

© Copyright 1996

Gail C. Murphy



Lightweight Structural Summarization  
as an Aid to Software Evolution

by

Gail C. Murphy

A dissertation submitted in partial fulfillment  
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by \_\_\_\_\_  
(Chairperson of Supervisory Committee)

Program Authorized  
to Offer Degree \_\_\_\_\_

Date \_\_\_\_\_



University of Washington

Abstract

Lightweight Structural Summarization as an Aid to Software Evolution

by Gail C. Murphy

Chairperson of the Supervisory Committee: Professor Alan Borning

Department of Computer Science  
and Engineering

To effectively perform a change to an existing software system, a software engineer needs to have some understanding of the structure of the system. All too often, though, an engineer must proceed to change a system without sufficient structural information because existing software understanding techniques are unable to help the engineer acquire the desired knowledge within the time and cost constraints specified for the task.

The thesis of this research is that an approach based on summarization can overcome the limitations associated with existing approaches, enabling an engineer to assess, plan, and execute changes to a software system more effectively. Summarization involves the production of overviews of vast amounts of user-selected information in a timely manner. I describe two techniques developed to support the summarization approach. The first technique, the *software reflexion model technique*, enables an engineer to summarize selected structural information in the context of a task-specific high-level model. The second technique, the *lexical source model extraction technique*, supports the summarization process by facilitating the scanning and analysis of system artifacts for structural information that is difficult or impossible to extract at low cost using existing approaches. Each of these techniques is lightweight and iterative: the engineer is able to quickly and easily gain access to partial and approximate structural information, and may then balance the completeness and accuracy of the information needed with the cost of further



applying the technique. I demonstrate the viability of the approach by describing its use on a variety of change tasks and systems, including the use of the reflexion model technique by an engineer at Microsoft Corporation to aid with an experimental reengineering of the million-line Excel spreadsheet product.



# Table of Contents

List of Figures . . . . .	v
List of Tables . . . . .	vii
Chapter 1: Introduction . . . . .	1
1.1 The Change Process . . . . .	3
1.2 The Role of Software Structure . . . . .	5
1.3 A Brief Survey of Structure Understanding Techniques . . . . .	8
1.3.1 Program Visualization . . . . .	8
1.3.2 Reverse Engineering . . . . .	11
1.3.3 Metrics . . . . .	15
1.4 The Utility of Summary Information . . . . .	16
1.5 An Overview of Lightweight Structural Summarization . . . . .	19
1.5.1 An Introduction to the Software Reflexion Model Technique . . . . .	19
1.5.2 An Introduction to the Lexical Source Model Extraction Technique . . . . .	24
1.6 The Organization of the Dissertation . . . . .	27
Chapter 2: Software Reflexion Models . . . . .	28
2.1 An Example . . . . .	29
2.1.1 Selecting a High-Level Model . . . . .	30
2.1.2 Extracting a Source Model . . . . .	31
2.1.3 Defining a Map . . . . .	31
2.1.4 Computing a Reflexion Model . . . . .	32
2.1.5 Interpreting a Reflexion Model . . . . .	34
2.1.6 Refining a Reflexion Model . . . . .	36
2.1.7 Using a Reflexion Model . . . . .	38
2.2 Formal Characterization . . . . .	39
2.2.1 Reflexion Model . . . . .	40
2.2.2 Computing a Reflexion Model . . . . .	42
2.2.3 Displaying a Reflexion Model . . . . .	45

2.2.4	A Family of Reflexion Models . . . . .	45
2.3	Summary . . . . .	46
Chapter 3:	Computing Software Reflexion Models . . . . .	48
3.1	Tools . . . . .	48
3.1.1	Source Entity Description Language . . . . .	51
3.2	Complexity . . . . .	55
3.3	Speeding up the Computation . . . . .	57
3.3.1	Trading Space for Time . . . . .	57
3.3.2	Optimizing the Regular Expression Comparison . . . . .	61
3.3.3	Combining Optimizations . . . . .	64
3.3.4	Incremental Computation . . . . .	65
3.4	Summary . . . . .	73
Chapter 4:	Refining the Software Reflexion Model Technique . . . . .	75
4.1	Multiple Relations . . . . .	75
4.1.1	Typed Source Model . . . . .	76
4.1.2	Typed High-level Model . . . . .	78
4.2	Managing Information . . . . .	82
4.2.1	Tagging . . . . .	82
4.2.2	Annotations . . . . .	84
4.3	Usability . . . . .	88
4.3.1	Assessing Coverage . . . . .	88
4.3.2	Debugging Computations . . . . .	91
4.4	Summary . . . . .	92
Chapter 5:	A Discussion of the Software Reflexion Model Technique . . . . .	93
5.1	Models . . . . .	93
5.1.1	Syntactic Models . . . . .	93
5.1.2	Models as Collections of Binary Relations . . . . .	96
5.1.3	A Non-Hierarchical Structural Comparison . . . . .	97
5.2	Maps . . . . .	97
5.2.1	The Form of a Map . . . . .	98
5.2.2	The Content of a Map . . . . .	99
5.3	Visualization . . . . .	102
5.4	Summary . . . . .	103
Chapter 6:	Lexical Source Model Extraction . . . . .	104
6.1	The Technique . . . . .	106
6.2	The Specification Language . . . . .	109

6.2.1	Specifying Patterns . . . . .	109
6.2.2	Specifying Actions . . . . .	111
6.2.3	Specifying Analysis Operations . . . . .	112
6.3	Example . . . . .	116
6.4	Using the Technique . . . . .	122
6.5	Producing Source Models for Computing Reflexion Models . . . . .	123
6.6	Summary . . . . .	125
Chapter 7:	The Lexical Source Model Extraction Tools . . . . .	127
7.1	The Generated Tools . . . . .	127
7.1.1	Scanner . . . . .	128
7.1.2	Analyzer . . . . .	133
7.2	Performance . . . . .	133
7.3	Summary . . . . .	137
Chapter 8:	A Discussion of the Lexical Source Model Extraction Technique . . . . .	138
8.1	Expressiveness . . . . .	138
8.2	Accuracy . . . . .	141
8.3	Heuristics . . . . .	143
8.4	Engineering Trade-offs . . . . .	144
8.5	Summary . . . . .	145
Chapter 9:	Validation . . . . .	146
9.1	Excel: A Case Study in Assessing and Planning Change . . . . .	147
9.1.1	The Environment . . . . .	148
9.1.2	The Process . . . . .	149
9.1.3	Results . . . . .	158
9.2	Program Restructuring Tool: A Case Study in Design Conformance . . . . .	159
9.2.1	The Environment . . . . .	159
9.2.2	The Process . . . . .	160
9.2.3	Results . . . . .	161
9.3	SPIN: A Case Study in the Use of Approximate Information . . . . .	161
9.3.1	The Environment . . . . .	162
9.3.2	The Process . . . . .	162
9.3.3	Results . . . . .	165
9.4	Additional Case Studies . . . . .	166
9.5	Summary . . . . .	167

Chapter 10: Related Work . . . . .	168
10.1 Software Reflexion Models . . . . .	168
10.1.1 Forward Engineering . . . . .	168
10.1.2 Reverse Engineering . . . . .	172
10.1.3 Knowledge-based Approaches . . . . .	177
10.1.4 Program Visualization . . . . .	179
10.1.5 Model Comparison . . . . .	179
10.2 Lexical Source Model Extraction . . . . .	181
10.2.1 Tools Based on Regular Expressions . . . . .	182
10.2.2 Tools Based on Parsing . . . . .	183
10.3 Summarization Techniques . . . . .	185
Chapter 11: Conclusion . . . . .	186
11.1 Contributions . . . . .	187
11.2 Future Work . . . . .	188
11.2.1 Software Reflexion Models . . . . .	190
11.2.2 Lexical Source Model Extraction . . . . .	192
Bibliography . . . . .	193
Appendix A: A Short Overview of Z . . . . .	204
A.1 Data Types and Schemas . . . . .	204
A.2 The Meaning of Symbols . . . . .	206
Appendix B: A Formal Characterization of Typed Software Reflexion Models . . . . .	210
B.1 Typed Reflexion Model . . . . .	210
B.2 Computing a Typed Reflexion Model . . . . .	214
B.3 Displaying a Typed Reflexion Model . . . . .	218
B.4 Tagging a Reflexion Model Arc . . . . .	219
B.4.1 Tagging an Arc . . . . .	220
B.4.2 Applying Tags to a Reflexion Model Arc . . . . .	220
B.5 Annotating a Reflexion Model Arc . . . . .	222
B.6 Families of Typed Reflexion Models . . . . .	223
Appendix C: Lexical Source Model Extraction Patterns . . . . .	224
C.1 A Pattern for Identifying Functions in C . . . . .	224
C.2 Patterns for Identifying Calls in Non-Preprocessed C Source . . . . .	225
C.3 Patterns for Identifying Calls in Preprocessed C Source . . . . .	225

# List of Figures

1.1	A Visualization of the Calls Structure of the groff Equation Preprocessor	10
1.2	An Initial Graph Display Using Rigi . . . . .	13
1.3	A Reverse Engineered Model of the groff Equation Preprocessor . . . . .	14
1.4	A Summary Produced by a Web Search Engine . . . . .	17
1.5	The Process of Using the Reflexion Model Technique . . . . .	20
1.6	A High-level Model and a Software Reflexion Model for gcc . . . . .	23
1.7	A Sample of Calls in gcc . . . . .	23
1.8	A Pattern to Extract Calls from C Source Code . . . . .	26
2.1	A High-level Model of a Unix Virtual Memory Subsystem . . . . .	30
2.2	A Map for the NetBSD Virtual Memory Subsystem . . . . .	32
2.3	High-level and Reflexion Models for the NetBSD Virtual Memory Subsystem	33
2.4	Investigating Source Through a Reflexion Model . . . . .	35
2.5	A Refined Reflexion Model for the NetBSD Virtual Memory Subsystem .	38
3.1	Tools for Computing Reflexion Models . . . . .	50
3.2	Source Entity Naming Tree . . . . .	53
3.3	Source Entity Frequency Distributions . . . . .	60
3.4	Incremental Tools for Computing a Reflexion Model . . . . .	68
3.5	A Modified Map for the NetBSD Virtual Memory Subsystem . . . . .	69
3.6	A Fragment of the Data Structure Used for Incremental Computation . .	70
4.1	Comparing a Typed Source Model and an Untyped High-level Model . . .	79
4.2	Comparing a Typed Source Model and Typed High-level Model . . . . .	79
4.3	Comparing a Typed Source Model and a Partially-typed High-level Model	81
4.4	An Annotated Reflexion Model . . . . .	86
4.5	Information on the Mapping . . . . .	90
4.6	Information on Unmapped Source Model Entities . . . . .	90
5.1	A Mediator-based Realization of a Programming Environment . . . . .	99

6.1	The Lexical Source Model Extraction Technique . . . . .	107
6.2	The Architecture of the Lexical Source Model Extraction System . . . . .	108
6.3	Analysis Specification for Computing a Global C Calls Source Model . . . . .	114
6.4	Patterns to Extract Events from Field Source Code . . . . .	118
6.5	A Pattern to Extract Event Registrations from a Structured Data File . . . . .	120
6.6	Implicitly-Invokes Analysis Specification . . . . .	121
6.7	Embedding Lexical Patterns in a Source Model Entity Naming Tree . . . . .	124
7.1	Generated Deterministic Finite State Machine for a Pattern . . . . .	129
7.2	Generated Deterministic and Non-Deterministic Finite State Machines . . . . .	132
9.1	A High-Level Model of Excel . . . . .	150
9.2	A Fragment of the Initial Reflexion Model for Excel . . . . .	151
9.3	Applying the Reflexion Model Technique at Microsoft Corporation . . . . .	153
9.4	A Fragment of a Refined Reflexion Model for Excel . . . . .	157
9.5	Patterns to Extract Calls between Modula-3 Procedures . . . . .	163
9.6	A Pattern to Extract Calls to Externally Declared C Functions . . . . .	164
C.1	C Function Definition Pattern. . . . .	225
C.2	Patterns for Extracting Calls from Non-Preprocessed C Source. . . . .	226
C.3	Patterns for Extracting Calls from Postprocessed C Source. . . . .	227

# List of Tables

1.1	Examples of Software System Components . . . . .	7
1.2	Examples of Software System Structural Interactions . . . . .	7
3.1	Naive Reflexion Model Computation . . . . .	56
3.2	Reflexion Model Computation with a Hash Table . . . . .	59
3.3	Reflexion Model Computation with a Cache . . . . .	60
3.4	A Comparison of the Speed of Regular Expression Packages . . . . .	62
3.5	Reflexion Model Computation with Regular Expression Optimizations . .	64
3.6	Reflexion Model Computation with Multiple Optimizations . . . . .	65
3.7	The Performance of the Incremental Algorithm . . . . .	73
5.1	A Comparison of Content Styles for a Structural Relation . . . . .	95
6.1	Icon Functions to Support Relational Operations . . . . .	115
6.2	Uses of the Lexical Source Model Extraction Technique . . . . .	123
7.1	Comparing the Speed of Tokenizing an Input Stream . . . . .	134
7.2	A Comparison of Various Tools that Extract C Calls Information . . . . .	136
8.1	Execution Statistics for Generated Scanners . . . . .	144
A.1	Z Set Notation Used in the Dissertation. . . . .	207
A.2	Z Relation Notation Used in the Dissertation. . . . .	207
A.3	Other Z Notation Used in the Dissertation. . . . .	209

## Acknowledgments

But words are things, and a small drop of ink,  
Falling like dew, upon a thought, produces  
That which makes thousands, perhaps millions, think;  
—Lord Byron from *Don Juan*, Canto III, Stanza 88.

If the words in this dissertation cause even a small fraction of thousands to think, it is in large part due to the help, support, and encouragement I have received from a number of people.

First and foremost, I thank my advisor, David Notkin. David's encouragement made this research happen; his advice made it infinitely better; and his sense of humour made it fun. I will always be grateful to him for teaching me what research is all about. Thanks also to Kevin Sullivan for some stimulating conversations that led to reflexion models and his insightful comments on many aspects of this research. I also thank Alan Borning and Nancy Leveson for their constructive comments; they greatly improved the quality of the dissertation. I also acknowledge Alan's help in serving as the Chair of my Supervisory Committee under rather strange and interesting circumstances. I owe a special thanks to William Griswold who has tirelessly read and commented on drafts of both papers and the dissertation, and who never fails to ask the really tough questions.

I am grateful to several people for participating in the case studies: the engineers at Microsoft who remain anonymous by choice, Dylan McNamee who provided expertise on virtual memory systems, and Pok Wong who applied reflexion models for a design conformance task. For their constructive comments on this research in a variety of forms, I thank a number of people: Gregory Abowd, Robert Allen, Paul Bugni, Kingsum Chow, David Garlan, Richard Helm, Daniel Jackson, Michael Jackson, David Lamb, Sui-Ching Lan, Bob Monroe, Stephen North, Kurt Partridge, Bob Schwanke, Nancy Staudenmeyer, Michael VanHilst, John Vlissides, Steve Wampler, Daniel Weise, and the students who served as guinea pigs in CSE 503.

I am also grateful to Paulin Laberge who sparked my interest in software engineering by throwing tough problems my way at MPR and the “ninja coders” for teaching me about software systems.

Last, but certainly not least, I thank my spouse, Michael, and my extended family for their patience and unfailing moral support.

I also acknowledge the financial support provided by a Canadian NSERC post-graduate scholarship and by a University of Washington Department of Computer Science & Engineering Educator’s fellowship, and equipment provided by Microsoft Corporation.

Without all of this support, the thoughts in this dissertation would never have found the necessary ink—well, make that bits for the digital age.



# Chapter 1

## Introduction

Any useful software system continually evolves [Lehman and Belady 1978; Parnas 1994]. The changes marking a system's evolution are initiated for a variety of reasons, including requests by clients to add new features into the system, market pressures that force support for new hardware and software technology, and business decisions to improve the reusability, maintainability, and quality of the source code.

To perform a change to a system effectively, a software engineer needs to have some understanding of the system [Boehm et al. 1976; Blum 1989; Von Mayrhauser and Vans 1995]. In particular, an engineer may benefit from having some understanding of the structure of the software system—the organization of the source code into components and the interactions between those components—since the difficulty of performing a change is dependent to a great extent on the system's structure [Parnas 1972; Lehman and Belady 1978].

Providing the software engineer with useful structural information requires techniques and tools for producing adaptable views in a timely cost-effective way. Adaptable views are necessary because it is difficult to pre-configure appropriate views for the wide range of changes that an engineer may perform [Harris et al. 1995]. It is necessary to produce these views in a timely and cost-effective way because of pressures to incorporate as many changes as possible before the next release. Typically, the number of desired changes exceeds the ability of the development organization to perform the changes in the time

available; Lehman and Belady, for instance, describe how desired changes flowed over from one release to another during the development of the IBM OS/360 operating system in the 1960s [Lehman and Belady 1980, p. 428].

Currently, there are three approaches available to aid an engineer in understanding a system's structure during the change process. The first approach, exemplified by program visualization techniques, produces direct textual and graphical representations of the structure of the source code (Section 1.3.1). These techniques provide benefits for source comprising a few thousand lines of code, but they suffer from scale problems when an engineer needs to gain an overview of parts of a larger system.

Reverse engineering techniques, which are representative of the second approach, attempt to overcome the scale problem by automatically or semi-automatically abstracting from the system's source to create higher-level models of the system's structure (Section 1.3.2). These higher-level models record the essence of a particular view of the system's structure in terms of a few abstracted elements, making it easier for an engineer to comprehend the information. Current reverse engineering techniques, though, are limited in their ability to produce a variety of views of a system on-demand at low cost.

The third approach, metrics, condenses information about the software structure into one or more quantitative values or qualitative ranges (Section 1.3.3). Although these values or ranges may provide the engineer with some understanding of the structure relative to some norm, the information does not aid the engineer in understanding a given system's structure with respect to a particular change task.

The thesis of this research is that an alternate approach, in which structural information is *summarized* in the context of a software engineer's high-level view of a system's structure, can overcome these limitations, enabling an engineer to assess, plan, and execute changes to a system more effectively. Furthermore, it is claimed that it is often reasonable and useful to trade accuracy and completeness in summarized structural information for faster, lower-cost access in order to satisfy the time and cost constraints

associated with change tasks.

In this dissertation, I describe two techniques developed to support structural summarization. The first technique, the *software reflexion model* technique, enables an engineer to produce a summary of structural information extracted from a system's source in the context of a high-level model selected as appropriate for reasoning about a given change task. The second technique, the *lexical source model extraction* technique, supports the summarization process by facilitating the scanning and analysis of system artifacts for structural information that is difficult or impossible to extract at low cost using existing techniques. Each of these techniques is lightweight and iterative: the engineer is able to quickly and easily gain access to partial and approximate structural information, and may then balance the completeness and accuracy of the information needed with the cost of further applying the technique.

To assess the benefits of these techniques, I present several case studies about the use of structural summarization to support various aspects of change tasks. These case studies range from the use of the techniques on small systems within an academic environment to use on a million lines of code production system within an industrial environment.

## 1.1 The Change Process

The process by which a change is initiated and performed on a software system has itself been changing over the last few years. Traditionally, most software development processes divided the software life-cycle into at least two different phases: development consisting of the determination of the requirements of the system through the delivery of the system to the customer, and maintenance consisting of all changes to the system after the original release [Boehm 1976]. Over the last few years, however, this process has been evolving into a more fluid approach in which the activities of development and maintenance are not considered as separate or disparate. Instead, development is viewed as the ongoing construction and evolution of a series of versions of a software product [Lehman and Belady 1980; Blum 1989; Cusumano and Selby 1995].

In an evolutionary approach, software development engineers and managers must deal, in a timely fashion, with a stream of change requests. Some of these change requests affect the resulting behaviour of the software system and are typically initiated either by users of the product, or from analyses of competing products. Other change requests affect non-functional characteristics of the software system and are initiated by the development organization itself. For instance, change requests to modify the structure of the software system to improve the reusability of software components fall into this latter category. This continuity of change has led Schneidewind to argue that the activities often labeled as maintenance activities are better characterized as change management activities [Schneidewind 1989].

Managing change requires software development personnel to assess, prioritize, and plan changes. Changes then need to be performed according to the plans. Performing these activities requires proposed and executed changes to be analyzed from both managerial and technical viewpoints.

When assessing and prioritizing a change, for example, several managerial questions must be answered. What is the change? What are the benefits of the change? What are the risks of the change? What resources are required to make the change? From a technical standpoint, a different set of questions arises. Is the change possible? What is the effect of the change on the current design? Does the change require other modifications? Answers to these technical questions can help address some of the managerial questions. For instance, the effect of the change on the current design may affect its priority. If the change cannot be easily accommodated, the priority of the change may be downgraded.

Similarly, planning a change often requires an interleaving of managerial and technical decisions. For example, once a change has been selected, there may be more than one way to implement the change. Technical input is required to assess the ramifications of each alternative on various aspects of development, such as the cost of implementing other changes and the cost of testing.

After executing a change, engineers need to check the conformance of the resulting

artifact to the plan. This conformance check attempts to identify deviations between the intended artifact and the actual artifact that results from the change. For example, when performing a change to improve the quality of the code, a conformance check may consist of executing regression tests against the changed artifacts. Conformance checks need not be limited to system behaviour, but may also be used to prevent structural entropy, or unstructuredness, in a system [Lehman 1974; Lehman and Belady 1978].

Ideally, engineers would be able to provide technical answers about the feasibility and the effect of changes based on analyses of the system artifacts. Approaches that support these kind of analyses are referred to as impact analysis techniques. Although a few specialized impact analysis techniques have been proposed, such as traceability metrics [Pfleeger and Bohner 1990] and a WHAT-IF tool [Ajila 1995], in practice, most software engineers answer these questions based on experience and intuition rather than on information collected from and about the system.

## 1.2 The Role of Software Structure

Because the structure of the system plays an important role in the cost of change, information about structure may be beneficial to an engineer reasoning about a system during the change process. Parnas, for instance, in his seminal work on information hiding, showed how two different modularizations, or structures, of a system affected the ability to change the system [Parnas 1972]. Lehman and Belady, based on a study of the development of the OS/360 operating system, noted that as the structure of a system deteriorates over time, the cost and time per unit of work performed on the system tends to increase [Lehman and Belady 1978].

A relationship between structure and the cost of change is also found in other engineering pursuits. The physical structure of a bridge, for instance, dictates the cost of such changes as adding another lane of traffic. For a bridge, the structure is evident and an engineer can start with the physical artifact to analyze the trade-offs associated with various changes. For software, however, the structure is less evident for as Brooks' has

said, the “reality of software is not inherently embedded in space” [Brooks 1986, p. 12].

The structure of software is less evident, in part, because the structure may take many forms. In this dissertation, I use the term, *structure*, in the same broad, flexible sense defined by Ossher:

Any system consists of parts, such as modules, procedures, classes and methods. The *structure* of the system is the *organization* and *interactions* of those parts. [Ossher 1987, p. 219]

To give a sense of the reason for this broad definition, consider the variety of structural information in the artifacts of two example software systems—the Chiron user interface system [Keller et al. 1991] and the GNU groff document formatting system. These two systems were chosen as examples for three reasons. First, the artifacts comprising the systems are publicly available.<sup>1</sup> Second, the systems are implemented in different programming languages; Chiron is implemented primarily in Ada [GPO 1983] and C [Kernighan and Ritchie 1978], and groff is implemented primarily in C++ [Stroustrup 1986]. Third, the systems are of moderate size with Chiron comprising about 20,000 lines of source, and groff comprising about 50,000 lines of source.

Each system is comprised of a variety of parts or *components*. Table 1.1 lists some examples of components in the two systems. Some components, like files and data items (or members), are found in both systems. Others, like packages and classes, depend on the programming language used to implement the system. The components defined for a system are not limited to identifiable pieces of the static system artifacts, but may extend to the system’s dynamic state during execution. In Chiron, for instance, which is designed as several intercommunicating Unix processes, a process may be considered as a component.

Similarly, a variety of interactions, or *relations*, may occur between the components. Table 1.2 enumerates some of these relations. As with the components, the relations are not limited to static properties of the system artifacts, such as the “knows-about”

---

<sup>1</sup>Release 1.4 of Chiron and version 1.10 of groff were used in this study.

Table 1.1: Examples of Software System Components.

System	Types of Components
Chiron	procedure, data type, data item, package, file, directory, process
groff	function, method, class, data member, file, directory, object, executable program

relation between Ada packages, but extend to dynamic relations as well, such as the event interactions between the Chiron system and the underlying X Window System [Scheifler and Gettys 1986].<sup>2</sup>

Table 1.2: Examples of Software System Structural Interactions.

System	Types of Relations
Chiron	calls between procedures, knows-about references between packages, event interactions between procedures, data flow between Unix processes
groff	calls between methods, include references between files, tool-to-tool invocations, assigns interactions between objects

This diversity of components and relations makes it difficult for a software engineer to gain an understanding of a system's structure when performing a change task and also complicates the construction of tools to support the engineer in the understanding process. A second difficulty facing both the engineer and the tool provider is the amount of structural information that must be considered. A simple analysis of the Chiron source, for instance, suggests there are about 76 Ada packages, 84 C header files, 39 C source files, and 162 instances of use of the `with` clause that defines the "knows-about" relation. The core of the groff distribution includes 48 C header files, 10 C source files, 75 C++

---

<sup>2</sup>X Window System is a trademark of the Massachusetts Institute of Technology.

source files, and 37 make utility [Feldman 1979] files. The equation preprocessor alone, which is just one of the approximately five tools and four drivers in the groff environment, includes 40 classes and 223 methods. Running a static analyzer on the source for the equation preprocessor tool reveals approximately 745 calls between methods.

The provision of structural information in a manageable form to the engineer becomes more difficult as a system grows larger than Chiron or groff. As Brooks has noted:

a scaling up of a software entity is not merely a repetition of the same elements in larger size, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some non-linear fashion and the complexity of the whole increases much more than linearly. [Brooks 1986, p. 11]

Techniques and tools to support an engineer during the change process, then, must be flexible to handle the variety of structural information present in a system, must scale to support the large amounts of structural information embedded in most software systems, and must permit the engineer to access the structural information within the time constraints of the change activities.

### **1.3 A Brief Survey of Structure Understanding Techniques**

Various techniques have been proposed and investigated to help meet these requirements. These techniques may be classified into three categories: program visualization, reverse engineering, and metrics. An overview of these categories is provided below; more information is provided in Chapter 10.

#### **1.3.1 Program Visualization**

Program visualization techniques support the display of static and behavioural properties of software system artifacts [Price et al. 1993]. For example, in the SEE program visualization system [Baecker and Marcus 1986; Baecker and Marcus 1990], static properties of C are used to pretty-print the source code as a technical manual with a table of

contents, chapters, and indices. The CIA system [Chen et al. 1990; Chen 1995] presents static properties of C source code in a different form by textually or graphically displaying information extracted from the source, such as the references between functions. The Field system [Reiss 1990; Reiss 1995], amongst other features, combines static and behavioural properties by animating the dynamic execution path in a graphical view of the calling structure. This category also includes systems that permit an engineer to create and query a cross-reference database comprised of information analyzed from a program's source, such as Masterscope [Teitelman and Masinter 1981], GraphLog [Consens et al. 1992], and many commercial programming environments.

Program visualization techniques provide a *direct* representation of selected structural information that may be useful to an engineer performing a change. By direct, I mean that the visualization technique displays or responds to queries in terms of components defined in the source code for the system. The relations displayed by the technique may either be defined in the source or derived from information defined in the source. Figure 1.1 shows a direct visualization produced using the Field system of the calls between methods extracted from the source for the equation preprocessor tool in the groff system.

One advantage of the program visualization approach is that an engineer can easily relate the displayed information to the system artifacts being manipulated. However, since each structural element is typically represented by one visual element—such as a pixel, a line, or a graphical box—an overwhelming amount of data must be displayed for even a moderately-sized system. For instance, it is difficult for an engineer to easily gain an understanding of the structure of the groff equation preprocessor given the visualization shown in Figure 1.1. This density of information is not atypical; for instance, Schwanke and several colleagues report that a display of an *includes* graph for a 56,000 line C program was 25 screens wide and had 48,000 edge crossings [Schwanke et al. 1989]. Engineers attempting to use these systems to perform tasks on large, production-oriented, software systems thus often run into problems associated with scale.

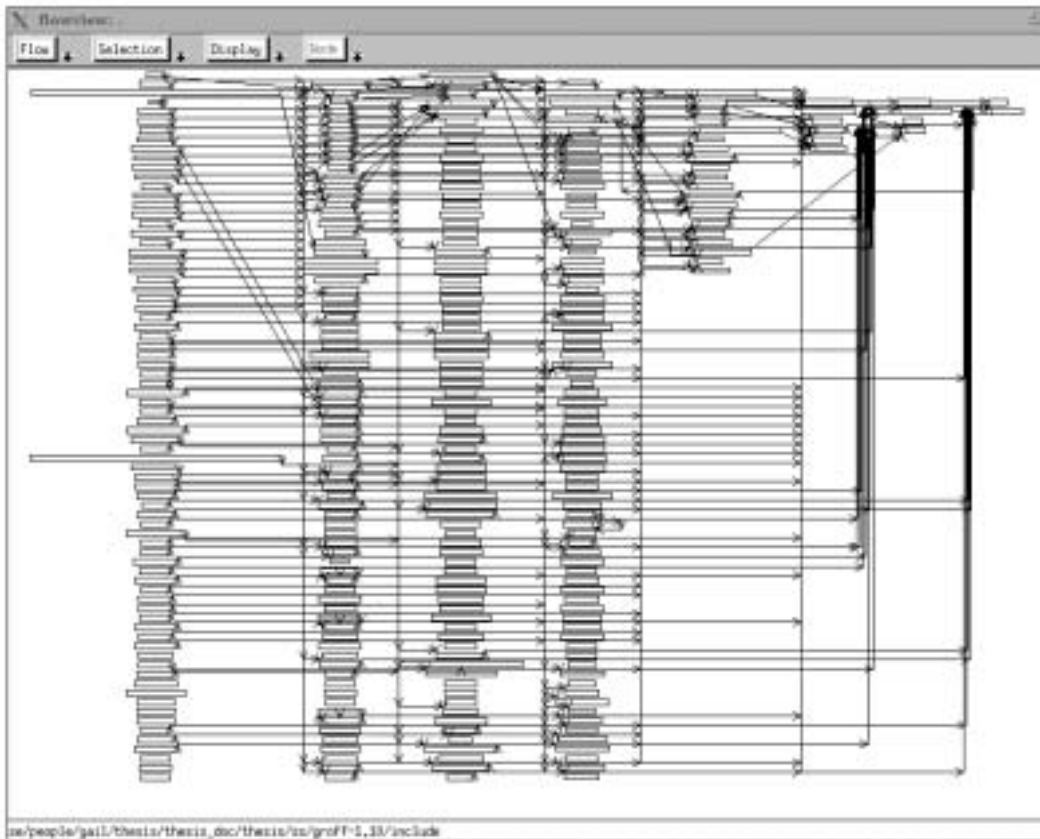


Figure 1.1: A Visualization of the Structure of the groff Equation Preprocessor. This visualization produced by the Field system shows the calls between methods extracted by a static scan of the C++ source for the groff equation preprocessor. Each box represents a method. An arc between two boxes represents a potential call.

### 1.3.2 Reverse Engineering

In part to address the scale problem associated with program visualization techniques, reverse engineering techniques create representations of the system at a higher level of abstraction [Chikofsky and Cross II 1990]. In essence, these techniques derive, either automatically or semi-automatically, a model from information extracted from system artifacts. These models differ from the views produced by program visualization techniques in that the reverse engineered models consist of abstract components rather than components found within the source. The reverse engineering techniques pertinent in this dissertation are those focusing on the abstraction of structural information.

Automated reverse engineering techniques identify higher-level structural components either by numerical methods, or by identifying common structural patterns. A technique characteristic of the former category is Hutchens and Basili's approach that produces a hierarchical module decomposition of a system by clustering procedures based on the number of data bindings between the procedures [Hutchens and Basili 1985]. The technique of Harris, Reubenstein, and Yeh, in which architectural styles expressed as entities and relations are translated into queries over an abstract syntax tree of a program [Harris et al. 1995], is characteristic of the latter category. One benefit of these automated techniques is that a high-level structural model may be produced without requiring a significant amount of the engineer's time. However, these techniques are limited to considering precoded kinds of structural information that are not necessarily suitable for the change task facing the engineer. In addition, these techniques have not been shown to scale for use with large systems.

