The LtMenu trait.

---

contains a specification of LtMenu trait, where Menu is defined as a new type consisting of a set of buttons. The type Button is assumed to be predefined. Only one button is allowed to be selected at one time. Each button is associated with one event. As long as some button is selected the associated event is fired and the state will be changed. Since only state independent properties are specified in traits, the operator act will be described in the interface level. The submenu denotes the Motif's PulldownMenu which is a menu pane for all types of pulldown menu systems, including menus of a menu bar, cascading submenus, and the menu associated with the option menu. A PulldownMenu is associated with a CascadeButton. A PulldownMenu can contain PushButton, ToggleButton, and CascadeButton. Here, we represent the event displaying all types of such buttons by the operator submenu that can popup

a new **Menu** object and the associated event is **DisplaySubmenu**. Various kinds of events can be defined by the users.

---

```

LtScrolledText(ScTxt): trait
  includes LtScrolled, LtText
  introduces
    all_visible: ScTxt → Bool
    visible: ScTxt, View → Bool
    visible_content: ScTxt → View
  asserts
    all_visible(visible_content(ScTxt)) == true
    visible(ScTxt, visible_content(ScTxt)) == true

```

Figure 8.15. The **LtScrolledText** trait.

---

The **LtScrolledText** trait can easily be specified by including predefined traits **LtScrolled** and **LtText** (Figure 8.15). The operator **all\_visible** checks if the entire text is visible to the user. The operator **visible\_content** will return the contents of the visible text. Finally, the trait for the scrolled text editor with its menu is formally described in the **LtScrolledTextMenu** trait (Figure 8.16). The specifier may include any primitive traits into the final design, e.g., deciding that either **popupbar** or **menubar** is needed in the text editor.

### 8.4.3 The Interface Level

An interface specification language is not tailored to specifying the behavior of an entire program; instead it is tailored to specifying the behavior of a part of a program (a module) [86], e.g. the Xt/Motif widgets. The LSL trait utilized in the interface language describes the abstract values and some vocabulary that is used to manipulate the abstract values. For example, Figure 8.17 shows part of the specification for



---

```

LtScrolledTextMenu(ScrolledTextMenu): trait
  includes LtScrolledText, LtMenu
  introduces
    with_popupmenu: ScrolledTextMenu → Bool
    with_menubar: ScrolledTextMenu → Bool
    label: ScrolledTextMenu → String
  asserts
    label(set_attr(ScrolledTextMenu, Label, Value)) == Value
    with_popupmenu(set_attr(ScrolledTextMenu, Popup, _)) == true
    with_menubar(set_attr(ScrolledTextMenu, MenuBar, _)) == true

```

Figure 8.16. The `LtScrolledTextMenu` trait.

---

primitive widgets based on the Larch generic interface language. Each **method** in the specification component `Primitive` corresponds to an operation associated with this component. *Select* takes a displayable widget as a parameter and makes the widget be in a state of having been selected. If the widget is already selected, then it remains selected. Thus, the **ensures** clause states that the return value of the Boolean operator `selected` applied to the chosen widget is always *true*. *Unselect* is very straightforward in that there are no preconditions, and the effect of the operation is that the `selected` operator returns *false*. *Move(w, delta)* moves each widget *w* by *delta*. For each widget *w*, the value of *w* after *Move* is the result of setting the position of the widget to its previous position plus *delta*. Once the *Move* is performed, the widget is unselected. This action/behavior is reflected in the second clause of the **ensures** clause. The specification of *Resize* is similar; it takes one widget as a parameter. The ensures clause changes both the position and size of the selected widget and unselects the widget.

Figure 8.18 states the behavior of a `Popup` widget in a window. The operation *Push(p)* makes a popup widget viewable at the topmost level among current viewable widgets. On the other hand, *Release(p)* makes it disappear and become unselected.

---

**component** *PrimitiveWidget* based on *Widget* from *LtPrimitiveWidget*

**method** *Select*(*w*: *Widget*)  
     **requires** (*w* ∈ *DisplayableSet*)  
     **modifies** (*w.view*)  
     **ensures** *selected*(*w'*) == *true*

**method** *Unselect*(*w*: *Widget*)  
     **requires** *true*  
     **modifies** (*w.view*)  
     **ensures** *selected*(*w'*) == *false*

**method** *Move*(*w*: *Widget*, *delta*: *Coord*)  
     **requires** *true*  
     **modifies** *w*  
     **ensures** *selected*(*w*) == *true* ⇒  
             (*pos*(*w'*) == *pos*(*w*) + *delta*) ∧ *selected*(*w'*) == *false*

**method** *Resize*(*w*: *Widget*, *pos*: *Coord*, *size*: *Size*)  
     **requires** *selected*(*w*)  
     **modifies** *w*  
     **ensures** *w'* == *set\_size(set\_pos(w,pos),size)* ∧ *selected*(*w'*) == *false*

Figure 8.17. Part of the Interface Specification of **Primitive Widget**.

---

The component **Popupmenu** inherits the operations from **Popup** and adds more operations, and we can say **Popupmenu** is a subclass of **Popup**. The interface specification can be reused by inheritance. A subclass inherits its parent class's specification; that is, the data member declarations and member methods of parent classes are inherited by the a derived class.

Specification inheritance provides a simple flexible mechanism for specifying a widget without respecifying existing widgets. In C++, only *public* functions can be inherited by derived class. This constraint is not imposed on our widget specifications for brevity purposes. Figure 8.19 is an example of a derived widget that inherits the

---

```

component Popup
  inherits Primitive

method Push(p: Popup)
  requires  $\neg displayable(p)$ 
  modifies p.view
  ensures  $displayable(p') \wedge selected(p) \wedge top\_level(p)$ 

method Release(p: PopupWidget)
  requires  $displayable(p)$ 
  modifies p.view
  ensures  $\neg displayable(p') \wedge \neg selected(p)$ 

```

Figure 8.18. The Larch Interface Specification of *Popup*.

---

member functions of the widget specification component *Popup* and introduces more member functions belonging to itself only, i.e., *CreateItem*, *DeleteItem*, *Choose*, and *Actor*. The operation *Choose* selects an item from the popup menu and fires the corresponding event. *Actor* makes the state of the current system change to reflect the fired event.

## 8.5 Applying Reuse Processes to Specification Components of Xt/Motif Widgets

This section describes an example that applies the reuse processes (construction, retrieval and modification) to the specification components of *Motif* widgets. More specifically, this section provides the reader with a documented example of a moderate size reusable component library and gives the reader some feel for the applicability of the reuse processes for real-world problems. We have specified about 80 widgets that have been implemented and described informally in the *Motif Programming*

---

```

component Popupmenu
  inherits Popup
  uses Menu from LtMenu

method CreateItem(p: Popupmenu, i: Item, e: Event)
  requires  $i \notin \text{ItemSet}(p)$ 
  modifies p
  ensures  $i \in \text{ItemSet}(p) \wedge \text{associated}(i, e)$ 

method DeleteItem(p: Popupmenu, i: Item)
  requires  $i \in \text{ItemSet}(p)$ 
  modifies p
  ensures  $i \notin \text{ItemSet}(p)$ 

method Choose(p: Popupmenu, i: Item)
  requires  $\text{selected}(p) \wedge \neg \text{selected}(i)$ 
  modifies i
  ensures  $(\text{associated}(i, e) \Rightarrow \text{fire}(e) \wedge \text{selected}(i))$ 
            $\wedge (i \neq j \Rightarrow \neg \text{selected}(j))$ 

method Actor(s: State, e: Event)
  requires  $\text{fire}(e)$ 
  modifies s
  ensures  $s' == \text{act}(s, e) \wedge \neg \text{fire}(e)$ 

```

Figure 8.19. The Larch Interface Specification of *Popupmenu*.

---

**Reference** [70].

Figure 8.20 shows a set of unstructured widget components that are to be used as an example. The system allows several operations on the components and their methods. The user may *add*, *delete*, or *view* some component in the workspace. Similarly, the user may also *add*, *delete* or *edit* some method of a component. The constructed hierarchy can also be saved in the system library that contains a set of reusable components.

### 8.5.1 Construction Process

Figure 8.21 gives a snapshot of the result of applying the *subsumption test* to the above set of widget components, i.e., a low-level hierarchy is formed among the above widget components. The *parent-child* relationship between two components reflects their *generality* relationship that is described in Chapter 4. Then applying the *clustering test* to the set of the most general components of the *lower-level* hierarchy generates a unified two-tiered hierarchy of the widget components which is shown in Figure 8.22, where the box represents a *real* specification component of some widget and the ellipse represents a *meta* component which is created by the clustering algorithm. As for the construction of the two-tiered hierarchy of the above components, it took about 10 minutes to construct the hierarchical structure of the widget components. Actually, most of the computation time is due to the calculation of the subsumption relationship among components. The subsumption test algorithm is applied to a set of “flat” components so the time complexity is proportional to the square of the number of the components. In Chapter 4, some algorithms are provided to improve the efficiency of this process. However, in most cases, the time complexity that it takes to construct the hierarchy is not critical since we are more interested in the ability to search and retrieve a set of reusable components efficiently and modifying them to satisfy the query specifications correctly.

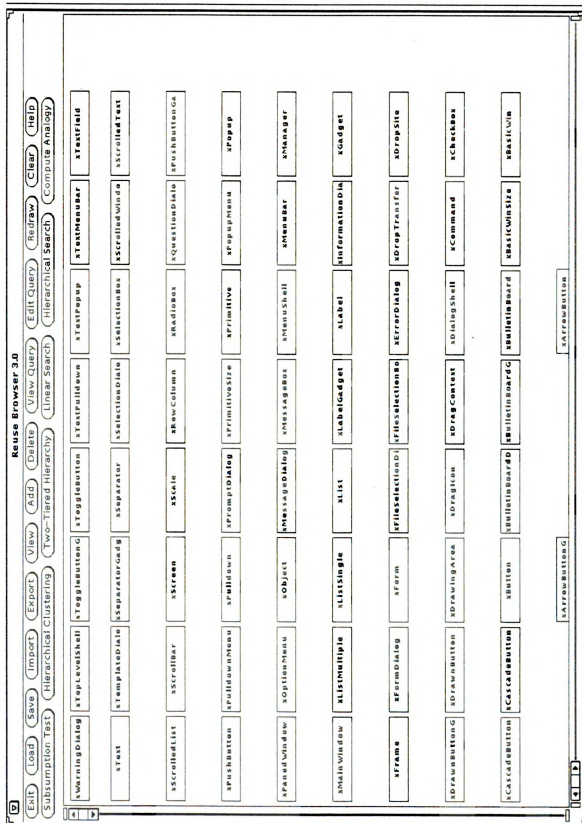


Figure 8.20. The unstructured widget components.

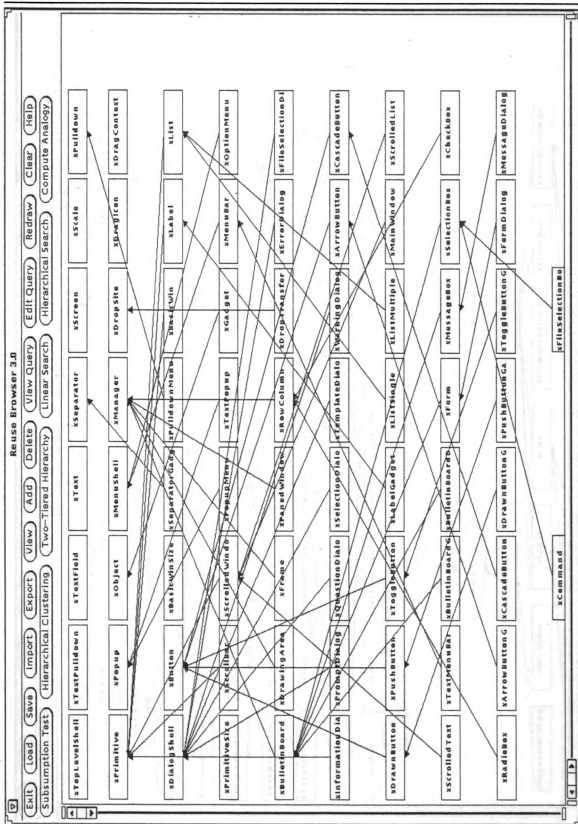


Figure 8.21. The lower-level hierarchy of widget components.





### 8.5.2 Retrieval Process

The time spent in the construction process is compensated by the time saved in the *retrieval* process. Due to the hierarchical structure of the components, a user is able to retrieve a set of components that exactly match the query specification, or are more *general* than the query specification. Figure 8.23 gives an example of retrieving an exactly-matched component. Given the query specification `win_ScrolledText` which is partially shown in the left side of the browser, the system returns the exactly-matched component `xScrolledText`. Figure 8.24 gives another snapshot of the result of retrieval process, where a group of more general components than the query component `win_ScrolledText` is returned to the user. The specification of `win_ScrolledText` is shown in Appendix C.3. The system takes only several seconds to return the set of more general components (or exactly-matched component) to the user and this example shows the applicability of the retrieval process in the framework of a two-tiered hierarchy.

### 8.5.3 Modification Process

As illustrated in Chapter 7, the matching process helps the user to recognize the similarity (or difference) between two components' methods and to modify an existing component to fit the query specification. Figure 8.25 gives an example of applying the matching process to the method `select` of the query component `win_PopMenu` and the method `push` of an existing component `xPullDownMenu`. The **Candidate Analogies** window displays the matchings found between the above two methods. The result shows that these two methods almost have the same form except that the term `pulldown` in `xPullDownMenu::push` should be replaced by the term `popup` in `win_PopMenu::select`. The same process can be applied to other methods of these two components to obtain similar results.

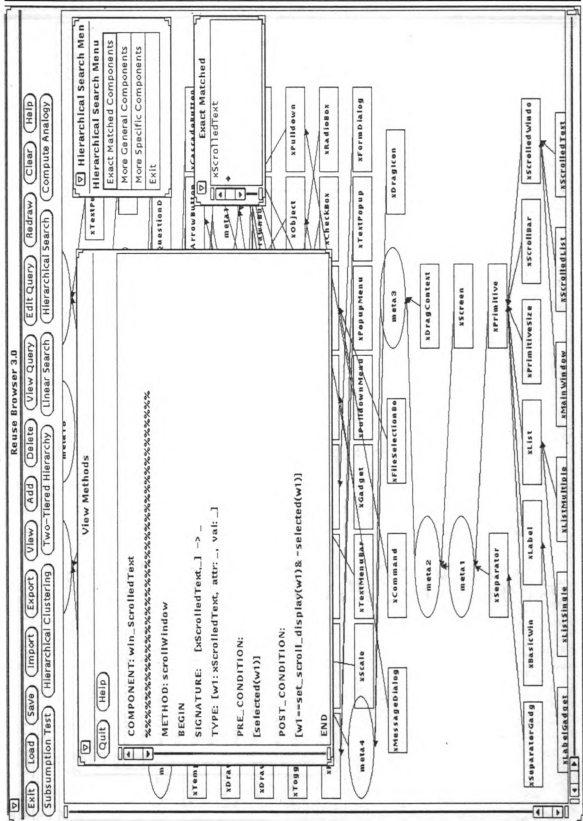


Figure 8.23. An example of retrieving an exact-matched component.

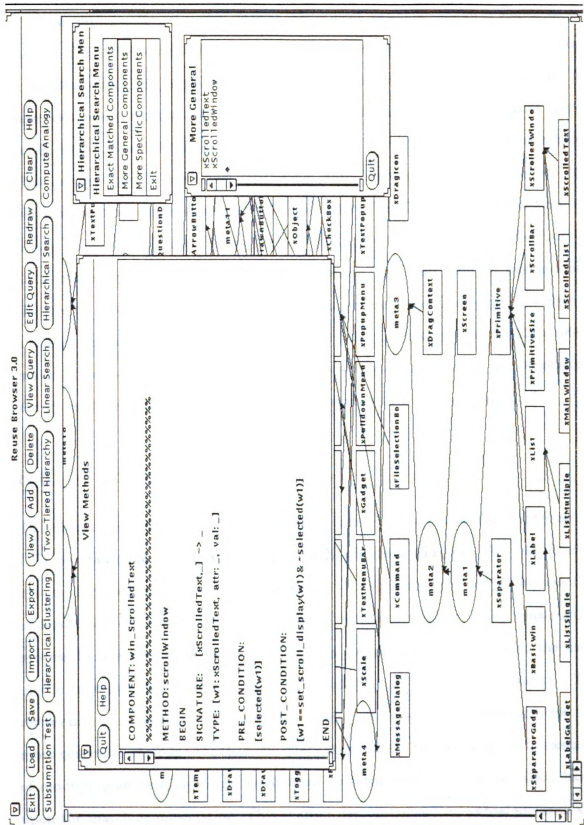


Figure 8.24. An example of retrieving a set of more general components.

The implementation of the existing component `xPullDownMenu` is shown in Appendix E.5. The output of this implementation is displayed in Figure 8.26 that is a window containing a pulldown menu. Based on the matching information, the user can easily modify the implementation of `xPullDownMenu` to support popup menu type. The only thing we need to do is to specify a set of new parameters that are required in a window containing popup menus. Motif defines two convenient functions `XmCreatePopupMenu` and `XmCreatePulldownMenu` to initiate the popup menu and pulldown menu respectively. Therefore, we exploit the function `XmCreatePopupMenu()` to obtain a popup menu. The main routine was modified slightly to move the menu out of the menubar and into the `DrawingArea` widget as a popup menu. The menu item declarations from the program of `xPullDownMenu` may still be used. The final implementation of the query component `win_PopMenu` is shown in Appendix E.6. The output of this program is given in Figure 8.27.

## 8.6 Summary

This chapter provides a case study of applying formal methods to a real world example GUI programming, specifically in terms of X window's widgets. The specification process formally describes the Motif widget's components and helps the users to understand more about the behavior of the window's widgets. The construction process built a two-tiered hierarchy of widget components and can assist the users in the browsing, the retrieving, and the modifying of existing components. The retrieval process has shown the ability to find a set of existing components that are more general or more specific than the query component. Finally, we gave an example of modifying an existing widget component to satisfy the query specification based upon the matching information. This chapter demonstrates how formal methods can be used in the specification of the software components, construction of the software

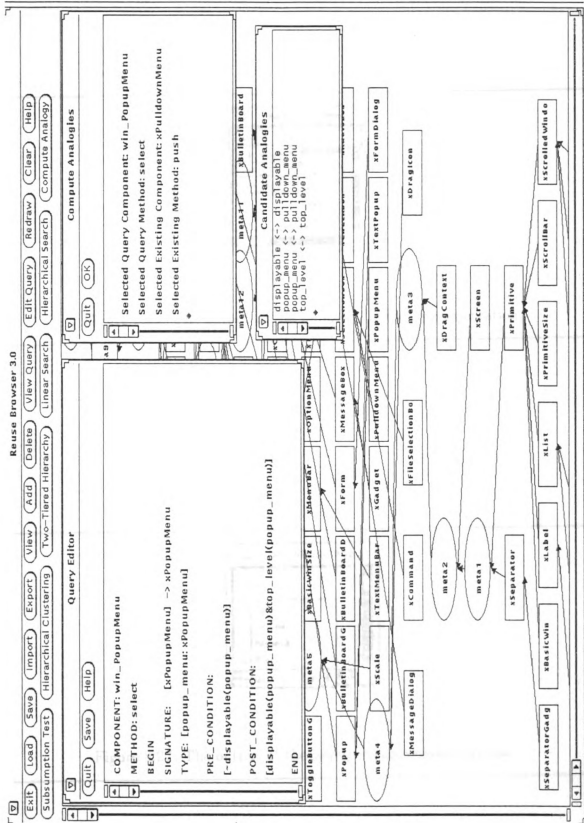


Figure 8.25. An example of computing matching between two methods.

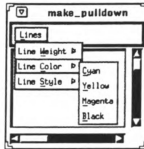


Figure 8.26. The output of the pulldown menu implementation.

---



Figure 8.27. The output of the popup menu implementation.

---

comp

retrie

component hierarchy, retrieval of the reusable components and modification of the retrieved reusable components.



# CHAPTER 9

## Related Work

This chapter describes systems that have been built to perform software component reuse, including description of the specific methods used by each system. The related work discussed is categorized according to software reuse, uses of analogy, and specifications of graphical user interface. While this survey is comprehensive, it is certainly not exhaustive.

### 9.1 Related Work for Reuse

#### Caldiera and Basili

Caldiera and Basili [25, 22] propose a reuse framework that defines a project organization and an experience factory. The project organization performs activities specific to the implementation of the system to which it is dedicated. The project organization engineers may request a list of reusable components from the experience factory. The experience factory's process model is twofold: it satisfies requests for components from the project organization, but it also prepares the pool of choices for the requests. The work in this paper is similar to the latter task in that we also prepare a component library, however, our library is hierarchically organized. We also use a formal representation for the software components in order to facilitate an

automated approach for determining software reuse, whereas their approach requires domain experts to select a set of reusable components. They produce software components by developing a component production plan - the experience factory extracts reusable components from existing systems or generalizes components previously produced on request from the project organization. They need domain analysts to study each component in order to determine the service it can provide. Then components are stored in the repository with all information that has been obtained about them. However, we prefer our approach for reasons of cost and extensibility. First, many domains cannot be easily described by domain analysts and domain analysis is very difficult and expensive. Our library can be automatically constructed without human intervention. Our library can be built for any domain and can easily be modified.

## **GURU**

The GURU project [14, 15] automatically assembles large components by using information retrieval techniques. The construction of the library consists of two steps. First, attributes are automatically extracted from natural language documentation by using an indexing scheme. Then a hierarchy is automatically generated using a clustering technique similar to our hierarchical clustering algorithm. The indexing scheme is based on the analysis of natural language documents obtained from manual pages or comments. The assumption is that natural language documentation is a rich source of conceptual information. However, natural language is not a rigorous language to specify the behavior of software components. One concept may be expressed in different ways or styles between two natural language documentations. In contrast, a formal specification language can serve as a contract, and a means of communication among a client, a specifier, and an implementer [19] with applicable formal verification techniques. Because of their mathematical basis, formal specifications are more precise and more concise than natural language documentation. Another presumption of their indexing scheme is that 98 percent of lexical relations

relate words that are separated by at most five words within a single sentence. Therefore, they extract pairs of words by sliding a window over the target-text.

## MAPS

Nishida and others [17] developed a method of semi-automatic specification refinement and program generation using library modules. Users write their specifications, modify and rearrange them so that the specification can be refined with the aid of the library modules. When a specification is given, a refinement system called MAPS searches for library modules applicable to the given specification, replaces the specification with a more detailed description written in the operation part of the modules, and convert the refined specification into a programming language designated by the user. Case-like expressions or pseudo-natural language expressions are used for describing user's specification and specifications for library modules. MAPS exploits the unification capability to search through reusable modules in library. However, their library is not hierarchically organized, thus the search space could become very large once the number of software modules in the library increases.

## Draco

The Draco project [87] focuses on the use of domain engineering of software reuse [26]. The goal of this project is to increase the productivity of software engineers in the construction of similar systems by organizing reusable software components by domain analysis. Draco was among the first systems to promote the reuse of products from all phases of software life cycle. from analysis, design to implementation (components). The most important aspect of Draco is the *domain language* that describes objects and operations of a particular domain and hence represents analysis information about that domain. The object and operation also describes the design information. There is a reusable component associated with each domain language object or operation. Since there is a potentially large number of components within a domain, Draco uses a classification scheme for the components called *faceted classification* to

aid in organizing and retrieving the components [88, 89, 23].

Using faceted classification, each component is described by a set of attributes. The attributes are chosen to best characterize the components of a particular domain. Each attribute is filled with a term from a well-defined vocabulary. A thesaurus is provided to determine the proper term to apply. A query is a tuple with selected terms used as keys to search the database. In general, a query session begins with with the most specific query, that is, all attributes are filled in. If the results of the query are unsatisfactory, the user may generalize the query by inserting a wildcard character (\*) for attribute values. Draco maintains a measure of conceptual closeness for the term lists of each attribute as a weighted, acyclic, directed graph. This way, an unsuccessful search can be tried again using an alternative but similar term in one of the attributes. The advantage of using faceted classification is that it is conceptually simple for users and relatively easy to implement. However, the disadvantage to faceted classification is that it is not suitable for unconstrained domains. Also, even with a conceptual closeness measure, semantically similar components may be overlooked, especially components from different domains.

### **The Reusable Software Library**

The Reusable Software Library is a system designed to make software reuse an integral part of the software development process [90]. The components of RSL are stored in the database with attributes that provides a basis for retrieval. RSL has two methods to search for components: standard multi-attribute search and natural language. The former method provides a menu-driven interface in which the user selects the attributes with which to perform the search. Alternatively, the user may express a query in the form of natural language. The system parses the input, extracts keywords from the query and uses those keywords as attributes to perform the search. The natural language front end is apparently easy to use, however, it is also expected to be inaccurate and slow because of the nature of the natural language

parser.

### **The Programmer's Apprentice**

The goal of the Programmer's Apprentice is to exploit artificial intelligence techniques in an effort to automate the programming process [91, 18, 92]. It is designed to provide intelligent assistance in all phases of a programming task. The designer thinks of the Apprentice as an agent in the process rather than as a tool. A reusable component in the Programmer's Apprentice is called a *cliché* that represents a commonly used combination of elements. A formalism called a Plan Calculus has been developed to represent a *cliché*. The Program's Apprentice automates *cliché* recognition as a means to understand existing programs and facilitate program optimization. It uses graph parsing to recognize *cliché* in program in order to recognize a program's design. A maintenance tool called Recognizer automatically finds all occurrences of a given set of *clichés* in a program and builds a hierarchical description of the program in terms of *clichés* found. Since plan is essentially a directed graph, the system uses graph-parsing to identify sub-graphs that are then replaced with more abstract operations. However, it is not efficient to search desired *clichés* via purely structural exhaustive strategy graph parsing. Some heuristics are expected to be added in order to enhance the system's performance.

### **Steigerwald**

Steigerwald [27] describes a tool used in Computer Aided Prototyping System (CAPS), developed at the Naval Postgraduate School, which retrieves reusable components from a software base using a formal specification as a search key. The query specification that represents a design requirement is compared to formal specifications of Ada reusable components stored in an object-oriented database management system. A syntactic search compares specification interfaces, identifying reusable candidates based on types of parameters. The semantic search rank orders a set of candidates based on semantic similarity to the query. The method, query by consis-

tency, compares terms that are reduced in the axioms of each specification. Specifications are normalized to facilitate the matching between query specification and reusable component specifications in the retrieval. A formal proof verifies that query consistency can retrieve components to meet specified requirements. The framework of this tool is similar to ours, however, there is no classification scheme exploited in it to enhance retrieval.

### **Zaremski and Wing**

Zaremski and Wing [93] propose *signature matching* as a method for achieving software reuse by using signature information easily derived from a reusable component. They consider two kinds of software components, functions and modules, and hence two kinds of matching, function matching and module matching. The signature of a function is simply its type; the signature of a module is a multiset of user-defined types and a multiset of function signatures. They consider both the *exact match* and *relaxed match*. In their work, the pre- and postconditions are not used as a key to find a set of reusable components.

## **9.2 Related Work for Analogy**

Several researchers have applied analogical reasoning to real-world problems e.g. automated theorem proving, education, artificial intelligence, and software engineering. This section provides a brief overview of some analogical applications in the AI and software engineering fields.

### **9.2.1 Analogy in AI**

Much of the work done in AI not use analogies in problem solving, and therefore turns out to be less interesting to our concerns in the application of analogy in solving reuse problems. This section is presented for the purpose of giving a brief history of analogy

in the AI community.