Semi-automated reverse engineering techniques require the engineer's involvement to derive an abstract model from selected structural information. In the approach supported by the Rigi environment, for example, an engineer repeatedly determines the criteria to use to cluster components in a displayed graph of structural information [Müller and Klashinsky 1989]. The criteria an engineer may use include the naming conventions of

the components and the characteristics of the graph itself, such as the strongly connected components. Figure 1.2 shows a typical graph that an engineer may start with in Rigi; this graph shows the calls between classes extracted from the groff equation preprocessor source code. The window on the upper left of Figure 1.3 shows a model built from the calls between classes graph. This model includes one clustered component, `strong_component`, that results from clustering 43 classes strongly connected to the `accent_box` class. The window on the upper right of Figure 1.3 shows a representation of the 43 classes. The engineer has applied clustering to these classes based on naming conventions: all 34 classes with the string `box` in their name have been clustered under the `classes_named_box` node. The window at the bottom of Figure 1.3 shows the 34 classes with `box` in their name.

The kind of semi-automatic approach supported by Rigi is attractive because the engineer can choose appropriate structural information for the change task from which to begin the clustering process. However, since this information is similar to that used by the program visualization techniques, the engineer must again deal with the scale issues outlined for the program visualization techniques. Furthermore, there is a risk that the high-level model derived using the technique may not be the view of interest to the engineer for performing the task. An example of this kind of situation is described by the Rigi developers based on an initial experience of applying Rigi to a large IBM system:

For our first experiment, we generated a view of the entire call graph without considering any SQL/DS-specific domain knowledge. The result was not as encouraging as we would have liked. The developers did not recognize the abstractions we generated, making it difficult for them to give us constructive feedback. This reaffirmed our belief that successful reverse engineering must do more than manipulate system representations independent of their domain; the results must add value for its customers. Informal information and application specific knowledge provided by existing documentation and expert system developers are rich veins of data that should be mined whenever possible. [Wong et al. 1995, p. 51]

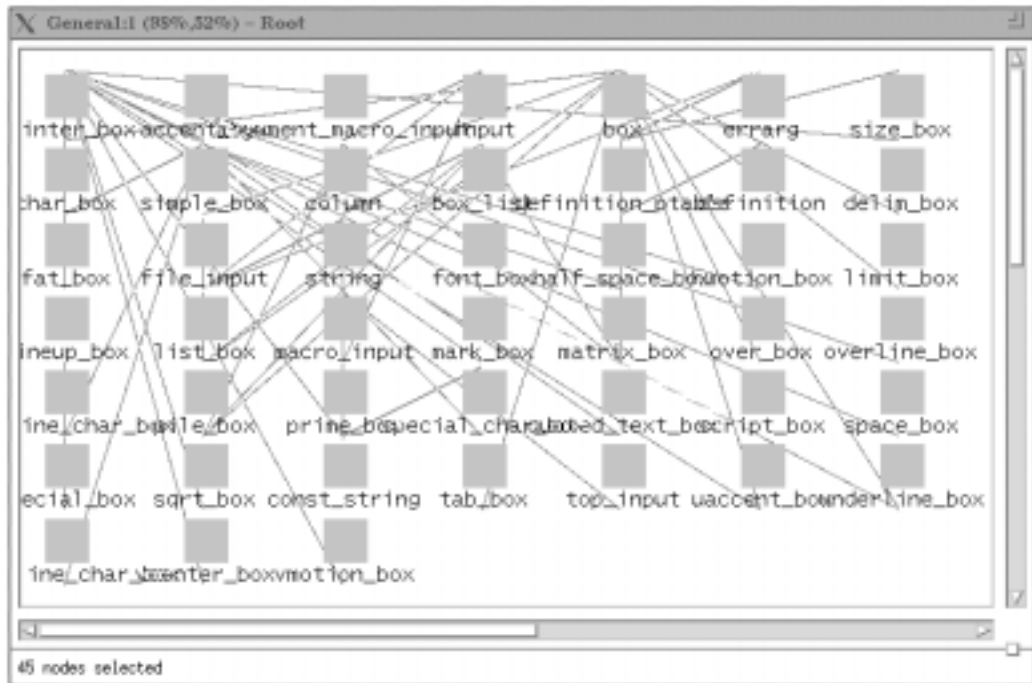


Figure 1.2: An Initial Graph Display Using Rigi. This graph consists of boxes representing classes in the groff equation preprocessor source code and arcs representing statically extracted calls between those classes. This display is similar to one that might be produced by a program visualization technique. With the Rigi semi-automated reverse engineering tool, an engineer starts with this kind of graph and applies a sequence of clustering operations to derive a high-level model of the system's structure.

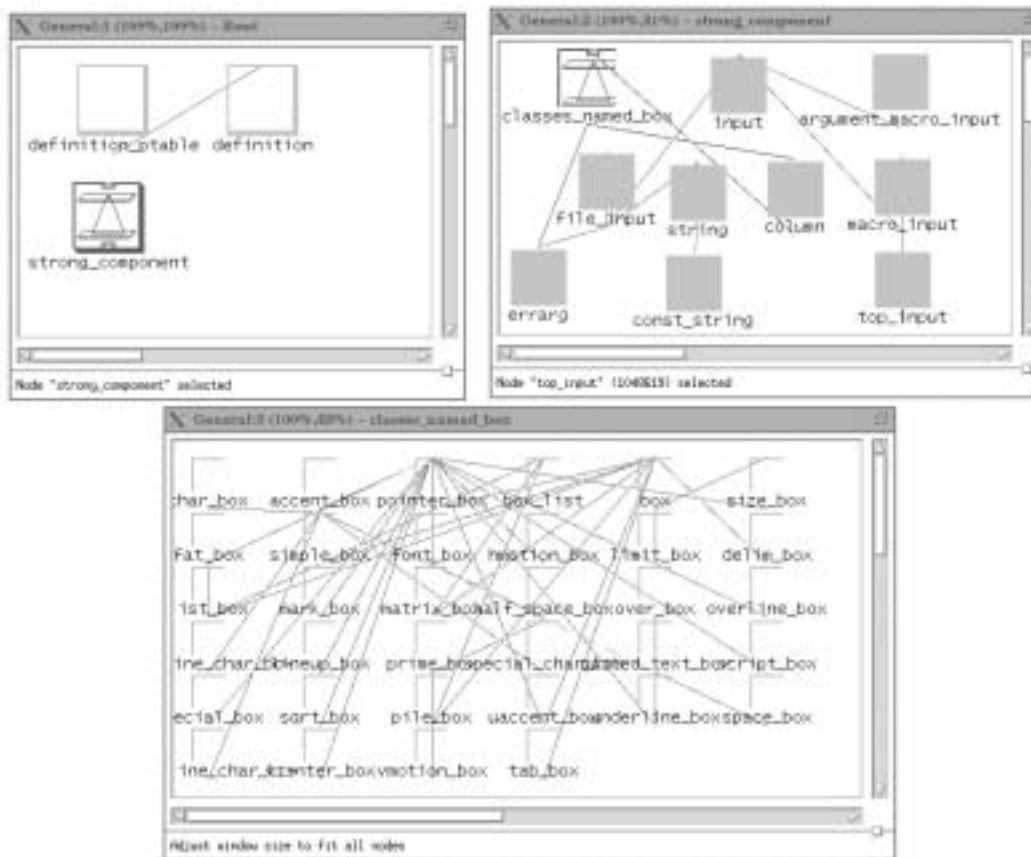


Figure 1.3: A Reverse Engineered Model of the groff Equation Preprocessor. This model was derived using Rigi from information about the calls between classes. The call information was extracted by scanning the C++ source comprising the groff equation preprocessor. This model is hierarchical. The window on the upper left is at the top of the hierarchy. The window of the upper right represents parts of the model clustered under the `strong_component` node shown in the window on the upper left. The window on the bottom represents parts of the model clustered under the `classes_named_box` node shown in the window on the upper right. A node with an icon represents a clustered node; a plain node represents a C++ class.

### 1.3.3 Metrics

Metrics are another way to provide engineers with information about the structure of a system. Boehm defines a metric as “a measure of the extent or degree to which a product...possesses and exhibits a certain (quality) characteristic” [Boehm et al. 1976, p. 596]. Most metrics associated with software structure provide a quantitative measure of structural complexity. For example, given a routine written in a structured programming language, McCabe’s cyclomatic complexity measure calculates the number of linearly independent paths in the control-flow graph for the routine [McCabe 1976]. The numbers returned from such measures have no inherent value, but when they are correlated with one or more desirable software qualities, such as testability, understandability, or maintainability, they may be used as input to the development process. McCabe, for instance, reported a strong correlation between particular ranges of the cyclomatic complexity measure and ease of testing. With such a correlation, this kind of measure may be useful during a code review session [Heimann 1995].

Other measures are more qualitative. Coupling, for example, is based, in part, on the kinds of connections and types of communication between modules [Stevens et al. 1974]. Coupling is lower, for instance, if two modules refer to each other by name through argument passing rather than by referring to each other’s internal elements. Similar to the quantitative measures, these qualitative values are useful only when correlated with desirable qualities.

Many researchers have proposed structural complexity metrics, including McClure [McClure 1978], Yau and Collofello [Yau and Collofello 1980], and Henry and Kafura [Henry and Kafura 1981]. Although these metrics may be useful in identifying parts of a system that are likely to be difficult to change [Kafura and Reddy 1987], they do not help an engineer answer many of the questions of interest in this dissertation, such as which parts may need to be studied for a given change, whether the change requires other modifications, or whether an executed change conforms to the selected design.

## 1.4 The Utility of Summary Information

Each of these existing approaches has one or more serious limitations that hinders—and sometimes prevents—its use when an engineer is assessing, planning, or executing a particular change on a specific system. The basic problem may be characterized as the need to support the comprehension of a large body of information, from multiple viewpoints, within a constrained amount of time.

We all regularly face this basic problem in our information rich world. Monthly personal bank and telephone statements that used to be a page or two have now expanded to multiple pages because they contain descriptions of many more transactions such as each use of a debit card, each collect call, and each calling card call. The World Wide Web gives us almost instantaneous access to archives of information around the globe ranging from quilting patterns to stock quotes. Although this information may prove useful to us, say for optimizing our financial return, there is generally too much of it to comprehend in the time we have available.

One common method we use to deal with these large quantities of information is to summarize. For example, many people enter their financial information into software products running on home computers that permit a person to tally the value and to enumerate all transactions meeting specified criteria, such as all cheque and credit card transactions related to home maintenance. Many web search engines, in response to a query, return a description of a collection of pages matching the criteria. Figure 1.4 shows the results of a query to a web search engine that includes a fragment of information about each page, thus providing a summary of summaries.

These summaries have several important characteristics. First, the summary provides an overview of a collection of information. This overview aids the user in comprehending the information. The collection of short fragments of pages returned by a web search engine, for instance, aids a user in comprehension as compared to a response that would force the user to peruse each selected page in its entirety. Second, the user is involved

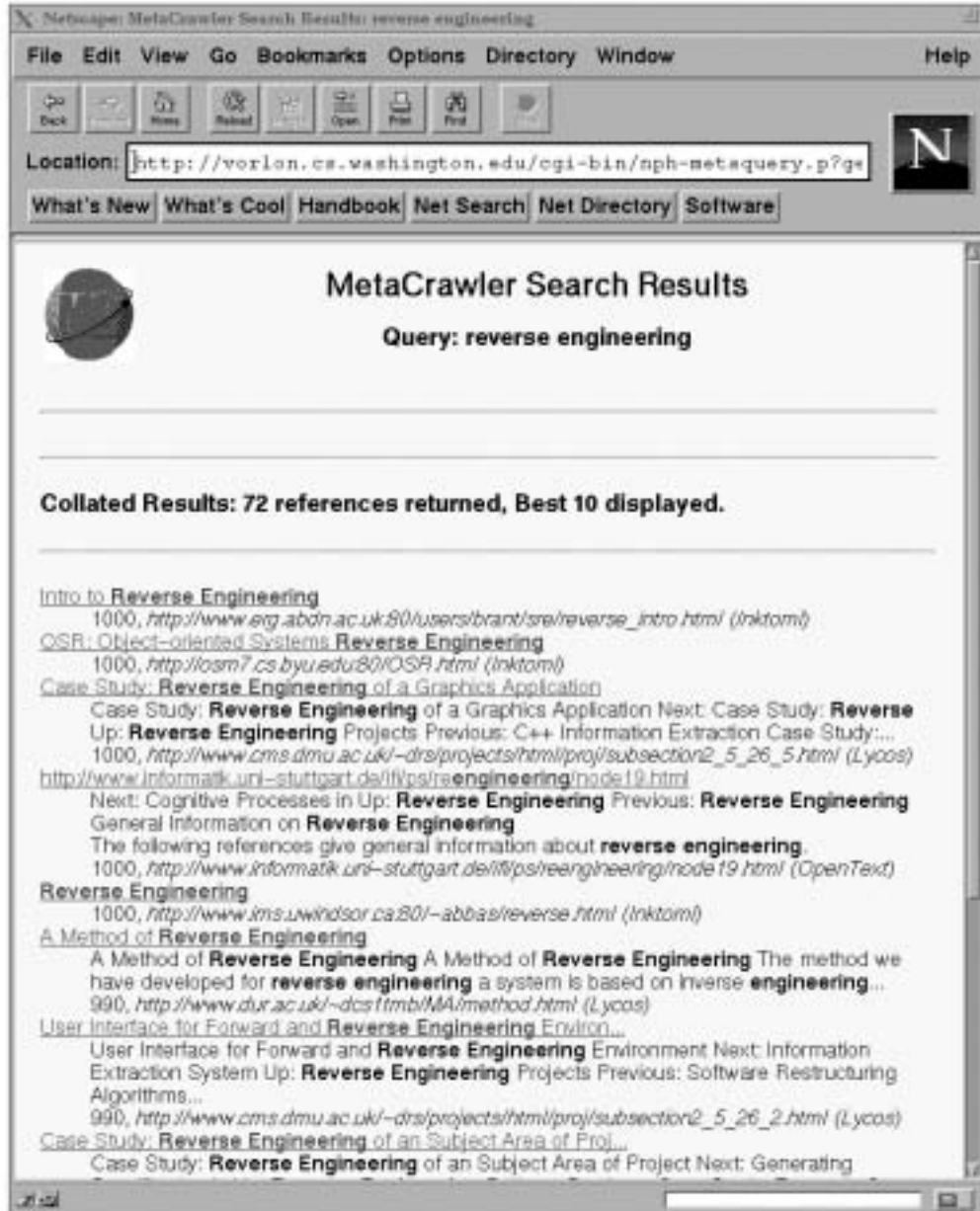


Figure 1.4: A Summary Produced by a Web Search Engine. This window shows a summary of some World Wide Web pages related to reverse engineering produced by the Metacrawler search engine [Selberg and Etzioni 1995]. A summary of each page is also provided.

in selecting the information from which the overview is produced. For instance, the user of the home financial system specifies criteria to define the transactions of interest. Finally, a user can produce a summary in a timely manner; most web search engines, for example, have simple query languages and can begin to provide results from the search almost instantaneously.

How does the process of summarization compare to the processes of filtering and abstraction found in the existing approaches to structural understanding? Filtering, in the context of program visualization, generally refers to the process of altering a stream of selected information. For instance, elision, the suppression of all but a subset of information, may be considered as a simple form of filtering. When using a financial management system, for example, a user might elide all transactions except those concerned with home maintenance. Ball and Eick use this form of filtering to help visualize programs [Ball and Eick 1996]. Even after filtering, though, the set of information might still be too large for the user to reason about it effectively. Furthermore, unlike summarization, the process does not provide any overview of the selected transactions from which an engineer may begin investigating the information. Finally, even if a filter is found, the engineer runs the danger of removing structural information pertinent to the task.

Summarization and abstraction are more closely related. In computer science, abstraction generally refers to the process of creating a construct that *represents* selected information in a more compact form than the information itself. In the context of reverse engineering, this typically means the creation of a construct to hide the details of the underlying information. The process of summarization is similar to abstraction in that an overview of the information is created. Summarization, however, differs in intent from abstraction in that the produced overview is not meant to stand for the selected information. Instead, summarization encourages the details of the selected information to seep through within the context of the overview. This detail helps a user viewing the overview comprehend the information. For instance, letting some of the detail of each

web page show through into the response of a query to a web search engine allows a user to more quickly and easily evaluate both the query and the returned information.

## 1.5 An Overview of Lightweight Structural Summarization

This dissertation describes the use of summarization to aid an engineer in understanding the structure of a software system during the change process. The summarization approach is supported by two separate, but companion, techniques. The *software reflexion model* technique supports the summarization of structural information extracted or collected from a system in the context of a high-level view selected by the engineer [Murphy et al. 1995]; the *lexical source model extraction* technique supports the summarization process by broadening the kinds of structural information that an engineer may easily extract from system artifacts [Murphy and Notkin 1995]. Using this approach, an engineer may begin to reason about a change task based on up-to-date structural information from the system artifacts rather than relying solely on intuition and experience.

### 1.5.1 An Introduction to the Software Reflexion Model Technique

To illustrate the key features of the software reflexion model technique, I present a sketch of its use to aid an engineer with a representative change task. The task facing the engineer is to assess the feasibility of reusing the back-end of the GNU gcc compiler with an existing graphical front-end development environment. Before the back-end may be reused, an engineer must first extract the components comprising the back-end from the source for the compiler.

Since the gcc system is comprised of about 297,000 lines of source code,<sup>3</sup> it is difficult for an engineer to gain an understanding of the system's structure directly from the source code. To apply the software reflexion model technique to aid with this task, the engineer applies the four basic steps outlined in Figure 1.5.

---

<sup>3</sup>Version 2.72 of gcc was used in this study.

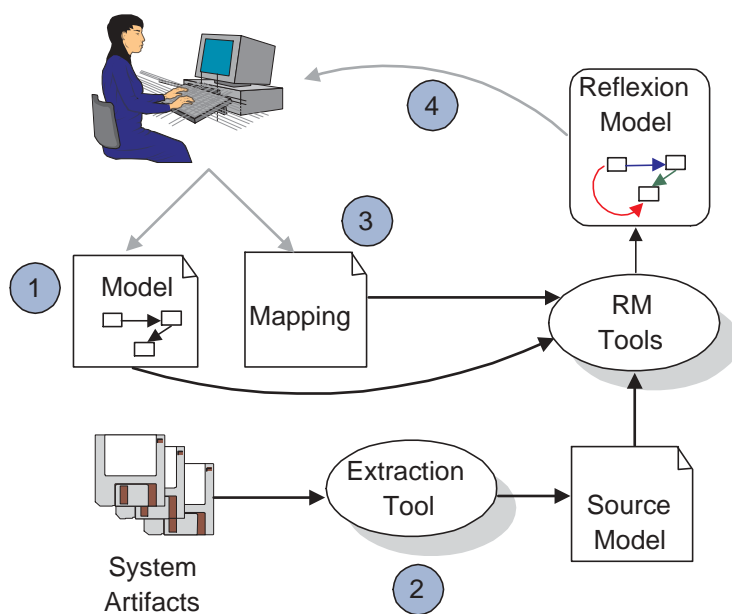


Figure 1.5: The Process of Using the Reflexion Model Technique. The circled numbers refer to the steps in the process.

First, the engineer selects a high-level structural model suitable for reasoning about the task. The high-level model consists of a set of entities, and zero or more relations between the entities. For instance, a high-level model may be a Booch object diagram [Booch 1993] or it may be a more informal sketch of the calls between modules comprising the system. For the extraction task on the `gcc` system, the engineer selects the model shown in Figure 1.6a which is based on a model described by Aho, Sethi and Ullman [Aho et al. 1986, p. 10]. This model consists of a set of modules that primarily represent stages of the compiling process and data interactions between those modules. Based on this view, it appears that it may be relatively simple to separate the back-end from the front-end of the compiler; the engineer need only extract the `Optimization`, `CodeGeneration`, and `SymbolTableManager` modules and remove some dependences from the `SymbolTableManager` module to the `Parsing` and `RTLGeneration` modules.

In the second step of the technique, the engineer extracts structural information—

called a source model—from the artifacts of the system. The source model consists of one or more relations between source components. For instance, a source model may consist of an inheritance relation between classes in an object-oriented system or a relation describing the message sends between objects or both. A source model may be produced either by statically analyzing the system’s source or by collecting information during the system’s execution. In this case, the engineer uses a third-party tool to extract a source model comprising information about approximately 8,000 calls between functions from the C source code comprising `gcc`.

In the third step, the engineer describes a mapping between the extracted source model and the stated high-level structural model. For `gcc`, this mapping contains entries like the following.

```
[ file=^loop\.c mapTo=Optimization ]
```

This entry causes the association of all functions in the `gcc` source file `loop.c` with the `Optimization` module. The mapping for `gcc` consists of 39 entries of this form created from information in the system’s documentation. The language used in the mapping is parameterized so that an engineer may tailor the language to ease the specification of the mapping for a particular system. For instance, an engineer may tailor the mapping language to refer to classes and objects if working with a system implemented in an object-oriented language.

In the fourth step, the engineer uses a tool to compute a software reflexion model which provides a comparison between the source model and the posited high-level structural representation. The computed software reflexion model for this change task on `gcc` is shown in Figure 1.6b. In the reflexion model, solid lines, called *convergences*, indicate interactions discovered in the source that were expected by the engineer in the high-level model. (Figure 1.6a shows the expected interactions.) Dashed lines, called *divergences*, indicate discovered interactions that were not expected by the engineer, and dotted lines, called *absences* indicate interactions that were expected but not found. The number

attached to each arc indicates the number of calls in the source associated with the interaction. The software reflexion model shown in Figure 1.6b summarizes 3704 calls found in the `gcc` source.

Using the reflexion model, the engineer can reason about the change task more effectively. For instance, the divergences (dashed arcs) from the back-end `Optimization` and `CodeGenerator` modules to the front-end `Parsing` module suggest that separating the back-end from the front-end may be more difficult than anticipated from the high-level view (Figure 1.6a) because additional dependences must be removed. To assess this difficulty, an engineer may investigate the software reflexion model by following a link from an arc shown on the reflexion model to the calls represented by the arc. For example, an engineer can view the calls associated with the arc from `Optimization` to `Parsing` (Figure 1.7). This traversal is similar to those made by a user following links from a page summarizing the results of a search with a web search engine to the actual pages of interest for more detail. Also in a similar fashion to a user of web search engine, based on an investigation of the software reflexion model, the engineer may decide to refine one or more of the inputs to the computation, and may iteratively recompute a reflexion model. For instance, the engineer may broaden the structural information summarized by adding to the source model references by functions to global variables.

The software reflexion model provides the engineer with a summary of the interactions in the source model in the context of a view the engineer selected as useful for reasoning about a change task. This summary is provided in the form of a comparison between two models: a high-level model and a source model. As is characteristic of summaries, details about the underlying information are not hidden, but rather, the details show through as the numeric values associated with each arc in the reflexion model. This transparency of detail, together with the ability to jump to the underlying interactions, helps an engineer to more easily move between the different cognitive system models engineers use when performing maintenance tasks [von Mayrhauser and Mans 1996]. Furthermore, software reflexion models can be produced in the necessary timely and low

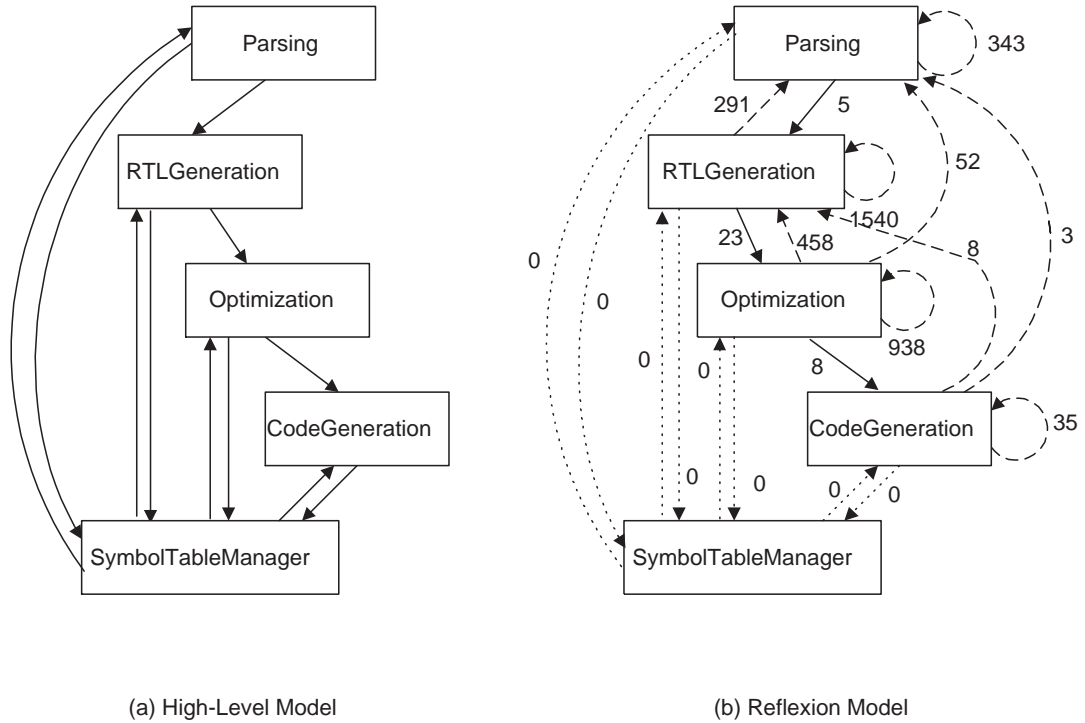


Figure 1.6: A High-level Model and a Software Reflexion Model for `gcc`. The boxes in both models represent modules. The arcs in the high-level model represent the flow of data, while the arcs in the reflexion model represent call interactions between modules. Three kinds of arcs are shown in the reflexion model: solid arcs are called convergences and represent expected interactions, dashed arcs are called divergences and represent unexpected interactions, and dotted arcs are called absences and represent interactions that were expected but not confirmed in the comparison. The numbers attached to reflexion model arcs indicate the number of source model interactions associated with that arc.<sup>5</sup>

```

combine_instructions in combine.c  calls  mode_for_size in stor-layout.c
simplify_comparison in combine.c  calls  mode_for_size in stor-layout.c
flow_analysis       in flow.c     calls  oballoc      in tree.c

```

Figure 1.7: A Sample of Calls in `gcc`. Three sample calls from the `gcc` source that are associated with the divergence from the `Optimization` module to the `Parsing` module.

cost fashion since the technique is *lightweight*; an engineer has often been able to specify the inputs and compute a software reflexion model in about an hour.

The software reflexion model technique also scales. Two properties of the technique, in particular, are helpful in managing scale. First, the technique is *iterative*, allowing an engineer to compute an initial software reflexion model based on an *approximate* (coarse-grained) association of structural source entities with the high-level model, and then, over time, compute a succession of progressively more refined software reflexion models. At any time in this progression, the engineer may stop and use the current software reflexion model for reasoning about and for performing the task at hand. An engineer may thus balance the cost of applying the technique with the degree of refinement necessary for the task at hand. Second, the technique is *partial*, enabling an engineer to compute a software reflexion model for only those pieces of a system pertinent to the task at hand.

### 1.5.2 An Introduction to the Lexical Source Model Extraction Technique

All software structure understanding techniques share one, unsurprising, requirement: the need for structural information. Sometimes, the structural information may be gathered by monitoring the execution of a system. For example, an engineer interested in how one system interacts with another across an OLE interface [Brockschmidt 1995] might use an external monitoring tool to track the desired interactions. Other times, the structural information may be gathered by applying static analysis techniques to the source code for the system. For example, an engineer might also determine the possible OLE interactions by statically analyzing the source code for the system.

In trying to apply existing lexical and syntactic static analysis tools to the extraction of structural information from source—both for the purposes of computing reflexion models and for using existing software visualization and reverse engineering tools—I discovered several problems. Existing syntactic tools, like the Field and CIA tools,

---

<sup>5</sup>For presentation purposes, all reflexion models shown in this dissertation have been redrawn from the output produced by the reflexion model tools that are described in Chapter 3.

were difficult to use for a wide variety of systems because they were sensitive to the condition of the source. These tools, for instance, were not tolerant when the system was written in a slight variant of a programming language, or when the system was not compilable because it was in the midst of being ported. Moreover, without significant modifications, these tools could not extract some structural interactions of interest such as event interactions, nor were they capable of scanning many of the system artifacts containing structural information such as data files or documentation files.

To access the structural information of interest, it was thus often necessary to write special-purpose extractors using existing lexical tools like `awk` [Aho et al. 1979] and `perl` [Wall 1990]. Although these lexical tools support the scanning of many different kinds of system artifacts, the lexical specifications needed to generate an appropriate extractor were often long, complex, and difficult to maintain.

The lexical source model extraction technique was developed to overcome some of these limitations by retaining the *flexibility* and *tolerance* inherent with a lexical approach, while providing a *lightweight* style of specification. In essence, the technique permits engineers:

- to use regular expressions to describe patterns of interest in system artifacts;
- to specify actions to execute when a pattern is matched to part of an artifact; and
- optionally, to specify operations for combining matched information to compute structural interactions.

For example, an engineer desiring to compute the calls between functions from the C source code for `gcc` might use the patterns shown in Figure 1.8 as input to the lexical source model extraction tools. The first (parent) pattern describes how to recognize function definitions in K&R-style C code, while the second (child) pattern describes the form of calls within a function body. The first pattern states that a function definition might start with an optional type declaration followed by the name of a function, an opening

parenthesis, optional formal arguments, a closing parenthesis, optional argument type declarations, finally ending with an opening curly brace. From these patterns, a scanner can be generated that will scan a system artifact for these patterns. The generated scanner will attempt to match zero or more call (child) patterns for every occurrence of the function definition (parent) pattern. When a call is matched to the source code, the action code (within @ symbols) is executed, writing out the call interaction. The technique also supports the generation of analyzers that compute a desired source model from information collected from one or more scans of the system artifacts.

```
(1) [ <type> ] <functionName> \( [ { <formalArg> }+ ] \)
    [ { <type> <argDecl> ; }+ ] \{

(2) <calledFunctionName> \( [ { <parm> }+ ] \) @
    write ( functionName, " calls ", calledFunctionName ) @
```

Figure 1.8: A Pattern to Extract Calls from C Source Code.<sup>6</sup>

Several features of the technique simplify the description and extraction of structural information from system artifacts. For instance, the ability to specify patterns hierarchically simplifies the description of many constructs found in source code. Heuristics built into the generated scanners ease the matching of patterns to the appropriate portions of a system artifact.

Similar to the software reflexion model technique, the lexical source model extraction is applied iteratively. An engineer writes a coarse specification to extract *approximate* and *partial* structural information. The specification is then refined over several iterations until the desired information is produced.

Although the lexical source model extraction approach can be used to generate source models for the software reflexion model technique, the two techniques and supporting tools are separate. Source models extracted with the lexical source model extraction technique may be used as input for other structural understanding approaches such as

---

<sup>6</sup>Details of the pattern language are provided in Chapter 6.

program visualization and reverse engineering techniques. As described in the previous section, a source model for computing a reflexion model may be produced using another extraction approach.

## 1.6 The Organization of the Dissertation

The core of this dissertation begins with a description and discussion of the software reflexion model technique. In Chapter 2, I describe the essence of the technique both informally, by presenting a detailed example, and formally, by presenting a specification of the technique written in the Z formal language [Spivey 1992]. Chapter 3 then addresses the computation of reflexion models, describing tools and algorithms developed to support the use of the technique on large systems. In Chapter 4, several refinements to the technique motivated by its use on large (million line) systems are described. As these refinements require further validation, I describe them separately from the core technique. Finally, in Chapter 5, I discuss choices made in the design of the technique.

The dissertation then moves into a description and discussion of the supporting lexical source model extraction technique. In Chapter 6, I describe the technique from the viewpoint of an engineer trying to extract structural information from system artifacts. Chapter 7 describes the tools that support the technique and Chapter 8 discusses issues surrounding the design of the technique.

When developing new techniques, it is critical to determine if the concepts underlying the technique have value. I investigated the benefits and drawbacks of the lightweight structural summarization approach through several case studies that are described in Chapter 9. In particular, I describe how an engineer at Microsoft Corporation used the software reflexion model technique to support an experimental reengineering of the Excel spreadsheet product which consists of over a million lines of C source code.

The final two chapters conclude the thesis. Chapter 10 covers related work. In Chapter 11, I review the concept of structural summarization to aid an engineer evolving a software system, describe the contributions of the research, and discuss future work.

## Chapter 2

# Software Reflexion Models

The difficulty of dealing with structural information embedded in a system's source often leads software engineers to rely on high-level models that are separate from the source. Sometimes these high-level models are part of the documentation of a system. For instance, the documentation for a system might contain an architectural diagram describing the major components of a system and how those components interact. Other times, the models are “mental models” that engineers' have developed from experience with systems. These models often appear as box and arrow sketches on white boards or on serviettes when engineers discuss a proposed change to a system.

Although these high-level models can be useful for reasoning about a change, they are also dangerous because they may not be an accurate representation of a system's source. In the end, it is the source that the engineer must update to affect the desired change. As Lindvall notes about a domain object model contained in the documentation of commercial mobile telecommunications product:

To be usable in the future, the domain object model has to be updated continuously. As long as it contains wrong information, it will never be used. If it is used, but not updated—it is dangerous. [Lindvall 1994, p. 68–69]

Instead of requiring high-level models to be continuously updated, the software reflexion model technique reduces the risk associated with reasoning in terms of them by bridging the gap between the models and the source. In essence, the engineer selects and

uses a task-specific high-level model as a framework on which to summarize structural information extracted or collected from the software system's artifacts. Using the hints provided in the reflexion model, the engineer may then selectively investigate aspects of the system's source pertinent to the task. The technique is sufficiently lightweight and iterative so that an engineer, based on the particular needs of the task, may cost-effectively explore a number of structural aspects of the system in varying levels of detail.

In this chapter, I describe the software reflexion model technique from two perspectives. First, I describe the technique from the perspective of an engineer performing a change task on a software system. Then, to make precise the meaning of a software reflexion model, I present a formal characterization of the technique. The tools built to support the technique and refinements added to manage scale are discussed in the following two chapters.

## 2.1 An Example

Consider an engineer asked to estimate the cost of changing the virtual memory subsystem of the NetBSD<sup>1</sup> Unix operating system to page across a network rather than to a file system. The engineer, who is an expert Unix virtual memory system developer, but who has no experience with the NetBSD implementation, is asked to provide the estimate within five working days.

To provide an estimate with a reasonable level of confidence, the engineer needs to understand the basic structure of the virtual memory subsystem of the NetBSD implementation. The NetBSD implementation comprises about 250,000 lines of C [Kernighan and Ritchie 1978] source code spread over approximately 1900 source files.<sup>2</sup> This immensity of source makes a direct perusal of the code impossible for the engineer working under a five day time limit.

Instead, the engineer might reason about the task in terms of a high-level model that

---

<sup>1</sup>NetBSD is a public-domain implementation of a Unix operating system available for a variety of platforms.

<sup>2</sup>To aid the reader, citations are included to the first reference of a work in each chapter.

is based on the engineer's experience with other Unix virtual memory subsystem. This model, shown in Figure 2.1, consists of a modularization of a virtual memory subsystem implementation and the call paths between those modules. Based on this model, an engineer might reason that the desired change is easy to accomplish by cloning the file system module and migrating it to a module that accesses the network. But, how much confidence can the engineer have in estimates that are based on this model when there is little evidence that this model is a reasonable representation of the NetBSD implementation?

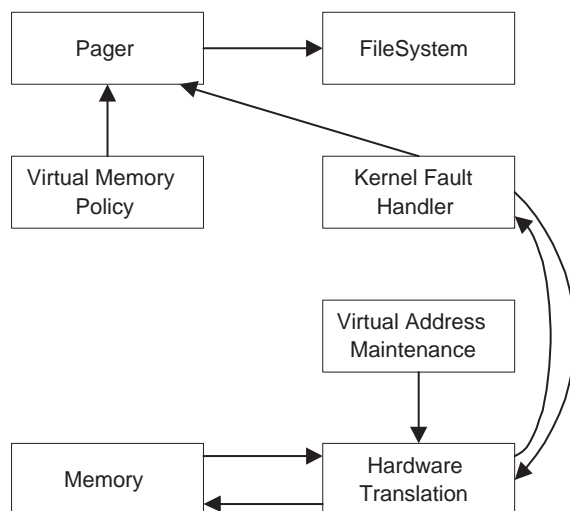


Figure 2.1: A High-level Model of a Unix Virtual Memory Subsystem. The boxes represent modules and the arcs represent call paths between those modules.

The reflexion model technique provides a means of increasing the engineer's confidence in producing estimates for the change task by enabling a comparison between the model shown in Figure 2.1 and structural information extracted from the NetBSD implementation. The approach consists of four basic steps (illustrated earlier in Figure 1.5).

### 2.1.1 Selecting a High-Level Model

First, an engineer defines a high-level model suitable for understanding the virtual memory subsystem of NetBSD. Since the engineer needs to understand the interactions be-

tween the pager and other components of the virtual memory subsystem, the engineer posits the high-level model shown earlier in Figure 2.1.<sup>3</sup>

### 2.1.2 Extracting a Source Model

Second, the engineer uses a tool to extract structural information from the artifacts comprising the NetBSD implementation to form a source model. Since, the source code was compilable, the engineer uses a syntactically-based tool—in this case, the Field programming environment [Reiss 1990; Reiss 1995]—to extract a source model consisting of the calls between functions in the NetBSD implementation. This source model comprises over 15,000 calls between functions.

### 2.1.3 Defining a Map

Third, the engineer defines a mapping between the entities in the source model (i.e., functions) and the entities in the high-level model (i.e., modules). One way to create associations of source model entities to high-level model entities is to enumerate the associations explicitly. However, this approach is infeasible given that the source model for NetBSD has over 3000 functions. This seeming difficulty is mitigated using three techniques to collapse the size of the mapping specification. First, the map may be partial so that the engineer need only provide associations for those portions of the system of interest for the task at hand. Second, the engineer may use the physical (e.g., directory and file) and logical (e.g., functions and classes) structure of the source to name many source model entities in a single map entry. Finally, the engineer may use regular expressions to take advantage of naming conventions in the system artifacts.

Using these techniques, a declarative mapping for the virtual memory subsystem of NetBSD consists of the six entries shown in Figure 2.2. The regular expressions in the map obviate the need to enumerate a large set of structures. The first entry in the map

---

<sup>3</sup>Although the task at the heart of this example is hypothetical, the inputs to this reflexion model computation were defined by an expert Unix virtual memory subsystem developer.

```

[ file=.*pager.*      mapTo=Pager ]
[ file=vm_map.*      mapTo=VirtualAddressMaintenance ]
[ file=vm_fault.c    mapTo=KernelFaultHandler ]
[ directory=.*fs     mapTo=FileSystem ]
[ file=pmap.*        mapTo=HardwareTranslation ]
[ file=vm_pageout.c  mapTo=VirtualMemoryPolicy ]

```

Figure 2.2: A Map for the NetBSD Virtual Memory Subsystem. The map consists of six entries; each entry is enclosed in square brackets. An entry consists of a list of keyword and value pairs describing zero or more source model entities, and a list of keyword (always the phrase `mapTo`) and value pairs describing high-level model entities. The keywords available for use in describing source model entities are defined by a source entity description language described in Section 3.1.1. In this example, the keywords `file` and `directory` are used. Values to specify source model entities may be regular expressions.

shown in Figure 2.2, for example, names all four files whose name includes the string “pager”. The structure of the source itself is used to obviate the need to name each piece of substructure. The second entry, for example, associates all 37 functions defined in files containing “vm\_map” as part of their filename with the `VirtualAddressMaintenance` module in the high-level model. Although all entries in the example associate zero or more source model entities with only one high-level model entity, an entry may associate zero or more source model entities with multiple high-level model entities.

Details of the mapping language are discussed further in Section 3.1.1.

#### 2.1.4 Computing a Reflexion Model

From the selected high-level model, the extracted source model, and the defined mapping, a reflexion model may be computed and displayed. The high-level model and the computed reflexion model for the NetBSD virtual memory subsystem are shown side-by-side in Figure 2.3.

As described in Chapter 1, the solid lines in the reflexion model (Figure 2.3b) show the

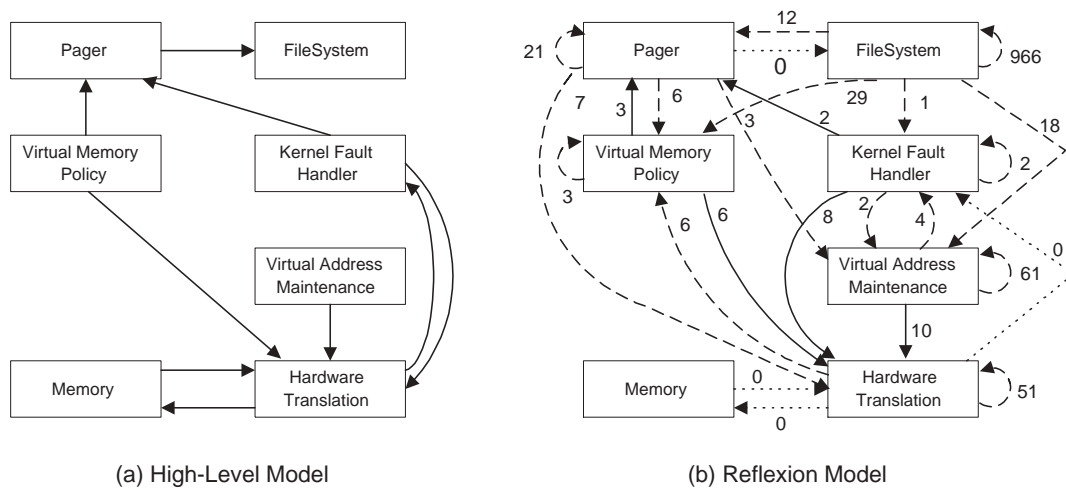


Figure 2.3: High-level and Reflexion Models for the NetBSD Virtual Memory Subsystem. The high-level model is reproduced from Figure 2.1. The solid lines in the reflexion model are convergences indicating predicted source interactions; the dashed lines are divergences indicating unpredicted source interactions; the dotted lines are absences indicating predicted, but unsubstantiated, interactions.

*convergences*, where the source model agrees with the high-level model. For instance, as the engineer expected, there were three calls found in the source between functions in modules implementing `VirtualMemoryPolicy` and functions in modules implementing `Pager`. The dashed arrows show the *divergences*, where the source model includes interactions not predicted by the high-level model. For instance, the dashed line from `FileSystem` to `Pager` indicates that functions within modules mapped to `FileSystem` make twelve calls to functions within modules mapped to `Pager`. The dotted lines show the *absences*, where the source model does not include interactions predicted by the high-level model. For instance, no calls were found between modules mapped to `Pager` and modules mapped to `FileSystem`. The number associated with each arc in the reflexion model is the number of source model relation values mapped to the arc; absence arcs, by definition, always have zero mapped values.

In essence, the reflexion model display in Figure 2.3b provides a comparison between the source model and the high-level model. This reflexion model summarizes 1221 calls

related to the virtual memory subsystem from the NetBSD source model.

It takes about 30 seconds on a DECStation 3000/300X running OSF/1 V3.2 to compute and display this reflexion model. I discuss the performance of the tools for computing reflexion models in more detail in Chapter 3.

### 2.1.5 Interpreting a Reflexion Model

A static display of the reflexion model does not provide enough information for the engineer to reason about the estimation task. Rather, the engineer needs to understand the connection between the high-level model and the source model for the parts of the model relevant for the task at hand.

An engineer learns about the correspondences between the two models by investigating and interpreting the source model interactions that are mapped to a particular arc in the reflexion model. The screen snapshot in Figure 2.4 shows the result of selecting the divergence between `FileSystem` and `Pager`. The 12 values in this window show the calling and called functions with their associated directory and file information.

An engineer's experience and judgment are essential in interpreting reflexion models. Convergences tend to increase an engineer's belief in the trustworthiness of the model, while divergences and absences tend to decrease this belief.

As expected, for instance, two of the three calls mapped to the convergence from `VirtualMemoryPolicy` to `Pager` correspond to the policy module requesting a pager to page out a page. However, a convergence can also be deceiving if it appears because of a different set of calls than expected by the engineer. The third call between the `VirtualMemoryPolicy` and `Pager` modules, for instance, is the result of a pageout daemon in the policy module asking all pagers to clean up from any asynchronous paging activities. The engineer may not have anticipated this interaction when stating the calling relationship between the two modules, but it may affect the features that need to be supported by a network paging module. This knowledge will affect the estimates for the change task produced by the engineer.

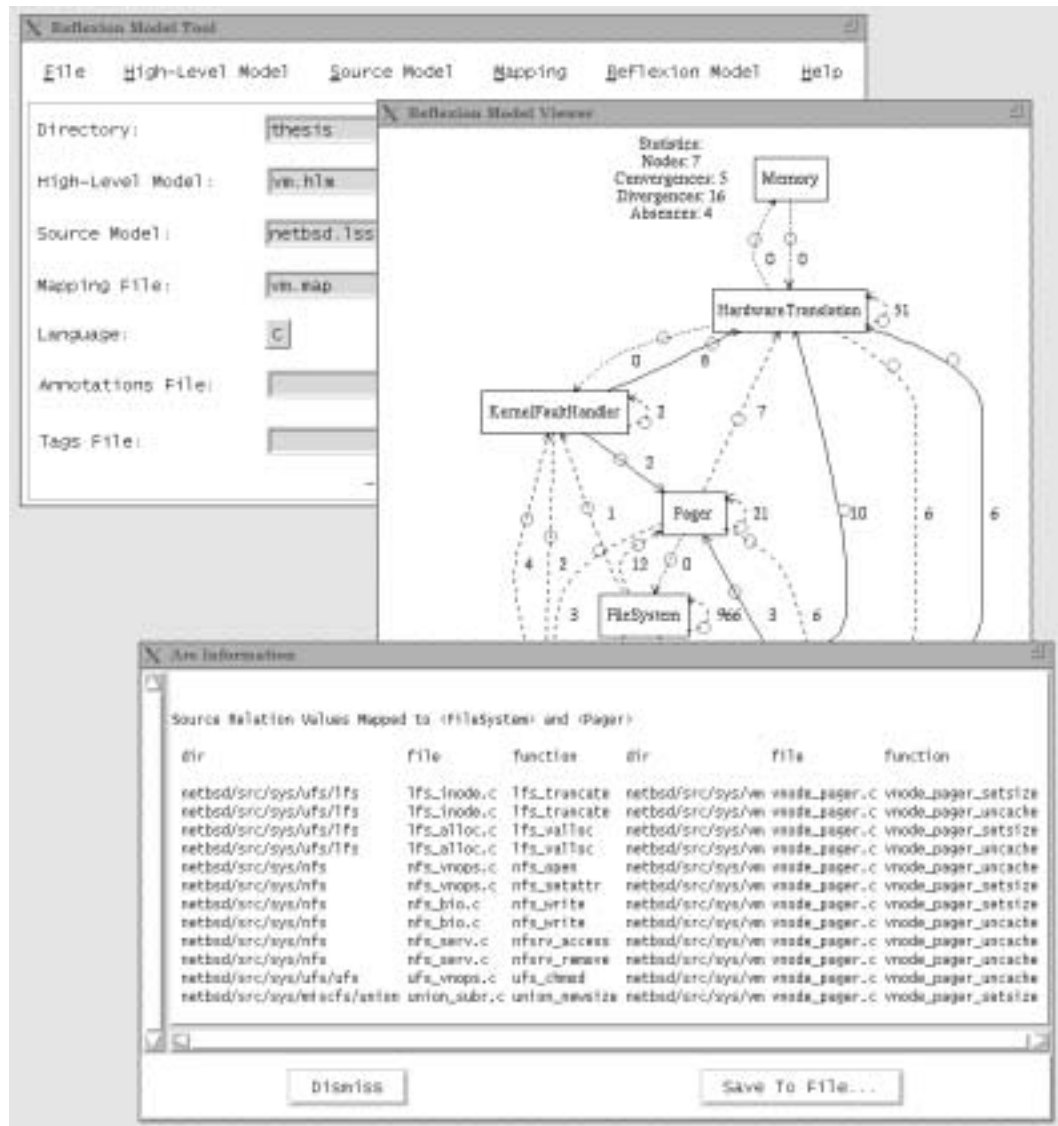


Figure 2.4: Investigating Source Through a Reflexion Model. The engineer has selected the arc from `Filesystem` to `Pager` in the displayed reflexion model and requested a display of the twelve calls in the source model that correspond to the interaction.

Similarly, divergences must be carefully interpreted. The divergence from the `Pager` to the `HardwareTranslation` module is the result of the pager directly translating addresses rather than using the `KernelFaultHandler` for this purpose. This interaction may be the result of a performance optimization. As a result, the interaction is not necessarily bad, but it may affect how the `Pager` interacts with a new network paging module. Again, this knowledge will impact the estimation task being performed.

Neither are absences necessarily bad; absences often arise where a high-level model entity has no identified representation in the source code. In the reflexion model shown in Figure 2.3, for example, the absences to and from the `Memory` entity arise because the engineer has not included an entry in the mapping file that names the `Memory` module. These absences may affect the confidence level an engineer associates with estimates produced using knowledge obtained from the reflexion model.

### 2.1.6 Refining a Reflexion Model

To further investigate the structure of the system while performing the estimation task, the engineer may compute a succession of reflexion models. Any of the three inputs—the high-level model, the map, or the source model—may be varied between computations.

For instance, sometimes an engineer may decide, during the interpretation of a reflexion model, to refine the high-level model. The engineer investigating the reflexion model for the virtual memory subsystem of the NetBSD system might decide, for example, that the divergence from `Pager` to `HardwareTranslation` represents an arc omitted from the high-level model. The engineer may choose to refine the high-level model to explicitly remember and capture this interaction, and then may recompute the reflexion model. This recomputation transforms the divergence into a convergence. The goal of the engineer is not necessarily to eliminate divergences from the model, but rather, to use the refinement of the high-level model to document, even temporarily, knowledge about the Unix virtual memory subsystem.

At other times, an engineer may discover that refinements to the mapping are needed

to correct inappropriate entries in the map file, or to increase the number of source model values mapped by the map. The engineer, for instance, may add to the end of the map file the following map entry.

```
[ dir=arch/sparc/sparc file=mem mapTo=Memory ]
```

This entry associates all functions in files with “mem” in their name that reside in the sparc architecture directory with the **Memory** module. With this map entry, the engineer is attempting to include source module values concerned with the **Memory** module in the computation of the reflexion model. The addition of this map entry will likely increase the number of calls summarized by the reflexion model and may transform some existing absences into convergences, or add some new divergences.

Less frequently, an engineer may determine that more structural information is needed in a source model. For the estimation task on the virtual memory subsystem, the engineer may also want to investigate the data coupling between modules related to virtual memory policies. To include this information, an engineer may use the Field system to extract the relevant data reference information and then may use this information to augment the source model. Performing this activity adds 21 tuples to the source model. The engineer must then update the map to include this newly extracted data reference information in the next reflexion model computation. The engineer adds the following entry to the beginning of the map used in the NetBSD reflexion model computation.

```
[ variable=page.*active mapTo=VirtualMemoryPolicy ]
```

This map entry associates all functions referencing global variables with “page” followed by “active” in their names to the **VirtualMemoryPolicy** module.

Figure 2.5 shows the high-level model and reflexion model for the NetBSD virtual memory subsystem after the changes described above. In comparison to the originally computed reflexion model shown in Figure 2.3b, this reflexion model indicates that there are interactions between the **Memory** module and the **VirtualMemoryPolicy** module; this

interaction may effect the estimation task because the `FileSystem` module also interacts with the `VirtualMemoryPolicy` module.

When changes are made either to the high-level model or to the mapping, a new reflexion model can typically be recomputed within a minute or two. (Section 3.3 discusses the performance of the computation in more detail.) Changes involving the extraction of additional structural information from the source require additional time, typically on the order of tens of minutes or longer for large systems. However, these changes generally have minimal impact on the iterative use of the technique because they are far less frequent.

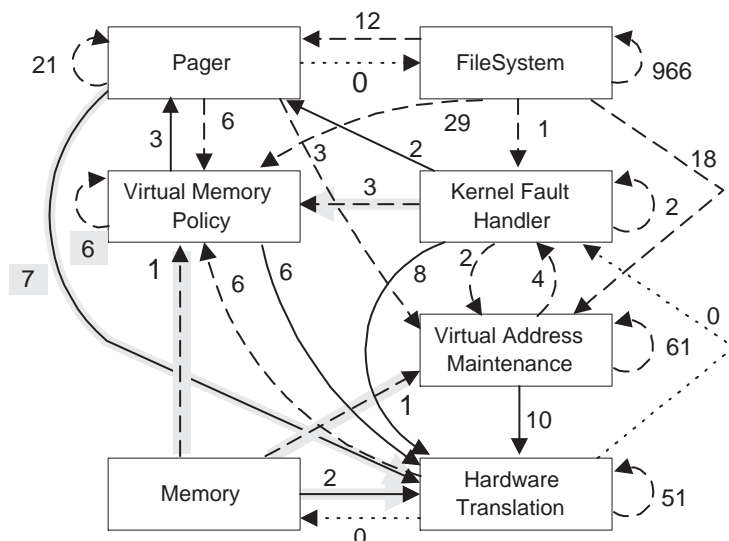


Figure 2.5: A Refined Reflexion Model for the NetBSD Virtual Memory Subsystem. This reflexion model summarizes both calls between functions and references from functions to global variables representing virtual memory policies. The areas in grey indicate portions of the reflexion model that are different from the reflexion model shown in Figure 2.3.

### 2.1.7 Using a Reflexion Model

The reflexion models computed for the NetBSD virtual memory subsystem provide valuable information to the engineer estimating the cost of changing the virtual memory

system to page over a network rather than page to a file system. For instance, it is likely that a lower-cost estimate would have been produced if the engineer had based the estimates on the original high-level model shown in Figure 2.1 because the model shows a `Pager` module with a simple one-way interaction to the `FileSystem` module. The eventual reflexion model shown in Figure 2.5 provides even more information, showing that the `Pager` has several unanticipated interactions with other modules of the system. For example, the `Pager` makes calls to the `VirtualMemoryPolicy` module. Moreover, the interaction between the `FileSystem` and the `Pager` is in the opposite direction than expected. Based on an inspection of the source model values associated with this divergence, and on an inspection of the related source code, the engineer determines that the pager requests file system operations through indirect function calls that are not included in the source model, and that the requests made by the file system of the pager are performance optimizations. From this information, the engineer can estimate the cost of the change more accurately. This information may be obtained at low cost because the engineer can quickly and easily compute a “rough” reflexion model and then iteratively refine the inputs to the computation as additional detail is necessary and as time permits.

Although the change task described in this section is hypothetical, the software reflexion model technique has been used by engineers on production systems. These uses of the technique are described in Chapter 9.

## 2.2 Formal Characterization

In this section, I present a formal specification of the reflexion model technique. This specification is provided for two reasons. First, the specification makes precise the meaning, computation, and display of a reflexion model. Precise descriptions of these aspects of the technique are necessary for implementors developing tools to support the technique, and may also be helpful to other researchers developing similar techniques. In particular, the specification highlights aspects of the technique that may be parame-

terized. Second, the specification is used, in later chapters, as a basis for discussing the computability of reflexion models. Readers primarily interested in the use of the technique and tools may choose to skip directly to Chapter 3.

The specification is written in the Z [Spivey 1992] language. A Z specification models the data of a system using mathematical data types and describes the effects of system operations using predicate logic. The specification is decomposed into several schemas linked by a natural language commentary. For readers unfamiliar with Z, I provide a summary of the Z notation used in this dissertation in Appendix A.

### 2.2.1 Reflexion Model

A static schema in Z describes a state the system can occupy as well as invariants that must be maintained across state transitions. Any system implementing the reflexion model technique must maintain the state components and invariants described in the *ReflexionModel* schema presented below.

The *ReflexionModel* schema uses two basic types, *HLEMENTITY*, which represents the type of a high-level model entity, and *SMENTITY*, which represents the type of a source model entity. Four type synonyms are also defined: *HLMRelation* and *SMRelation*, which define relations over high-level model entities and source model entities respectively, and *HLMTuple* and *SMTuple*, which define the types of tuples in the *HLMRelation* and *SMRelation*.

[*HLEMENTITY*, *SMENTITY*]

*HLMRelation* == *HLEMENTITY*  $\leftrightarrow$  *HLEMENTITY*

*SMRelation* == *SMENTITY*  $\leftrightarrow$  *SMENTITY*

*HLMTuple* == *HLEMENTITY*  $\times$  *HLEMENTITY*

*SMTuple* == *SMENTITY*  $\times$  *SMENTITY*

<i>ReflexionModel</i>	
$rmEntities : \mathbb{P} HLMENTITY$	
$convergences : HLMRelation$	
$divergences : HLMRelation$	
$absences : HLMRelation$	
$mappedSourceModel : HLMTuple \leftrightarrow SMTuple$	
$smBag : \text{bag } SMTuple$	
$convergences \cap divergences = \emptyset$	(1)
$divergences \cap absences = \emptyset$	(2)
$convergences \cap absences = \emptyset$	(3)
$\text{dom } mappedSourceModel = convergences \cup divergences$	(4)
$\text{ran } mappedSourceModel \subseteq \text{dom } smBag$	(5)
$\text{dom } convergences \cup \text{ran } convergences \cup \text{dom } divergences \cup \text{ran } divergences$ $\cup \text{dom } absences \cup \text{ran } absences \subseteq rmEntities$	(6)

The variables (above the dividing line) in the schema represent observations that can be made about the state of a reflexion model system. The *rmEntities* variable describes the set of nodes in a reflexion model; the relation described by the *convergences* variable defines where the high-level model agrees with the source model; the relation described by the *divergences* variable describes where the source model differs from the high-level model; and the relation described by the *absences* variable defines where the high-level model differs from the source model. The fourth variable, *mappedSourceModel*, is a relation that describes which source model values contribute to a convergent or divergent arc in the reflexion model. The information in *mappedSourceModel* supports operations that aid an engineer in interpreting a reflexion model. For example, the information in *mappedSourceModel* is necessary to support a query of the form shown in Figure 2.4. The final variable, *smBag*, is a bag that includes all source model values and describes the number of occurrences of each value in the source model.<sup>4</sup>

In addition to declaring the state components of a reflexion model, the static schema includes the definition of invariants (below the dividing line) that must be satisfied by a reflexion model system.<sup>5</sup> The first three invariants state that the values of the

<sup>4</sup>This variable helps support the treatment of the range of *mappedSourceModel* as a multigraph. This treatment is helpful when a source model used in a reflexion model computation is a multigraph; source models produced by monitoring the execution of a software system are often multigraphs.

<sup>5</sup>The invariants have been numbered in parentheses at the far right of the schema for ease of reference.

*convergences*, *divergences* and *absences* relations are disjoint. The fourth invariant states that the investigation of the source model values contributing to a reflexion model arc is supported only for values in the *convergences* and *divergences* relations; *absences* have no contributing source model values. The fifth invariant states that the source model values contributing to reflexion model arcs must be either be a subset of, or the same as, the values in the source model stored in *smBag*. The final invariant states that the *convergences*, *divergences*, and *absences* variables are relations amongst the entities of the reflexion model. This final invariant is weak enough to permit entities of the reflexion model that are not part of the *convergences*, *divergences*, or *absences* relations, enabling an engineer to compute a reflexion model in which one or more entities are disjoint from interactions occurring in the system. For instance, an engineer may wish to compute a reflexion model for a system by describing only the entities, and not the interactions between the entities, of a system. The resultant reflexion model may include disjoint entities that do not serve as an end-point for any computed divergence.<sup>6</sup> An example of this kind of use of a reflexion model is described in Section 9.3.

### 2.2.2 Computing a Reflexion Model

The dynamic schema, *ComputeReflexionModel*, presented below, describes the computation of a reflexion model from three inputs: a high-level model, a source model, and a mapping from the source to the high-level model.

The high-level model is described by two variables: the *hlmEntities?* variable describes the set of entities in the high-level model, and the *hlm?* variable describes a relation over high-level model entities. Splitting the description of the high-level model across these variables permits an entity to be part of a high-level model without requiring it to interact with any other high-level model entity. The source model (*sm?*) is described as a bag of tuples relating source model entities, and the mapping (*map?*) is an ordered list of map entries. Each map entry—defined by the type *MapEntry*—names zero or more source

---

<sup>6</sup>The computed reflexion model will include only divergences since no interactions were posited.

model entities and associates with these entities one or more high-level model entities. *SMENTITYDESC* is introduced to represent the type of a description naming zero or more source model entities (e.g., in the reflexion model tools I developed the description is a regular expression over logical and physical software structure).

Computing a reflexion model also requires a function (*mapFunc*) that matches entities from the source model to the specified map, producing a set of associated high-level model entities.

[*SMENTITYDESC*]

*MapEntry* == *SMENTITYDESC* × ( $\mathbb{P}$  *HLEMENTITY*)

| *mapFunc* : (seq *MapEntry* × *SMENTITY*) → ( $\mathbb{P}$  *HLEMENTITY*)

<i>ComputeReflexionModel</i>	
$\Delta$ <i>ReflexionModel</i>	
<i>hlmEntities?</i> : $\mathbb{P}$ <i>HLEMENTITY</i>	
<i>hlm?</i> : <i>HLMRelation</i>	
<i>sm?</i> : bag <i>SMTuple</i>	
<i>map?</i> : seq <i>MapEntry</i>	
<hr/>	
(dom <i>hlm?</i> ) ∪ (ran <i>hlm?</i> ) ⊆ <i>hlmEntities?</i>	(1)
∪(ran(ran <i>map?</i> )) ⊆ <i>hlmEntities?</i>	(2)
<i>rmEntities'</i> = <i>hlmEntities?</i>	(3)
<i>smBag'</i> = <i>sm?</i>	(4)
<i>mappedSourceModel'</i> = ∪ { <i>t</i> : <i>SMTuple</i>   <i>t</i> in <i>smBag'</i> • ( <i>mapFunc</i> ( <i>map?</i> , <i>first t</i> ) × <i>mapFunc</i> ( <i>map?</i> , <i>second t</i> )) × { <i>t</i> } }	(5)
<i>convergences'</i> = <i>hlm?</i> ∩ (dom <i>mappedSourceModel'</i> )	(6)
<i>divergences'</i> = (dom <i>mappedSourceModel'</i> ) \ <i>hlm?</i>	(7)
<i>absences'</i> = <i>hlm?</i> \ (dom <i>mappedSourceModel'</i> )	(8)

The first predicate after the dividing line is a pre-condition stating that the set of high-level model entities must be a superset of the entities appearing in the high-level model. In addition, the high-level model entities that are part of the map are constrained by the second pre-condition (after the dividing line) to be a subset of the *hlmEntities?* set. The set of entities defined by *hlmEntities?* forms the entities of the computed reflexion model as described by the third predicate.

The value of *mappedSourceModel* is computed by pushing the elements of each source model tuple through the map, resulting in two sets of high-level model entities.<sup>7</sup> The cross-product of these sets is taken and each element in the resultant set is associated (through another cross-product) with the original source model tuple. Once the *mappedSourceModel* is computed, the values of the *convergences*, the *divergences*, and the *absences* relations are easily determined through set intersection and set difference operations.

This formal specification clarifies that the map defined by an engineer between the source model entities and the high-level model entities consists of a sequence of map entries rather than a set of map entries. Characterizing the map as a sequence permits the definition of a mapping function that uses the ordering information in the sequence to produce the set of high-level model entities associated with the first match of a given source model entity to an entry in the map. Treating the map as a sequence is the most common configuration of a reflexion model system used by engineers because it permits the inclusion of “catch-all” entries at the end of the map to associate source model entities not associated by any other entry. The use of this feature by a Microsoft engineer applying the technique to aid an experimental reengineering of Excel is described in Section 9.1.

The definition of other mapping functions is possible. For instance, the reflexion model tools support a mapping function that returns the set of high-level model entities resulting from the union of all matches found in the map. This definition effectively treats the map as a set of entries rather than a sequence. Writing the formal specification helped to clarify the parameterization of the mapping function, as the original version of the tools had only supported the treatment of the map entries as a sequence.