### Evans

One of the earliest computational accounts of analogy is due to Evans [94]. Evans wrote a program that attempted to solve geometric problems of the kind that occur in intelligence tests. These problems test the ability to perceive relations and analogies, or rather the ability to perceive the same analogy as the person who defined the problem. The analogies are not used for any external purposes. Evans' early work on analogy is an important part of the development of analogy with AI. Evans was the first person to articulate analogies as symbolic correspondences between formal representations. The notion of what an analogy exists implicitly in the work of both Hesse [65] and Polya [95], but is not developed to the stage of being made explicitly.

### Kling

Kling [96] developed an analogical reasoning system for use with an existing resolution theorem proving system. It was the first attempt to automate the use of analogies to solve problems. That is, Kling was the first to introduce the paradigm for reasoning by analogy. Kling was mainly concerned with the analogies that exist between different branches of abstract algebra, particularly group theory and ring theory.

### Carbonnell

Carbonnell [97] has discussed how a problem solving system can be augmented with analogy components. He proposes various operators that may be applied to a faulty plan in a hope of transforming it into a valid target solution. The branching rate at the solution transformation level is much greater than that at the original state space level, and the solution level may lie at greater depth than the object level; thus, without some account of the control regime at the transformation level, the use of the analogy is likely to make the target problem harder to solve. Carbonnell proposed a new model for reasoning by analogy within an automated problem solver [98], where he coined the phrase *derivational analogy*. Instead of looking for analogies

between problem representations alone, analogies are sought between initial segments of problem solving activity between the target and the base. That is, the target problem is attacked by the problem solver (presumably with some general purpose search technique); the entire trace of the problem solver's search at any stage is retained, including all failed branches and intermediate states; matching takes place between this structure and the corresponding initial trace for candidate base problems; once an adequate match is found, the later stages of the process proceed in a way similar to Carbonnell's earlier model.

### **Indurkha**

Indurkha [99] describes a knowledge representation scheme, very similar to first-order logic, and defines a notion of analogy within it. While Indurkha does consider the problem of reasoning with analogies, he requires an analogy to be an isomorphism between formulas, where predicates and objects may be associated with different predicates and objects, but only in a strictly consistent manner. There can be no permutation of arguments between associated terms and formulas, and there can be no unmatched arguments.

### **Gentner et al**

Gentner and colleagues perform investigation with analogy from both a psychological and computational standpoint [100, 101]. They have developed a model for analogical reasoning called *structural mapping theory*. In the mapping stage, a symbolic match is formed between the base and target descriptions. Such a mapping is then used to make predictions about the target domain from known properties of the base domain. Gentner's work has focused on modeling famous scientific analogies such as Rutherford's solar system model of the atom and the fluid flow model of electricity, and also analogies used in cognitive experiments. Gentner uses what is known as a restrictive notion of analogy match; matches must be isomorphisms between parts of the base and target descriptions, with relations being matched only with identical



relations. Gentner has also considered the problem of base filetering: that is the determination of a small set of plausible base problems/descriptions from a potentially vast base of descriptions.

### **Kedar-Cabelli**

Kedar-Cabelli [102] introduces *purpose* as an additional constraint on a structure-mapping mechanism. Their motivation is that the similar purpose for two problems often lead to similar solutions. With the constraint purpose, the analogical reasoning system needs to generate causal structures of both base and target problems for the mapping process. Purpose-directed analogy satisfies the constraint of mapping only those causal relations that share similar purposes.

### **Owen**

Owen [103] addresses the use of analogy to guide the search in theorem proving systems. The goal has been to develop mechanisms for constructing and exploiting simple analogies between mathematical problems. As with others work on genuine problem solving analogies, a flexible notion of an analogy match is necessary. Much of his effort has been towards a more flexible, powerful, and unstandable matching algorithm than those of Munyer and Kling.

## **9.2.2 Analogy in Software Engineering**

### **Dershowitz**

Dershowitz [104] suggested the formulation of program of analogies as a basic tool in program abstraction. An analogy is first sought between the specification of the given programs; this process yields an abstract specification that may be instantiated to any of the given concrete specifications. The analogy is then used as a basis for transforming the existing program into abstract schemas helping to verify and complete the analogy. A given concrete specification of a new problem may then be compared with the abstract specification of the schema to suggest an instantiation of

L  
E  
sp  
tr  
de  
co  
aut

the schema that yields a correct program.

### **Goldberg**

Goldberg [105] explores how an implementation of a modified specification can be realized by replaying the transformational derivation of the original implementation and modifying it as required by changes made to the specification. They structure derivations using the notion of tactics and record derivation histories as an execution trace of the application of tactics. One key idea is that tactics are compositional: higher level tactics are constructed from more rudimentary ones using defined control primitives. Given such a derivation history and a modified specification, the correspondence between program parts of the original and modified program is established. Their approach uses a combination of name association, structural properties, and associating components to one another by intensional descriptions of objects defined in the transformations themselves.

### **Lubars, et al**

The ROSE-2 project [106] is based on the knowledge-based refinement paradigm, which is a software development process in which user-supplied requirements are used to select and customize a high-level design. The paradigm is supported by a knowledge base of high-level design abstractions called *design schemas* and refinement rules. The schemas and rules are used to customize the user's designs to satisfy the user's requirements and design decisions.

### **Bhansali**

Bhansali [107] describes the derivation of a concrete program from a semi-formal specification of a problem. He used a transformational approach based on a set of transformational rules that produce a top-down decomposition of a problem statement down to the level of target language primitives. The top-down decomposition process combines ideas from AI research in planning to generate programs efficiently. The author emphasizes the reuse of domain specific knowledge. APU is a system which

uses the proposed paradigm to synthesize UNIX programs (shell scripts) from semi-formal specifications of programs. He proposes an analogy approach to automate software derivation. . . A knowledge base contains formal and informal specifications from past experience. The formal specification is described in pre- and postconditions. In order to describe a target problem, its informal specification is provided in the form of a systematic representation of information about the domain: objects, attributes, relations between objects, and problem descriptions. The informal specification is analyzed and compared to extract analogies. An analogy algorithm is proposed to detect the analogies from the knowledge base, which is then used to derive the formal specification for the target problem.

#### **Maiden et al**

Maiden and Sutcliffe [108] investigate the potential of specification reuse by analogy and its possible benefits for requirements analysis. They have developed two non-simple examples to examine the potential for specification reuse by analogy. The first example illustrates an analogy between air-traffic controller (ATC) and a flexible manufacturing system (FMS). The second example shows analogy between the ATC and a classroom administration system (CAS), and the FMS and the CAS. They propose a software engineering analogy model based upon three types of knowledge: solution knowledge, domain knowledge, and goal knowledge.

#### **Lung et al**

Lung and Urban [109] have proposed an analogy model for software reuse. In addition to the constraints proposed by Maiden and Sutcliffe, more constraints are added for software analogy analysis due to the complexity of the software system. They have proposed an analogy-based domain analysis method that can support a high level reuse across domains [110]. The purpose is to help users better understand a domain and support potential future reuse in a different domain. Their method generalizes that of Maiden and Sutcliffe.

**Coen-Porisini et al.**

Alberto Coen-Porisini, et al. [111] developed a technique and an environment-supporting specialized software components. The technique is based on symbolic execution. It allows one to transform a more general software component into a more specific and more efficient component. Specialization is motivated as a technique to facilitate software reuse. They assume that a library of generalized components exists and the environment supports a designer in customizing a generalized component when the need arises for it under more specialized conditions.

### **9.3 Related Work of Specifying GUI**

There are several Larch Shared Language available, e.g. Larch Handbook and the extended example of simple windowing system in [76]. Larch also has been used to specify properties of objects in transaction-based distributed systems [77, 41]. Many formal specifications have been proposed to describe computer graphics systems [112, 113, 114, 115, 116, 117, 118, 119, 120] however none of them specify, implementation-independent window systems. Mallgren [121, 122] introduces a lower-level specification for the computer graphics languages, e.g. geometry and color transformations, but the specification of higher-level GUI interface like window widgets is not provided. The similar work closely related to [122, 121] is that by Guttag and Horning [123]. They have designed a hierarchy of types to serve as the basis for display device interface and specified them algebraically. Sufrin [124] presents the formal specification of a simple, display-oriented text editor. It is an early effort to incorporate formal specification in the design stage. Although their editor specification is tedious, the experience can be used for future attempts of specifying GUI in a formal means. Narayana and Dharap [125] give a formal specification of the look manager of a dialog system. The look manager deals with the presentation of visual aspects

of objects and the editing of those visual objects. They provide a formal model for specifying the look of the objects. Jacob [126, 127] describes the specification of the user interface module for the family of message systems and provides the surveys of formal specification techniques that can be applied to human-computer interfaces.

# **CHAPTER 10**

## **Concluding Remarks**

In this chapter, we summarize the contributions of this work and discuss potential future investigations.

### **10.1 Summary**

This dissertation contributes to the field of software engineering and component reuse along the following dimensions:

1. Construction of component hierarchy.
2. Incorporation of formal methods to software reuse.
3. Retrieving reusable components from the component hierarchy.
4. Modifying more general components.
5. Modifying analogous components.

Each of these contributions are discussed in further detail.

#### **10.1.1 Construction of Component Hierarchy**

We have proposed classification schemes and algorithms for automatically constructing a hierarchy of software components that provide a means for representing, storing,

browsing, retrieving and modifying reusable components. The hierarchical relationships of the software components based on a *generality* relationship and similarities between software components. The similarities are calculated with respect to a partition of operators into equivalence classes. The resulting library structure consists of lower-level and higher-level hierarchies. The lower-level hierarchy is created by a *subsumption test algorithm* that determines whether one component is more general than another. Based on the *generality* relationship, the most general components are placed at the top of the hierarchy and the more detailed or restrictive components at the bottom. All the generated components undergo another grouping scheme to yield the higher-level hierarchy. The higher-level hierarchy is generated by a classical *hierarchical clustering algorithm* that groups the most syntactically similar components together. The end result is a connected hierarchy of software components organized from the most general to the most specific.

### 10.1.2 Incorporation of Formal Methods to Software Reuse

Using formal specifications to represent software components facilitates determination of reusability because the formal representation more precisely characterizes the functionality of the software. In addition, the well-defined syntax of formal specification languages makes processing amenable to automation. In Chapter 8, a case study involving the specification of X window widgets is described to demonstrate how to perform reuse for solving a real-world problem.

### 10.1.3 Retrieving Reusable Components from the Hierarchy

Based on the two-tiered hierarchy of reusable components and formal description of each component, we can easily locate the components that have the *generality* rela-



1  
If  
w  
th  
be  
re  
th  
ha

tionship with a user's query that is also represented by a formal specification. This approach is in contrast to another popular method used today, classification schemes. The classification scheme approach attempts to store and retrieve components based on attributes whose values are selected from a finite set of keywords. Retrieving components, hence, needs some organizational knowledge of the software structure and the knowledge of the keyword set. However, query by formal specification requires no knowledge of the software component hierarchy from the user.

#### 10.1.4 Modifying More General Components

From the framework of a two-tiered hierarchy of reusable software components, the candidate components that are more *general* than the query specification will be retrieved from the hierarchy. A more general retrieved component is compared to the query specification to determine what changes need to be applied to the corresponding program component in order to make it satisfy the query specification. The weakest preconditions of the retrieved specifications with respect to the corresponding program component are obtained and returned to the users to provide information to guide the program modification.

#### 10.1.5 Modifying Analogous Components

If there are no components in the hierarchy exhibiting the *generality* relationships with the query specification, then the candidate components that are *analogous* to the query specification are retrieved from the hierarchy. A set of analogical *matches* between an analogous retrieved component and query specification are computed and returned to the user. Based on the information, the user can determine what part of the program needs to be modified to satisfy the query specification. This dissertation has described a new approach to modifying the analogous components, the relevant

issues about this approach, and several examples to demonstrate the usefulness of this approach.

## 10.2 Future Work

Based upon the framework that has been developed thus far, the approach of construction and retrieval processes will be extended in several aspects. First, new techniques are being developed that may be more efficient than the currently used approach for determining functional similarity between two software components. The abstraction scheme for forming meta nodes of software components will also be further investigated. The formal representation for the inheritance relationship and the genericity of software components will be further investigated in order to better exploit the properties of object-oriented development.

Regarding program modification, several issues need to be addressed in the future. As mentioned in Chapter 7, program modification is a combination of analogical reasoning, verification, transformation, and synthesis. These features should be supported by an integrated system that contains programming tools, verification assistant, and a domain-specific knowledge base. Some guidelines for programming by modification need to be provided to the programmers in order to support the development of correct programs. Some rules such as *wp* semantics are required to assist the programmers verify that the “modified” program based on the results of the matching process. Another set of rules may be needed to assist the programmers update the unexecutable statements, i.e., change the non-primitive operators into primitive operators.

As for the matching process, we recognize some problems should be solved before the potential benefits of the matching process can be fully exploited. Supporting the understanding of existing programs is an important factor to the successful



specification-level reuse. An ARMP needs didactic support for comprehension of candidate specifications, which requires an explanation facility to help the software developer understand the query domain and the candidate specifications. Since an automated ARMP is unlikely to achieve a perfect match, explanation will also be necessary for evaluating the appropriate query specifications.

Regarding the relationship of the matching process and the two-tiered hierarchy used in the construction and retrieval processes, we have the following problems need to be solved:

- Apply the matching process to the abstraction of two real nodes into a meta node. Currently, the meta node is just an aggregation of its child nodes. If the abstraction is possible, then we can develop a middle layer in the hierarchy and the nodes of this layer represent the program schemata of their child nodes.
- Select a set of methods that are similar to the query specification and supply them to the matching process as candidate specifications. We now use the notion of *similarity* as a guidance, but we wish to have a selection scheme that is beyond the syntax-based approach.

Regarding the reuse system, we are investigating the integration of the reuse framework into a software development environment comprising tools for formal specification editing [29, 128], program visualization from formal specifications [129], and a tool that abstracts formal specifications from program code [130, 131].

# **APPENDICES**

# APPENDIX A

## Analogical Reasoning

In this appendix, we describe both *normative* and *cognitive* accounts of analogical reasoning [103]. By normative accounts, we mean those that attempt to provide an analytical justification for reasoning by analogy. In contrast, cognitive accounts refer to those that attempt to model human analogical reasoning and hence gain insight into how people are able to benefit from analogy.

Philosophers since Aristotle have tried to understand analogy by considering what reason we have for believing an analogy is useful. Hesse's theory of analogy [65] addresses the use of analogical models to make predictions in science. When scientists are investigating some new and unfamiliar system (the target system), with only a few properties of the system known to them, an analogy with a familiar system (the base system) may be applied in order to predict new properties of the unfamiliar system. The most famous examples are Newton's particle theory and Rutherford's solar system model of the atom. Typically, the base problem is well understood and there is a successful theory with the observed properties. The model is used to predict the target system because the corresponding properties are known to be true of the base.

Recently, cognitive psychologists have attempted to model human analogical reasoning [100]. Early models tended to concentrate on isolated analogy like the IQ test.

More recent models explain analogy as part of a wider human reasoning. For example, Gentner's structure mapping theory (SMT) has presented a complete analogical reasoning model from the computational point of view. As illustrated in Figure A.1, the analogy retriever looks for potential analogies for the current state of working memory and past experiences stored in long term memory. Potential analogies are then passed to the analogy engine. The engine performs full matching on the potential analogies and makes candidate inferences about the target situation and structural evaluation of the strength of the analogy.

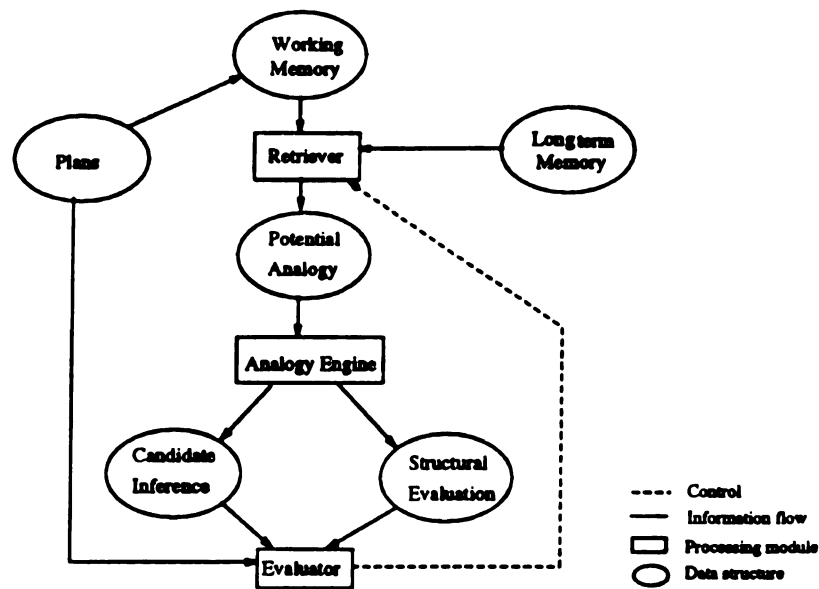


Figure A.1. Gentner's Model of Analogy

---

Gentner's model construes analogy matching and analogical inference as being isolated syntactic operations, i.e., it does not allow semantic or pragmatic factors to directly influence the analogy process. The PI model of Hoyoak and Thagard [132] takes a different approach to analogy. PI is a spreading activation model in which



currently active concepts, e.g. those involved in the current goals, spread activation to other concepts with which they are associated. As shown in Figure A.2, analogy is tightly coupled with normal problem solving. The loop in the figure represents the principal cycle of rule retrieval and application in PI. If the current goal stored in working memory has not been solved yet, then production rules that are relevant to the problem may be retrieved and matched. The use of analogy suggests plausible rules to fire for the current state, thus distinguishing PI from SMT [100].

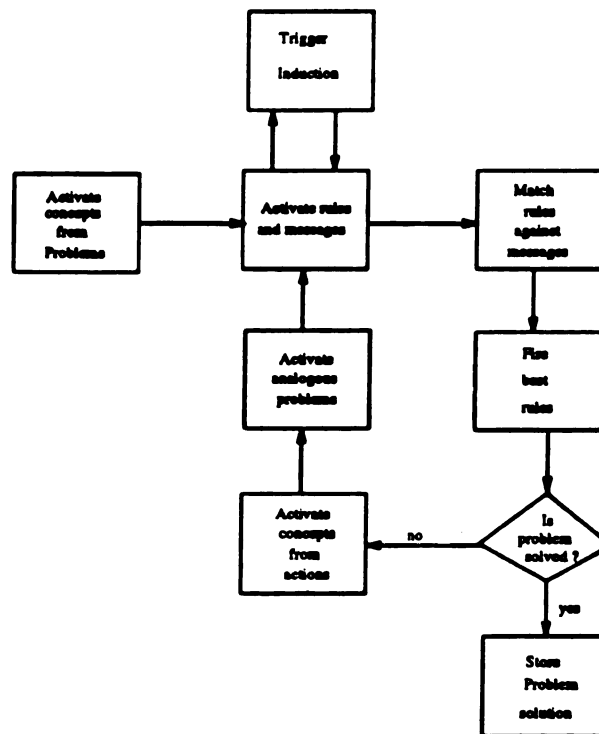


Figure A.2. PI Model

---

As described in Section 7.1, the goal of ARMP is to enable a program synthesizing system to reuse existing software components similar to a query specification. Thus ARMP may be considered to be a type of machine learning. A popular machine learn-

ing system

solving be

tion. Face

that sub

ate the

system

new pro

knowle

domain

exper

ing system is the *generalization system*. That is, given a set of examples of problem solving behavior, it forms a generalization (schema) of both the problem and the solution. Faced with a new problem, the generalization system looks for a generalization that subsumes the new problem. When one is found, the system tries to instantiate the generalization solution to solve the given problem. In contrast, the analogy system makes no generalization, but, instead, attempts to relate old problems with new problems by *matching*. The generalization systems rely on sophisticated domain knowledge. When such knowledge is lacking or insufficient, for example when a new domain is to be explored, the analogy system could provide a means for reusing past experience. The relationships of analogy and generalization are shown in Figure A.3.

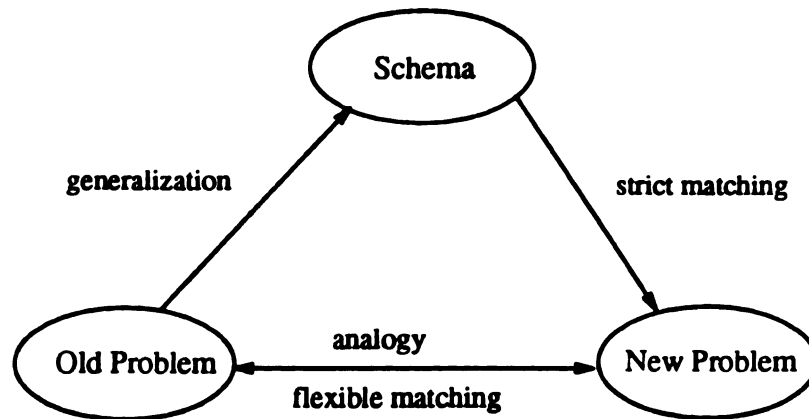


Figure A.3. Analogy and Generalization

---

# APPENDIX B

## Bottom-Up Matching Approach

The algorithm presented in Section 7.5 is a top-down matching algorithm that emphasizes the higher-level predicate or function symbols. However, for the matching problem, *bottom-up* matching is an alternative approach. It is believed that specifications can be represented as trees and the symbolic computation, analogical matching in our case, can be described as tree replacement. In the papers [133, 134, 135], the authors' approach to automatic proving of equational theorems is to treat a set of equational axioms as replacements rule and transform one side of the equation to be proved into the other by a sequence of tree replacements. Knuth and Bendix [55] discuss some of the cases in which tree replacements yield efficient theorem provers. We present formally the tree-matching problem, but the detailed bottom-up tree-patterning algorithms will not be discussed in this report. Suppose we are given a finite ranked alphabet  $\Sigma$  of function symbols, including constants as nullary functions.  $S$  denotes the set of  $\Sigma$ -terms, formally defined as follows.

**Definition B.1**  $\Sigma$  terms:

- (1) For all  $b$  in  $\Sigma$  of rank 0,  $b$  is a  $\Sigma$ -term.
- (2) If  $a$  is a symbol of rank  $q$  in  $\Sigma$ , then  $a(t_1, \dots, t_q)$  is a  $\Sigma$ -term provided each of the  $t_i$  is.
- (3) Nothing else is a  $\Sigma$ -term.

A nullary symbol  $\nu$  is used as a placeholder for a  $\Sigma$ -tree. We define the set  $\Sigma \cup \{\nu\}$  just as  $\Sigma$ -terms, denoted by  $S_\nu$ .

**Definition B.2** *A tree pattern is any term in  $S_\nu$ . If  $b(t_1, \dots, t_q)$  is a term, then define  $\text{child}_i(t_1, \dots, t_q)$  to be  $t_i$  for  $1 \leq i \leq q$ .*

**Definition B.3** *A pattern  $p$  in  $S_\nu$  with  $k$  occurrences of the symbol  $\nu$  matches a subject tree  $t$  in  $S$  at node  $n$  if there exist  $\Sigma$ -trees  $t_1, \dots, t_k$  in  $S$  such that the  $\Sigma$ -tree  $p'$ , obtained from  $p$  by substituting  $t_i$  for the  $i$ th occurrence of  $\nu$  in  $p$ , is equal to the subtree of  $t$  rooted at  $n$ .*

**Definition B.4** *The Tree Matching Problem.*

*A tree matching problem consists of a finite set of patterns  $p_1, \dots, p_k$  in  $S_\nu$  and a subject tree in  $S$ . A solution to a matching problem is a list of all pairs  $\langle n, i \rangle$ , where  $n$  is a node in  $t$  and  $p_i$  matches at  $n$ .*

All suggested matched methods for tree matching should be compared to the naive algorithm (based on the simple form of unification), which merely tries every pattern at every position in the subject tree. The main objectives of the bottom-up matching algorithm is to find, at each point in the subject tree, all patterns and all parts of patterns that match at this point. Let  $n$  be a node in the subject tree labeled with the  $q$ -ary symbol  $b$ , and suppose we wish to compute the set  $M$  of all those pattern subtrees other than  $\nu$ . Suppose we have already computed such sets for each of the children of  $n$ , and call these sets, from left to right,  $M_1, \dots, M_q$ . Then  $M$  contains  $\nu$  plus exactly those pattern subtrees  $b(t_1, \dots, t_q)$  such that  $t_i$  is in  $M_i$ , for  $1 \leq i \leq q$ . Therefore, we could compute  $M$  by forming trees  $b(t_1, \dots, t_q)$  for all combinations of  $(t_1, \dots, t_q)$ , where  $t_i$  are chosen from  $M_i$ , and then asking whether each candidate for membership in  $M$  is a subpattern. Once we have assigned these sets to each node in the subject tree, then the tree matching problem is solved.

# APPENDIX C

## The Specifications for Motif Widgets

### C.1 The Functionalities of Motif Widgets

This appendix contains the functionalities of Motif window widgets and formally describes a set of Motif window widgets, which include their LSL traits and several examples of the LIL specifications for application widgets.

#### Display Widgets

Motif provides many widgets whose primary purpose is to display information or interact with the user. These widgets include:

<code>XmArrowButton</code>	<code>XmDrawButton</code>
<code>XmList</code>	<code>XmPushButton</code>
<code>XmSeparator</code>	<code>XmText</code>
<code>XmScale</code>	<code>XmCreateButton</code>
<code>XmLabel</code>	<code>XmToggleButton</code>
<code>XmScrollBar</code>	<code>XmTextField</code>

#### Container Widgets

Motif provides many widgets that can be used to combine other widgets into composite panels. Composite widgets allow endless combinations of buttons, scrollbars, text

panes, and so on, to be grouped together in an application. These widgets include:

<b>XmDrawingArea</b>	<b>XmFrame</b>
<b>XmRowColumn</b>	<b>XmForm</b>
<b>XmPanedWindow</b>	<b>XmBulletinBoard</b>
<b>XmMainWindow</b>	<b>XmScrollWindow</b>

### **Menu Widgets**

Motif provides many widgets that allow applications to create popup, pulldown and option menu. A menu pane consists of a popup Shell widget that manages an XmRowColumn widget containing buttons, labels, and other types of widgets. The buttons are selectable entries in the menu. By combining different types of widgets, the programmer can create different types of menus: pulldown, popups, cascading pulldowns, cascading popups, option menus, and so on. The following widgets can be used to construct menus:

<b>XmRowColumn</b>	<b>XmToggleButton</b>
<b>XmPushButton</b>	<b>XmCascadeButton</b>
<b>XmLabel</b>	<b>XmSeparator</b>

### **Dialog Widgets**

The Motif widget set builds on the underlying popup facilities provided by the Xt Intrinsics to provide a versatile set of dialog widgets. Motif uses a subclass of the Shell widget class, the XmDialogShell widget class directly because Motif provides convenience functions that create different types of dialogs and popups. Most Motif widgets know they are the children of the XmDialogShell widget class and automatically popup up and down the parent shell when managed and unmanaged. The following are the Motif widgets classes designed to be used as dialog widgets.

<b>XmBulletinBoard</b>	<b>XmFileSelectionBox</b>
<b>XmCommand</b>	<b>XmForm</b>
<b>XmMessageBox</b>	<b>XmSelectionBox</b>

By setting appropriate resources, many types of dialogs can be created from these

basic widget classes. Motif widgets provide a large set of convenience routines to make dialogs easier to create and use. These convenience functions create the following types and dialogs:

BulletineBoardDialog	ErrorDialog
InformationDialog	MessageDialog
SelectionDialog	WarningDialog
FormDialog	QuestionDialog
FileSelectionDialog	WorkingDialog
PromptDialog	

These dialogs are not really *new* widget classes, but are combinations of an `XmDialogShell` widget and one of the manager widgets described earlier in this section.