---

<sup>7</sup>Note that for the purposes of computing a reflexion model, the source model is treated as a relation rather than a bag. The number of times each tuple appears in the source model is used in the computation of the numeric value associated with a reflexion model arc as defined in Section 2.2.3.

### 2.2.3 Displaying a Reflexion Model

The dynamic schema, *ComputeReflexionModelArcValue*, below, describes an operation to determine the numeric label of an arc in a computed reflexion model. This operation takes a reflexion model arc (*rmArc?*) and produces the numeric output value in the *arcVal!* variable. A user interface for a reflexion model system might use this operation repeatedly, once for each arc in the union of the *convergences*, *divergences*, and *absences* relations, to display a reflexion model with associated arc values (as in Figure 2.3b).

The operation also requires a function, *addArcCounts*, which given a selected set of source model tuples and the source model bag, sums the number of times each tuple appears in the bag and produces the result. If the specified set of source model tuples is empty, *addArcCounts* returns zero.

$$\mid \text{addArcCounts} : (SMRelation \times (\text{bag } SMTuple)) \rightarrow \mathbb{N}$$

$\frac{\begin{array}{l} \text{ComputeReflexionModelArcValue} \\ \exists ReflexionModel \\ rmArc? : HLMTuple \\ arcVal! : \mathbb{N} \end{array}}{\begin{array}{l} rmArc? \in (convergences \cup divergences \cup absences) \\ arcVal! = addArcCounts((mappedSourceModel(\{rmArc?\})), smBag) \end{array}}$
--

One pre-condition is specified for the operation; the arc for which the value is to be computed must be an arc in the reflexion model. The value returned by the operation is the result of applying the *addArcCounts* function to the set of source model values that contribute to the specified reflexion model arc.

### 2.2.4 A Family of Reflexion Models

This formal specification defines a family of reflexion model systems. Different kinds of systems result, depending on the choices made for representing the source model entity descriptions in a map (*SMENTITYDESC*) and for defining the mapping function

(*mapFunc*). The tools described in the next chapter support a source model entity description language based on physical and logical structure information, and permit an engineer to choose between two different mapping functions when computing a reflexion model. Writing this specification helped to clarify these points at which the technique and supporting tools could be parameterized to extend their usefulness.

### 2.3 Summary

In this chapter, I have presented an overview of the software reflexion model technique. The technique permits an engineer to summarize structural information extracted from the source within the framework of a high-level model. The technique is:

- *lightweight*, in that an engineer can quickly and easily compute a reflexion model,
- *approximate*, in that the map used in the computation of a reflexion model may coarsely associate source model entities with high-level model entities,
- *partial*, in that the map may not include in the computation all entities in a given source or high-level model, and
- *iterative*, in that the engineer may selectively refine the inputs to compute a series of reflexion models until the desired information is obtained.

Similar to reverse engineering techniques, the reflexion model technique provides a view of a system in terms of abstract components. In comparison to reverse engineering techniques, though, the engineer selects the abstract components rather than those components being created through the process of using the technique. By selecting the abstract components, the engineer is assured the components are those useful for reasoning about the task being performed. Chapter 10 provides a further comparison between the reflexion model technique and existing approaches to enhancing structural understanding.

To make precise the reflexion model technique, a *Z* specification for a reflexion model system was also presented. This specification clarifies the inputs, outputs, and operations associated with the technique, and identifies ways in which the technique may be generalized. This specification may be useful to implementors of reflexion model systems and to others developing similar techniques. In the next chapter, I use this specification as a basis for reasoning about the complexity of computing a reflexion model.

## Chapter 3

# Computing Software Reflexion Models

Ensuring that it is both easy for an engineer to prepare the inputs to a reflexion model computation, and that a reflexion model may be computed quickly from the specified inputs, requires the careful design of tools to support the technique. In this chapter, I describe the developed tools and discuss how they were engineered to support the iterative use of the reflexion model technique on a variety of large software systems.

### 3.1 Tools

The architecture of the system developed to support the software reflexion model technique consists of a number of programs implemented in C++ [Stroustrup 1986] that communicate through the file system. The programs, referred to as tools, may be used though either a command-line interface or through a graphical user interface implemented in TCL/TK [Ousterhout 1994].<sup>1</sup> This architecture was chosen to ease the development of the tools for multiple operating systems including various Unix workstation operating systems, MS-DOS, and Windows NT.<sup>2</sup> This flexibility in the execution platform permit-

---

<sup>1</sup>In this chapter, I describe the architecture and report on the performance of version 1.7 (Build 1) of the software reflexion model tools. The version 1.7 distribution includes fifteen tools that comprise approximately 12,000 lines of C++ [Stroustrup 1986] and 3,000 lines of TCL/TK [Ousterhout 1994] code.

<sup>2</sup>MS-DOS is a registered trademark and Windows NT is a trademark of Microsoft Corporation.

ted the use of the tools in several environments for the purpose of validation as described in Chapter 9.

Three tools are used in the computation of a reflexion model:

- the `m12map` tool compiles a given map into an internal map format,
- the `computeArcs` tool pushes the tuples of the source model through the compiled map producing a set of mapped interactions between entities of the reflexion model, and
- the `produceRM` tool compares the mapped interactions with the high-level model specified by the engineer to produce the reflexion model.

Two members of the family of reflexion model systems described by the formal specification presented in the previous chapter are supported by these tools. Each member of the family uses a *SMENTITYDESC* type based on physical and logical software structure information that is specified using regular expressions. Support for this type is built into the `m12map` tool. The two members of the family are differentiated by the mapping function, *mapFunc*, selected by an engineer when invoking the `computeArcs` tool: the first function treats the map as a sequence, returning the set of high-level model entities resulting from the first match of a source model entity to the compiled map, whereas the second function ignores the sequential nature of the map, returning the union of the high-level model entities resulting from any match of a source model entity to an entry in the compiled map. Together, the `m12map` and `computeArcs` tools compute the value of *mappedSourceModel* in the *ComputeReflexionModel* schema. The `produceRM` tool computes the values of *convergences*, *divergences*, and *absences*. The *ComputeReflexionModel* schema is implemented as three tools instead of one to provide flexibility in the interface presented to the engineer.

The flow of data between these tools is shown in Figure 3.1. The `m12map` tool uses a specification written in the source entity description language to compile a mapping

file. The `computeArcs` tool uses the compiled mapping file to produce an image arcs file, a mapped arcs file, and an unmapped arcs file. The image arcs file describes the arcs between high-level model entities that result from applying the map to the source model. The file also contains numeric values describing how many source model tuples contribute to each arc. The numeric values are included in the image arcs file because the `computeArcs` tool also implements the *ComputeReflexionModelArcValue* schema. The mapped arcs file describes which source model arcs contribute to each arc in the image arc file. The unmapped arcs file describes the source model arcs not mapped by the map file. The `produceRM` tool compares the image arcs file and the specified high-level model to produce a description of the reflexion model. The computed reflexion model may be displayed using the AT&T graphviz graph display program [Gansner et al. 1988; Chen et al. 1995].

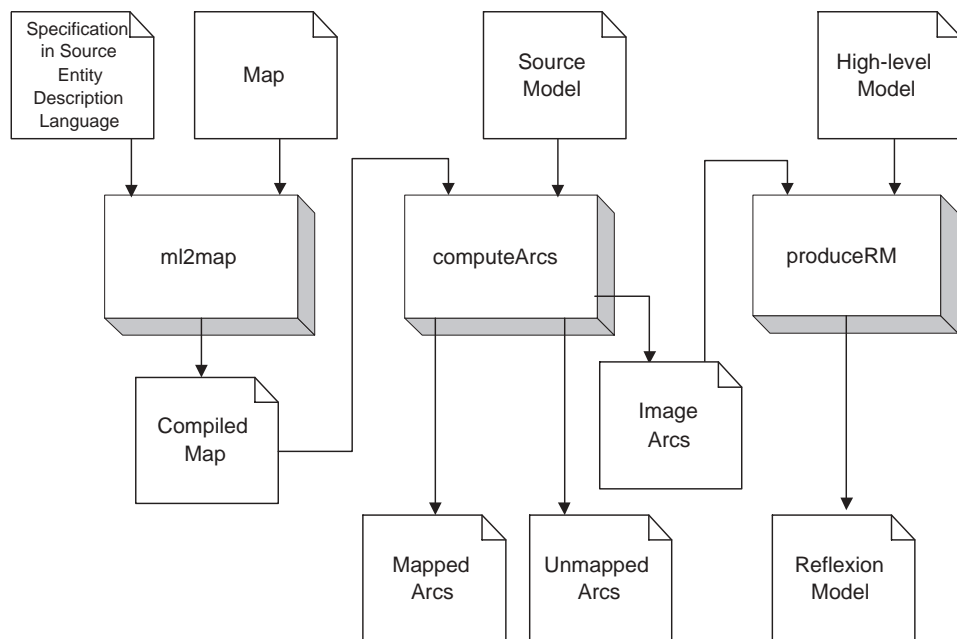


Figure 3.1: Tools for Computing Reflexion Models. The shadowed boxes represent tools in the reflexion model system. The boxes with turned down corners represent input and output files to the tools. The arrows show the flow of information.

### 3.1.1 Source Entity Description Language

The computation of a reflexion model requires the engineer to specify a map between entities of the source and the high-level models. As defined by the formal model, this map consists of a sequence of map entries. Each map entry specifies zero or more source model entities and one or more high-level model entities. Since the engineer selects the high-level model to use in the computation based on the view of the system necessary for a particular task, the number of entities in the high-level model is generally small; as described in Chapter 9 this has been the case in practice with the largest high-level model consisting of 17 entities. As a result, it is reasonable, as part of the mapping language, to require the engineer to list the high-level model entities involved in a map entry explicitly. However, the large number of source model entities requires a language that permits an engineer to easily describe multiple source model entities.

A natural way to provide the expressiveness necessary for naming source model entities is to leverage the organizational information inherent in the implementation of the software system. Two kinds of organizational information are typically available: physical and logical. The physical organization refers to the separation of the software into components for storage on a device. For instance, C [Kernighan and Ritchie 1978] source code may be stored, using a Unix file system, into files that are placed into a hierarchical directory structure. The logical organization refers to the separation of the software into components using language and environment mechanisms. For instance, C source code is organized into functions, whereas Ada [GPO 1983] source code is organized into functions and packages. The organizational structure varies for different programming languages, environments, and file systems. A software system implemented within a Smalltalk environment [Goldberg and Robson 1983], for example, may have little physical organizational information, but will have lots of logical information such as methods, classes, and protocols. In contrast, a software system implemented in assembler may have little logical organization, but may have some physical file organization.

To support these differences between languages, environments, and systems, the reflexion model tools are parameterized by a specification describing the organization of the software system that is the subject of the reflexion model computation. This specification is written in a source entity description language. This language defines a tree-based naming mechanism to denote the physical and logical organizational information recorded about a source model entity.

Each node in a naming tree defines a keyword that describes one aspect of the organizational information. The defined keywords need only be unique on a path of the tree. For example, Figure 3.2 shows a possible naming tree for function and variable entities within C source code stored in a Unix file system. This tree permits the naming of function entities by specifying values for any combination of physical directory and file information, or logical function name information. For example, the phrase,

```
directory=vm file=pager.c function=writeToFile
```

names a function `writeToFile`, stored within the `pager.c` file in the `vm` directory. A C variable, `pagePolicy`, declared within the scope of the `pager.c` file that exists within the `vm` directory may be specified using the following phrase.

```
directory=vm file=pager.c variable=pagePolicy
```

The naming tree specified in Figure 3.2 also declares a keyword named `variable` as a child of the function node to permit the naming of variables declared within the scope of a function. If a `pagePolicy` variable was also declared within the `writeToFile` function, for instance, it could be named using the following phrase.

```
directory=vm file=pager.c function=writeToFile variable=pagePolicy
```

When a keyword is used on more than one path in the tree, as is the case with the `variable` keyword in the naming tree of Figure 3.2, the naming path to use in interpreting a phrase is chosen based on a breadth-first search of the provided keywords. For instance, the phrase

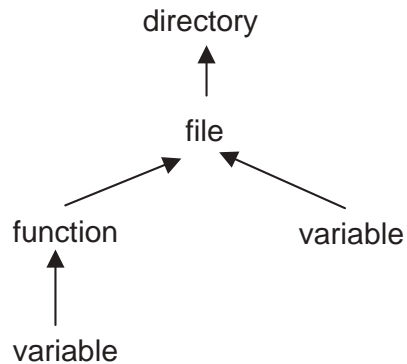


Figure 3.2: Source Entity Naming Tree.

```
variable=pagePolicy
```

denotes the following path.

```
directory -> file -> variable
```

On the other hand, the phrase

```
function=writeToFile variable=pagePolicy
```

denotes the path shown below.

```
directory -> file -> function -> variable
```

The description of the source entity description language provided as input to the reflexion model tools consists of the fully qualified names of each node in the tree. The naming tree shown in Figure 3.2, for instance, would be described to the reflexion model tools as follows.

```
directory
directory.file
directory.file.function
directory.file.function.variable
directory.file.variable
```

The source entity description languages also affects the source model as the source model must be encoded according to the source entity description language specification. The encoding consists of converting the appropriate information about each source model entity's physical and logical organization into a string. The string consists of a header for each node in the naming tree followed by the value of the node. The value of any node may be null. For instance, the encoded string for a function entity given the naming tree of Figure 3.2 will have empty variable nodes. Each node of the structure is identified by assigning it a number in a depth-first ordering of the naming tree. For example, the encoded string of the `writeToFile` function entity described above is given by:

```
@1@vm@2@pager.c@3@writeToFile@4@@5@
```

if “@” is used as the separator character. The encoded string of the `pagePolicy` variable declared within the scope of the `writeToFile` function is given by:

```
@1@vm@2@pager.c@3@writeToFile@4@pagePolicy@5@.
```

The appropriate encoding can be produced by the source model extraction tool described in Chapter 6. Simple scripts in text processing languages such as `awk` [Aho et al. 1979] or `perl` [Wall 1990] are generally enough to translate source models extracted with other tools. For example, a simple `awk` script was used to translate the calls between functions information extracted from the NetBSD source with the `Field` [Reiss 1990; Reiss 1995] programming environment into a suitable format for the reflexion model tools.

To further ease the specification of the map, the `m12map` tool permits the use of regular expressions for keyword values. Furthermore, keywords may be left unspecified in a map, causing them to be implicitly treated as the regular expression, `.*`, that matches all sequences of characters.<sup>3</sup> For example, the phrase

```
directory=vm file=pager.c
```

---

<sup>3</sup>Regular expressions are specified in `egrep-style`.

names all functions and variable entities in the `pager.c` file found in the `vm` directory. Based on feedback from the engineers who have used the reflexion model tools, the specification of the map has been further simplified by treating non-anchored regular expression values as being implicitly surrounded by wildcards. Thus, the phrase above names all functions and variables appearing in files with `pager.c` and `vm` in their file and directory names. If an engineer desires to override this implicit wildcarding, the regular expression anchor characters may be placed around the keyword value.

### 3.2 Complexity

Understanding how to make the tools run fast enough to support the iterative computation of reflexion models requires an understanding of both the theoretical complexity of the computation, and also of the way in which the technique is used in practice.

Insights into the theoretical complexity of the reflexion model computation can be gained from a consideration of the formal characterization presented in Section 2.2. From the dynamic Z schema, *ComputeReflexionModel*, it is evident that the time complexity of computing a reflexion model is dependent on the cost of forming the *mappedSourceModel* relation, and on the cost of comparing that computed relation to the high-level model. An upper bound on this time complexity is given by:

$$O(\#(dom\ sm) \times ((\#map \times t_{comparison}) + (\#hlmEntities)^2)) + O((\#hlm)^2)$$

where  $\#(dom\ sm)$  is the number of unique source model values in the source model bag,  $\#map$  is number of entries in the map,  $t_{comparison}$  is the cost of comparing a source model entity to a source model entity description in a map entry,  $\#hlmEntities$  is the cardinality of the set of high-level model entities, and  $\#hlm$  is the size of the high-level model relation.<sup>4</sup> Since the number of entities in the high-level model is generally small, particularly in comparison to the size of the source model and the map, the terms in-

---

<sup>4</sup>The question marks are dropped from the variable names for simplicity of presentation.

volving *hlmEntities* and *hlm* can, in practice, be treated as a constant, yielding:

$$O(\#(dom\ sm) \times \#map \times t_{comparison}).$$

To show what this complexity means in practice, Table 3.1 provides data on the execution time required to naively compute reflexion models on a DECStation 3000/300X running OSF/1 V3.2 for three different tasks.<sup>5</sup> In the naive computation, the map entries are held in memory, and as each tuple is read in from the source model file, each component of the tuple is matched against the map, and the cross-products of the resultant sets are calculated. The table includes data describing the number of source model tuples, the number of entries in the map, the number of tuples in the high-level model, and the total number of comparisons executed. Data on three tasks are reported: the estimation task on the NetBSD system (described in Chapter 2); a design conformance task on a meaning-preserving program restructuring tool (described in Section 9.2); and an experimental reengineering task on Microsoft's Excel spreadsheet product (described in Section 9.1).

Table 3.1: Naive Reflexion Model Computation. The performance of the tools was measured on a DECStation 3000/300X running OSF/1 V3.2.

System	Source Model (# Tuples)	Map Entries (#)	High-Level Model (# Tuples)	Comparisons (#)	Exec. Time (min)
NetBSD	15,656	6	9	108,898	:07
restruct. tool	5,855	215	11	1,727,305	:48
Excel	119,637	971	96	203,449,441	79:03

The naive computation approach is satisfactory for small source models and maps, as in the case of NetBSD where the computation required less than ten seconds, but is too

---

<sup>5</sup>The reported time is the sum (rounded down) of the user and system seconds as reported by the Unix time command to run the `m12map`, `computeArcs`, and `produceRM` tools.

costly to support the iterative and exploratory use of the reflexion model technique as the size of the source model and the map grow. For instance, the naive computation of a reflexion model for the Excel case required almost 80 minutes.

### 3.3 Speeding up the Computation

The `computeArcs` tool accounts for approximately 98% of the time required to compute a reflexion model for the program restructuring tool reported in Table 3.1. An execution profile of this tool indicates that over 98% of its execution time is spent performing regular expression matches.<sup>6</sup> This information suggests at least two ways of speeding up the computation of a reflexion model. First, the speed may be improved by reducing the number of regular expression matches performed. Second, the speed may be improved by decreasing the time required to compare an encoded source model entity with a regular expression.

I added optimizations to the reflexion model tools in each of these categories. These optimizations are sufficient to support the iterative use of the technique for tasks that involve the use of source models with a few hundreds of thousands of values and maps with a few thousand entries, as was the case in the use of the technique on Excel. For even larger source models and maps, I developed a version of the reflexion model tools that supports the incremental computation of a reflexion model. These incremental computation tools have been tested but have not been validated in practice. In the following sections, I describe the optimizations and incremental computation algorithm in more detail.

#### 3.3.1 Trading Space for Time

The number of regular expression matches attempted during the computation of a reflexion model can be reduced by trading space for time. To investigate the effect on performance when space is traded for time, I implemented two mutually exclusive ap-

---

<sup>6</sup>Profiling was performed using `pixie` on the DECStation 3000/300X.

proaches: one based on hash tables, and another based on a cache. In general, each of these space trade-offs will improve the computation of a reflexion model, but since both are based on redundancy in the source model (i.e., that the same source model entity appears in multiple source model tuples), the actual benefits of these optimizations are dependent on the source model used in the computation.

*The Hash Table Approach.* The first approach uses a hash table to store the set of high-level model entities mapped to a particular source model entity the first time the source model entity is seen and matched against the map. When the source model entity is encountered again later, it may be used to index into the hash table, saving the need to reexecute the costly regular expression comparisons of the source model entity against the map entries. This optimization is implemented in the `computeArcs` tool.

The performance of this optimization is summarized in Table 3.2. The data in the table includes the number of comparisons between source model entities and the map performed during the computation, the number of hash table hits, the size of the `computeArcs` process during the computation,<sup>7</sup> the execution time of the computation,<sup>8</sup> and the speedup relative to the naive execution times.<sup>9</sup> The size of the source model, the map, and the high-level model are unchanged from those reported in Table 3.1.

The speedup column of Table 3.2 shows that all the computations were performed as fast or faster using the hash table optimization as compared to the naive computation approach, with the Excel computation performing over ten times faster. Even for the largest computation configuration (Excel), the memory usage for the hash table is modest at less than 5 Mb.

---

<sup>7</sup>The size reported is the virtual address size of the `computeArcs` tool measured by executing the Unix `ps` command at the end of the execution of the tool.

<sup>8</sup>The time reported is the sum of the number of user and system seconds (rounded up to support a conservative comparison with the naive computation) as reported by the Unix `time` command for executing the `computeArcs` tool.

<sup>9</sup>The speedup is calculated as the execution time of the naive version of `computeArcs` tool for each system (7 seconds for NetBSD, 47 seconds for the program restructuring tool, and 4742 seconds for Excel) divided by the execution time of the version of the tool using the hash table optimization.

Table 3.2: Reflexion Model Computation with a Hash Table. Performance was measured on a DECStation 3000/300X running OSF/1 V3.2.

System	Comparisons (#)	Hash Hits (#)	Memory Usage (Mb)	Exec. Time (min)	Speedup From Naive
NetBSD	13,628	17,704	2.65	:07	1
restruct. tool	398,607	7,954	2.80	:13	3.6
Excel	15,622,867	221,022	4.66	7:19	10.8

The performance of this optimization is dependent on the redundancy of source model entities within the source model. To help understand the degree of redundancy in the source models for the measured computations, the frequency of occurrence of source model entities in each of the Excel, NetBSD, and program restructuring tool source models was calculated. Figure 3.3 shows a histogram of these data. The histogram shows the percentage of entities in the source model in each of four frequency occurrence ranges. For instance, the first range shows that 14% of the entities in the Excel source model appear only once, whereas 55% of the entities in the source model appear between two and ten times. Since most of the source model entities appear between two and one hundred times and since many entities appear even more frequently than ten times, there is enough redundancy to benefit from using a hash table.

*The Cache Approach.* In some cases, for instance when the computation involves a very large source model, or when the number of times each source model entity appears is small, an optimization that uses less memory than the hash table approach may be desirable. To investigate a lower memory use space-time trade-off, I incorporated a rolling cache into the `computeArcs` tool that attempts to benefit from locality of source model entities within the source model. The rolling cache optimization tested maintains the last two source model entity to high-level model entity mappings seen during the computation. This two entity cache approach assumes that the information in the source

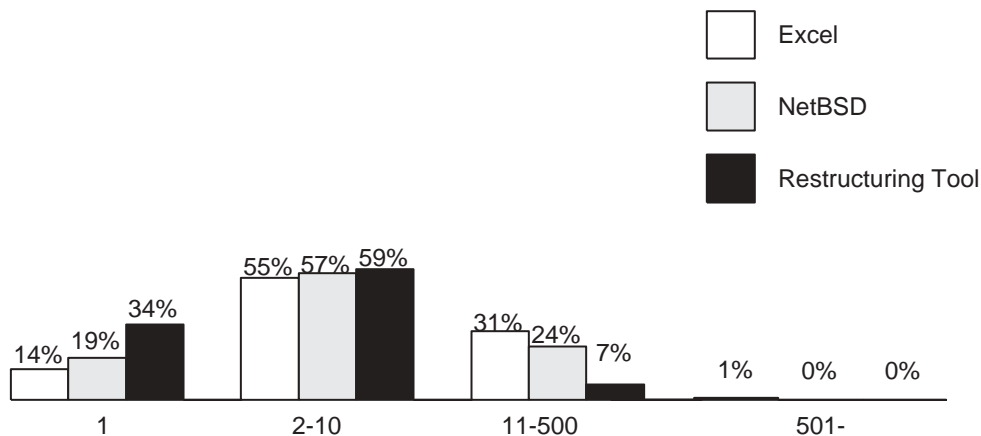


Figure 3.3: Source Entity Frequency Distributions. The distributions are shown for the Excel, NetBSD, and program restructuring tool source models. The  $x$  axis indicates frequency ranges of the occurrence of entities within a source model. The bars indicate the percentage of source model entities in that frequency range.

model is localized around at least one of the entities.

The performance of this optimization is summarized in Table 3.3. The data in the table are similar to the data reported for the hash table optimization except the number of cache hits is reported instead of the number of hash table hits. As expected, the memory usage is smaller, and the computation times, although in general showing modest improvement over the performance of the naive computation, are slower than the execution times that result from the use of the hash table optimization.

Table 3.3: Reflexion Model Computation with a Cache. The performance was measured on a DECStation 3000/300X running OSF/1 V3.2.

System	Comparisons (#)	Cache Hits (#)	Memory Usage (Mb)	Exec. Time (min)	Speedup From Naive
NetBSD	93,503	13,382	1.92	:08	0.8
restruct. tool	1,260,730	4,139	2.16	:35	1.3
Excel	119,270,378	98,945	2.39	50:11	1.5

### 3.3.2 Optimizing the Regular Expression Comparison

The optimizations described above trade space for time to improve performance. The hash table optimization, in particular, minimizes the number of comparisons required for a computation. Even with the hash table optimization, a profile of the `computeArcs` tool indicated that almost 50% of the cycles during execution were spent performing regular expression comparisons between source model entities and encoded source model entity descriptions from the map. To improve the performance of an individual regular expression comparison, I added five optimizations to the `computeArcs` tool. Four of these optimizations are, in essence, in-lined optimizations to try to eliminate the need to call a general regular expression package. The fifth optimization uses knowledge of the encoding of a source model entity description to a regular expression, implemented as part of the `m12map` tool, to reduce the cost of a comparison.

The `computeArcs` tool uses a regular expression library package implemented by Henry Spencer. This package is a reimplementation of the Unix V8 `regexp(3)` package. To determine the effect of using this package on the performance of the reflexion model tools, I compared its execution speeds with four other regular expression packages:

- the regular expression package distributed with OSF/1 V3.2, and
- the GNU regular expression package (Version 0.12) accessed in three different ways:
  - through the POSIX compliant interface,
  - through the GNU interface, and
  - through the GNU interface with the `fastmap` option intended to speed searching through a long string.

To compare these packages, I wrote a C program that read in a set of test cases, each of which consisted of compiling a regular expression, and then attempted, 10,000 times, to match a stated string to the compiled regular expression. Each test case description

included an indication of whether the regular expression match should be successful. The 12 test cases used were selected from the Excel and NetBSD case study data. Half of these test cases resulted in successful matches; the other half resulted in unsuccessful matches. Table 3.4 summarizes the execution speeds—reported as the sum of the user and system times reported by the Unix `time` command over five executions of the test—on a DECStation 3000/300X for the various regular expression packages. This data shows that Spencer’s package is a fast implementation.

Table 3.4: A Comparison of the Speed of Regular Expression Packages.

Package	Execution Speed (s)
Spencer’s	1.49
OSF/1 V3.2	11.22
GNU POSIX	6.98
GNU	7.57
GNU FAST	6.53

In most of the uses of the reflexion model tools reported on in this dissertation, the regular expressions used in the maps were simple and of limited types. For instance, fewer than 18% of the entries in the map used for Excel use wildcards as part of the value for a keyword. None of the regular expressions in entries in the maps used for NetBSD, the restructuring tool, or Excel make use of bracketed expressions. To improve the performance of the reflexion model tools, I added checks for these common types of regular expressions to attempt to eliminate unnecessary calls to the general regular expression package. The implemented checks, described in the order of their application, are:

- an end-of-string-literal check that performs a simple string comparison if the source entity description forms a string consisting of wildcards followed by a literal string,
- an end-of-string-filter check that can discard a potential match if there are no

regular expression characters at the tail of the encoded source model entity description, and the two encoded strings (one the description and the other a source model entity from the source model) do not match at the tails of the strings,

- an escapes-only check that checks if the only regular expression characters used in the source entity description are escapes, and if so, removes the escapes and uses a C language library provided string comparison routine (i.e., `strstr`) to compare the two encoded strings,
- a limited-regexp-only check that checks if the only regular expression characters used in the source entity description are escapes and wildcards (i.e., `.*`), and if so, removes the escapes and forms the ordered sequence of substrings between wildcards, and then does a special string comparison looking for the ordered sequence of substrings in the encoded source model entity, and
- a longest-substring check that determines the longest substring without regular expression characters in the encoded source entity description and then uses the C language substring search operation (i.e., `strstr`) to eliminate any unnecessary call to the regular expression package.

In essence, the end-of-string-literal, end-of-string-filter, longest-substring, and escapes-only checks are optimizations that are inlined from the regular expression package. The limited-regexp-only check takes advantage of the particular structure of the regular expressions compiled by the `m12map` tool. As described in Section 3.1.1, each component of the source entity description is anchored by a unique pattern within the encoded string. The substrings extracted by the limited-regexp-only check include this anchoring information to ensure that substrings match to the appropriate source entity description components.

The execution times of the `computeArcs` tool with only the regular expression optimizations are provided in Table 3.5. Also included in the table are the number of

times each optimization was applied during the computation, and the speedup relative to the naive performance of the `computeArcs` tool. As described in Section 3.1.1, when the `m12map` tool encodes a source entity description, unless the regular expression for a specific component is head or tail anchored, wildcards are inserted. As a result, the limited-regexp-only check is frequently applied.

Table 3.5: Reflexion Model Computation with Regular Expression Optimizations. The performance was measured on a DECStation 3000/300X running OSF/1 V3.2.

System	End-of-String-Literal (#)	End-of-String-Filter (#)	Escapes-Only (#)	Limited-RegExp-Only (#)	Longest-Substring (#)	Exec. Time (min)	Speed-up From Naive
NetBSD	0	0	0	74,698	34,053	0:07	1.0
restruct. tool	0	158,910	0	195,601	1,306,591	0:14	3.3
Excel	169,735,679	0	26,068,029	7,645,733	0	12:51	6.1

### 3.3.3 Combining Optimizations

The best time performance of the `computeArc` tool, and hence the reflexion model tools, may generally be achieved by applying both the hash table and the regular expression optimizations. Table 3.6 shows the data summarizing the performance of the tool with both optimizations including the execution time of the tool, the memory usage of the tool, and the execution speedup relative to the naive performance. The optimizations better the performance of the tool in each of these three cases. For the Excel case, the optimizations increase the speed of execution a factor of 49 times. Moreover, the speed of execution is sufficient to support the iterative use of the reflexion model tools even for large source models. For Excel, a reflexion model can be computed in less than two minutes on a DECStation 3000/300X and under five minutes on a 486 running at 100Mhz.

Table 3.6: Reflexion Model Computation with Multiple Optimizations. This table reports the performance of the tools when both the hash table and regular expression optimizations are used. The performance was measured on a DECStation 3000/300X running OSF/1 V3.2.

System	Exec. Time (min)	Memory (Mb)	Speedup From Naive
NetBSD	0:05	2.65	1.4
restruct. tool	0:05	2.80	9.4
Excel	1:36	4.66	49

### 3.3.4 Incremental Computation

Is a 100,000 tuple source model large? Is a 1000 entry map large? Certainly there are many systems in existence of more than a million lines of code for which it is reasonable to expect an extracted source model to be even larger than 100,000 tuples. Source models for smaller systems may also grow larger than 100,000 tuples if additional structural relations are included, for instance, perhaps augmenting call and data relations with file inclusion relations and data flow relations. It is also reasonable to expect that maps comprised of thousands of entries may not be uncommon when engineers require detailed views of portions of a large software system.<sup>10</sup> When source models and maps grow even larger in size than those used in the Excel case, the combined optimization approach described above may not provide enough speed to permit the engineer to interactively and iteratively apply the reflexion model technique. Furthermore, even when source models and maps are modest in size, the amount of time software engineers need to wait for the recomputation of a reflexion model should be proportional to the changes made to the inputs. To better support the engineer in these circumstances, I investigated incremental versions of the reflexion model tools.

---

<sup>10</sup>Although large maps require a significant effort by the engineer, it is possible that the cost in creating a large map may be amortized over many uses of the reflexion model. For instance, a map may be reused to compute new reflexion models to aid new personnel in gaining familiarity with a system.