### Gadgets

In addition to widgets, Motif provides user interface components known as *gadgets*. Gadgets are nearly the same as widgets, except that they have no windows of their own. A gadget must display text or graphics in the window provided by its parent, and also must rely on its parent for input. The Motif gadget classes include:

<code>XmArrowButtonGadget</code>	<code>XmLabelGadget</code>
<code>XmSeparatorGadget</code>	<code>XmToggleButtonGadget</code>
<code>XmPushButtonWidget</code>	<code>XmCascadeButtonGadget</code>

Nearly all widgets can be customized by the user or the programmer. Widgets are reuse and customized by specifying values for various resources by the widget. Based on the set of Motif widgets, the programmer can create any reusable customized GUI components with desired behavior.

## C.2 The LSL Traits of Motif Widgets

This appendix describes the Larch traits of the widgets and gadgets available in Motif toolkits. Since trait only describes state-independent properties of target objects, the state dependent specifications are described in next section. Because of limited space,



not all developed Motif widgets will be formally described here. The prefix of the Larch traits is “Lt” which stands for “Larch Trait”.

## 1. Basic Window

```
LtBasicWindow: trait
  assumes LtCoordinate
  includes LtRegion, LtView, Displayable(Window)
  asserts Window tuple of pos: Coord,
                                size: Size,
                                cont, clip: Region,
                                front, back: View,
                                id: WId
  forall w: Window, cd: Coord
    cd ∈ w == cd ∈ w.clip
    w[cd] == if cd ∈ w.cont w.front else w.back
  implies converts [-], ∈: Coord, Window → Bool
```

## 2. Color

```
LtColor(C): trait
  includes Set, LtPizelMap
  introduces
    color1: → C
    color2: → C
    :
    colorn: → C
    equal: C, C → Bool
    -⊕-: C, C → C
    -⊖-: C, C → C
```

## 3. Compound String

```
LtCompoundString(CS): trait
  includes LtString(CS, CSSet)
  introduces
    conver_c_string: String → CS
    deconvert_c_string: CS → String
```

## 4. Coordinate

```

LtCoord: trait
  introduces
    origin  $\rightarrow$  Coord
     $--: \text{Coord}, \text{Coord} \rightarrow \text{Coord}$ 
  asserts forall cd: Coord
    cd - cd == origin

```

## 5. Displayable

```

LtDisplayable(T): trait
  assumes LtCoord
  includes LtRegion(T)
  introduces
     $[-]: T, \text{Coord} \rightarrow \text{Color}$ 

```

## 6. label

```

LtLabel(L): trait
  includes LtCompoundString(CompString)
  introduces
    set_label: L, CompString  $\rightarrow$  L
    get_label: L  $\rightarrow$  CompString
    delete_label: L, CompString  $\rightarrow$  L

```

## 7. Line

```

LtLine(Line): trait
  includes Set(Line for E, LineSet for C), LtCoord(Coord)
  introduces
    create Coord, Coord  $\rightarrow$  Line
    delete Line  $\rightarrow$ 
    meet Line, Line  $\rightarrow$  Coord
    parallel Line, Line  $\rightarrow$  Bool

```

## 8. Manager Window

```

LtManager(M): trait
  includes LtView
  assumes LtPrimitive
  M tuple of pos: Coord,

```

*size: Size,*  
**introduces**  
*create:  $\rightarrow M$*   
*realize:  $\rightarrow M$*   
*size:  $M \rightarrow \text{Size}$  % observers*  
*pos:  $M \rightarrow \text{Coord}$*   
*id:  $M \rightarrow \text{Integer}$*   
*child:  $M, \text{Real} \rightarrow \text{Widget}$*   
*child\_size:  $M \rightarrow \text{Size}$*   
*child\_pos:  $M \rightarrow \text{Coord}$*   
*realize\_child:  $\text{Widget} \rightarrow \text{View}$*   
*derealize\_child:  $\text{Widget} \rightarrow \text{View}$*   
**asserts**  
 *$M$  generated by create*  
 *$M$  partitioned by id*

## 9. Pixel Map

*LtPixelMap: trait*  
**includes** *LtRegion( $R$ )*  
**asserts**  
*Pixel tuple of int: Intensity,*  
*pos: Coord*  
*PixelMap enumeration of Pixel*

## 10. Primitive Window

*LtPrimitive: trait*  
**includes** *LtBasicWindow, LtView, Set( $\text{Widget}$ ,  $\text{WidgetSet}$ )*  
*Widget tuple of pos: Coord*  
*size: Size*  
*id: Real*  
**introduces**  
*create:  $\rightarrow \text{Widget}$*   
*destroy:  $\rightarrow \text{Widget}$*   
*open:  $\text{Widget} \rightarrow \text{View}$*   
*quit:  $\text{Widget} \rightarrow \text{View}$*   
*selected:  $\text{Widget} \rightarrow \text{Bool}$*   
*move:  $\text{Widget}, \text{Coord} \rightarrow \text{Widget}$*   
*size:  $\text{Widget} \rightarrow \text{Size}$  % observers*  
*set\_size:  $\text{Widget}, \text{Size} \rightarrow \text{Widget}$*   
*pos:  $\text{Widget} \rightarrow \text{Coord}$*   
*set\_pos:  $\text{Widget}, \text{Coord} \rightarrow \text{Widget}$*   
*overlap:  $\text{Widget}, \text{Widget} \rightarrow \text{Bool}$*   
*id:  $\text{Widget} \rightarrow \text{Integer}$*   
*parent:  $\text{Widget} \rightarrow \text{Widget}$*

```

    child: Widget, Real → WidgetSet
    is_child: Widget, Widget → Bool
asserts
    ∀ w: Widget
        move(w).pos == coord_map(w.pos)
        move(w).size == w.size
        move(w).id == w.id
        resize(w).id == coord_map(w.pos)
        resize(w).id == length_map(w.size)
        resize(w).id == w.id
    w generated by create
    w partitioned by id

```

## 11. Region

```

LtRegion(R): trait
assumes LtCoord
introduces
    _∈_: Coord, R → Bool
    nullregion: → R
    point: Coord → R
    lineseg: Coord, Coord → R
    intersec: R, R → R
    union: R, R → R
    difference: R, R → R
    contains: R, R → Bool
    equal: R, R → Bool
    empty: R → Bool

```

## 12. Scrolled Text Window

```

LtScrolledText(ScTxt): trait
includes LtScrolled, LtText
introduces
    all_visible: ScTxt → Bool
    visible: ScTxt, View → Bool
    visible_content: ScTxt → View
asserts
    all_visible(visible_content(ScTxt)) == true
    visible(ScTxt, visible_content(ScTxt)) == true

```

## 13. Scrolled Text Editor with Menu

```

LtScrolledTextMenu(ScrolledTextMenu): trait

```

```

includes LtScrolledText, LtMenu
introduces
  with_popupmenu: ScrolledTextMenu → Bool
  with_menubar: ScrolledTextMenu → Bool
  label: ScrolledTextMenu → String
asserts
  label(set_attr(ScrolledTextMenu, Label, Value)) == Value
  with_popupmenu(set_attr(ScrolledTextMenu, Popup, -)) == true
  with_menubar(set_attr(ScrolledTextMenu, MenuBar, -)) == true

```

## 14. Scrolled Window

```

LtScrolled: trait
includes LtView, LtManager, LtPrimitive
Scrolled tuple of type: {vertical, horizontal},
  policy: {automatic, app_defined},
introduces
  create: → Scrolled
  destroy: Scrolled →
  scroll_up: Scrolled → Scrolled
  scroll_down: Scrolled → Scrolled
  scroll_left: Scrolled → Scrolled
  scroll_right: Scrolled → Scrolled
asserts
  scroll_up(create).cont == Empty
  scroll_down(create).cont == Empty
  scroll_left(create).cont == Empty
  scroll_right(create).cont == Empty
  scroll_up(scroll_down).cont == scroll_down(scroll_up).cont
  scroll_left(scroll_right).cont == scroll_right(scroll_left).cont

```

## 15. SetResource

```

LtSetResource(Attr, Val): trait
includes Set(Attr for E, AttrSet for C), LtObject(Ob for Obj)
introduces
  valid_attr: Ob, Attr → Bool
  valid_value: Attr, Val → Bool
  get_value: Ob, Attr → Val
  set_attr: Ob, Attr, Val → Ob
asserts
  get_value( set_attr (Ob, Attr, Val), Attr') ==
    if equal(Attr,Attr')
    then Val
    else get_value(Ob, Attr')

```

## 16. Size

```

LtSize(S): trait
  includes Real
  introduces
    create_size Real, Real  $\rightarrow$  S
    null_size  $\rightarrow$  S
    equal_size S, S  $\rightarrow$  Bool

```

## 17. Text Window (Text Editor)

```

LtText(Text): trait
  includes LtTextField, LtView, LtLine (LN for Line_Number)
    LtCharacter (CN for Character_Number), LtPrimitive
  introduces
    cursor: LN, String, String  $\rightarrow$  Text
    create:  $\rightarrow$  Text
    insert: Text, Stream  $\rightarrow$  Text
    delete: Text, Stream  $\rightarrow$  Text
    move_left: Text  $\rightarrow$  Text
    move_right: Text  $\rightarrow$  Text
    move_up: Text  $\rightarrow$  Text
    move_down: Text  $\rightarrow$  Text
    first: Text  $\rightarrow$  LN
    last: Text  $\rightarrow$  LN
    line:  $\rightarrow$  LN
    contents: Text, LN  $\rightarrow$  Stream
    clear: Text  $\rightarrow$ 
  asserts forall c: Char; cn:CN; s,s1,s2: Stream; n, n1, n2: LN; t: Text
    first(create) == Empty
    last(create) == Empty
    first(cursor(s1,s2)) == cursor(Empty,concat(s1,s2))
    last(cursor(s1,s2)) == cursor(concat(s1,s2),Empty)
    insert(c,cursor(s1,s2)) == cursor(append(s1,c),s2)
    delete(cursor(s1,Empty)) == cursor(s1,Empty)
    delete(cursor(s1,append(c,s2))) == cursor(s1,s2)
    move_left(cursor(Empty,s)) == cursor(Empty,s)
    move_left(cursor(append(s1,c),s2)) == cursor(s1,append(c,s2))
    move_right(cursor(s,Empty)) == cursor(s,Empty)
    move_right(s1,cursor(append(c,s2))) == cursor(append(s1,c),s2)
    line(move_up((cursor(s1,s2)))) ==
      if equal(line(cursor(s1,s2)),1)
      then 1 else line(cursor(s1,s2)) - 1
    line(move_down((cursor(s1,s2)))) ==
      if equal(line(cursor(s1,s2)),line.max)

```

```

    then line.max else line(cursor(s1,s2)) + 1
clear(t) == cursor(Empty,Empty)

```

## 18. Text Field (Line Editor)

***LtTextField(TF): trait***

**includes** *LtView, String, LtCharacter, LtPrimitive*

**introduces**

*cursor: String, String → TF*

*create: → TF*

*insert: TF → TF*

*delete: TF → TF*

*move\_left: TF → TF*

*move\_right: TF → TF*

*first: TF → Char*

*last: TF → Char*

*clear: → TF*

**asserts forall** *c: Char; s,s1,s2: String; tf: TF*

*insert(c,cursor(s1,s2)) == cursor(append(s1,c),s2)*

*delete(cursor(s1,Empty)) == cursor(s1,Empty)*

*delete(cursor(s1,append(c,s2))) == cursor(s1,s2)*

*move\_left(cursor(Empty,s)) == cursor(Empty,s)*

*move\_left(cursor(append(s1,c),s2)) == cursor(s1,append(c,s2))*

*move\_right(cursor(s,Empty)) == cursor(s,Empty)*

*move\_right(s1,cursor(append(c,s2))) == cursor(append(s1,c),s2)*

*first(cursor(s1,s2)) == cursor(Empty,concat(s1,s2))*

*last(cursor(s1,s2)) == cursor(concat(s1,s2),Empty)*

*clear(tf) == cursor(Empty,Empty)*

## 19. View

***LtView(View): trait***

**includes** *LtCoord, LtSize, LtWidget, PixelMap*

**introduces**

*create: → View*

*quit: View →*

*size: View → Size*

*pos: View → Coord*

*parent: View → Widget*

*empty: View → Bool*

*intensity: View → Real*

*coord\_map: Coord → Coord*

*length\_map: Size → Size*

*display: View → PixelMap*

## 20. Well Formed Widgets

```

LtWFWidget(WFWidget): trait
includes LtPrimitive, Set(Widget, WidgetSet), LtSetResource, Object
introduces
  ancestor: Widget, Widget → Bool
  descendant: Widget, Widget → Bool
  is_well_formed: Widget → Bool
  managed_by: Widget, Widget → Bool
  extract_wf: ObjSet → WidgetSet
  delete_wf_widget: Widget, Widget → Widget
  delete_objs: Widget, ObjSet → Widget
  delete_children: Widget → Widget
asserts for all w, w': WFWidget
  is_well_formed(w) ==
    (managed_by(w, parent(w)) ∧
     (for all w'::descendant(w', w) ⇒ ¬ ancestor(w', w)))
  parent(w, w') == child(w', w)
  ancestor(w, w') == descendant(w', w)
  for all w' ∈ extract_wf(w) ⇒ is_well_formed(w')

```

## C.3 The LIL Specifications of Widgets

This appendix describes the Larch interface specifications of the widgets and gadgets available in Motif toolkits. They are high-level specifications written using the Larch Generic Interface Language (GIL). Mice or other indicators are not explicitly mentioned. Note that a ' ('prime') is the new-value operator. That is, if *x* is a variable, then *x'* represents its value after the application of a transition. An unprimed *x* represents the old-value of *x*. We restrict operations to be pure functions; that is, each operation returns a value that depends only on the parameters of the call, and there are no side effects, either on the parameters or any other part of the system state.