Based on the use of reflexion models to date (Chapter 9), changes between computations of a reflexion model are more frequently made to the high-level model and to the map than to the source model. The high-level model may be changed in four ways: new entities may be added, existing entities may be deleted, new interactions between entities may be added, or existing interactions between entities may be deleted. The latter two classes of change affect only the execution of the `produceRM` tool that performs the comparison between the mapped arcs and the high-level model arcs. The complexity of this tool is low in comparison to the complexity of the `computeArcs` tool since it depends on the size of the high-level model rather than the sizes of the map or the source model. As a result, no special incremental support of the `produceRM` tool is warranted.

The former two classes of change—changes to the entities of the high-level model—may also involve changes to the map. For instance, entries may be added to the map to associate source model entities to a newly added high-level model entity, or existing map entries may be modified or deleted to remove a reference to a deleted high-level model entity. Changes may also be made to the map independent from changes to the high-level model such as adding an entry to the NetBSD map to include source model values concerned with the `Memory` module (Section 2.1.6). The incremental algorithm and tools developed focus on changes—as additions and deletions—to the map, and more specifically, to changes made to an ordered map since this is the most common member of the reflexion model family in use.<sup>11</sup>

The incremental version of the `computeArcs` tool, named `incComputeArcs`, analyzes the changes made to an ordered map and recomputes the image arcs file from the mapped and unmapped arcs files produced by a previous execution of either the `computeArcs` or the `incComputeArcs` tools (Figure 3.4). To support incremental computation, I modified the `computeArcs` tool to record in the mapped arcs file a pair of sequence numbers indicating the map entries that mapped each of the source model entities appearing in a source model tuple. In addition, I also changed the `computeArcs` tool to assign an

---

<sup>11</sup>A change to an existing map entry is treated as a deletion and subsequent addition.

identifier to each source model value; this identifier is recorded in the mapped arcs file.

The `incComputeArcs` tool is provided, as an input, a file that describes the differences between the previous map and the new map. This file may be produced using the Unix `diff` command. This information is used to create a data structure that describes, for each entry in the previous map, the status of the entry (i.e., whether it is deleted or renumbered in the sequence), and any entries in the new map that either possibly or definitely affect source model entities mapped by a given entry. The construction of the data structure involves four steps.

1. Determine the map entries that have been added or deleted.
2. Number the non-deleted entries in the map sequentially.
3. Determine, for each deleted map entry, the sequence of entries in the new map that may now map those source model entities previously mapped by the deleted entry. This sequence includes entries added before the deleted entry, and any entries appearing in the sequence below the deleted map entry for which it cannot be proven that the described set of source model entities is disjoint from the source model entities described by the deleted map entry.
4. Determine, for each added map entry, the sequence of entries in the new map that may be impacted by the addition. The impacted entries are classified into two sets. One set consists of the set of entries that are definitely remapped by the newly added entry because the added entry names a superset of the source model entities described by an existing entry. The second set consists of the set of entries that are possibly remapped because it cannot be proven that the newly added entry names a disjoint set of entities from the existing entry.

For example, consider the map for the NetBSD virtual memory system shown in Figure 3.5. The entries in the map marked with a plus sign at the beginning of the line have been added, and the entries marked with a minus sign have been deleted. A fragment

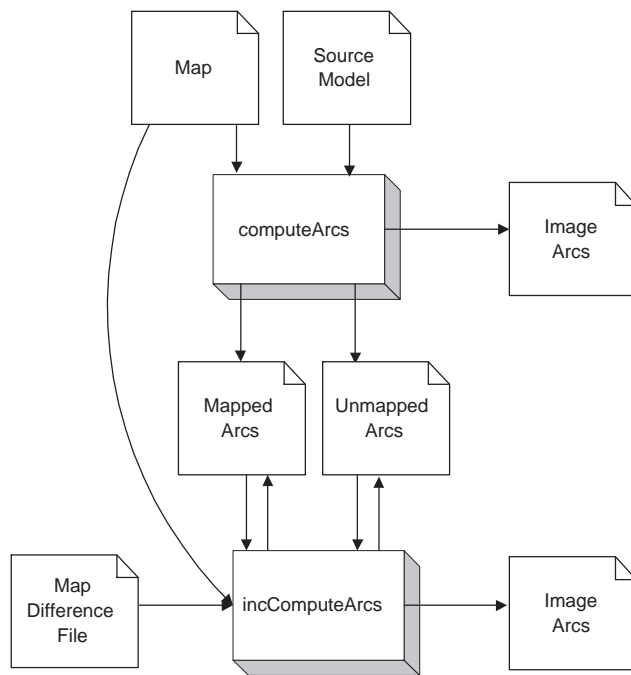


Figure 3.4: Incremental Tools for Computing a Reflexion Model. The shadowed boxes represent tools in the reflexion model system. The boxes with turned down corners represent input and output files to the tools. The arrows show the flow of information.

```

+ [ function=~vm_object_allocate$      mapTo=VMUtils ]
  [ file=.*pager.*                    mapTo=Pager ]
- [ file=~vm_map.*                    mapTo=VirtualAddressMaintenance ]
  [ file=~vm_fault.c$                 mapTo=KernelFaultHandler ]
- [ directory=.*fs                    mapTo=FileSystem ]
+ [ directory=.*fs$                   mapTo=NewEntity ]
  [ file=pmap.*                        mapTo=HardwareTranslation ]
  [ file=vm_object\.c$ function=~vm_object_allocate$ mapTo=Pager ]
+ [ directory=.*vm$                    mapTo=VMUtils ]

```

Figure 3.5: A Modified Map for the NetBSD Virtual Memory Subsystem.

of the data structure constructed when executing the `incComputeArcs` tool with this map information is shown in Figure 3.6. A node in the data structure, corresponding to a horizontal strip with a source entity description in Figure 3.6, describes an entry in the map. The nodes whose source entity descriptions are greyed represented deleted entries; the nodes whose source entity descriptions have a box around them indicate added entries.

The data structure to support incremental computation is indexed by the position of entries in the map before modification. For example, the second node in Figure 3.5 is indexed by the number one indicating that it was the first entry in the map prior to modification. Each existing and deleted node also has as an ordered list of pointers to nodes that may possibly or definitely impact the mapping of source model entities. This pointer information is shown on the left side of the data structure representation in Figure 3.6. To clarify how these pointer structures are derived, the information about the relationship between source model entity descriptions for the added and deleted map entries is annotated on the right side of the data structure representation. For instance, the figure shows that the original second entry in the map names a set of source model entities that is disjoint from the entities named by the original third entry, and the added first entry describes a superset of source model entities described by the last entry. This disjoint and superset information is used in the construction of the data structure, but is not explicitly maintained during execution.

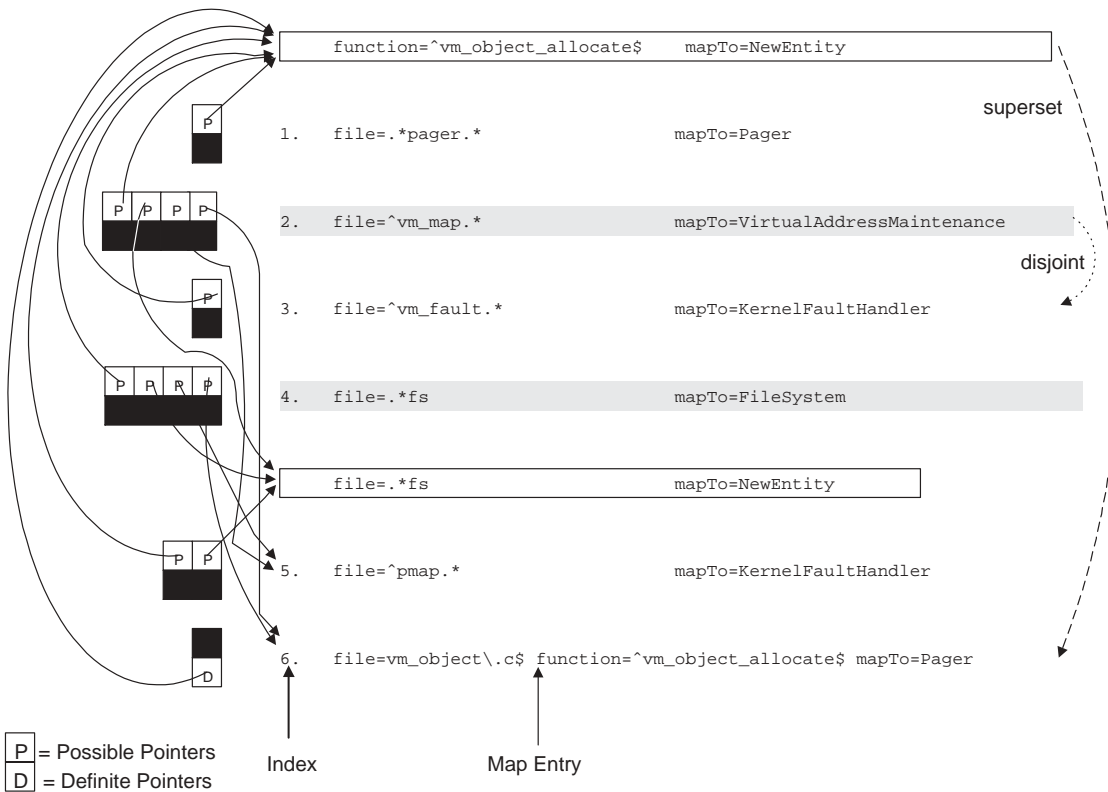


Figure 3.6: A Fragment of the Data Structure Used for Incremental Computation. Map entries are shown horizontally. The boxes to the left of a map entry are lists of pointers to other entries that possibly or definitely remap source model entities currently mapped by the given entry. The lists are ordered from left to right. Map entries that are boxed are entries that have been added to the map. Map entries that are greyed are entries that have been deleted from the map. The indices refer to the sequential order of entries in the previous map. The arrows at the right indicate regular expressions that are either disjoint from, or are supersets of, each other.

After this incremental data structure is constructed, each line in the mapped arcs file is processed. Processing consists of indexing each source model entity described in the line from the mapped arcs file into the data structure (using the associated line number information) and determining one of three possible actions:

1. retaining the mapping of the source model entity to high-level model entity recorded in the line because no change has impacted the map entry,
2. retaining the mapping of the source model entity to the high-level model entity recorded in the line but updating the number of the map entry involved in the association because changes to the map have shifted the position of the map entry, or
3. matching the source model entity against an ordered list of map entries that may supersede the previous mapping of the source model entity.

For instance, consider a line in the mapped arcs file describing a call from a `vm_object_allocate` function in the `vm_object.c` file to the `malloc` function. The `vm_object_allocate` function would previously have been mapped by the sixth entry in the map and this would have been recorded in the line in the mapped arcs file. When the line was encountered, the node indexed by six would be accessed in the data structure, and a match would now be assumed to the added first entry because of the definitely remaps pointer stored in the sixth entry. A similar indexing and processing would occur with the second half of the tuple described in the line from the mapped arcs file.

The performance of this incremental approach is dependent on the size of the mapped arcs file, the number of added and deleted entries in the map, and the number of entries in the possible and definite pointer lists for the existing and deleted entries.

To gauge the performance improvements with the incremental approach, I measured the execution speed and size of the incremental approach for a one entry change to a 1367 entry map for Excel and compared the data with the time and space necessary to

recompute the reflexion model from primary inputs. The change to the map consisted of the addition of an entry after the existing seventeenth entry in the map. The new map entry remaps an Excel function to a different high-level model entity than the function was previously mapped to by an entry referring to the function's file. This change, referred to as a *logical remodularization* (Section 9.1.2), was a change so frequently performed by the Microsoft engineer during the use of the reflexion model tools that the engineer created special scripts to support the operation. The source model used for the comparison was a 77,746 arc file describing calls between functions extracted from the Excel source.

Table 3.7 shows the execution time, the memory requirements, and the speedup with respect to recomputing from primary inputs when using two forms of the incremental approach. The first form of the incremental approach, `incrementalfile`, reads the existing map, mapped arcs, and unmapped arcs files, applies the mapping modifications, and writes out new versions of the mapped and unmapped arcs files along with a new version of the image arcs file. The second incremental form, `incrementalmemory`, reads the mapping modifications and applies the changes to versions of the mapped and unmapped arcs files held in memory before writing out the image arcs file.<sup>12</sup> The `incrementalmemory` version forms the basis of a server that given modifications to a map could produce and redisplay an updated reflexion model. The performance measurements reported in Table 3.7 include the use of the hash table and regular expression optimizations described earlier in this chapter. The data shows that the `incrementalfile` version of the reflexion model tool can process the change almost 3 times faster than when the change is recomputed from the primary inputs. The `incrementalmemory` version provides a faster turn-around on a change, executing 14 times faster than when recomputing from the primary inputs. This increase in the speed of execution is at the cost of almost ten times as much memory.

---

<sup>12</sup>The times reported for this incremental version do not include the time required to read in the mapped and unmapped arcs files.

Table 3.7: The Performance of the Incremental Algorithm. The performance was measured on a DECStation 3000/300X running OSF/1 V3.2.

Tool	Exec. Time (min)	Memory (Mb)	Speedup From Combined Optimizations
recompute <sub>primaryInputs</sub>	1:39	4.61	1
incremental <sub>file</sub>	0:35	5.4	2.8
incremental <sub>memory</sub>	0:07	45.8	14.1

The performance of the incremental tools might be improved by refining the determination of the definite and possible pointer lists for map entries. Currently, the determination of these lists uses information derived from the optimization of the regular expressions described in Section 3.3.2 to determine whether an entry names a disjoint set or a superset of source model entities when compared to another map entry. Often, a conservative answer must be assumed. More precise, yet still fast, algorithms for comparing the sets named by the regular expressions might decrease the number of entries that need to be considered for a change to the map, thus reducing the execution time necessary to process a change.

### 3.4 Summary

In this chapter, I have described the tools and techniques developed to ensure an engineer is able to both easily describe and quickly compute desired reflexion models. To ease the specification of the maps required for a reflexion model computation, the reflexion model tools were parameterized by a source entity description language. To improve the speed of the reflexion model computation, several optimizations were implemented. In general, the time to compute a reflexion model is reduced most significantly through the combination of a hash table and optimizations for common regular expressions. In the case of the Excel experimental reengineering task, these optimizations permit a reflexion

model to be computed 49 times faster than when a naive computation approach is used. In cases where the source model is very large—describing on the order of several hundred thousand interactions—and the iterative refinements to the map are small, an incremental algorithm may be used that recomputes a reflexion model 14 times faster than the optimized non-incremental tools for the common map changes seen in the Excel example.

## Chapter 4

# Refining the Software Reflexion Model Technique

Engineers have used the reflexion model technique to aid the performance of a variety of change tasks on several different systems (Chapter 9). These uses of the technique have motivated three categories of refinements to the basic technique. First, to provide more assistance to an engineer in understanding different aspects of software structure, the technique has been refined to support multiple kinds of relations in both the source and the high-level models. Second, operations have been added to aid the engineer in managing the investigation of the sizeable volume of information that may be summarized from large software systems. Finally, additional tools have been defined and implemented to improve the usability of the technique. As these refinements require further validation, I present them in this chapter separate from the description of the basic technique that appeared in Chapter 2.

### 4.1 Multiple Relations

Understanding the structure of a software system often requires the simultaneous consideration of more than one kind of interaction, or relation, between system components. In a system designed and implemented using object-oriented techniques, for example, it is often useful to consider both the inheritance and the aggregation structure between

classes of the system to understand the data to which an object of a class may have access. This need to consider multiple interactions concurrently is found in the diagrams resulting from many software design techniques such as Structured Design [Yourdon and Constantine 1979] and OMT [Rumbaugh et al. 1991].

Sometimes, when computing a reflexion model, it is sufficient to union the multiple relations between system components into a single relation. For instance, when reasoning about the estimation task for the NetBSD virtual memory system in Section 2.1.6, it was sufficient to treat the call and data coupling between modules uniformly. The reflexion model shown in Figure 2.5 was computed from the union of the call and data coupling source model relations. The union of these two relations effectively creates a source model comprised of a single “communicates-with” relation.

In other cases, it is useful to maintain a separation of the relations. For instance, if different estimating functions were used for call coupling versus data coupling, the engineer might want to maintain a separation between the two relations in the NetBSD virtual memory example. As another example, an engineer computing a reflexion model based on an existing OMT diagram [Rumbaugh et al. 1991] containing both inheritance and aggregation information may find it beneficial to separate the relations.

When computing a reflexion model, an engineer may require support for multiple relations in only the source model, or in both the source and the high-level model. In this dissertation, a model with multiple relations is referred to as a typed model. In the following sections, I discuss typed source and high-level models and describe the effect of using typed models when computing a reflexion model. A formal characterization of typed reflexion model is presented in Appendix B for those readers desiring further clarification.

#### 4.1.1 Typed Source Model

In a typed source model, each interaction between source model entities has an associated type. The type indicates the relation to which an interaction belongs. A typed source

model for NetBSD, for instance, might associate a “calls” type with each of the function to function call interactions, and might associate a “refers-to” type with each of the references from functions to global data variables.<sup>1</sup>

A typed software reflexion model may be computed from a typed source model and an untyped high-level model. In this case, the types associated with values in the source model induce types on the arcs of the reflexion model. The computation of a typed software reflexion model is a variant of the computation presented in Section 2.2.2. Each source model value in each relation is pushed through the map to compute one or more arcs between high-level model entities. The computed arcs are assigned the same type as that associated with the source model value contributing to the arc. Each computed arc is then compared with the high-level model. If the arc matches an interaction in the high-level model, a convergence is created of the type of the computed arc. If the arc does not match, a divergence is created of the type of the computed arc. Interactions remaining unmatched in the high-level model after the comparison with all the computed arcs are absences; absences remain untyped. Section B.2 in Appendix B formally describes this computation by extending the Z [Spivey 1992] specification presented in Chapter 2.

For example, Figure 4.1b shows a typed reflexion model for the estimation task on the NetBSD virtual memory subsystem. This reflexion model was computed from the typed source model described above and the untyped high-level model presented in Chapter 2 (Figure 4.1a reproduces the high-level model). Two colours—black and grey—and line thickness are used to represent the type of an arc.<sup>2</sup> Thick black lines represent arcs of the type “refers-to”, thin black lines represent the “calls” type, and grey lines represent untyped interactions. As before, the line attributes, solid, dashed, and dotted, represent convergences, divergences, and absences respectively. Thus, the

---

<sup>1</sup>To be precise, the relations comprising a source model are bags that include a count of the number of times each tuple appears in the relation. Since the distinction between bags and relations is not critical in this discussion, the term relation is used for ease of presentation. Appendix B should be consulted when clarification is required.

<sup>2</sup>The colours used in this dissertation are limited to black and grey to ease the production of the dissertation. The reflexion model tools support the use of all colours available in the graphviz [Gansner et al. 1988; Chen et al. 1995] display program.

thick black line—a “refers-to” divergence—from the `KernelFaultHandler` module to the `VirtualMemoryPolicy` module represents unexpected references from functions mapped to the `KernelFaultHandler` module to global data defined in the `VirtualMemoryPolicy` module. As another example, the thin black solid line—a “calls” convergence—from the `VirtualMemoryPolicy` module to the `HardwareTranslation` module indicates an expected call interaction between the two modules. Since the high-level model is untyped, any absences, such as the one from the `Pager` module to the `FileSystem` module, remain untyped and are drawn as dotted grey lines.

Separating the different kinds of source model relations with types and then computing reflexion models using the typed information can provide more information to the engineer viewing the reflexion model. For instance, in Figure 4.1, two self-divergences appear for the `VirtualMemoryPolicy` module: one representing source model values of type “calls”; and the other representing source model values of type “refers-to”. In comparison, only one self-divergence appears when the source model is untyped (Figure 2.5).

#### 4.1.2 Typed High-level Model

In a typed high-level model, an interaction between high-level model entities may have an associated type. Similar to a typed source model, the type of a high-level model interaction indicates the relation to which the interaction belongs. In contrast to a typed source model in which all source model values must be typed, an interaction in a typed high-level model may, under some conditions, remain untyped.

A fully-typed high-level model for the NetBSD virtual memory subsystem example, in which each interaction has an associated type, is shown in Figure 4.2a. In this high-level model, the engineer anticipates all interactions between modules to be of the “calls” type, except for the interaction between the `VirtualMemoryPolicy` and the `Pager` module that is assigned the “refers-to” type.

A typed software reflexion model may be computed from a typed source model and a typed high-level model. In this case, the comparison between the source and high-level

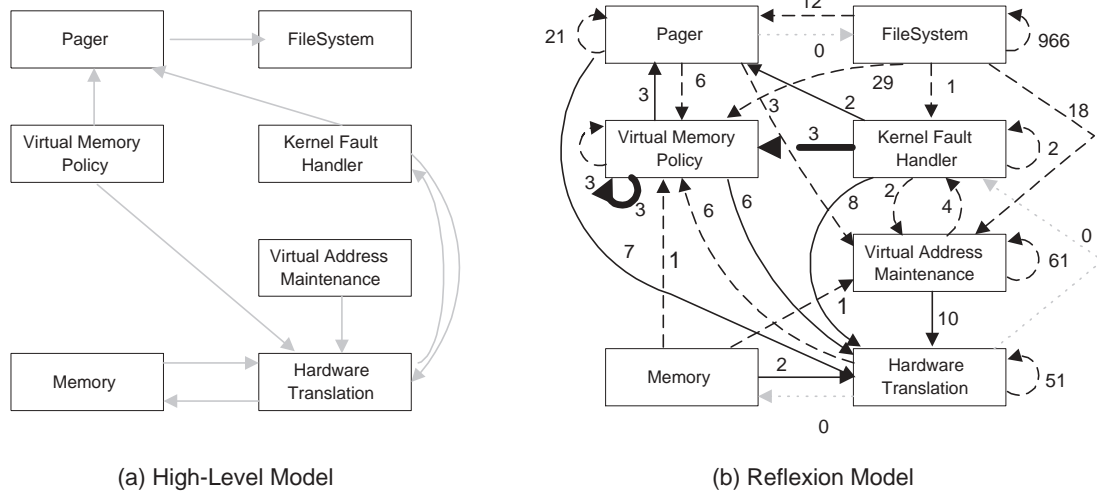


Figure 4.1: Comparing a Typed Source Model and an Untyped High-level Model. Untyped arcs are shown in grey. Arcs of type “calls” are shown as thin black lines. Arcs of type “refers-to” are shown as thick black lines. All arcs in the high-level model are untyped. There are three untyped arcs in the reflexion model: the arc from `Pager` to `FileSystem`, the arc from `HardwareTranslation` to `KernelFaultHandler`, and the arc from `HardwareTranslation` to `Memory`.

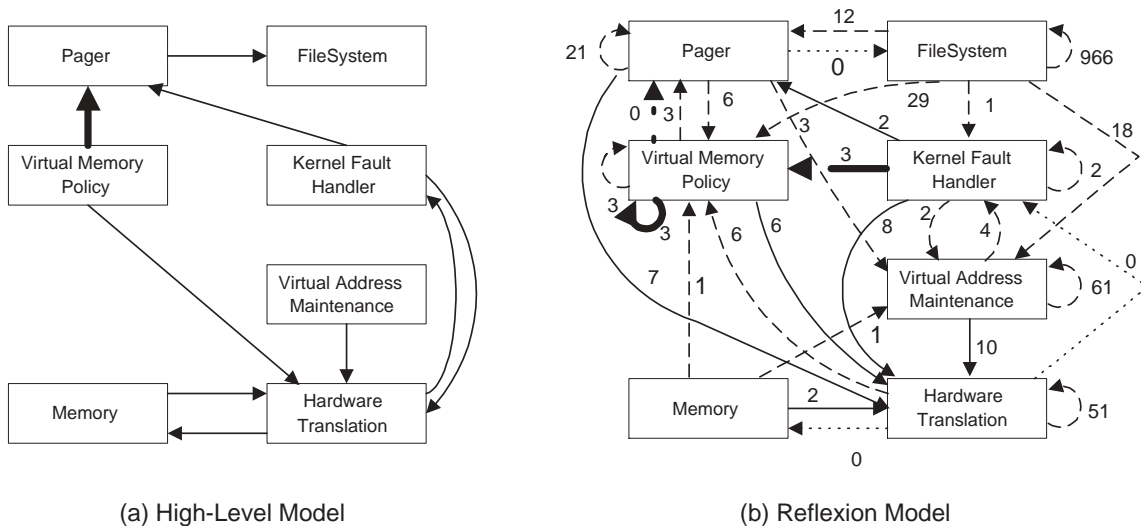


Figure 4.2: Comparing a Typed Source Model and Typed High-level Model. Arcs of type “calls” are shown as thin black lines. Arcs of type “refers-to” are shown as thick black lines. There are no untyped arcs in this reflexion model.

models includes a comparison of the types of the interactions. Figure 4.2b, for example, shows the reflexion model computed from the typed NetBSD source model described in Section 4.1.1 and the typed high-level model shown in Figure 4.2a.

Compared to the reflexion model computed using the typed source model and untyped high-level model in Figure 4.1b, typing arcs in the high-level model leads to an absent arc of type “refers-to” from `VirtualMemoryPolicy` to `Pager` and a divergent arc of type “calls” between the two modules. As was the case with typing the source model, typing the high-level model can provide more information to the engineer at the level of the reflexion model without requiring a detailed investigation of the source model values associated with a given reflexion model arc.

A high-level model is considered partially-typed if some interactions do not have associated types. A partially-typed high-level model must satisfy the constraint that an interaction may be untyped only if it is between two high-level model entities for which no defined typed interactions are specified. This form of computation is useful when an engineer does not require equal levels of detail in all parts of the reflexion model. For example, for the NetBSD virtual memory subsystem, the engineer may wish to compute a reflexion model in which only the interactions between the `VirtualMemoryPolicy`, `Pager`, and `FileSystem` modules are typed (Figure 4.3a). In the reflexion model computed from this high-level model, the “call” type is induced on untyped high-level model arcs like `KernelFaultHandler` to `Pager`. As before, for high-level model interactions with associated types like `VirtualMemoryPolicy` to `Pager`, the types are considered in the computation resulting in an absent arc of type “refers-to”, and a divergent arc of type “calls”, between the `VirtualMemoryPolicy` and `Pager` modules. Partially-typed high-level models support the goal of a lightweight technique by reducing the burden on the engineer to define a type for each high-level model interaction; an engineer may focus on those parts of the model where typing will provide the most benefit.

The reflexion model tools described in Chapter 3 also support the computation of typed software reflexion model and permit an engineer to define a configuration file that

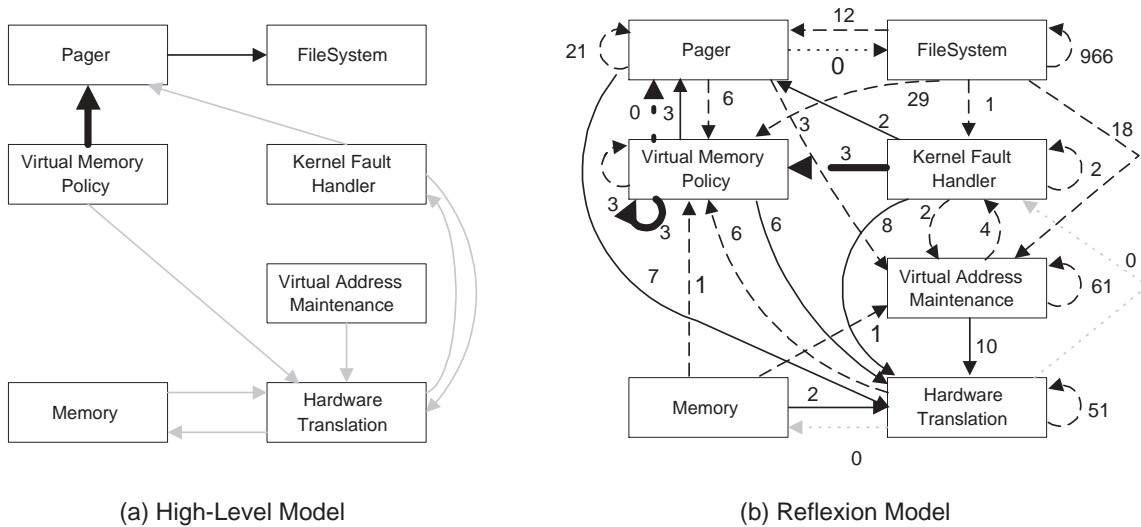


Figure 4.3: Comparing a Typed Source Model and a Partially-typed High-level Model. Untyped arcs are shown in grey. Arcs of type “calls” are shown as thin black lines. Arcs of type “refers-to” are shown as thick black lines. All arcs in the high-level model are untyped except for the arc of type “refers-to” from `VirtualMemoryPolicy` to `Pager` and the arc of type “calls” from `Pager` to `FileSystem`. The reflexion model has two untyped arcs: the arc from `HardwareTranslation` to `Memory` and the arc from `HardwareTranslation` to `KernelFaultHandler`.

describes the colours to use when displaying a typed reflexion model. Although the computation of a typed reflexion model requires the consideration of type information when performing comparisons to the high-level model, the worst-case complexity of the computation remains the same as that for computing an untyped reflexion model since the computation continues to be bounded by the comparison of the source model tuples to the map.

## 4.2 Managing Information

A reflexion model, whether typed or untyped, is able to summarize many structural interactions from a software system for investigation and interpretation by an engineer. The typed reflexion model for estimating the cost of a particular change to the NetBSD virtual memory subsystem shown in Figure 4.1b, for instance, summarizes 1231 call and data coupling interactions to the engineer.<sup>3</sup> The complexity of the view presented to the engineer is dependent not on the number of summarized interactions, but rather, it is dependent on the number of arcs appearing in a reflexion model. Although the engineer has some control over this complexity through the selection of the reflexion model entities, the information presented to the engineer can still be overwhelming.

To permit an engineer to more effectively explore a succession of reflexion models, I refined the process of applying the reflexion model technique by adding two operations: the *tagging* operation permits an engineer to hide interactions in a computed reflexion model from view, while the *annotation* operation permits an engineer to track the investigation of interactions in a computed reflexion model.

### 4.2.1 Tagging

The tagging operation is a form of elision that permits an engineer to hide a reflexion model arc from view until one of the following occurs: the engineer chooses to again

---

<sup>3</sup>Since an untyped reflexion model is a special case of a typed reflexion model, a distinction between typed and untyped reflexion models will only be made in the remainder of this dissertation when necessary.

view the interaction, the interaction changes designation when a succeeding reflexion model is computed (e.g., from a convergence to a divergence), or during the computation of a succeeding reflexion model, the source model values contributing to the interaction change. This operation is useful when an engineer wants to focus on a particular part of the reflexion model for some period of time and wants to retain the flexibility to consider the information summarized within a larger scope.

For instance, in the NetBSD example, the engineer may choose to focus the investigation for a period of time on the interactions between `FileSystem` and `Pager`. The engineer may tag the `FileSystem` to `VirtualMemoryPolicy` interaction because the engineer considers the interaction outside the scope of the task being performed. This tagging operation will hide the interaction between `FileSystem` and `VirtualMemoryPolicy` from view. Alternatively, an engineer could choose to redefine the high-level model to remove these entities from consideration. Redefining the model has two drawbacks. First, the source model values associated with the arc are removed from the summarization. Second, the engineer will not be notified if the values associated with the arc change. Losing this information may lessen the engineer's confidence in reasoning about the task at hand in terms of the model.

I integrated the tagging operation into the reflexion model tools as part of the graphical user interface and as part of the `produceRM` tool. When an engineer tags an arc in a displayed reflexion model, information about the arc and the contributing source model values is written to a file. When a succeeding reflexion model is computed, the saved arc and source model value information is compared to the newly computed reflexion model (as part of the execution of `produceRM`), and based on the definition of the operation above, arcs may be removed from the model before display.

Since each reflexion model arc may be tagged, the worst-case time complexity of the filtering step is given by:

$$O(\#(\text{convergences} \cup \text{divergences}) \times \#(\text{dom } sm)).$$

In other words, the complexity is dependent on the number of arcs in the reflexion model that have contributing source model values multiplied by the number of interactions in the source model.<sup>4</sup>

As an engineer successively tags more arcs, the operation could negatively impact the performance of iteratively computing reflexion models. However, since the more arcs an engineer tags, the less useful the reflexion model generally becomes for reasoning about a task, this possible performance problem is mitigated. More commonly, an engineer might tag only a fraction of the arcs in a reflexion model. When only a few arcs are tagged, the filtering step does not impede the iterative use of reflexion models. For instance, when all convergent and divergent arcs involving the `VirtualMemoryPolicy` in the reflexion model shown in Figure 4.2 are tagged, the execution of the `produceRM` tool takes about four seconds of CPU time compared to just under one second when the tagging information is not considered.