### 1. Primitive

```

component PrimitiveWidget
  based on Widget from

```



*LtPrimitive Widget*

```

method Select(w: Widget)
  requires (w ∈ DisplayableSet)
  modifies (w.view)
  ensures selected(w') == true

method Unselect(w: Widget)
  requires true
  modifies (w.view)
  ensures selected(w') == false

method Move(w: Widget, delta: Coord)
  requires true
  modifies w
  ensures selected(w) == true ⇒
    (pos(w') == pos(w) + delta) ∧ selected(w') == false

method Resize(w: Widget, pos: Coord, size: Size)
  requires selected(w)
  modifies w
  ensures w' == set_size(set_pos(w,pos),size) ∧ selected(w') == false

method ChangeAttr(w: Widget, attr: Attr, val: Val)
  requires valid_attr(w, attr) ∧ valid_value(attr,val)
  modifies w.attr
  ensures w' = set_attr(w,attr,val)

method Open(w: Widget)
  requires selected(w) ∧ displayable(w) == false
  modifies w.view
  ensures displayable(w') == true

method Close(w: Widget)
  requires selected(w) ∧ displayable(w) == true
  modifies w.view
  ensures displayable(w') == false

method Delete(w: Widget)
  requires selected(w)
  modifies w
  ensures selected(w') == undefined

```

## 2. Popup

**component** *Popup*  
 inherits *Primitive*

**method** *Push*(*p*: *Popup*)  
 requires  $\neg \text{displayable}(p)$   
 modifies *p.view*  
 ensures  $\text{displayable}(p') \wedge \text{selected}(p) \wedge \text{top\_level}(p)$

**method** *Release*(*p*: *PopupWidget*)  
 requires  $\text{displayable}(p)$   
 modifies *p.view*  
 ensures  $\neg \text{displayable}(p') \wedge \neg \text{selected}(p)$

### 3. Popupmenu

**component** *Popupmenu*  
 inherits *Popup*  
 uses *Menu* from *LtMenu*

**method** *CreateItem*(*p*: *Popupmenu*, *i*: *Item*, *e*: *Event*)  
 requires  $i \notin \text{ItemSet}(p)$   
 modifies *p*  
 ensures  $i \in \text{ItemSet}(p) \wedge \text{associated}(i, e)$

**method** *DeleteItem*(*p*: *Popupmenu*, *i*: *Item*)  
 requires  $i \in \text{ItemSet}(p)$   
 modifies *p*  
 ensures  $i \notin \text{ItemSet}(p)$

**method** *Choose*(*p*: *Popupmenu*, *i*: *Item*)  
 requires  $\text{selected}(p) \wedge \neg \text{selected}(i)$   
 modifies *i*  
 ensures  $(\text{associated}(i, e) \Rightarrow \text{fire}(e) \wedge \text{selected}(i))$   
 $\wedge (i \neq j \Rightarrow \neg \text{selected}(j))$

**method** *Actor*(*s*: *State*, *e*: *Event*)  
 requires  $\text{fire}(e)$   
 modifies *s*  
 ensures  $s' == \text{act}(s, e) \wedge \neg \text{fire}(e)$

### 4. List

**component** *List* based on *List* from *LtList*

**method** *Choose*(*w*: *List*, *i*: *Item*, *e*: *Event*)

**requires**  $\neg \text{selected}(i) \wedge \text{selected}(p)$   
**ensures**  $\text{associated}(i,e) \Rightarrow \text{fire}(e)$

**method** *CreateItem*(*w*: *List*, *i*: *Item*, *e*: *Event*)  
**requires**  $i \notin p$   
**modifies** *w*  
**ensures**  $i \in w \wedge \text{associated}(i,e)$

**method** *DeleteItem*(*w*: *List*, *i*: *Item*)  
**requires**  $i \in w$   
**modifies** *w*  
**ensures**  $i \notin w$

**method** *ChangeAttr*(*w*: *List*, *attr*: *xxx*, *val*: *xxx*)  
**requires**  $\text{valid\_attr}(w, \text{attr}) \wedge \text{valid\_value}(\text{attr}, \text{val})$   
**modifies** *w*  
**ensures**  $w' == \text{set\_attr}(w, \text{attr}, \text{val})$

**method** *Actor*(*w*: *List*, *e*: *Event*, *s*: *State*)  
**requires**  $\text{fire}(e)$   
**modifies** *s*  
**ensures**  $s' == \text{act}(s,e) \wedge \neg \text{fire}(e)$

# APPENDIX D

## Algorithm to find LGCs

As described in Chapter 5, given a query specification, the process of searching a set of candidate components is performed in the lower-level hierarchy that is basically a partial ordering set. The components called real nodes in the lower-level hierarchy obey the subsumption relationship ( $\supseteq_{comp}$ ) that is a partial ordering. Since reusing a more specific component by abstraction is believed to be difficult, we are interested in reusing a more general component. In order to minimize the effort needed to modify a reusable component, a *least general component* (LGC) is only sought from the lower-level hierarchy. The definition of LGC will be given shortly. In this appendix, we attempt to explore the complexity of finding a LGC from a partial ordering set, which can be regarded as the upper bound of our retrieval algorithm. Some basic definitions are presented that are necessary for the following proof.

**Definition D.1** *Least General Component (LGC).* Assume  $L$  is a set with partial ordering  $\supseteq_{comp}$ . Given a component  $x \notin L$ , if there exists a component  $y \in L$  such that  $(\forall y' \in L \setminus \{y\}: (y' \supseteq_{comp} x) \Rightarrow \neg(y \supseteq_{comp} y'))$  then  $y$  is called a *Least General Component (LGC)* for  $x$  in the set  $L$ .

Our LGC problem is specified as follows:

INS

QU

mc

col

ne

re

pe

st

in

-

A

I

C

INSTANCE: A partial ordering set  $L$  and a given component  $x$ .

QUESTION: Find a set of LGCs for  $x$  in  $L$ .

The LGC problem is just a restricted version of our problem of retrieving a set of more general components from a lower-level hierarchy given some query specification component. In practice, we do not really need to find a set of “least” general components. However, studying this LGC problem helps us to understand how difficult the retrieval problem would be if the search space is not limited explicitly.

The lower-level hierarchy can be a directed graph  $G = (V, E)$ , where each component is represented by a node in  $V$  and every edge  $(u, v) \in E$  represents the subsumption relationship  $v \sqsupseteq_{comp} u$ . The algorithm of finding a set of LGC is described in Figure D.1.

---

**Algorithm 10** *LGC*

**Input:**  $G = (V, E)$  with partial ordering  $\sqsupseteq$  and query component  $x$ .

**Output:** A set of LGCs for  $x$  in  $G$ .

**1. Coloring**

*Given a query component  $x$ , we color every node  $v \in V$  by comparing  $v$  and  $x$ . If  $v \sqsupseteq x$  then  $v$  is colored as a black node, otherwise it is colored as a white node.*

**2. Connecting**

*For every pair of nodes  $(u, v)$ , if  $u \sqsupseteq^* v$  then add an arc  $(v, u)$  to the set  $E$ , where  $\sqsupseteq^*$  is the transitive closure of  $\sqsupseteq$ .*

**3. Collecting**

*For every black node  $u$ , if no other white node  $v$  exists such that  $(v, u) \in E$  then  $LGC \leftarrow LGC \cup \{u\}$ .*

---

Figure D.1. The algorithm to find a set of LGCs.

---

Figure D.2 is an example of finding a set of LGCs in a partial ordering set. The black nodes C,F,I,H,J are the nodes that subsume the input query specification. The solid lines represent the subsumption relationships. The dotted lines represents the transitive closure among the nodes. Assume there are  $n$  nodes in  $G$ ,  $indegree_{max}$  denotes the maximum number of incident arcs for the nodes in  $G$  and  $height_{max}$  denotes the maximum height of  $L$ . The final LGC set contains two nodes C and H. The complexity of the coloring step is  $O(n)$ , the connecting step is  $O(n \times height_{max})$  and the collecting step is  $O(n \times indegree_{max})$ . Hence, the overall complexity is  $O(n \times max(indegree_{max}, height_{max}))$ . Therefore, we know that  $LGC$  is in the complexity class  $P$ .

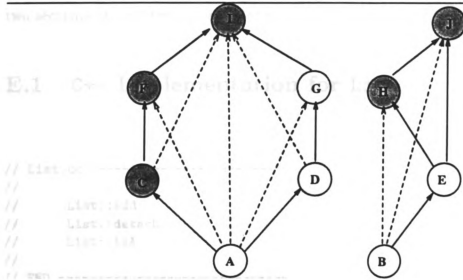


Figure D.2. Coloring, joining, and collecting a partial ordering set.

# APPENDIX E

## Program Listing

This appendix contains several programs that are mentioned in previous chapters. The programs in the first four section are written in C++. The programs in the last two sections are written in C and Motif.

### E.1 C++ Implementation for List

```
// List.cc -----  
//  
//      List::add  
//      List::detach  
//      List::isA  
//  
// END -----  
  
List::~~List()  
{  
    while( head != 0 )  
    {  
        ListElement *temp = head;  
        head = head->next;  
        delete temp;  
    }  
}  
  
void List::add( Object& toAdd )  
{
```



```

    ListElement *newElement = new ListElement( &toAdd );
    newElement->next = head;
    head = newElement;
    itemsInContainer++;
}

void List::detach( const Object& toDetach, int deleteObjectToo )
{
    ListElement *cursor = head;

    if ( *(head->data) == toDetach )
    {
        head = head->next;
    }
    else
    {
        ListElement *trailer = head;

        while ( cursor != 0 )
        {
            cursor = cursor->next;
            if ( *(cursor->data) == toDetach )
            {
                trailer->next = cursor->next;
                break;
            }
            else
            {
                trailer = trailer->next;
            }
        }

        }

        if( cursor != 0 )
        {
            itemsInContainer--;
            if ( deleteObjectToo )
            {
                if( cursor->data ) delete cursor->data;
                cursor->data = 0;
            }
            else
            {
                cursor->data = 0;
            }
            delete cursor;
        }
    }
}

```

```

List List::isA() const
{
    return listClass;
}

char *List::nameOf() const
{
    return "List";
}

```

## E.2 C++ Implementation for Array

```

// Array.cc -----
//
//      Array::addEnd
//      Array::addAt
//      Array::clear
//      Array::isA
//
// End -----

Array::~Array()
{
}

classType Array::isA() const
{
    return arrayClass;
}

void Array::addEnd( Object& toAdd )
{
    while( theArray[ whereToAdd ] != ZERO && whereToAdd <= upperbound )
    {
        whereToAdd++;
    }

    if( whereToAdd > upperbound )
    {
        reallocate( whereToAdd - lowerbound + 1 );
    }
}

```

```

    theArray[ whereToAdd - lowerbound ] = &toAdd;
    whereToAdd++;
    itemsInContainer++;
}

void Array::addAt( Object& toAdd, int atIndex )
{
    if( atIndex > upperbound )
    {
        reallocate( atIndex - lowerbound + 1 );
    }

    if ( theArray[ atIndex ] != ZERO )
    {
        delete theArray[ atIndex ];
        itemsInContainer--;
    }
    theArray[ atIndex - lowerbound ] = &toAdd;
    itemsInContainer++;
}

```

### E.3 C++ Implementation for Stack

```

// stack.cc -----
//      Stack::push
//      Stack::pop
//      Stack::top
// End -----

void Stack::push( Object& toPush )
{
    theStack.add( toPush );
    itemsInContainer++;
}

Object& Stack::pop()
{
    Object& temp = theStack.peekHead();
    if( temp != NOOBJECT )
    {
        theStack.detach( temp );
    }
}

```

```

        itemsInContainer--;
    }
    return temp;
}

Object& Stack::top() const
{
    return theStack.peekHead();
}

int Stack::isEmpty() const
{
    return theStack.isEmpty();
}

classType Stack::isA() const
{
    return stackClass;
}

char *Stack::nameOf() const
{
    return "Stack";
}

```

## E.4 C++ Implementation for DoubleList

```

// dbllist.cc -----
//      DoubleList::addAtHead
//      DoubleList::addAtTail
//      DoubleList::detachFromHead
//      DoubleList::detachFromTail
// End -----

DoubleList::~DoubleList()
{
    while( head != 0 )
    {
        DoubleListElement *temp = head;
        head = head->next;
        delete temp;
    }
}

```

```

void DoubleList::addAtHead( Object& toAdd )
{
    DoubleListElement *newElement = new DoubleListElement( &toAdd );

    if ( head )
    {
        head->previous = newElement;
        newElement->next = head;
        head = newElement;
    }
    else
    {
        tail = head = newElement;
    }
    itemsInContainer++;
}

void DoubleList::addAtTail( Object& toAdd )
{
    DoubleListElement *newElement = new DoubleListElement( &toAdd );

    if ( tail )
    {
        tail->next = newElement;
        newElement->previous = tail;
        tail = newElement;
    }
    else
    {
        head = tail = newElement;
    }
    itemsInContainer++;
}

void DoubleList::detachFromHead( const Object& toDetach, int deleteToo )
{
    if( head )
    {
        DoubleListElement *cursor = head;

        if ( *(head->data) == toDetach )
        {
            if( head->next == 0 )
            {
                tail = 0;
                head = head->next;
                head->previous = 0;
            }
        }
    }
}

```

```

        else
        {
            while ( cursor != 0 )
            {
                if ( *(cursor->data) == toDetach )
                {
                    cursor->previous->next = cursor->next;
                    cursor->next->previous = cursor->previous;
                    break;
                }
                else
                {
                    cursor = cursor->next;
                }
            }
        }
    }
    if( cursor != 0 )
    {
        itemsInContainer--;
        if ( deleteToo )
        {
            if( cursor->data ) delete cursor->data;
            cursor->data = 0;
        }
        else
        {
            cursor->data = 0;
        }
        delete cursor;
        cursor = 0;
    }
}

void DoubleList::detachFromTail( const Object& toDetach, int deleteToo )
{
    if( tail )
    {
        DoubleListElement *cursor = tail;

        if ( *(tail->data) == toDetach )
        {
            if( tail->previous == 0 )
            {
                head = 0;
                tail = tail->previous;
                tail->next = 0;
            }
        }
    }
}

```

```

    }
    else
    {
        while ( cursor != 0 )
        {
            if ( *(cursor->data) == toDetach )
            {
                cursor->previous->next = cursor->next;
                cursor->next->previous = cursor->previous;
                break;
            }
            else
            {
                cursor = cursor->previous;
            }
        }
    }
    if( cursor != 0 )
    {
        itemsInContainer--;
        if ( deleteToo )
        {
            if( cursor->data ) delete cursor->data;
            cursor->data = 0;
        }
        else
        {
            cursor->data = 0;
        }
        delete cursor;
        cursor = 0;
    }
}
}
}

```

## E.5 A Window with Pulldown Menu (Existing Component)

```

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>

```

```

#include <Xm/CascadeBG.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>

typedef struct _menu_item {
    char          *label;          /* the label for the item */
    WidgetClass   *class;          /* pushbutton, label, separator... */
    char          mnemonic;        /* mnemonic; NULL if none */
    char          *accelerator;     /* accelerator; NULL if none */
    char          *accel_text;      /* to be converted to compound string */
    void          (*callback)();    /* routine to call; NULL if none */
    XtPointer     callback_data;    /* client_data for callback() */
    struct _menu_item *subitems;    /* pullright menu items, if not NULL */
} MenuItem;

Widget
BuildPulldownMenu(parent, menu_title, menu_mnemonic, items)
Widget parent;
char *menu_title, menu_mnemonic;
MenuItem *items;
{
    Widget PullDown, cascade, widget;
    int i;
    XmString str;

    PullDown = XmCreatePulldownMenu(parent, "_pulldown", NULL, 0);

    str = XmStringCreateSimple(menu_title);
    cascade = XtVaCreateManagedWidget(menu_title,
        xmCascadeButtonGadgetClass, parent,
        XmNsubMenuId, PullDown,
        XmNlabelString, str,
        XmNmnemonic, menu_mnemonic,
        NULL);
    XmStringFree(str);

    for (i = 0; items[i].label != NULL; i++) {
        if (items[i].subitems)
            widget = BuildPulldownMenu(PullDown,
                items[i].label, items[i].mnemonic, items[i].subitems);
        else
            widget = XtVaCreateManagedWidget(items[i].label,
                *items[i].class, PullDown,
                NULL);

        if (items[i].mnemonic)
            XtVaSetValues(widget, XmNmnemonic, items[i].mnemonic, NULL);
    }
}

```



```

        if (items[i].accelerator) {
            str = XmStringCreateSimple(items[i].accel_text);
            XtVaSetValues(widget,
                XmNaccelerator, items[i].accelerator,
                XmNacceleratorText, str,
                NULL);
            XmStringFree(str);
        }

        if (items[i].callback)
            XtAddCallback(widget, XmNactivateCallback,
                items[i].callback, items[i].callback_data);
    }
    return cascade;
}

void
set_color(widget, color)
Widget widget;
char *color;
{
    printf("Setting color to %s\n", color);
}

MenuItem color_menu[] = {
    { "Cyan", &xmPushButtonGadgetClass, 'C', "Meta<Key>C", "Meta+C",
        set_color, "cyan", (MenuItem *)NULL },
    { "Yellow", &xmPushButtonGadgetClass, 'Y', "Meta<Key>Y", "Meta+Y",
        set_color, "yellow", (MenuItem *)NULL },
    { "Magenta", &xmPushButtonGadgetClass, 'M', "Meta<Key>M", "Meta+M",
        set_color, "magenta", (MenuItem *)NULL },
    { "Black", &xmPushButtonGadgetClass, 'B', "Meta<Key>B", "Meta+B",
        set_color, "black", (MenuItem *)NULL },
    NULL,
};

MenuItem drawing_menus[] = {
    { "Line Weight", &xmCascadeButtonGadgetClass, 'W', NULL, NULL,
        0, 0, NULL },
    { "Line Color", &xmCascadeButtonGadgetClass, 'C', NULL, NULL,
        0, 0, color_menu },
    { "Line Style", &xmCascadeButtonGadgetClass, 'S', NULL, NULL,
        0, 0, NULL },
    NULL,
};

main(argc, argv)
int argc;

```

```

char *argv[];
{
    Widget toplevel, main_w, menubar, drawing_a;
    XtAppContext app;

    toplevel = XtVaAppInitialize(&app, "Demos", NULL, 0,
        &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_w",
        xmMainWindowWidgetClass, toplevel,
        XmNscrollingPolicy, XmAUTOMATIC,
        NULL);

    menubar = XmCreateMenuBar(main_w, "menubar", NULL, 0);
    BuildPulldownMenu(menubar, "Lines", 'L', drawing_menus);
    XtManageChild(menubar);

    drawing_a = XtVaCreateManagedWidget("drawing_a",
        xmDrawingAreaWidgetClass, main_w,
        XmNwidth, 500,
        XmNheight, 500,
        NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app);
}

```

## E.6 A Window with Popup Menu (Modified Component)

```

#include <Xm/RowColumn.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/CascadeBG.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>

typedef struct _menu_item {
    char          *label;          /* the label for the item */
    WidgetClass *class;           /* pushbutton, label, separator... */

```

```

char          mnemonic;      /* mnemonic; NULL if none */
char          *accelerator;  /* accelerator; NULL if none */
char          *accel_text;   /* to be converted to compound string */
void          (*callback)(); /* routine to call; NULL if none */
XtPointer      callback_data; /* client_data for callback() */
struct _menu_item *subitems; /* pullright menu items, if not NULL */
} MenuItem;

```

#### Widget

```
BuildPopupMenu(parent, menu_title, menu_mnemonic, items)
```

```
Widget parent;
```

```
char *menu_title, menu_mnemonic;
```

```
MenuItem *items;
```

```

{
    Widget PopUp, cascade, widget;
    int i;
    XmString str;

    PopUp = XmCreatePopupMenu(parent, "_popup", NULL, 0);

    for (i = 0; items[i].label != NULL; i++) {

        if (items[i].subitems)
            widget = BuildPopupMenu(PopUp,
                                    items[i].label, items[i].mnemonic, items[i].subitems);
        else
            widget = XtVaCreateManagedWidget(items[i].label,
                                                *items[i].class, PopUp,
                                                NULL);

        if (items[i].mnemonic)
            XtVaSetValues(widget, XmNmnemonic, items[i].mnemonic, NULL);

        if (items[i].accelerator) {
            str = XmStringCreateSimple(items[i].accel_text);
            XtVaSetValues(widget,
                            XmNaccelerator, items[i].accelerator,
                            XmNacceleratorText, str,
                            NULL);
            XmStringFree(str);
        }

        if (items[i].callback)
            XtAddCallback(widget, XmNactivateCallback,
                           items[i].callback, items[i].callback_data);
    }
    return cascade;
}

```

```

void
set_color(widget, color)
Widget widget;
char *color;
{
    printf("Setting color to %s\n", color);
}

MenuItem color_menu[] = {
    { "Cyan", &xmPushButtonGadgetClass, 'C', NULL, NULL,
      set_color, "cyan", (MenuItem *)NULL },
    { "Yellow", &xmPushButtonGadgetClass, 'Y', NULL, NULL,
      set_color, "yellow", (MenuItem *)NULL },
    { "Magenta", &xmPushButtonGadgetClass, 'M', NULL, NULL,
      set_color, "magenta", (MenuItem *)NULL },
    { "Black", &xmPushButtonGadgetClass, 'B', NULL, NULL,
      set_color, "black", (MenuItem *)NULL },
    XmVaSEPARATOR,
    { "Quit", &xmPushButtonGadgetClass, 'Q', "Ctrl<Key>c", "Ctrl-C",
      NULL, NULL, (MenuItem *)NULL },
    NULL,
};

main(argc, argv)
int argc;
char *argv[];
{
    Widget menu, toplevel, main_w, menubar, drawing_a;
    void popup_cb(), input();
    XtAppContext app;

    toplevel = XtVaAppInitialize(&app, "Demos", NULL, 0,
                                &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_w",
                                       xmMainWindowWidgetClass, toplevel,
                                       XmNscrollingPolicy, XmAUTOMATIC,
                                       NULL);

    drawing_a = XtVaCreateManagedWidget("drawing_a",
                                       xmDrawingAreaWidgetClass, main_w,
                                       XmNwidth, 500,
                                       XmNheight, 500,
                                       NULL);

    menu = BuildPopupMenu(drawing_a, "Lines", 'L', color_menus);
    XtAddCallback(drawing_a, XmNinputCallback, input, menu);
}

```

```

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app);
}

void
input(widget, popup, cbs)
Widget widget;
Widget popup; /* popup menu associated with drawing area */
XmDrawingAreaCallbackStruct *cbs;
{
    if (cbs->event->xany.type != ButtonPress ||
        cbs->event->xbutton.button != 3)
        return;

    /* Position the menu where the event occurred */
    XmMenuPosition(popup, (XButtonPressedEvent *)(cbs->event));
    XtManageChild(popup);
}

```

# **BIBLIOGRAPHY**

## BIBLIOGRAPHY

- [1] Ted J. Biggerstaff. An Assessment and Analysis of Software Reuse. In Marshall C. Yovits, editor, *Advances in Computers*, volume 34, pages 1–57. 1992.
- [2] Ted J. Biggerstaff, editor. *Software Reusability Vol. 1: Concepts and Models*. ACM Press, New York, 1989.
- [3] Ted J. Biggerstaff, editor. *Software Reusability Vol. 2: Applications and Experience*. ACM Press, New York, 1989.
- [4] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
- [5] P.A.V. Hall, editor. *Software Reuse and Reverse Engineering in Practice*. UNICOM Applied Information Technology, Chapman & Hall, 1992.
- [6] E. Horowitz and J.B. Munson. An Expansive View of Reusable Software. *IEEE Transaction on Software Engineering*, 10(5):477–487, September 1984.
- [7] M. Sitaraman. *Mechanisms and Methods for Performance Tuning of Reusable Software Components*. PhD thesis, Ohio State University, Columbus, Ohio, 1990.
- [8] W. Tracz. Where does reuse start? *ACM SIGSOFT Software Engineering Notes*, 15(2):42–46, April 1990.
- [9] B.W. Weide, W.F. Ogden, and S.H. Zweben. Reusable Software Components. *Advances in Computers*, 33:1–65, 1991.
- [10] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. Technical Report STP-378-88, MCC, November 1988.
- [11] M.D. McIlroy. Mass Produced Software Components. In *Proceedings of NATO Conference on Software Engineering*, pages 88–98, New York, 1969. Petrocelli/Charter.

- [12] B.J. Cox. *Object-Oriented Programming - An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts, 1991.
- [13] James W. Hooper and Rowena O. Chester. *Software Reuse: Guidelines and Methods*. Plenum Press, New York and London, 1991.
- [14] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An Information Retrieval Approach for Automatic Constructing Software Libraries. *IEEE Trans. Software Engineering*, 17(8):800–813, August 1991.
- [15] R. Helm and Y.S. Maarek. Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. In *Proceedings of OOPSLA '91*, pages 47–61, 1991.
- [16] R.L. London. Specifying Reusable Components Using Z: Realistic Sets and Dictionaries. *ACM SIGSOFT Software Engineering Notes*, 14(3):120–127, May 1989.
- [17] F. Nishida, S. Takamatsu, Y. Fujita, and T. Tani. Semi-Automatic Program Construction from Specification Using Library Modules. *IEEE Transaction on Software Engineering*, 17(9):853–870, 1991.
- [18] Charles Rich and Richard C. Waters. Formalizing Reusable Software Components. In *Proceedings of Workshop on Reusability in Programming*, pages 152–158, Newport, RI, September 1983.
- [19] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.
- [20] D. R. Smith. KIDS: A semiautomatic program development system. 16(9):1024–1043, September 1990.
- [21] Betty Hsiao-Chih Cheng. *Synthesis of Procedural and Data Abstractions*. PhD thesis, University of Illinois at Urbana-Champaign, 1990. Tech Report UIUCDCS-R-90-1631.
- [22] G. Caldiera and V. Basili. Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2):61–70, February 1991.
- [23] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, January 1987.



- [24] C.W. Krueger. Models of Reuse in Software Engineering. Technical Report CMU-CS-89-188, Carnegie Mellon University, December 1989.
- [25] Victor R. Basili, Gianluigi Caldiera and Giovanni Cantone. A Reference Architecture for the Component Factory. *ACM Transaction on Software Engineering and Methodology*, 1(1):53–80, January 1992.
- [26] Rubén Prieto-Díaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.
- [27] Robert Allen Steigerwald. *Reusable Software Component Retrieval via Normalized Algebraic Specifications*. PhD thesis, Naval Postgraduate School, Monterey, California, 1991.
- [28] Tom Danieli and Betty H.C. Cheng. Construction of Formal Specifications from an Object-Oriented Decomposition of Informal Problem Descriptions. Technical Report MSU-CPS-92-08, Department of Computer Science, Michigan State University, 1992.
- [29] Robert H. Bourdeau and Betty H.C. Cheng. An object-oriented toolkit for constructing specification editors. In *Proceedings of COMPSAC'92: Computer Software and Applications Conference*, pages 239–244, September 1992.
- [30] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press and McGraw-Hill, Cambridge, 1986.
- [31] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood, New Jersey, 1990.
- [32] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [33] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Yourdon Press, Prentice Hall, Englewood. New Jersey, 1990.
- [34] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood, New Jersey, 1988.
- [35] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-oriented Programming in C++*. John Wiley & Sons, 1990.