Details of the tagging operation are presented in Appendix B as part of the formal characterization of typed reflexion models.

#### 4.2.2 Annotations

When investigating reflexion models that summarize many structural interactions, it is also difficult for an engineer to track which interactions in the reflexion model, and more specifically, which contributing source model values to a reflexion model arc have been investigated. The annotation operation on a reflexion model helps an engineer track the exploration by recording arcs that have been investigated along with notes describing the relevance of the arc to the task (or high-level model). A visual indication of the effect of annotations is provided to the engineer in a displayed reflexion model by labeling any annotated reflexion model arc with the number of annotated source model values. The

---

<sup>4</sup>This complexity formula assumes the sets of source model values that must be compared for each tagged reflexion model arc are stored in a form that permits an  $O(n)$  comparison.

annotation numeric value is provided in addition to the number of contributing source model values typically shown on a reflexion model arc.

For example, an engineer investigating the NetBSD virtual memory subsystem determines that two of the three source model values contributing to the call divergence between `VirtualMemoryPolicy` and `Pager` are the result of a request to page out a page. To annotate this information using the reflexion model tools, the engineer would record a regular expression describing the divergent arc and the source model values of interest.<sup>5</sup> For instance, the engineer may state the following regular expression:

```
VirtualMemoryPolicy Pager .*vm_pageout_page.* .*vm_pager.c.*
```

As documentation, the engineer may also record, with this regular expression, text describing that these values refer to a paging request. The display of a computed reflexion model with annotations applied is shown in Figure 4.4. This reflexion model includes numeric values in parentheses that indicate how many of the source model values are annotated. The arc between `VirtualMemoryPolicy` and `Pager` is annotated with a two to indicate the two values resulting from requests to page.

I developed this operation as a generalization of a tracking method used by a Microsoft engineer applying the software reflexion model technique to an experimental reengineering of the Excel spreadsheet product. The operation used by the Microsoft engineer is described in Section 9.1.

The operation is supported in the reflexion model tools as a filter step after the execution of the `computeArcs` tool that pushes the source model through the map, but prior to the execution of the `produceRM` tool that compares mapped arcs with the specified high-level model.

The worst-case time complexity of the operation depends, in part, on the comparison of the stated regular expressions to values in the mapped arc file (which contains the mapping of source model values to mapped arcs), and in part, on the determination

---

<sup>5</sup>The regular expression may refer either to the high-level model entities or to the source model values mapped to the high-level model interaction or both.

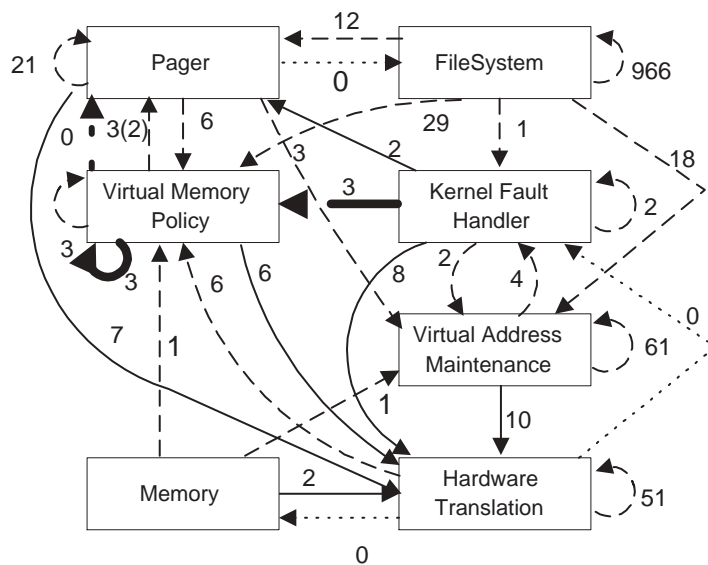


Figure 4.4: An Annotated Reflexion Model. The arc from `VirtualMemoryPolicy` to `Pager` shows 3 mapped calls with 2 annotated (the 2 is in parentheses). With the reflexion model tools, all remaining arcs would show a zero in parentheses since all remaining arcs are unannotated. These have been elided for presentation purposes.

of the numeric value to attach to each reflexion model arc. At worst, determining the numeric values requires visiting each value in the mapped arcs file. Comparing the regular expression to values in the mapped arcs file has a worst-case time complexity given by:

$$O(\#regExp \times \#mappedArcFile \times t_{comparison})$$

where  $\#regExp$  is the number of regular expressions,  $\#mappedArcFile$  is the number of entries in the mapped arcs file, and  $t_{comparison}$  is the cost of comparing a regular expression to a value. Providing a true upper bound for the number of regular expressions is not possible as the number of regular expressions is defined by an engineer. However, a reasonable estimate of the upper bound is the size of the mapped arcs file since the regular expressions name values in this file. The size of the mapped arcs file is bounded by the product of the size of the source model and the square of the number of entities in the reflexion model. The worst-case time complexity of the comparison part of the annotation operation is then:

$$O((\#(dom\ sm) \times (\#rmEntities)^2)^2 \times t_{comparison}).$$

Including the complexity of the numeric value determination would add in a term for the size of the mapped arcs file. Since this term is dominated by the complexity of the comparison part, the above formula also gives a bound for the time complexity of the annotation operation.

As shown by the analysis of the performance of the reflexion model tools in Chapter 3, regular expression comparisons can be costly. The application of the annotations, then, could negatively impact the iterative computation of reflexion models. This impact is managed in two ways. First, since the engineer is in control of defining the number of annotations (the dominant factor in the complexity formula), it is unlikely that the number of annotations would reach the stated bound of the size of the mapped arcs file.

For example, a set of similar regular expressions created by the Microsoft engineer when working with Excel comprised 993 entries, far less than the 119,355 entries in the mapped arcs file for the Excel reflexion model computation. Applying the 993 annotations to the computation of the reflexion model for Excel takes 64 CPU seconds on a DEC 3000/300X. This time is significant as it is 67% of the recomputation time for the Excel reflexion model of 96 seconds reported earlier (Table 3.6). An engineer can balance the cost of the annotation operation with the benefits it provides by using the second impact management technique, the selective application of the annotation filtering step.

Readers interested in the details of the annotation operation are directed to Appendix B which includes a formal description of the operation.

### **4.3 Usability**

The reflexion model tools have been used in both industrial and academic environments. The use of the tools in these environments identified two issues regarding the usability of the technique: the need for an engineer to assess the coverage of the information summarized by a reflexion model, and the need for an engineer to debug a reflexion model computation.

#### **4.3.1 Assessing Coverage**

An engineer's confidence in the information summarized by a reflexion model is dependent, in part, on the inclusion of appropriate structural information for reasoning about the task being performed. One way an engineer can gain confidence in this dimension is to investigate the source model values contributing to a specific reflexion model arc. Another way is for an engineer to assess the coverage of the source model by the reflexion model. The coverage refers to the source model values mapped by the reflexion model computation.

I added three operations to the reflexion model tools to permit an engineer to quantitatively and qualitatively assess the coverage of a computation. First, an engineer may

view a list summarizing the effects of a mapping on the source model. This list describes, for each high-level model entity, the source model entities that have been mapped to that high-level model entity. Figure 4.5 shows the results of an engineer requesting this mapping list for the computation of the typed software reflexion model for NetBSD defined in Section 4.1.2. The window shows, for instance, that the `vm_fault` function located in the `vm_fault.c` file has been associated with the `KernelFaultHandler` high-level model entity. This list aids an engineer in understanding the correspondence of source model entities to high-level model entities. In particular, the list may be used, when trying to understand absences in a computed reflexion model as an engineer may peruse the list to determine if a high-level model entity has any corresponding source model entities defined by the mapping. For instance, an engineer may use the list to assess whether the absences to and from the `VirtualMemoryPolicy` module in the reflexion model shown earlier in Figure 4.2 are the result of the map not associating any source model entities with the `VirtualMemoryPolicy` module.

Second, an engineer may view a list of the source model values (and hence, entities) that are not mapped by the computation. A quantitative count of the unmapped entities is also provided. Figure 4.6 shows a window containing this information for the typed reflexion model for NetBSD defined in Section 4.1.2. For instance, one entry in the window indicates that the `vm_allocate` function in the `vm_user.c` file in the `netbsd/src/sys/vm` directory is not mapped. This list aids an engineer in understanding what aspects of the source model are not being included in the reflexion model summarization.

Third, an engineer may request a quantitative summary of the source model covered by a mapping. This value reports the percentage of the source model that is mapped. This percentage provides a broad basis on which an engineer may assess the amount of information being considered. Using this command for the typed reflexion model for NetBSD, an engineer may determine that 7% of the source model tuples are being summarized by the reflexion model. This coverage is low because the engineer is using the reflexion model to look at a portion, the virtual memory subsystem, of the NetBSD im-

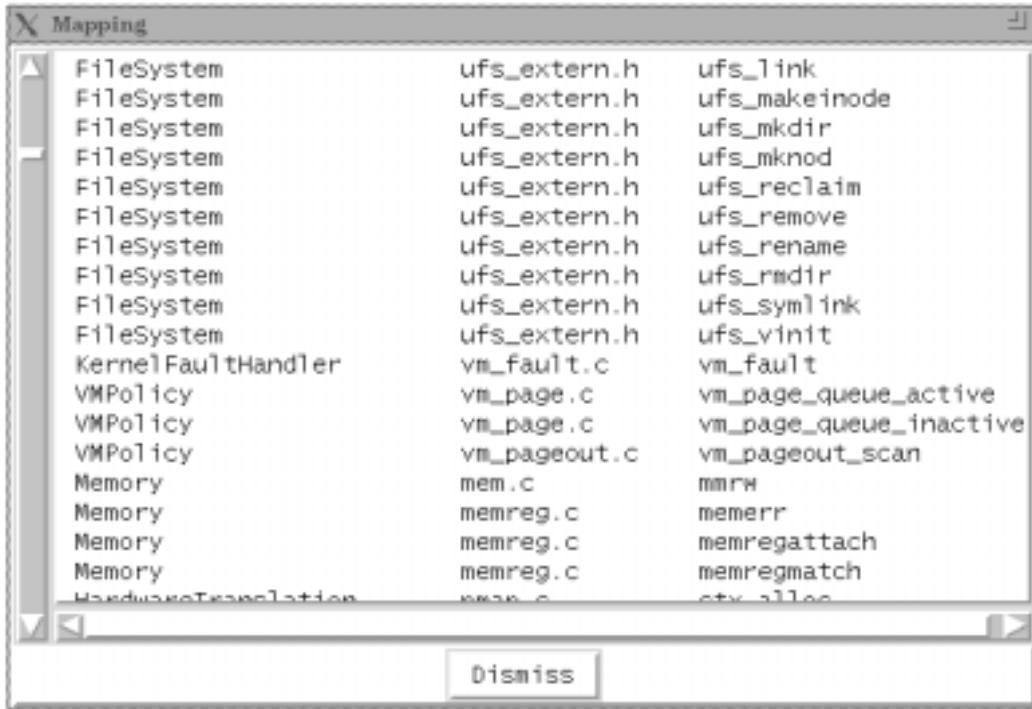


Figure 4.5: Information on the Mapping. A window showing the effects of the mapping for a computation of a reflexion model for NetBSD. For presentation purposes, the name of the `VirtualMemoryPolicy` is shortened to `VMPolicy` in this window.



Figure 4.6: Information on Unmapped Source Model Entities. A window showing the unmapped source model entities for a computation of a reflexion model for NetBSD.

plementation. The Microsoft engineer used a reflexion model with 99% coverage when performing the experimental reengineering task described in Chapter 9.

### 4.3.2 Debugging Computations

Although the mapping language eases the description of a reflexion model computation for the engineer, it can be difficult for the engineer to understand the effect of any given entry in the map on the computation. The operation described in the previous section that displays the source model entities mapped to any particular high-level model entity provides one way in which an engineer may gain insight into the computation. However, this operation does not help an engineer determine if any entries in the map are not having any effect on the computation.

To help the engineer assess the map file, I added an operation to the reflexion model tools that reports, quantitatively, the affect of each map entry. For example, applying this operation to the typed NetBSD virtual memory subsystem example, the following information is returned in a file.

```

6 [ _ _ function = .*page.*active.*$ mapTo=VirtualMemoryPolicy ]
75 [ _ file = .*pager.* _ mapTo=Pager ]
160 [ _ file = .*vm_map.* _ mapTo=VirtualAddressMaintenance ]
24 [ _ file = .*vm_fault.c.* _ mapTo=KernelFaultHandler ]
1992 [ dir = .*fs.* _ _ mapTo=FileSystem ]
141 [ _ file = .*pmap.* _ mapTo=HardwareTranslation ]
60 [ _ file = .*vm_pageout.c.* _ mapTo=VirtualMemoryPolicy ]
4 [ dir = .*arch/sparc/sparc file = .*mem.* _ mapTo=Memory ]

```

Each line in this file corresponds to a line in the map defined by the engineer. The number at the far left of each line is an indication of how many times the map entry was used to map a source model entity. In this case, all map entries are being used to associate source model entities with high-level model entities.

#### 4.4 Summary

In this chapter, I have described three categories of refinements to the basic reflexion model technique. To support the summarization of additional kinds of structural information, I presented a refinement of the computation to support the typing of arcs in both the source and high-level models. The second two categories of refinements affect the process of using the reflexion model technique. First, I presented two operations to help an engineer manage the immensity of information displayed in a reflexion model: the tagging operation permits the engineer to hide stable information during successive computations of reflexion models, and the annotation operation permits an engineer to track parts of the reflexion model that have been investigated. Second, I presented several operations to help support the usability of the tools. These operations include queries to determine the effect of the mapping and to help support the debugging of reflexion model computations. These refinements are implemented in the reflexion model tools described in the previous chapter. Further validation is need to determine the strengths and weaknesses of these refinements.

## Chapter 5

# A Discussion of the Software Reflexion Model Technique

Various choices were made in the design of the software reflexion model technique. In this chapter, I discuss these design decisions in terms of three categories: the representation of models; the definition of maps; and the visualization of reflexion models.

### 5.1 Models

Models are central to the software reflexion model technique. In designing the technique, I made three decisions related to the representation of models. The first decision was to base the technique on syntactic, rather than semantic, aspects of the models. The second decision was to represent models as a collection of binary relations, and the third decision was to use non-hierarchical models. I discuss the ramifications of each of these decisions in turn.

#### 5.1.1 Syntactic Models

Reflexion models, as described in the formal characterization of Section 2.2 (and Appendix B), are computed without any knowledge of the intended semantics of the high-level or the source models; the semantics are applied only by the engineer when specifying and interpreting a reflexion model. Syntactic models were chosen over semantic models

for three reasons. First, syntactic models provide significant flexibility, allowing engineers to investigate many different kinds of structural interactions in a software system (e.g., calls, data dependences, event interactions, etc.). This flexibility has been beneficial in practice, as reported in Chapter 9. For instance, engineers have exploited this flexibility by changing the meaning of the models across the iterative computation of a succession of related reflexion models. Both a virtual memory developer applying reflexion models to the NetBSD system and the Microsoft engineer applying reflexion models to the Excel system, for example, first used a calls source model for computing reflexion models and later, as successive reflexion models were computed, augmented the source models with static dependences of functions on global data. By adding static dependences, both engineers implicitly shifted their high-level model from a calls diagram to a communicates-with diagram. In both cases, the changes were driven by the need to understand additional aspects of the system structure for the task being performed.

The second reason for choosing syntactic models was that these models support the goal of providing a lightweight technique since they require minimal specification by an engineer. Finally, I chose syntactic models because I was interested in investigating the boundaries of usefulness of these models.

Using syntactic models for the comparison has several consequences. One consequence is that the engineer must ensure that it makes sense to compare a selected high-level model with an extracted source model. For example, comparing the calls between modules diagram of the NetBSD virtual memory system (Figure 2.3a) with a source model consisting of an extracted calls relation makes sense (Figure 2.3b), but comparing the same high-level model with a source model representing the `#include` structure of NetBSD would probably be meaningless. Inappropriate comparisons have not arisen in practice.

A second consequence of syntactic models is that the engineer is responsible for selecting, or at least understanding, the appropriate content style for each structural relation in a source model. Each structural relation has one of four different content styles:

precise, conservative, approximate, or optimistic. Table 5.1 shows how these styles are differentiated by the occurrences of false positives and false negatives. A *precise* structural relation is one that contains exactly those values for which the relation is true in the system artifacts (i.e., an inherits-from relation extracted from C++ [Stroustrup 1986] object-oriented code may be precise). A *conservative* structural relation is one in which there are values that are false positives (i.e., for a calls source model, values that represent calls that do not really occur in any execution of the program), but in which there are not any false negatives (e.g., calls that exist in some execution of the program for which there is not a relation value).<sup>1</sup> An *approximate* structural relation is one that admits both false positives and false negatives. Finally, an *optimistic* structural relation is one that admits false negatives but not false positives (i.e., a calls model extracted from the dynamic execution of a system).

Table 5.1: A Comparison of Content Styles for a Structural Relation.

	<i>FalsePositives</i>	$\neg$ <i>FalsePositives</i>
<i>FalseNegatives</i>	approximate	optimistic
$\neg$ <i>FalseNegatives</i>	conservative	precise

The content style of each relation in the source model must be suitable for the kind of reasoning the engineer wants to perform with the reflexion model. For instance, the engineer’s level of confidence that two entities in the reflexion model that are not connected by any arc are indeed independent is lower if the source model relation(s) are *optimistic* or *approximate* than if they are *conservative* or *precise*. As is discussed later in Section 8.2, it is often difficult for an engineer to assess the content style of a structural relation extracted by a tool. Placing this responsibility on the engineer to be conscious of what may be reasoned about is not unique to the reflexion model technique, but also exists when using software visualization and reverse engineering tools. One

<sup>1</sup>The meaning of conservative is dependent on the structural relation of interest. For the purposes of this thesis, we will use conservative in the sense defined above.

benefit of the architecture of the tools supporting the reflexion model technique is that the engineer must explicitly select a tool to produce the source model rather than the more common approach of bundling and hiding the tools to produce the source model within an environment. Requiring the engineer to select the extractor helps bring the issue to the forefront so that the engineer may make an effective decision for the task being performed.

To date, requests by users of the software reflexion model technique have been primarily for scripts and tools to gain access to additional kinds of structural information in the source, such as including calls through function pointers in C, and to gain access to different styles of source models, such as gaining access to calls in data initialization code to form a more conservative source model, rather than for semantic models. Additional experience with the technique is necessary, however, to begin to characterize the kinds of tasks that an engineer may confidently and easily reason about using the syntactic models found in the software reflexion model technique.

### 5.1.2 Models as Collections of Binary Relations

In essence, each of the high-level and source models are described as collections of binary relations. A high-level model consists of a collection of binary relations between high-level model entities where each relation represents a different type of interaction. Similarly, a source model consists of a collection of binary relations between source model entities where each relation represents a different type of interaction.<sup>2</sup> Binary relations are often used to represent information about software systems [Linton 1984; Meyer 1985]. Meyer describes why binary relations were chosen to represent information about a software project in a software knowledge base (SKB):

The SKB system only uses binary relations; the reason is that binary relations are mathematically simple, have nice properties, and provide an intuitively

---

<sup>2</sup>This discussion uses the more abstract notion of a source model that ignores the treatment of the model as a multigraph since the information concerning the number of times a particular source model interaction occurs does not impact the kind of structural information represented.

appealing way to describe structural properties of systems. From the theoretical standpoint, any system that can be described using general relations (as e.g. with a relational data base management system) can be described with binary relations [Bracci et al. 1976], and algorithms have been proposed to efficiently translate a binary schema into a more efficient  $n$ -ary one [Rishé 1985]. [Meyer 1985, p. 160]

For the same reasons, binary relations were chosen, and are sufficient, to represent structural information when computing reflexion models.

### 5.1.3 A Non-Hierarchical Structural Comparison

The software reflexion model technique supports the comparison of non-hierarchical structural descriptions at different levels of abstraction. If an engineer desires to consider different hierarchical levels of software structure within a reflexion model, for instance, to compare the abstract design of module nesting with the nesting described in the source, the engineer has two choices: the engineer may express the hierarchical structure as a relation in the high-level model, or the engineer may compute different reflexion models for each level of the hierarchy.

This design choice was made as part of the philosophy of the research, namely not to add features into the techniques unless the features were required by users. To date, engineers have not found this use of non-hierarchical structural descriptions restrictive.

## 5.2 Maps

Equally as important as design decisions related to the representation of models are design decisions related to the form and content of the map that relates parts of a source model to parts of a high-level model. The form of the map, chosen to be an entity-to-entity correspondence in the software reflexion model technique, affects the range of computations that may be performed, while the content, chosen to be a parameterized declarative language, affects the usability of the technique. I consider each of these design decisions in turn.

### 5.2.1 The Form of a Map

A map for a reflexion model computation states a correspondence between some (or all) of the entities of the source model and some (or all) of the entities of the high-level model. This entity-to-entity form for the map was chosen because it supports the projection of interactions from the source model to a higher level of abstraction; this projection has been shown to be useful for software engineers performing in a range of software change tasks (Chapter 9).

Other choices for the form of this map are also possible. For instance, an engineer may desire to map entities in a source model to interactions within a high-level model. Sullivan describes such a case, where an engineer may model a portion of an integrated programming environment as an **Editor** that may cause an action in a **Compiler** on certain events (top portion of Figure 5.1) [Sullivan 1994, p. 11–14]. However, in the implementation, the relationship between the two tools may be implemented as an entity called a **mediator** (bottom portion of Figure 5.1). This kind of association is not supported directly by a reflexion model computation. Rather, an engineer must choose one of the following approaches: introduce a mediator entity in the high-level model, associate the source level **mediator** entity with one or both of the **Editor** and **Compiler** high-level model entities, or pre-process the source model to compute the appropriate interpretation of interactions between the **mediator** and the **editor** and **compiler** to interactions involving only the **editor** and the **compiler**. Similar difficulties arise if an engineer desires to associate interactions in a source model with interactions in a high-level model. More research is needed to understand when different forms for the map are appropriate and the consequences of the desired forms on a reflexion model computation.

Another design choice made in the current map form was to support the specification of partial maps in which an engineer states only a partial correspondence between the two models. Similar to syntactic models, partial maps are beneficial because they contribute to the lightweight nature of the technique. An engineer, for example, may focus on a

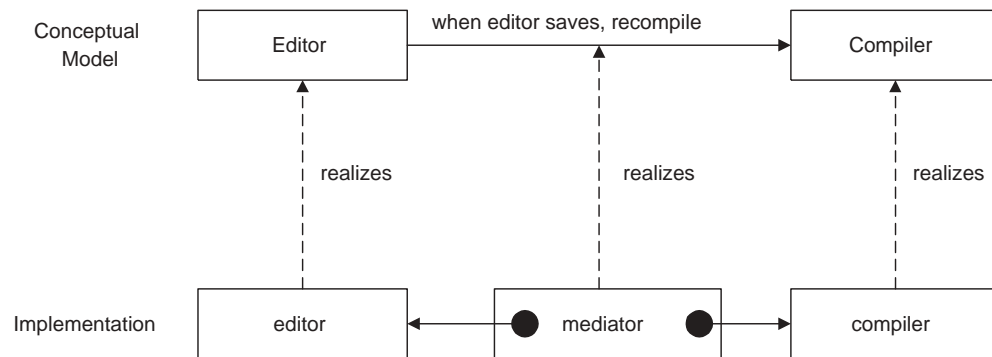


Figure 5.1: A Mediator-based Realization of a Programming Environment. This figure is based on Figure 1.7 in Sullivan’s dissertation [Sullivan 1994].

particular subsystem important to a task and then successively include other subsystems as necessary by expanding the map. Partial maps are also beneficial in handling scale. Rather than having to provide a full correspondence for a system like NetBSD that contains 3000 source model entities before computing a reflexion model, an engineer may focus on those aspects of the source model pertinent to the task, enabling an engineer to manage the time spent in defining maps with the benefits accrued from using the technique.

### 5.2.2 The Content of a Map

For the software reflexion model technique to be usable, it is imperative that a software engineer be able to specify the content of a map easily, even when there are many entities. One design choice made to achieve this goal was to parameterize the language of a map by the source entity description language. The source entity description language, based on a naming tree, is sufficiently flexible to describe the physical and logical organization of a wide-range of source structures. The language has been used, for example, to describe the physical and logical arrangement of Ada [GPO 1983], C++, C [Kernighan and Ritchie 1978], CLOS [Bobrow et al. 1989], Eiffel [Meyer 1992], TCL/TK [Ousterhout 1994],

and Field structured data files [Reiss 1990; Reiss 1995] for the purposes of computing a software reflexion model. Furthermore, the parameterization of the map language by the source entity description language permits an engineer to use the vernacular of the system to describe the desired map, making the specification of the map more natural to the engineer. For example, when analyzing a system implemented in C++, an engineer may define a description language to permit the reference of entities by familiar terms such as file, class, method, or function.

An alternative design considered for the source entity description language was to create a superset of all language structures. An engineer using such a language to compute a reflexion model would be able to choose the particular keywords pertinent for the implementation language entities named in the source model. This design was rejected because different implementation languages use similar terms to refer to semantically different entities. For example, a package in Ada is a very different kind of entity with a very different kind of structure than a package in CLOS. If the names of any of these entities were changed, even to the extent of referring to an `Ada_package` or a `CLOS_package`, the vernacular of the map would begin to deviate from the terminology commonly used by the engineers.

Generally, an engineer can define a source entity description language once and then use the language for many different reflexion model computations. For instance, the source entity naming tree for naming functions and variables in C code shown in Figure 3.2 may be used for a wide range of reflexion model computations. Thus, an engineer need not always be burdened with the definition of this input to the computation. Sometimes, however, it is necessary for an engineer to define the language. In these cases, an engineer must take care to keep the language close to the structure of the source code, as embedding more abstract structure into the language (i.e., adding a “class” construct for C code) places inappropriate pressure on the construction of a source model adhering to the naming language.

The design of the source entity description language supports reference to entities in

a multi-lingual system by the appropriate terms. For instance, if a system is implemented in both C++ and TCL/TK, an engineer may define an appropriate naming tree—perhaps a union of the naming trees commonly used for each language—to refer to entities using the appropriate terms. Support for multiple languages is particularly helpful when analyzing large systems that are often composed of programs written in different languages.

The source entity description language also allows an engineer to define a semantically approximate map in which an entry coarsely associates source model entities with high-level model entities. For example, an engineer may define an entry that associates, by way of substructure, all C functions in a particular file with a specified set of high-level model entities. This mapping may be approximate because one or more of the functions, to support reasoning about the task, may need to be remapped to a different set of high-level model entities; that is, the mapping by structure alone is not sufficiently precise. The source entity description language thus admits false positives in a map in its ability to name groups of entities. The language also admits false negatives in allowing entities to remain unmapped.<sup>3</sup> Approximate maps that were later refined were used by the Microsoft engineer applying the reflexion model technique to Excel (Section 9.1). The ability to approximately map entities until an engineer finds that a finer granularity is needed is beneficial in permitting an engineer to focus attention, through successive map refinements, on the areas of structure of most interest to a task at hand. Similar to the case of partial maps described above, approximate maps also contribute to the lightweight nature of the technique and aid in handling scale.

---

<sup>3</sup>The concept of the source entity description language admitting false negatives is related to the partiality of the map. These two concepts differ in the intent of the engineer in defining the map. An engineer defines a partial map when interested in a subset of the system. A map has false negatives when, given a defined subset of the system of interest, the engineer defines a map that leaves desired entities unmapped.

### 5.3 Visualization

Although the reflexion model technique has a visual component, the graphical visualization of a computed reflexion model has not been essential to the effective use of the technique. As described in detail in Section 9.1, the Microsoft engineer investigated a computed reflexion model primarily from the textual output of the reflexion model tools. Some of this mode of the use of the tool may be a result of the lack of stability in the layout of a reflexion model by the graphviz [Gansner et al. 1988; Chen et al. 1995] tool from computation to computation. When laying out a reflexion model graphically, it is generally desirable to fix the location of the nodes (entities) in the graph even if this leads to a higher degree of edge (arc) crossings. Fixing the nodes allows an engineer to more easily compare the input high-level model with a computed reflexion model, and to compare successively computed reflexion models. The graphviz tool provides only limited capabilities to express constraints on the location of nodes, often leading to a great variance in the location of nodes between successive reflexion models.

Even if more stability in the graphical layout of reflexion models was achieved, some information embedded in a reflexion model may be more useful in textual form. For instance, the Microsoft engineer drove the investigation of a reflexion model from the counts attached to arcs in a reflexion model. This information is easy to understand in textual form by sorting the arcs and associated counts, but may be difficult if the information is only embedded within a graphical layout of a reflexion model.

To understand the role of visualization in the reflexion model technique, more experience must be gathered of the use of the tools in a variety of situations. Based on experience from a usability study of the existing tools conducted by another University of Washington graduate student, Kurt Partridge, the situations that are studied should vary both in the software engineering task being performed and in the expertise of the engineer applying the tool. The usability study consisted of observing pairs of students from a graduate software engineering class attempt to use the tool to perform a simple

task on a compiler comprised of a few thousand of lines of C++ code. In contrast to the use of the technique by the Microsoft engineer (Section 9.1), the students requested more graphical features, such as being able to define the high-level model through a graphical interface, and they were slow to use the textual iterative features, such as modifying the map, available to them. The differences in usage between the students and the Microsoft engineer may be attributable to the study format; students were typically observed in the first hour or two of their use of the tool. However, the differences may also arise from the variation in the tasks being performed or the variation in the development experience of the students compared to the Microsoft engineer. Similar differences are also apparent in the uses of the tools described in Chapter 9. Any study to assess the role of visualization in the reflexion model technique must account for these factors.

#### **5.4 Summary**

In this chapter, I have described several of the design decisions made during the development of the software reflexion model technique including decisions related to the representation of models, of maps, and of computed reflexion models. In the next three chapters, I describe the companion lexical source model extraction technique that was motivated, in part, by attempts to apply the reflexion model technique to a number of systems. Readers primarily interested in the software reflexion model technique may wish to jump to Chapter 9, which describes several case studies of the use of the technique.

## Chapter 6

# Lexical Source Model Extraction

To compute a software reflexion model, an engineer must extract structural information from system artifacts to form a source model. Sometimes, an engineer produces the source model by instrumenting the dynamic execution of the software system.<sup>1</sup> Other times, an engineer produces the model by applying static analysis tools to the system artifacts. For many change tasks, the static analysis approach is preferable because, among other reasons, the structure over various configurations of the system is of interest or because the system is not executable.

One class of tools often used by engineers to statically analyze an artifact is the class of lexical tools that includes `grep`, `awk` [Aho et al. 1979], and `lex` [Lesk 1975]. An engineer produces a source model with these tools by specifying regular expressions to be matched to text within the artifacts. One advantage of these lexical tools is their versatility; few constraints are placed on the kinds of artifacts to which they may be applied. For example, regular expressions can be applied to both source code and documentation. Lexical approaches are also generally fast and easy to use.

When the desired source model requires recognition of syntactic constructs such as call graphs, though, the use of existing lexical tools quickly becomes unwieldy. A seductive alternative available to engineers to overcome the problems associated with the lexical

---

<sup>1</sup>For example, software reflexion models have been computed using a source model based on the output of the `gprof` [Graham et al. 1982] profiling tool.

approaches are parser-based approaches. But existing parser-based approaches have two drawbacks. First, parser-based approaches generally place stringent constraints on the artifacts from which source models are to be extracted, precluding their use during some maintenance activities. Many parser-based tools, for example, require that all system header files be present and correct; this is an important constraint for compilation, but is overly strict for computing a call graph while a system is being ported. Second, modifying an existing parser—for instance, to produce a new source model—can be quite complicated in practice.

Expanding a tool’s capabilities to include additional source languages and additional analyses, while seemingly conceptually simple, can often be quite difficult. The statement that “all you have to do is add a new parser” is deceptively appealing. [Reubenstein et al. 1993, p. 117]

This brittleness often drives engineers back towards lexically-oriented, and often ad hoc, approaches. For example, Wong and several colleagues recently described a case in which they decided against writing a parser to produce a source model during a structural redocumentation task, instead extracting the information using “a collection of Unix’s `csh`, `awk`, and `sed` scripts...” [Wong et al. 1995, p. 49].

I have developed a lightweight source model extraction approach that extends the kinds of source models that can be easily produced from lexical information. Being lexical, this technique avoids the constraints and the brittleness of most parser-based approaches. This technique allows software engineers to generate flexible and tolerant source model extractors as needed for the software engineering tasks—including change tasks—being performed.

- By *lightweight*, I mean that the specifications for new extractors are reasonably small and easy to write. For example, specifications for C [Kernighan and Ritchie 1978] call graph extractors, event extractors for CLOS [Bobrow et al. 1989], and global variable references extractors for TCL [Ousterhout 1994] are all fewer than 25 lines.

- By *flexible*, I mean that there are few constraints on the structure of the artifact considered. For instance, the technique has been used to generate extractors for both source code and structured data files.
- By *tolerant*, I mean that there are few constraints on the condition of the artifact from which information is to be extracted. For instance, the technique has been used to extract call information from source that does not compile; this information may be valuable to software engineers trying to port the system.

Similar to existing lexical, and some syntactic, methods of extracting source models, the technique produces approximate information—not all intended constructs may be extracted, and some unintended constructs may be extracted. In general, the technique trades precision in extraction for improved efficiency in developing a desired extractor, and increased flexibility and tolerance in the generated tool.

In this chapter, I describe the technique and present an example of its use in extracting the implicitly-invokes relation between tools in the Field programming environment [Reiss 1990; Reiss 1995]. I also discuss how the technique may be used to produce source models suitable for use in computing a reflexion model. The tools supporting the technique and the features of the approach are discussed in the following two chapters.

## 6.1 The Technique

An overview of the lexical source model extraction (LSME) technique is shown in Figure 6.1. An engineer examines the system artifacts and writes a lexical specification describing the information to extract as a source model. Using this specification, the system produces a source model from the artifacts. As necessary, the engineer may refine the specification and recompute a new source model.

The system consists of two generators: a scanner generator and an analyzer generator. As shown in Figure 6.2, each generator reads the specification written by the engineer. In the specification, the engineer defines:

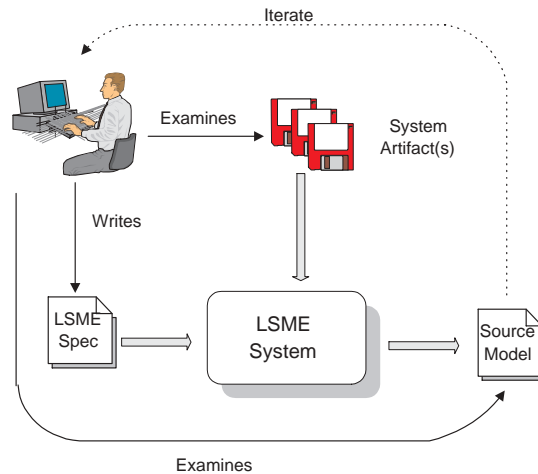


Figure 6.1: The Lexical Source Model Extraction Technique.

1. patterns describing constructs of interest in a system artifact,
2. actions to execute when a pattern is matched to information in an artifact being scanned, and
3. optionally, post-processing analysis operations for combining local information extracted from individual files into a global source model.

The scanner generator uses the first two parts—the pattern and action definitions—to generate a scanner that reads in a sequence of artifact files and produces a stream of local output. Sometimes, for instance when determining the imports relation between Ada [GPO 1983] packages, the desired source model may be produced by simply scanning individual files. Other times, for instance when determining the calls relation between C files, the desired source model must be computed by combining information scanned from individual files. When information must be combined, the output from a generated scanner may be an intermediate relational representation.

The analyzer generator uses the optional third part of the specification, the post-processing operations, to generate an analyzer that reads an intermediate representation stream produced by a scanner and computes the desired source model.

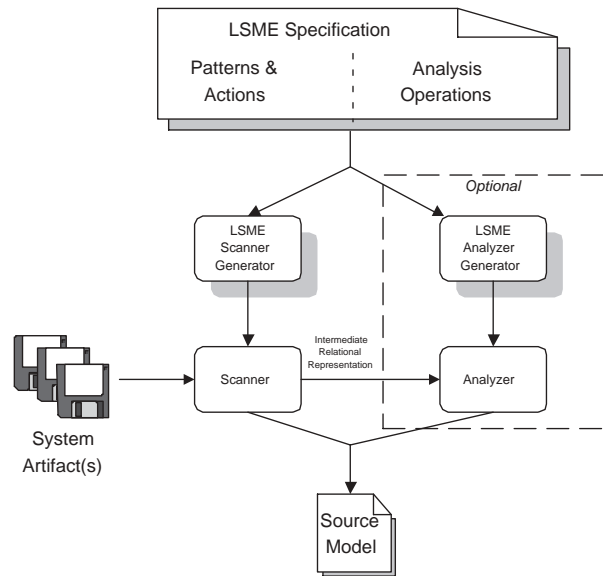


Figure 6.2: The Architecture of the Lexical Source Model Extraction System.

Sometimes, producing the desired source model requires scanning several different kinds of system artifacts. For instance, as described in Section 6.3, extracting the implicitly-invokes relation between tools in the Field programming environment involves two scanners: one to scan C source code and another to scan structured data files. Multiple scanners may also be used over the same system artifacts to produce a source model. For example, to produce a “communicates-with” relation from C code similar to the relation used in computing a refined reflexion model for the NetBSD estimation task described in Section 2.1.6, an engineer might use one scanner to extract information about calls between functions, and another scanner to extract information about references of functions to global variables. The results from multiple scans may either be appended to a source model produced directly from scanning, or may be appended to the intermediate relational representation. Appending this information is easy since both the source model and the intermediate representation stream are stored as ASCII files. As necessary, a generated analyzer may be used to combine the results from multiple scans.

## 6.2 The Specification Language

The language for specifying extractors has three parts: patterns describing constructs of interest to search for in system artifacts, action code to execute after a pattern is matched to a portion of a system artifact, and, optionally, analysis operations that compute a source model from an intermediate relational representation produced during scanning.

### 6.2.1 Specifying Patterns

The engineer describes the information to extract from the system artifacts as a set of hierarchical patterns. Each pattern uses regular expressions to describe a construct that may be found within the artifacts. For example, the following pattern may be used to extract the names of functions defined within a file containing K&R-style C source code:<sup>2</sup>

```
[ <type> ] <functionName> \( [ { <formalArg> }+ ] \) [ { <type> <argDecl> ; }+ ] \{
```

This pattern specifies that a function definition consists of an optional type specification, followed by the name of the function, a left parenthesis, an optional list of formal arguments, a right parenthesis, an optional list of declarations of the types of the formal arguments (each of which is terminated by a semicolon), and an opening curly brace for the start of the function body. The names appearing within angle brackets in the pattern introduce scanner variables that will be matched to tokens from the input stream during scanning. The scanner generated from this pattern has no semantic knowledge of the form or possible values of these scanner variables; for instance, the generated scanner is unaware of the names of the built-in types in the C programming language.

This pattern will not generally extract all function definitions; for example, it will not match definitions with an argument declared to be of a pointer type where the asterisk is separated by white space from both the type and the argument identifier

---

<sup>2</sup>The notation uses square brackets to indicate optional constructs, { } to indicate a non-empty sequence of constructs, { }+ to indicate one or more non-empty sequences of constructs, | to represent alternation, and backslashes to escape reserved single-character tokens.

(e.g., `int * x;`). Simple refinements of this basic pattern, though, can be used to find all function definitions for existing bodies of code. For example, this pattern has been iteratively refined to find all of the function definitions in the 18,000 lines of C, yacc [Johnson 1975], and lex code comprising the cross-reference tool of the Field software system (see Section C.1).

Patterns may be nested hierarchically. For instance, to extract a static calls relation between functions, the engineer may specify the following two patterns where the pattern after the blank line is a child of the first pattern.

```
[ <type> ] <functionName> \( [ { <formalArg> }+ ] \) [ { <type> <argDecl> ; }+ ] \{
<calledFunctionName> \( [ { <parm> }+ ] \)
```

During extraction, the source will be scanned for an occurrence of the first pattern. Once the first pattern pattern is matched, scanning will continue looking for both another occurrence of the first pattern and also occurrences of the second pattern. Searching for both patterns after a match ensures that the scanner will not miss the start of the next function declaration while still being tolerant to syntactical deviations in the source code. For example, the scanning is not dependent on a closing curly brace or, moreover, to perfectly matched braces in the definition of the function.

Disjunction is supported by permitting the description of multiple patterns at the same level of the hierarchy. For example, an engineer can search for global data declarations and function definitions by defining two patterns at the same hierarchical level.

In contrast to most scanning approaches that define tokens on a per-language basis (for instance, in lex), the patterns specified by the engineer implicitly define two classes of tokens. The first class of tokens is the class of single-character tokens. These tokens are defined by their appearance within a specified pattern. For instance, the escaped left and right parentheses in the patterns above become single-character tokens. The second class of tokens is the class of identifiers, consisting of any sequence of non-white space

characters that do not contain any single-character token.<sup>3</sup>

### 6.2.2 Specifying Actions

An engineer may attach action code to patterns to be executed when a pattern is matched in the source code. The action code can access the value of the scanner variables matched to scanned tokens and perform operations such as writing to the local output stream. The pattern shown below will write out a stream of the form “function calls function” as calls are matched in the source.

```
[ <type> ] <functionName> \( [ { <formalArg> }+ ] \) [ { <type> <argDecl> ; }+ ] \{
    <calledFunctionName> \( [ { <parm> }+ ] \)
    @ write ( functionName, " calls ", calledFunctionName ) @
```

The @ symbols introduce action code to be executed when the second pattern is matched in the input source. The tools developed to support the technique define actions in Icon [Griswold and Griswold 1983], a general-purpose imperative programming language with special features for string scanning, and built-in support for set, list and table data structures. Action code may be attached to any scanner variable or one-character token appearing within a pattern.

In addition to producing output, action code may be used to control the matching of constructs in the source to particular patterns. Specifically, an engineer may reject matches to a particular pattern by invoking the `fail` expression within the action code. This control is often used to reject matches when patterns are too general. For example, the child pattern for locating calls within a function specified above may also match control constructs such as `if` statements. An engineer can reject these matches by

---

<sup>3</sup>In most cases, white space consists of any number of space, tab, and newline characters. A mechanism is provided for redefining starting and end character sequences to identify blocks of comments to be ignored by the tokenizer. Sometimes, this may remove a newline from consideration as white space. Newlines may be mentioned within a pattern; this also removes newlines from consideration as white space.

testing the value of the `calledFunctionName` variable:<sup>4</sup>

```

<calledFunctionName>
  @ if calledFunctionName == "if" then
    fail
    write ( functionName, " calls ", calledFunctionName ) @
\ ( [ { <parm> }+ ] \ )

```

When a match is failed, the scanner will backtrack and try to match the next best alternative. This scheme is similar to the **REJECT** mechanism in the lex scanner generator, and as with lex, the fail mechanism may be used to match overlapping patterns. I discuss the affect of failing a match on the behaviour of the scanner in further detail in Section 7.1.1.

### 6.2.3 Specifying Analysis Operations

Sometimes, the desired source model cannot be produced directly during scanning. Instead, the source model must be computed at the conclusion of scanning from multiple kinds of information extracted from the artifacts. For example, a calls source model that includes information about the files in which the caller and callee functions are declared—referred to as a global calls source model in this dissertation—must go beyond the example above because the file of the callee can only be resolved after scanning is completed. An engineer defines the desired computation in an analysis section of the specification that is input to the tools. The computation is performed on the intermediate relational stream produced from scanning.

Consider the use of the tool to produce a global calls source model. First, the engineer must place the necessary information to compute this model on the intermediate relational stream by using the special **relation** function within action code attached to a pattern. For example, an engineer may use the **relation** function to record the file in

---

<sup>4</sup>The Icon operator `==` compares two strings.

which a function is defined when a function definition boundary is recognized:

```
[ <type> ] <functionName> \ ( [ { <formalArg> }+ ] \ ) [ { <type> <argDecl> ; }+ ] \ {
  @
  file := getArguments(1)
  relation ( "decl", "file=" || file, "function=" || functionName )
  @
```

The `relation` function writes one tuple of a binary relation to the intermediate stream. The first parameter to the `relation` function is the name of the relation to which the tuple being written belongs. The second and third parameters are the values of each part of the binary relation. In this case, the file location is recorded in a relation named “decl”. The tuple values are strings consisting of a space-separated list of keyword and value pairs. In the example above, for instance, the name of the file is retrieved as the first argument to the scanner program using the `getArguments` function. The second parameter is then formatted as a string consisting of the keyword “file” with the value of the retrieved name of the file.<sup>5</sup> The third parameter records the name of the function scanned as the value of the “function” keyword. In a similar fashion, the engineer can record the name of the caller and callee functions within a “calls” relation.

In the analysis section, where the desired computation is defined, the engineer may use relational operations such as selection and difference which are provided as built-in functions to a generated analyzer as well as general Icon code. For instance, to form the global calls source model, an engineer may use relational selection operations to determine the file information for each function involved in a tuple of the “calls” relation. Given the file information, the global “calls” source model may be output. The analysis section specified by an engineer to perform this computation is shown in Figure 6.3.

Table 6.1 describes the functions available for use in an analysis section of a specification.

---

<sup>5</sup>The Icon `||` operator concatenates strings.

```

analysis @
# A line starting with a # is a comment line.

# Retrieve all tuples from the calls relation.
callR := relationSelect ("calls", "", "" )

# Visit each tuple in the calls relation.
every tuple := !callR.records do {

    # Extract the caller from the tuple
    caller := tupleGetValue ( tupleProject ( tuple, 1 ), "function" )

    # Determine the file of the caller by querying the decl relation
    selectR := relationSelect ("decl", "function=" || caller, "" )
    callerFile := tupleGetValue
        ( tupleProject ( get (selectR.records), 1), "file" )

    # Extract the callee from the tuple
    callee := tupleGetValue
        ( tupleProject ( tuple, 2 ), "function" )

    # Determine the file of the callee by querying the decl relation
    selectR := relationSelect ("decl", "function=" || callee, "" )
    calleeFile := tupleGetValue
        ( tupleProject ( get (selectR.records), 1), "file" )

    # Write out the call information with associated files.
    write (callerFile, "/", caller, " ", calleeFile, "/", callee )
}
@

```

Figure 6.3: Analysis Specification for Computing a Global C Calls Source Model.

Table 6.1: Icon Functions to Support Relational Operations.

<code>relationCreate ( )</code>	Creates a new binary relation.
<code>relationMakeIndex ( theRelationName, field, keyWord, keyWordList )</code>	Make an index for the specified relation on the indicated field and for the specified keyword in the third and fourth parameters.
<code>relationAddTuple ( theRelation, theTuple )</code>	Add the specified tuple to the specified relation.
<code>relationMember ( theRelation, value1, value2 )</code>	Is the tuple described by value1 and value2 part of the specified relation?
<code>relationDifference ( relation1, relation2 )</code>	Compute the difference between two structurally similar relations.
<code>relationSelect ( theRelationName, key1, key2 )</code>	Select tuples from the specified relation using the criteria given in the second and third parameters.
<code>relationWrite ( aRelation )</code>	Write the specified relation to the output.
<code>tupleGetValue ( field, key )</code>	Extract the value associated with the given key from the field of a tuple specified as the first parameter.
<code>tupleProject ( tuple, field )</code>	Extract either the first or the second field—as specified by 1 or 2 in the field parameter—from the specified tuple.
<code>writeTuple ( tuple )</code>	Write the specified tuple to the output.

### 6.3 Example

To clarify the technique, I describe the use of the system to extract the “implicitly-invokes” [Garlan and Notkin 1991] source model between tools within the Field programming environment. Field tools communicate indirectly through a centralized message server. Tools express interest in events by registering regular expressions with the message server, and announce events by passing ASCII messages to the message server. The message server matches an incoming message to the stored regular expressions, and then it forwards the message to the appropriate tools. An understanding of the “implicitly-invokes” source model in Field may be useful for software engineers modifying an existing tool or integrating a new tool into the environment.

Event registrations and event announcements are coded as calls to C functions in Field. An examination of the source code comprising Field revealed that some of the C calls involved in event registration and announcement have, as a first parameter, a literal character string with the name of the event. Information about this coding style can be used to produce a static approximation of the dynamic interconnections of Field tools.

More precisely, Field tools register events through calls to the C function, *MSGregister*, and announce events using the C functions, *MSGsend*, *MSGsenda*, *MSGcall*, and *MSGcalla*. Patterns were written to scan the C source code for these functions with action code that output two relations to the intermediate stream: an event registration relation, and an event announcement relation. A generated analyzer was used to join these two relations on the event name to form a source model consisting of possible dynamic interconnections between Field tools.

*Scanning Event Information.* The specification to extract the registration and announcement of Field events from C source code is shown in Figure 6.4. Two nested patterns are defined. The first pattern matches K&R-style C function declarations. The second pattern matches calls within a function body that take a constant character string as a first argument. Action code is depicted within boxes in Figure 6.4. The point in

the pattern at which action code is attached is indicated with a grey dot.

The action code for the first pattern checks the name of the function matched and rejects any matches to control constructs (e.g., `if` statements, `forin` statements, etc.) by using the `fail` expression.<sup>6</sup>

Action code is also attached to two parts of the second pattern. The first body of action code (attached to  $\langle calledFuncName \rangle$ ) checks the name of the matched function call to ensure it is either an event registration or an event announcement.<sup>7</sup> Matches to any other function calls are rejected using the `fail` expression. The second piece of action code (attached to the end quote of the literal C string) writes a tuple of either the event registration (`registers`) or event announcement (`announces`) relation to the intermediate stream. The tuple includes the name of the tool, the names of the file and function being scanned, and the name of the event. The tool name is the name of the directory containing the file being scanned. The directory and file information are passed in as arguments to the scanner. The name of the event is taken from the first non-blank portion of the C literal string and is transformed into all lower-case letters by the call to the Icon `map` function.

For example, given the following snippet of C code from the *flowmenu.c* file in the *flow* directory, which handles messages for the Field call graph display tool:

```
void FLOW_menu_setup_trace(fw)
    FLOW_WIN fw;
    { ...MSGsenda("DDTR EVENT ADD %s * * 0 * * 0 CALL %1",fw->system);...}
```

the following tuple for the `announces` relation is output:<sup>8</sup>

---

<sup>6</sup>The `forin` construct is a macro used in the Field code.

<sup>7</sup>The Icon `find` function succeeds if the first parameter is a substring of the second parameter. This use of the `find` function to determine if the call is related to events may match other functions than the five MSG calls outlined above. The engineer makes this trade-off to ease the specification of the action code. Specifying all five calls in full string comparisons to the `calledFuncName` variable is another option that might increase an engineer's confidence in the actions of the scanner.

<sup>8</sup>For presentation purposes, this tuple is shown in a different format than output by the lexical source model extraction tools.

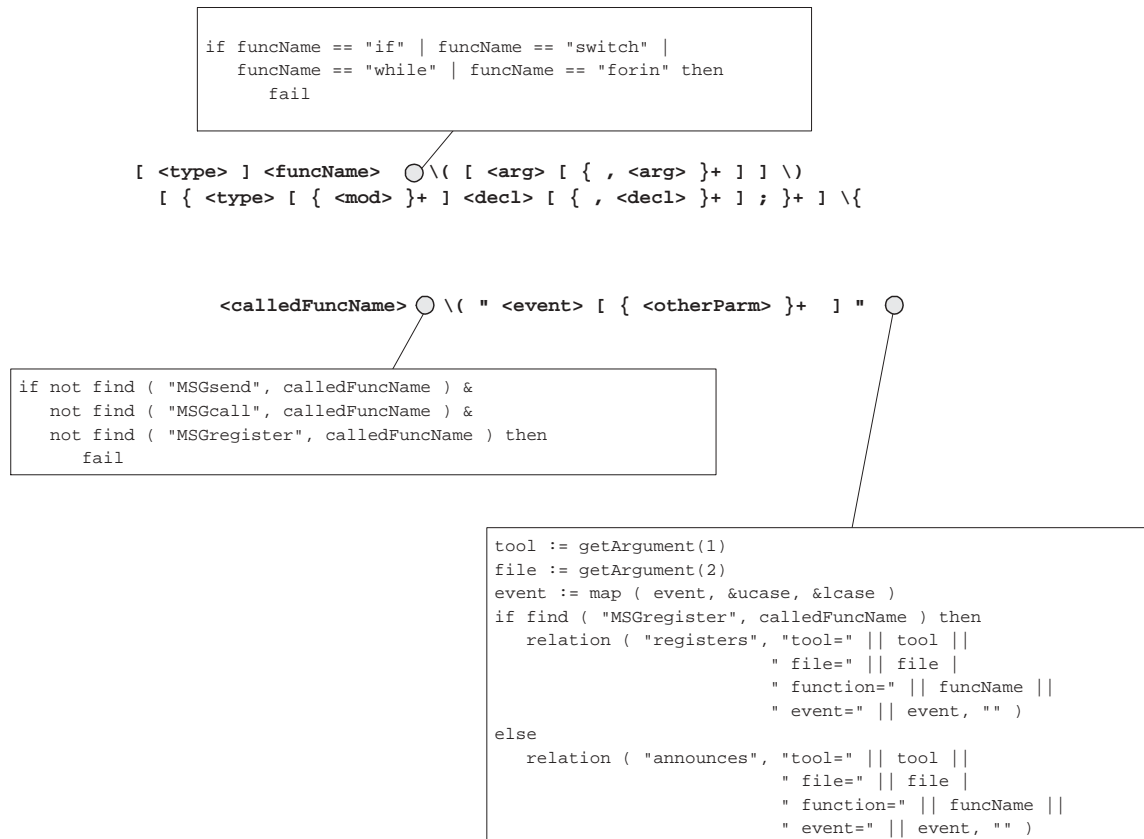


Figure 6.4: Patterns to Extract Events from Field Source Code.

```
tool=flow file=flowmenu.c function=FLOW_menu_setup_trace event=ddtr.
```

Scanning 160,000 lines of Field source code with the Icon program generated by the system takes about seven minutes on a DECStation 3000/300X running OSF/1 V3.2. The scanner generates 103 tuples for the `announces` relation and seven tuples for the `registers` relation. Seven tuples was a smaller number than I expected causing me to further inspect the Field source code. From this inspection, I determined that few tools used the *MSGregister* function with the event passed as a string. Instead, most tools called the registration function with a variable containing the name of the event. The values are generally set by reading an auxiliary data file containing blocks such as:

```
BUILD:
    SYSTEM_MSG = "DEBUG SYSTEM %1s"
    MAKE_MSG = "BUILD MAKE %1s"
    COMPILE_MSG = "BUILD COMPILE %1s"
    COMMAND_MSG = "BUILD COMMAND %1s %2s"
    COMPILEGO_MSG = "BUILD COMPGO %1s"
    FILEWD_MSG = "MSG FILE_WD"
    ;
```

Each block begins with the name of a tool followed by a colon. Several messages are then defined: each message is preceded by a message name, an equals sign, and a starting quote.

To extract event registrations from this structured data file, I wrote the specification shown in Figure 6.5. Each time a message definition is matched, action code is executed to transform the name of the tool and the name of the event to lower-case characters, and to output an event registration relation to the intermediate representation stream. Scanning the 181-line structured data file using the generated scanner takes less than a second, and generates 95 additional tuples for the `registers` relation.

`<tool> : { <name> = " <event> ○ [ { <otherParm> }+ ] }+`

```

tool := map ( tool, &ucase, &lcase )
event := map ( event, &ucase, &lcase )
relation ( "registers", "tool=" || tool ||
          " event=" || event, "" )

```

Figure 6.5: A Pattern to Extract Event Registrations from a Structured Data File. The data file is read at run-time by tools in the Field programming environment. The tools use data from the file to register events.

*Computing a Source Model.* To form the implicitly-invokes relation, I wrote the analysis specification shown in Figure 6.6. This specification creates an index for the `registers` relation, performs a relational join between the `announces` relation and the `registers` relation, and outputs the desired source model. A value in the source model describes a possible implicitly-invokes interaction between two Field tools, and also specifies a portion of the name of the event and the file and function from which the event announcement is made. For example, the source model value:

```
annot autoc annot autocbase.c handleMsg
```

describes that the `autoc` tool (second value) implicitly-invokes the `annot` tool (first value) through the `annot` event (third value), and that the announcement of the `annot` event is made from the `handleMsg` function in the `autocbase.c` file. Information about the location of the event registration is not provided because this information is not available for tuples extracted from the structured data file.

Analyzing the `announces` and `registers` relations extracted from the Field source code and the structured data file takes less than a second on the DEC Alpha. The computed source model consists of 61 event interactions between the 22 Field tools.

```

# This line is a comment line.

# Create an index for the registers relation.
eventList := ["event"]
relationMakeIndex ( "registers", 1, "event", eventList )

# Perform a join between the announces and registers relations
# based on the event name. tupleProject and tupleGetValue
# are functions that provide access to fields in the relation.

announcerR := relationSelect ( "announces", "", "" )
every tuple := !announcerR.records do {
  theEvent := tupleGetValue ( tupleProject ( tuple, 1 ), "event" )
  registerR := relationSelect ( "registers", "event=" || theEvent, "" )

  every rtuple := !registerR.records do {

    # Write out a value in the source model.

    registerValue := tupleProject ( rtuple, 1 )
    announceValue := tupleProject ( tuple, 1 )
    write ( tupleGetValue ( registerValue, "tool" ), " ",
            tupleGetValue ( announceValue, "tool" ), " ",
            theEvent, " ",
            tupleGetValue ( announceValue, "file" ), " ",
            tupleGetValue ( announceValue, "function" ) )
  }
}

```

Figure 6.6: Implicitly-Invokes Analysis Specification.

*Assessing the Source Model.* Determining the true implicitly-invokes relationship between Field tools is undecidable using static analysis techniques because Field allows events to be arbitrary strings that can be constructed at run-time. Unix's grep can be used to capture many of the invocations and registrations, but the quantity of information returned is great (380 lines), and data- and control-flow analysis would have to be performed to compute the relationship values. Performing even part of this analysis by hand would be, at best, a time-consuming activity.

Instead, I compared the extracted source model with one gleaned from reading the available literature and the man pages about Field. Because there are many ways of sending messages between tools beyond using the C functions with a constant character string parameter, the extractor missed some implicit invocations between tools. For example, the annot tool (the annotation editor) registers interest in some events announced by the cross-reference tool. The registration information about these events is stored in a second auxiliary data file (annot\_data.auxd). Since this data file was not scanned, the analyzer did not have enough information to determine the implicitly-invokes interaction between the cross-reference tool and the annotation editor. More events of this nature could have been determined by increasing the number of auxiliary data files scanned.

On the other hand, some interactions between tools that were automatically extracted were not found in a study of the documentation. For instance, the interaction between the auto-commenter tool (autoc) and the annotation editor tool (annot) was found by the extraction technique but is not readily apparent in the documentation.

No other source model extractions tools appear to easily extract this (or any similar) relation.

## 6.4 Using the Technique

In addition to the implicit-invocation example described above, the lexical source model extraction technique has been used to extract several different kinds of source models from a variety of types of system artifacts. Table 6.2 lists some of the uses of the

technique. The data in this table includes the number of lines of specification written by the engineer (the “Patterns & Actions” and “Analysis” columns) and the number of lines of supporting Icon code (the “Supporting Code” column).<sup>9</sup> Usually, less than 40 lines of specification and supporting code has been required. The specification and code required to support the extraction of call information from Eiffel [Meyer 1992] source code is somewhat larger because of the need to extract, track, and compute inheritance and type information. The technique has also been used to extract calls information from Modula-3 [Cardelli et al. 1989] source code as part of a design conformance task for the *SPIN* operating system development. This use of the technique is described in Section 9.3.

Table 6.2: Uses of the Lexical Source Model Extraction Technique.

Extractor	Patterns & Action Code (# Lines)	Analysis (# Lines)	Supporting Code (# Lines)	Total (# Lines)
CLOS Events	15	0	0	15
Eiffel Calls	110	15	94	219
Modula-3 Calls	15	20	0	35
TCL Globals	16	0	0	16

## 6.5 Producing Source Models for Computing Reflexion Models

The lexical source model extraction technique and tools may be used separately from the software reflexion model technique and tools. However, to ease the production of appropriately encoded source models for a reflexion model computation, special support has been built into the lexical source model extraction tools. Specifically, an engineer describes the hierarchical relationship between the lexical patterns using a source model entity naming tree; that is, an engineer embeds the patterns as annotations to the naming

---

<sup>9</sup>The supporting Icon code includes procedures to write out source model information, to compute inheritance hierarchies from relational information, etc.

tree. Figure 6.7 shows patterns for recognizing C calls embedded in a naming tree.

Furthermore, the keywords an engineer uses to express the structure of tuples to the special relational functions in the action and analysis code are defined by the source model entity naming tree in which the patterns are embedded. For example, the naming tree shown in Figure 6.7 defines four keywords: `directory`, `file`, `function`, and `call`. These keywords are available for use in the description of a tuple as a parameter to the `relation` function. This relationship between the naming tree and the keywords has the effect that tuples written out using the `relationWrite` function are encoded in the appropriate format for direct use by the reflexion model tools.

A final consequence of this interaction between the tools supporting each technique is that the same input file may be used for both tools. Using the same file ensures consistency between the encoding of the source model and the source model entity naming tree used in the reflexion model computation.

```

directory

directory.file

    directory.file.function
    %
    [(type)] functionName \([ { formalArg }+ ] \) [ { type } argDecl ; }+ ] \{
    %

    directory.file.function.call
    %
    (calledFunctionName) \([ { parm }+ ] \)
    %

```

Figure 6.7: Embedding Lexical Patterns in a Source Model Entity Naming Tree. The percent symbols denote the pattern attached to a naming tree entity. The first two entities in this tree (`directory` and `file`) have no attached patterns.

## 6.6 Summary

This chapter has described a lexically-based technique for extracting structural information from a wide variety of system artifacts. The specification language, as in other lexical systems, is based on regular expressions. In contrast to other lexical systems, the language for this lexical technique includes several features intended to ease the description of the source model to be extracted. First, the language simplifies the specification of hierarchically-related regular expressions, allowing, for example, an engineer to state that a regular expression describing a call construct may occur only after a match to a regular expression describing a function definition. These hierarchical regular expressions ease the description of syntactic constructs. Second, an engineer specifies a regular expression using tokens rather than characters. The use of tokens reduces the amount of information the engineer needs to define. Finally, like many other lexical approaches, the language permits an engineer to attach code to a regular expression; the code is executed when text from an artifact is matched to the expression. From the code, an engineer may access artifact text unified to different parts of the regular expression: this facilitates the production of the desired source model.

Similar to other lexical approaches, the technique is lightweight, flexible, and tolerant. The technique is lightweight in that specifications are easy to write and are typically small; specifications for C call graph extractors, event extractors for CLOS, implicitly-invokes extractors for Field, and global variable reference extractors for TCL, all have specifications of fewer than 25 lines. The technique is flexible in that it may be applied to many different kinds of system artifacts including source code and structured data files. Finally, the technique is tolerant in that few constraints are placed on the condition of the artifacts; the same pattern, for example, has been used to extract C call graphs from both pre-processed and post-processed source code.

In the next chapter, I describe the tools developed to support this technique and compare the performance of some instances of generated source model extractors with

existing parser-based tools. The accuracy of the approach is discussed in further detail in Chapter 8.

## Chapter 7

# The Lexical Source Model Extraction Tools

The lexical source model extraction technique is supported by two tools: a scanner generator, and an analyzer generator. In this chapter, I describe these tools and discuss how scanners and analyzers generated with these tools operate. The performance of two instances of C [Kernighan and Ritchie 1978] call graph extractors generated with the lexical source model extraction tools are then compared to some existing parser-based approaches.

### 7.1 The Generated Tools

Given a specification for a source model, the lexical source model extraction system generates two tools.<sup>1</sup> The first tool, a scanner, produces either an intermediate relational representation stream or a source model when given an artifact. The optional second tool, an analyzer, produces a source model from an intermediate stream.

---

<sup>1</sup>This dissertation describes version 1.5 (Build 13) of the lexical source model extraction tools. The tools are implemented in C++ [Stroustrup 1986] and are built upon the classes comprising the software reflexion model tools. The lexical source model extraction source consists of 16 unique classes comprising approximately 5,600 lines of code. The tools generate code for Version 9 of Icon. All performance tests reported on in this dissertation were run using the Version 9 Icon interpreter and compiler distributed from the University of Arizona.