- [36] Ann L. Winblad, Samuel D. Edwards, and David R. King. *Object-Oriented Software*. Addison-Wesley, Publishing Company Inc., 1990.
- [37] Kevin D. Jones. LM3: A Larch Interface Language for Modula-3 - A Definition and Introduction Version 1.0. Technical Report 72, Digital Systems Research Center, June 1991.
- [38] Gary T. Leavens and Yoonsik Cheon. Preliminary Design of Larch/C++. Technical Report 92-16a, Department of Computer Science, Iowa State University, May 1992.
- [39] Jeannette M. Wing. Using Larch to Specify Avalon/C++ Objects. *IEEE Transaction on Software engineering*, 16(9):1076–1088, September 1990.
- [40] J.V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Technical Report 5, Digital Equipment Corporation Systems Research Center, Palo Alto, California, July 24 1985.
- [41] Richard Allen Lerner. *Specifying Objects of Current Systems*. PhD thesis, Caregie Mellon University, May 1991.
- [42] Jun-Jang Jeng and Betty H.C. Cheng. Using formal methods to construct a software component library. In *Proceedings of the Fourth European Software Engineering Conference*, pages 397–417, Garmisch-Partenkirchen, Germany, September 13-17, 1993.
- [43] Betty H.C. Cheng and Jun-Jang Jeng. Formal methods applied to reuse. In *Proceedings of the Fifth Workshop in Software Reuse*, 1992.
- [44] Jun-Jang Jeng and Betty H.C. Cheng. Using Automated Reasoning to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, 1992.
- [45] John V. Guttag and James J. Horning. Introduction to LCL, a Larch/C Interface Language. Technical Report 74, Digital Systems Research Center, July 1991.
- [46] Gary T. Leavens. Modular Verification of Object-Oriented Programs with Subtypes. Technical Report 92-09, Department of Computer Science, Iowa State University, July 1990.
- [47] Gary Leavens. Modular Specification and Verification of Objected-Oriented Programs. *IEEE Software*, 8(4):72–80, July 1991.

- [48] Mike Laux. lb: A Larch Browser. Technical report, Dept. of Computer Science, MSU, 1992. in preparation.
- [49] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.
- [50] J.A. Robinson. A Machine Oriented Logic Based on Resolution Principle. *Journal of ACM*, 12(1):227–234, 1965.
- [51] K.H. Blasius and H.J. Burchert, editors. *Deduction Systems in Artificial Intelligence*. Ellis Horwood Series in Artificial Intelligence, 1989.
- [52] S. Miyamoto. *Fuzzy Sets in Informational Retrieval and Cluster Analysis*. Kluwer Academic Publishers, 1990.
- [53] D.H. Ballard and C.M. Brown. *Computer Vision*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
- [54] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proc. of the American Mathematical Society*, 1956.
- [55] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.
- [56] D. Gries. *The Science of Programming*. Springer-Verlag, New-York, 1981.
- [57] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
- [58] Ted Faison. *Borland C++ 3.1 Object-Oriented Programming*. SAMS, 1992.
- [59] Greg Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):571–561, October 1989.
- [60] Jun-Jang Jeng and Betty H. C. Cheng. Using Analogy to Determine Program Modification Based on Specification Changes. In *Proceedings of 5th International Conference on Tools with Artificial Intelligence, 1993*, pages 113–116. IEEE Computer Society, November 8–11, 1993.
- [61] K.E. Gorlen. *NIH Class Library*. National Institute of Health, Computer System Laboratory, Division of Computer Research and Technology, Bethesda, MD. Public Domain Software.

- [62] Roger P. Hall. Computational Approaches to Analogical Reasoning: A Comparative Analysis. *Artificial Intelligence*, 39:39–120, 1989.
- [63] Dedre Gentner. Mechanisms of Analogical Reasoning. In Stella Vosniadou and Andrew Ortony, editors, *Similarity and Analogical Reasoning*, pages 199–241. Cambridge University Press, 1990.
- [64] Nachun Dershowitz. *The Evolution of Programs*. Birkhauser, Boston, MA, 1983.
- [65] M.B. Hesse. *Models and Analogies in Science*. Notre-Dame, 1963.
- [66] Jun-Jang Jeng and Betty H. C. Cheng. Program Modifications Based on Specification Changes Part 1: More General Components. Technical report, Michigan State University, 1993.
- [67] Jun-Jang Jeng and Betty H.C. Cheng. Using Automated Reasoning to Determine Software Reuse. Technical Report MSU-CPS-ACS-64, Department of Computer Science, Michigan State University, East Lansing, Michigan, June 1992. accepted for publication in IJSEKE.
- [68] Jun-Jang Jeng and Betty H.C. Cheng. Using Formal Methods to Construct a Software Component Library. Technical Report MSU-CPS-92-11, Michigan State University, Department of Computer Science, 1992.
- [69] Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual*. O'Reilly & Associate, Inc., 1990.
- [70] Dan Heller. *Motif Programming Manual*. O'Reilly & Associate, Inc., 1991.
- [71] Douglas A. Young. *Object-Oriented Programming with C++ and OSF/Motif*. Prentice Hall Englewood Cliffs, 1992.
- [72] Dan Heller. *XView Programming Manual*. O'Reilly & Associate, Inc., 1990.
- [73] Michael Mehlich and Weishi Zhang. Specifying Interactive Components for Configuring Graphical User Interfaces, May 1993. personal communication.
- [74] Donald L. McMinds. *Mastering OSF/Motif widgets*. Reading, Mass. : Addison-Wesley, 1993.
- [75] John V. Guttag, James J. Horning, and Andres Modet. Report on the Larch Shared Language: Version 2.3. Technical Report 58, Digital Systems Research Center, April 1990.

- [76] Stephen J. Garland, John V. Guttag, and James J Horning. Debugging Larch Shared Language Specifications. *IEEE Transactions on Software Engineering*, 16(9):1044–1057, September 1990.
- [77] Jeannette M. Wing. Using Larch to Specify Avalon/C++ Objects. *IEEE Transactions on Software Engineering*, 16(9):1076–1088, September 1990.
- [78] Allan Heydon, Mark W. Maimone, J.D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró: Visual Specification of Security. *IEEE Transactions on Software Engineering*, 16(10):1185–1197, October 1990.
- [79] Jeannette M. Wing and Amy Moormann Zaremski. A Formal Specification of a Visual Language Editor. Technical Report CMU-CS-91-112, February 25 1991.
- [80] Amy Moormann Zaremski. A Larch Specification of Miró Editor. Technical Report CMU-CS-91-111, February 25 1991.
- [81] Jeannette M. Wing and Chun Gong. A Library of Concurrent Objects and Their Proofs of Correctness. Technical Report CMU-CS-90-151, July 1990.
- [82] Scott Nettles. A Larch Specification of Copying Garbage Collection. Technical Report CMU-CS-92-219, School of Computer Science, Carnegie Mellon University, December 1992.
- [83] Tim O'Reilly. *Xlib Programming Manu. sl*. O'Reilly & Associate, Inc., 1990.
- [84] Tim O'Reilly. *Xlib Reference Manu. sl*. O'Reilly & Associate, Inc., 1990.
- [85] Hartmut Ehrig. *Fundamentals of Algebraic Specification Vol 1 and 2*. Springer-Verlag, Berlin, New York, 1985.
- [86] Yoonsik Cheon and Gary T. Leavens. A Quick Overview of Larch/C++. Technical Report 92-18, Department of Computer Science, Iowa State University, June 1993.
- [87] J.M. Neighbors. The draco approach to constructing software from reusable components. *IEEE Transaction on Software Engineering*, 10(5):564–573, 1984.
- [88] Rubén Prieto-Díaz. Implementing Faceted Classification for Software Reuse. *Communication of ACM*, 34(5), May 1991.
- [89] Rubén Prieto-Díaz. Implementation Faceted Classification for Software Reuse. In *IEEE International Conference on Software Engineering*, pages 300–304, 1990.

- [90] Bruce A. Burton, Rhonda Wienk, et al. The Reusable Software Library. *IEEE Software*, 4:25–33, July 1987.
- [91] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. ACM Press, 1990.
- [92] Richard C. Waters. The programmer's apprentice : A session with KBE-macs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1981.
- [93] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Key to Reuse. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 182–190, December 1993.
- [94] T. G. Evans. A program for solution of geometric analogy intelligence test questions. 1968.
- [95] G. Polya. *Induction and Analogy in Mathematics*. Princeton University Press, 1954.
- [96] Robert E. Kling. A Paradigm for Reasoning by Analogy. *Artificial Intelligence*, 2:147–178, 1971.
- [97] J. Carbonnell. Learning by Analogy: formulating and generalizing plans from past experience. In *Machine Learning*. Tioga, 1983.
- [98] J. Carbonnell. Derivational analogy. In *AAAI-83*. 1983.
- [99] B. Indurkha. *A computational theory of metaphor comprehension and analogical reasoning*. PhD thesis, Boston University, 1985.
- [100] Brian Falkenhainer, Kenneth D. Forbus, and Dedre Gentner. The Structure-Mapping Engine: Algorithm and Examples. *Artificial Intelligence*, 41:1–63, 1989.
- [101] Dendre Gentner. Structural Mapping: a theoretical framework for analogy. *Cognitive Science*, 7, 1983.
- [102] S. Kedar-Cabelli. Analogy - from a Unified Perspective. In D.H. Helman, editor, *Analogical Reasoning*, pages 65–104. Klumer Academic Publishers, 1988.
- [103] Stephen Owen. *Analogy for Automated Reasoning*. Academic Press, 1990.

- [104] Nachum Dershowitz. Program Abstraction and Instantiation. *ACM Transactions on Programming Languages and Systems*, 7(3):446–477, July 1985.
- [105] A. Goldberg. Reusing Software Development. In *Proceedings of the 4th ACM SIGSOFT Symp. on Software Development Environment*, pages 107–119, Irvine, California, December 1990.
- [106] Mitchell D. Lubars. The ROSE-2 Strategies for Supporting High Level Software Design Reuse. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*. MIT Press, 1991.
- [107] Sanjay Bhansali. Domain-Based Program Synthesis Using Planning and Derivational Analogy. Technical Report UIUCDCS-R-91-1701, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1991.
- [108] Neil A. Maiden and Alistair G. Sutcliffe. Exploiting Reusable Specifications Through Analogy. *Communications of the ACM*, 35(4):55–64, August 1992.
- [109] Chung-Horng Lung and Joseph E. Urban. Analogical Approach for Software Reuse. In *Proceedings of Golden West International Systems, 1992*, Reno, Nevada, June 1-3, 1992.
- [110] Chung-Horng Lung and Joseph E. Urban. Integration of Domain Analysis and Analogical Approach for Software Reuse, 1993.
- [111] Alberto Coen-Porisini, Flavio De Paoli, Carlo Ghezzi, and Dino Mandrioli. Software Specialization Via Symbolic Execution. *IEEE Transactions on Software Engineering*, 17(9):884–899, September 1991.
- [112] D.B. Arnold, D.A. Duce, and G.J. Reynolds. An approach to the formal specification of configurable models of graphics systems. In *Eurographics '87*, pages 439–463, 1987.
- [113] George S. Carson. The specification of computer graphics systems. *IEEE Computer Graphics and Applications*, pages 27–41, September 1983.
- [114] George S. Carson. An approach to the formal specification of computer graphics systems. *Computers and Graphics*, 8(1):51–57, 1984.
- [115] D.A. Duce. GKS, structures and formal specification. In *Eurographics '89*, pages 271–287, 1989.

- [116] D.A. Duce and E.V.C. Fielding. Towards a formal specification of the GKS output primitives. In *Eurographics '86*, pages 307–323, 1986.
- [117] D.A. Duce, E.V.C. Fielding, and L.S. Marshall. Formal specification of a small example based on GKS. *ACM Transactions on Graphics*, 7(3):180–197, July 1988.
- [118] Eugene Fiume. Toward realistic formal specifications for non-trivial graphical objects. In *Eurographics '89*, pages 289–300, 1989.
- [119] Rupert Gnatz. Approaching a formal framework for graphics software standards. *Computers and Graphics*, 8(1):39–50, 1984.
- [120] Joseph A. Goguen. Modular Algebraic Specification of Some Basic Geometrical Constraints. Technical Report CLSI-87-87, March 1987.
- [121] William R. Mallgren. ACM Distinguished Dissertation Series, The MIT Press, Cambridge, Massachusetts, London, England, 1982.
- [122] William R. Mallgren. Formal specification of graphic data types. *ACM Transactions on Programming Languages and Systems*, 4(4):687–710, October 1982.
- [123] John Guttag and Jim J. Horning. Formal Specification as a Design Tool. In *Seventh Annual ACM Symposium on Principles of Programming Language*, pages 251–261, 1980.
- [124] Bernard Sufrin. Formal Specification of a Display-Oriented Text Editor. *Science of Computer Programming*, 1:157–202, 1982.
- [125] K. T. Narayana and Sanjeev Dharap. Formal Specification of a Look Manager. *IEEE Transactions on Software Engineering*, 16(9):1089–1103, September 1990.
- [126] Robert J.K. Jacob. Using Formal Specification in the Design of a Human-Computer Interface. *Communication of ACM*, 26(4):259–264, 1983.
- [127] Robert J.K. Jacob. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.
- [128] Michael R. Laux, Robert H. Bourdeau, and Betty H.C. Cheng. An integrated development environment for formal specifications. In *Proc. of IEEE International Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, July 1993.



- [129] M. V. LaPolla, J. L. Sharnowski, B. H. C. Cheng, and K. Anderson. Data parallel program visualizations from formal specifications. *Journal of Parallel and Distributed Computing*, May 1993.
- [130] Betty H.C. Cheng and Gerald C. Gannod. Constructing formal specifications from program code. In *Proc. of Third International Conference on Tools in Artificial Intelligence*, pages 125–128, November 1991.
- [131] Gerald C. Gannod and Betty H.C. Cheng. A two-phase approach to reverse engineering using formal methods. In *to appear in the Proc. of Formal Methods in Programming and Their Applications Conference*, June 1993. The proceedings will appear as part of the series of Lecture Notes in Computer Science (LNCS) published by Springer-Verlag.
- [132] K.J. Holyoak and P.R. Thagard. A Computational Model of Analogical Problem Solving. In A. Ortony and S. Vosniadou, editors, *Similarity and Analogic Learning*, pages 242–266. Cambridge University Press, 1990.
- [133] Kuo-Chung Tai. The Tree-to-Tree Correction Problem. *JACM*, 26(3):422–433, July 1979.
- [134] Christopher M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Tree. *JACM*, 29(1):68–95, January 1982.
- [135] Kaizhong Zhang, Dennis Shasha, and Jason Tsong-Li Wong. Approximate Tree Matching in the Presence of Variable Length Don't Cares. Technical Report CIS-91-21, Department of Computer Science, New Jersey Institute of Technology, Newark, New Jersey, 1991.

MICHIGAN STATE UNIV. LIBRARIES



31293010515496