### 7.1.1 Scanner

Based on the specification provided by the engineer, the scanner generator produces a description of a lexer and a description of a set of hierarchically-related deterministic finite state machines. These descriptions are combined into an Icon [Griswold and Griswold 1983] program that executes the finite state machines on input that is tokenized by the lexer.

Each pattern in a specification is translated into a separate deterministic finite state machine. Figure 7.1 shows the pattern for locating C [Kernighan and Ritchie 1978] function definitions from Chapter 6, and the state machine generated for that pattern. Each generated state machine has an initial state (marked by an oval in the figure) and one or more final states (marked by the diamond-shaped node in the figure). The generated finite state machines are related in the same hierarchy as the patterns from which they were generated.

Each transition in a state machine has an associated value indicating the kinds of tokens on which the transition can be taken. Transitions are represented by edges in Figure 7.1; the labels on the edges represent the transition values. The transition values are either single-character tokens or **identifiers**. A single-character token matches only that token in the input stream. An **identifier** matches any identifier returned by the lexer and also any single-character token that does not appear in the pattern from which the machine was generated.

The generated scanner program is responsible for both executing the appropriate state machines as tokens are produced, and recording the paths taken through the machines. At the start of the scanning process, only the state machines at the top of the multi-rooted tree are active. As tokens are produced by the lexer, transitions are taken, in parallel, in all active machines. On each token, a new path is also started from the initial state in all active machines.

The scanner always attempts to match the longest possible sequence of tokens. When

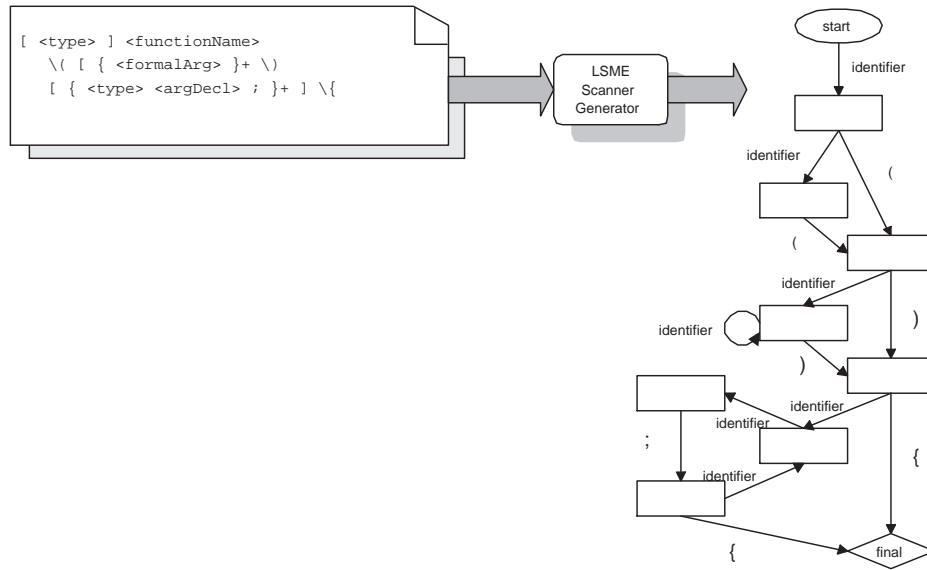


Figure 7.1: Generated Deterministic Finite State Machine for a Pattern.

a path reaches a final state, the scanner continues execution of all paths of equal or greater length in machines at the same or higher level of the hierarchy to determine if a longer match is possible. Scanning continues as long as paths grow.

Since multiple state machines are active concurrently, a final state may be reached by more than one path simultaneously. The scanner must choose one path to *reduce*. Reducing a path results in the unification of tokens to variables named in the matched pattern, and in the execution of action code. Two heuristics guide the choice of the path to reduce:

1. If more than one path has reached a final state, and the paths are in different state machines, reduce the path in the machine with the lowest number in a breadth-first ordering of the machines. The machines are ordered based on the definition of the associated patterns in the specification. This heuristic enables the scanner to reset. In the C calls example, the start of the definition of a new function resets the search from looking for calls to looking for functions. The heuristic also permits engineers to prioritize patterns at the same level of the hierarchy; patterns closer

to the top of the specification have a higher priority.

2. If more than one path reaches a final state at the same hierarchical level, reduce the path with the most matched identifiers and single-character tokens. This heuristic generally selects the most specific pattern.

If, while executing action code as part of reducing, a **fail** expression is encountered, the scanner halts the reduction and looks for another possible path to reduce. If no other path is ready to reduce, the scanner backtracks and continues the execution of the existing paths. After a successful reduction occurs, scanning proceeds by pruning all existing paths, by restarting all machines at the same or higher level of the hierarchy than the machine of the reduced path, and by restarting all machines that are children of the machine of the reduced path.

Active paths are pruned when no transition is possible with the current token. It is still possible, however, that the search space may explode if the patterns are not sufficiently specific. To bound the search space, a third heuristic is encoded into the scanner, pruning a path if more than a fixed number of tokens have been matched. The default value of the number of tokens that may be matched is 100. An engineer may specify a different value for this heuristic when a scanner is generated.

*Translating Patterns to State Machines.* Ensuring that the appropriate variables are unified with consumed tokens and that the appropriate action code is executed when a path reduces requires the maintenance of additional information linking a pattern to its deterministic state machine. The correct linkage is ensured by first performing a straightforward translation from a given pattern (i.e., regular expression) into a non-deterministic finite state machine with  $\epsilon$  moves. Transitions in the generated non-deterministic machine are labeled with either an  $\epsilon$ , a single-character token, or an **identifier**. For an **identifier** label, the appropriate variable to unify with a consumed token is also maintained on the transition. Action code is associated with non- $\epsilon$  transitions. The non-

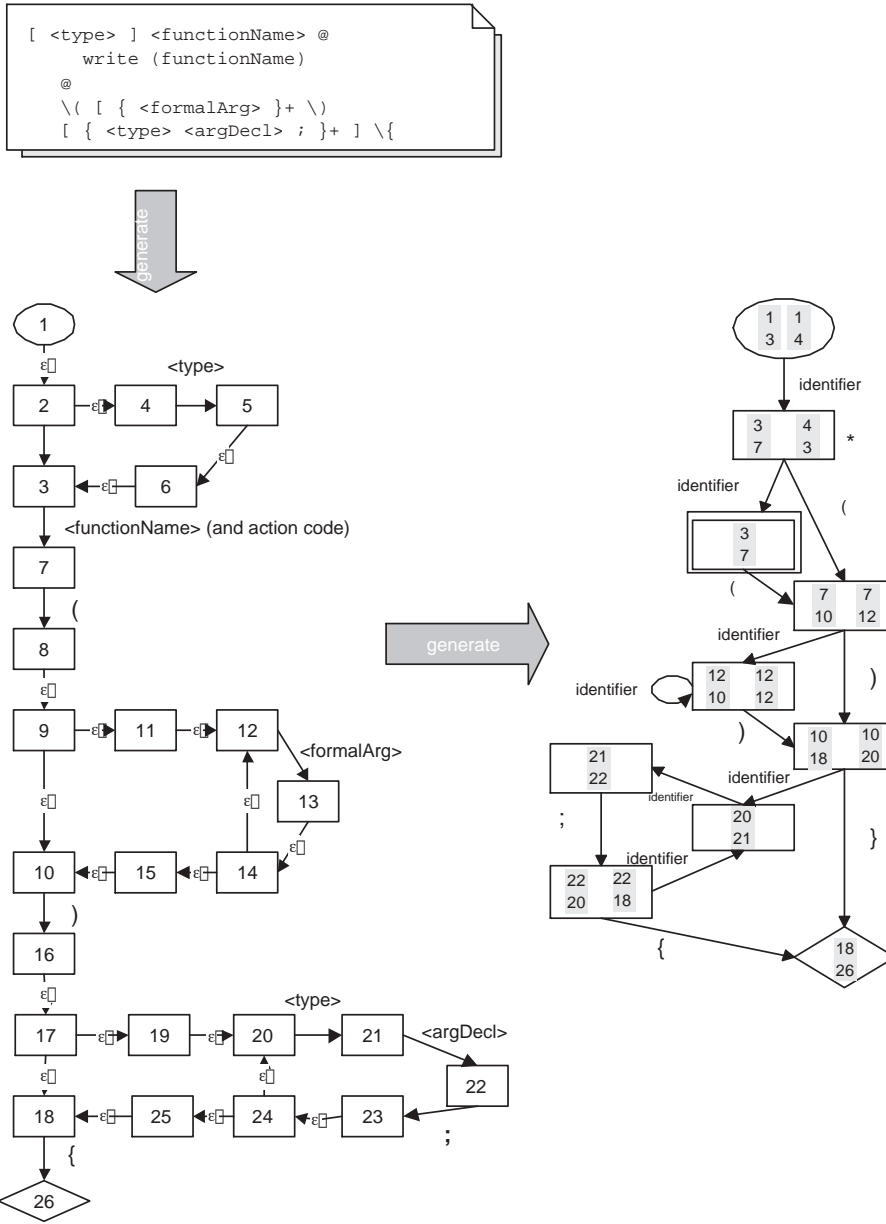
deterministic machine for the pattern to recognize C function definitions (with action code to write out the name of recognized functions) is shown in Figure 7.2.

The production of the deterministic machine from the non-deterministic machine follows the standard algorithm [Rabin and Scott 1959; Aho et al. 1986, p. 117] with one variation. In the standard algorithm, each state in the deterministic machine corresponds to a set of states in the non-deterministic machine. In the modified algorithm, each state in the deterministic machine corresponds to a set of pairs of non-deterministic machine states. These pairs encode path information from the non-deterministic machine into the deterministic machine. For example, consider the nodes of the deterministic machine for matching function definitions (also shown in Figure 7.2). Each deterministic machine node is annotated with a set of pairs (shown vertically) of non-deterministic machine states. The top element of each pair names a state in the non-deterministic machine that may transition to the non-deterministic machine state named in the bottom element of the pair when a token of the kind named on the input arc of the deterministic machine state is seen. For example, the second state of the deterministic machine (a double box in the figure) contains the pairs:

3	4
7	3

indicating that from state 3, if an identifier—the kind of token attached to the input arc—is returned by the tokenizer, the non-deterministic machine may transition to state 7, and from state 4 may transition to state 3.

As described earlier, the scanner maintains path information as tokens are consumed and deterministic machines are executed. The path information that is maintained is really the path information in the non-deterministic machine. As a transition in the deterministic machine is taken, the scanner determines the possible corresponding paths in the non-deterministic machine by matching the bottom element of each pair in the



(a) Non-deterministic state machine

(b) Deterministic state machine

Figure 7.2: Generated Deterministic and Non-Deterministic Finite State Machines.

current deterministic state with the top element of each pair in the new deterministic state. For example, if the deterministic machine shown in Figure 7.2 is in the second state, there are at least two active paths (each path is shown in []): [(1 3) (3 7)] and [(1 4) (4 3)]. If an identifier is then seen, the [(1 4) (4 3)] path is matched with the (3 7) pair, forming the path ((1 4) (4 3) (3 7)). The other path is pruned from consideration since there is no pair that matches to state 7 in the new deterministic state.

When a deterministic machine is reduced, the path from the non-deterministic machine is traversed, the variables are unified with consumed tokens, and the action code is executed.

### 7.1.2 Analyzer

The analyzer generator translates the analysis code from the source model specification file into an Icon program that reads tuples from the intermediate representation stream, forms the relations as defined by those tuples, and then performs the relational operations and processing specified by the analysis code.

## 7.2 Performance

The scanner and analyzer generators are fast; the generators complete execution in a few seconds on a Sparc 20/50 for all specifications that have been written. Although theoretically the conversion a generator performs from the non-deterministic state machines to the deterministic state machines could result in an exponential number of states in the deterministic machines, this has not been a problem in practice. For example, there are less than 20 states in the deterministic machines generated for the Field [Reiss 1990; Reiss 1995] event specification described in Chapter 6. One of the largest examples, a call graph extractor for Eiffel code, which consists of four patterns with 234 symbols (e.g., regular expression characters, variable names, single-character tokens, etc.), translates into a set of deterministic machines with a total of 87 states.

To support the iterative approach, it is also important that the generated scanners be efficient. While developing patterns, the Icon interpreter is often used to test the generated scanners on sample files. Interpreting the scanner generated for extracting Field events over an 840 line C file from Field takes 20 seconds on a Sparc 20/50. This interpretation speed is fast enough to support the iterative refinement of a specification.

Once the specification has been refined to extract the desired information, the generated Icon scanner program can be compiled for better efficiency. To give a sense of the speed of the compiled Icon, Table 7.1 shows a comparison of the time required to perform the basic operation of breaking an input stream into tokens with a scanner generated using the system, and a similar tokenizer generated using lex [Lesk 1975]. These lexers recognized the start and end of C-style comments, specific one-character tokens (i.e., opening and closing parentheses, a comma, and an opening curly bracket), and identifiers consisting of non-white space sequences of characters other than the specified one-character tokens. The performance data reported is based on executing each lexer on the 215 C source files comprising the source for the Field programming environment. The lexer generated with the lexical source model extraction technique's scanner generator (LSME lexer) is able to process approximately 103K lines per minute, compared to the 186K lines by lex.

Table 7.1: Comparing the Speed of Tokenizing an Input Stream. The performance was measured on a Sparc 20/50 running SunOS 4.1.3\_U1.

Tool	Scan Time (min)	Lines Scanned Per Minute
lex	:54	186K
LSME lexer	1:37	103K

The execution time of a generated scanner is heavily dependent upon the patterns in a specification and the content of the artifacts being scanned since these two factors affect the size of the search space explored. To give a sense of the performance of the

scanners, Table 7.2 compares the time required to run two different lightweight scanners for extracting C calls from the Field system source code with the execution times of the Field and CIA [Chen et al. 1990; Chen 1995] calls extractors on a Sparc 20/50. The first scanner (labeled LSME—no cpp) was generated from a specification containing two patterns (see Section C.2), and was run over the approximately 167,000 lines of preprocessed C source code comprising Field. The second scanner (labeled LSME—cpp) was generated from a specification containing three patterns (see Section C.3), and was run over the approximately 985,000 lines of postprocessed source code (similar to the Field and CIA tools). The data reported include the CPU time<sup>2</sup> to produce a “function calls function” source model,<sup>3</sup> the wall clock time,<sup>4</sup> and a calculation of the number of lines of source scanned per minute. The data in the table shows the dependence of the scanners on the specification and artifact; the second scanner, for instance, has a higher scanning rate than the first scanner although it encodes more patterns and scans more lines of code. Although the second scanner runs more slowly than the CIA and Field tools, its performance is adequate for supporting an engineer in most maintenance tasks.

Similar to speed, the space required by a generated scanner during execution depends on the patterns specified by an engineer, the artifacts being scanned, and the value of the third heuristic. To measure the space used by a generated scanner at execution, I altered the code of the two generated scanners to report, using the Unix `ps` command, the size of the process at the completion of a scan. These scanners each had the default third

---

<sup>2</sup>The CPU time reported is a sum of the user and system seconds required to produce the source model as reported by the Unix `time` command. For the CIA tool, the time reported includes the time to scan each source file (using the `cia` command), combine the results of the scans into a database (also using the `cia` command), and then query the database for the source model (using the `cref`) command. For the Field tool, the time reported includes the time to scan the source files and to then query the resultant database for the desired source model (using the `xrefdb` command in each case). The time reported for the LSME scanner generated from two patterns includes the time to scan the individual source files comprising the Field code base, as well as the time required to postprocess the resultant source model with `awk` to expand some common macros for ease of comparison to CIA. The time reported for the LSME generated scanner from three patterns is simply the time to scan the individual Field source files.

<sup>3</sup>The CIA tool actually extracts a “function refers-to function” source model. This source model is a superset of the “function calls function” source model.

<sup>4</sup>As reported by the Unix `time` command.

heuristic value of 100. The average space used across the scans of the 251 Field source files was modest; 237Kb was the average space used by the scanner generated from two patterns, and 409Kb was the average space used by the scanner generated from three patterns. The maximum space required by either of these two scanners was 2252Kb.

The performance of a source model extractor cannot be considered independent of the information it extracts. Table 7.2 also reports a comparison of the similarity between the source models extracted to the source model extracted by the CIA tool. As will be discussed in more detail in Section 8.3, the degree of approximation acceptable in a source model depends on the task being performed. For instance, when a software engineer needs to build an understanding of unfamiliar C source as was the case with the NetBSD estimation task described in Chapter 2, a source model consisting of over 90% of the calls may provide a sufficient source model.

The analyzers generated from the specifications perform sufficiently fast for the analysis specifications that have been written. For example, as described earlier, the generated analyzer for the Field events specification completes execution in less than a second on a Sparc 20/50 when given the tuples extracted by the generated scanner. When better performance or more sophisticated relational operations are required, other analyzers, such as commercial relational databases, may be used.

Table 7.2: A Comparison of Various Tools that Extract C Calls Information. The sample source was the Field source code. The performance was measured on a Sparc 20/50 running SunOS 4.1.3\_U1.

Tool	CPU Time (min)	Wall Clock Time (min)	Lines Scanned Per Minute	% Calls Same as CIA
CIA	7:22	15:44	133.2K	100
LSME (2 patterns, no cpp)	7:34	8:51	22.5K	85
Field	9:02	17:11	109.1K	97
LSME (3 patterns, cpp)	21:10	23:31	46.6K	92

### 7.3 Summary

The scanning of structural information from system artifacts using the lexical source model extraction technique is based on the specification of hierarchical regular expressions with attached action code and the later generation and execution of a hierarchy of deterministic finite state machines. The matching of text from an artifact to the specified regular expressions is controlled by the structure of the specification, heuristics, and the action code attached to the regular expressions. Structural information resulting from one or more scans of system artifacts may be combined using a generated analyzer.

Assessing the performance of a generator-based system is difficult. To give some sense of the performance of the system, I compared C call graph extractors generated with the tools to existing parser-based tools that extract a similar source model. Although the C call graph extractor tools generated using the lexical source model extraction system perform somewhat slower than comparable parser-based tools, the performance is sufficient to support the use of the technique on large systems. This performance is promising since the purpose of the system is not to supplant existing parser tools, but rather to take advantage of the benefits the system provides in being able to extract novel structural source models from possibly uncompileable source code and other system artifacts.

## Chapter 8

# A Discussion of the Lexical Source Model Extraction Technique

The lexical source model extraction technique exemplifies a point in the design space of source model extractor tools somewhere between existing lexical tools like `awk` [Lesk 1975] and existing parser-based tools like `yacc` [Johnson 1975]. In this chapter, I describe the trade-offs resulting from this point in the design space and discuss some of the choices made in the engineering of the tools. The chapter begins with a discussion of the expressiveness of the pattern language. I then consider issues surrounding the accuracy of the lexical approach and discuss the role of the heuristics embedded in the generated scanners. The chapter concludes with a description of some architectural trade-offs made in the design of the system compared to other similar tools.

### 8.1 Expressiveness

The pattern specification language is equivalent to regular expressions. For instance, each path in the pattern tree may be used to form a regular expression by joining parents to children with wildcard expressions, and by adding the appropriate iteration and alternation information. For example, the two patterns to search for call information

in C [Kernighan and Ritchie 1978] code shown in Chapter 6 may be joined by:

```
( functionPattern ( ( wildCardPattern )* callPattern )* )*
```

where `functionPattern` is the pattern to recognize a C function definition, `wildCardPattern` matches any characters and white space, `callPattern` is the pattern to recognize a C call, and the parentheses with a star enclose a regular expression that occurs zero or more times.

In contrast to other existing regular expression tools, the lexical source model extraction technique combines several features to simplify the specification and use of regular expressions for recognizing language features.

First, the input is simplified by deriving a lexer from the specified pattern. The engineer, for instance, does not need to specify a character-based regular expression pattern to represent identifiers, numbers, and such for the particular language of interest. One limitation of this approach is that it reduces the control an engineer has of the tokenization process, sometimes causing unexpected results. For example, a scanner generated from a pattern consisting of only a string may not match all occurrences of the string in a given piece of text. Rather, only those pieces of text in which the string is preceded by white space will be matched. In general, the benefit of easing the difficulty and reducing the time required to specify a desired source model generally outweighs the limitations associated with an implicit lexer. To handle cases in which more control is necessary, the system could be augmented with the ability to accept an explicit lexer defined by the engineer.

Second, allowing patterns to be defined hierarchically permits the engineer to easily refine and augment a collection of existing patterns. The ability to refine and augment patterns is important given the iterative nature of the technique. New patterns can be introduced into the hierarchy without modifying existing patterns; this is in contrast to existing lexical tools in which the current set of regular expressions must be modified and augmented (i.e., with iteration information) as new hierarchical information is

introduced.

Finally, the heuristics built into the generated scanners restrict the number of matches of the patterns to the text of an artifact. To the engineer, this more closely approximates the behaviour found when searching based on a grammar. For instance, the heuristic which prefers matching a pattern closer to the top of the hierarchy reduces the amount of code than an engineer must otherwise write to remove unwanted matches of regular expression patterns to artifact text.

In contrast to most parser-based tools, the pattern language of the lexical source model extraction technique is based on the concrete syntax of the language of the artifact rather than on an abstract syntax for the language.<sup>1</sup> Basing the patterns on the concrete syntax has the disadvantage that the expression of the source model of interest may be longer than that required for a tool that uses an abstract syntax representation. For example, Griswold and colleagues outline scripts for extracting call graph information from C code in a variety of lexically- and syntactically-based tools [Griswold et al. 1996]. In one case described, a script for the lexical source model extraction technique is 35 lines in length, while a script for Griswold's parser-based tawk tool [Griswold et al. 1996] is four lines:

```
[expression:FUNCTION:$fcall] {
    printf ("%s: %s\n", FunctionName($fcall),
           CallName($fcall));
}
```

This tawk script is small and declarative, in part, because it matches only on calls. Information related to the name of the containing function or macro is accessed via

---

<sup>1</sup>Not all parser-based approaches require an engineer to query against the abstract syntax representation of a program. For example, engineers using the parser-based Scruple tool [Paul and Prakash 1992] express the constructs based on the concrete syntax of the language of the artifact of interest; the matches, however, are performed on an abstract syntax tree created for the artifact. A disadvantage of the approach taken in Scruple is a requirement for the engineer to express all concrete syntax that appear in between constructs of interest (i.e., the engineer must explicitly provide wildcard constructs to describe the constructs that may appear between a function definition, a call site and the end of the function definition).

other nodes in the abstract syntax tree created by the tool when scanning the C source code. While this script is short, a disadvantage of the abstract syntax approach is the need for the engineer to understand the abstract syntax for the artifact of interest, and to understand how the constructs of interest in the artifact are mapped to the syntax.

Whether it is better to describe a construct embedded in system artifacts in terms of concrete or abstract syntax depends on a number of factors. For instance, is an abstract syntax form available for the language of the artifact? Is the abstract syntax form unique? Can the artifact be successfully translated into an abstract form? This variety indicates the range of the design space for extractor tools. One of the ramifications of basing the pattern language for the lexical source model extraction technique on regular expressions and, in essence, the concrete syntax of the language of the artifact of interest is that the pattern language is not well suited to extracting fine-grained, statement-oriented source models like control dependence graphs. The language is, however, sufficiently expressive to have been used to extract a number of different kinds of source models from a variety of types of system artifacts as outlined in Section 6.4.

## 8.2 Accuracy

The accuracy of a source model extracted using the system depends primarily upon three factors: the structure of the language used within an artifact (e.g., ANSI C, structured data files, etc.), the adherence to style within an artifact (e.g., limited nesting of calls, standardized placement of the dereferencing operator in C, etc.), and the effort spent in refining a specification. For example, a global variable references source model extracted from over 7000 lines of TCL [Ousterhout 1994] code is exact because a particular global variable construct, suitable for extraction with regular expressions, was used uniformly throughout the program. On the other hand, it is impossible to write a specification that will extract all calls from any C program because the regular expression based language limits the ability to express nested call constructs. Coding styles, however, can be exploited to decrease the number of calls missed (false negatives) by an extractor

generated with the system. For instance, if only two levels of embedded calls are allowed, regular expressions may be written to express the limited embedding.

One difficulty an engineer faces is the determination of accuracy of the extracted source model. This is not unique to this technique. An empirical study of four syntactically-based C call graph extractors showed that all tools generated some false positives (source model values reported that are not part of the “true” source model) and all tools generated some false negatives (calls that do occur but are not reported as tuples in the source model) [Murphy et al. 1996]. Understanding the sources of approximation in these syntactic approaches, as well as in this lexical technique, often requires careful analysis by the engineer. However, in contrast to most syntactically-based approaches, the iterative and lightweight nature of this technique enables an engineer to more easily improve the accuracy of the source model in a number of ways including, among others, refining the patterns in a specification or using multiple passes to extract different information that is then combined during analysis. For instance, by extracting calls from both pre-processed and post-processed Field system source code [Reiss 1990; Reiss 1995] and by expanding some macros during analysis, it was possible to extract 95% of the calls extracted for the same source by the CIA system [Chen et al. 1990; Chen 1995] and 96% of the calls extracted by the Field system.<sup>2</sup>

Approximate source models are useful for some, but not all, software engineering tasks. When I wanted to compute a reflexion model for a system implemented in Eiffel [Meyer 1992], there was no publicly or commercially available Eiffel call graph extractor; building a good, even if approximate, extractor in a couple of hours was beneficial for helping to understand an existing system written in Eiffel. Griswold and Atkinson have demonstrated with their Ponder system [Griswold and Atkinson 1995] that approximate control-flow information can be useful for constructing call graph displays used by engineers modifying an existing health care system. For other tasks, such as determining

---

<sup>2</sup>In comparison, the Field system found 97% of the calls found by CIA. Almost 3% of the calls found by Field, were not reported by CIA.

branch coverage for testing, or computing a graph of program dependences for automating restructuring tasks, approximate models may not be sufficient, and this technique may not be applicable. In many cases, though, the flexibility an engineer gains in being able to extract a desired source model is a fair trade for accuracy.

### 8.3 Heuristics

The heuristics embedded in the generated scanners tend to reduce the number of false positives that might otherwise be reported. Reducing the false positives often causes a corresponding decrease in false negatives. For instance, the heuristic that selects the pattern closest to the top of the hierarchy ensures that the C call graph extractor resets when a new function definition is seen; this may eliminate a false positive that would result from considering the function definition as a call from a previously defined function, as well as eliminating the false negatives that would arise from not considering the construct as a function definition when subsequent calls are matched.

The impact of the heuristics on the behaviour of a generated scanner is dependent upon the patterns specified by an engineer. To investigate the impact of the heuristics, I gathered statistics on the execution of scanners generated for three different specifications: the Field event specification shown in Figure 6.4, the Field structured data specification shown in Figure 6.5, and a C calls specification based on the patterns in Section 6.2.1. The first scanner was executed on the pre-processed C, yacc, and lex [Lesk 1975] source code comprising the Field system; the second scanner was executed on a 181-line Field structured data file; the third scanner was executed on a publicly available molecular biology application, Mapmaker,<sup>3</sup> comprised of approximately 46,000 lines of C code. Table 8.1 summarizes the data collected from the execution of these scanners. The data in the table includes statistics on the number of average and total times each heuristic was invoked, and the average and maximum number of active paths.<sup>4</sup>

---

<sup>3</sup>Version 3.0 of the MAPMAKER/EXP source code and version 1.1 of MAPMAKER/QTL source code were scanned.

<sup>4</sup>The average values were calculated by taking the average over all (average) values returned from the

The number of times heuristics were invoked during the scanning for Field events is small because the Field event specification is comprised of two clearly distinct patterns. In contrast, the C calls extractor is comprised of two similar patterns, resulting in higher values of heuristic invocations. The single pattern in the Field structured data specification ends in a repetitive sequence, causing several invocations of the second heuristic that chooses the longest possible match.

Table 8.1: Execution Statistics for Generated Scanners.

Specification	Heuristic 1 Hierarchy Avg (Total)	Heuristic 2 Longest Avg (Total)	Heuristic 3 Token Bound Avg (Total)	Number of Paths Avg (Max)
Field Events	0 (0)	0 (0)	0 (3)	3 (4)
Field Structured Data	0 (0)	15 (15)	2 (2)	1 (3)
C Calls	50 (2439)	57 (2783)	0 (0)	3 (12)

## 8.4 Engineering Trade-offs

An alternative approach to source model extraction is exemplified by software information systems, such as Masterscope [Teitelman and Masinter 1981], Omega [Linton 1983], cscope [Steffen 1985], CIA [Chen et al. 1990; Chen 1995], Field [Reiss 1990; Reiss 1995], LaSSIE [Devanbu et al. 1991], XL C++ Browser [Javey et al. 1992], and Sniff [Bischofberger 1992]. These systems derive a database from the target system's artifacts; desired source models are then extracted from the database. In contrast, with the lexical source model extraction technique, an engineer produces a separate source model extractor for each desired source model. This difference is in part a matter of engineering. To understand in which situations one approach may be better than the other requires consideration of a number of dimensions.

In the database approach, one must anticipate what information to include in the scan of each file.

database. If a new source model is needed that depends on information not in the database, the database structure must generally be modified, the tool that creates the database must be modified, and the tools that extract existing source models from the database may have to be modified. The lexical source model extraction approach is not dependent on anticipating these needs.

The lexical source model extraction technique of computing source models on demand, however, may be less effective if a number of source models need to be extracted from the same source code, since scanning is, in general, performed for each desired model. In contrast, the conventional approach amortizes scanning costs; once the database is computed, it is often inexpensive to extract source models from the database.

## 8.5 Summary

In this chapter, I have discussed various issues related to the design of the technique. My intent in designing the technique was to enable an engineer to more easily extract structural information that is present in software artifacts but that is not often considered when reasoning about a system because of the difficulty of accessing the information. The technique thus serves supplements existing techniques for extracting information from system artifacts. The examples of uses of the technique described in Chapter 6 provide evidence that the technique achieves this goal of broadening the information easily available to an engineer. This question is also considered in more depth in the next chapter which discusses validation efforts related to both the software reflexion model and the lexical source model extraction techniques.

## Chapter 9

# Validation

Are software reflexion models effective in helping an engineer assess, plan, or execute change tasks on large systems? Does the lexical source model extraction approach extract source models that are useful to a software engineer during a change task? Is partial and approximate structural information serviceable for supporting reasoning about change?

Answers to these questions are necessary to validate the claims made in this dissertation. However, as with most techniques aimed at improving parts of the software development process, it is not possible to answer these questions for the general case. Rather, I use empirical evidence from a set of exploratory case studies to consider whether there is support for the claims. As defined by Yin, “a case study is an empirical inquiry that:

- investigates a contemporary phenomenon within its real-life context; when
- the boundaries between phenomenon and context are not clearly evident; and in which
- multiple sources of evidence are used” [Yin 1984, p. 23].

The case studies chosen span several kinds of change tasks on systems of varying size implemented in several different kinds of programming languages. These variations are meant to build confidence that the techniques may generalize to an interesting set of software development scenarios.

Three case studies are described in-depth, beginning with a description of the use of the reflexion model technique by a software engineer at Microsoft Corporation to support an experimental reengineering of the Excel spreadsheet product. This case study provides evidence to support the claim that the software reflexion model technique is effective in helping an engineer assess and plan a change task on an existing system, and shows that the technique scales to use on a large (million lines of C [Kernighan and Ritchie 1978] code) system.

Next, I describe my use of the reflexion model technique to assess the conformance of a program restructuring tool implemented in CLOS [Bobrow et al. 1989] to a previously prepared software architecture. This case provides evidence to support the claim that the software reflexion model technique aids in the execution of a change task by enabling a conformance check between a system's design and its implementation.

Third, the use of both the reflexion model technique and the lexical source model extraction technique to promote an understanding of the structure of the *SPIN* operating system [Bershad et al. 1995] before a change is described. This case study provides evidence that approximate structural information can be useful for reasoning about change.

In addition to these in-depth accounts, condensed descriptions of two other case studies are provided at the end of the chapter.

The presentation of each case study includes a description of the task, a description of the environment and process under which the study was performed, and a description of the results. In each case, multiple sources of evidence were collected. The results of each study have also been discussed with the participants.

## 9.1 Excel: A Case Study in Assessing and Planning Change

A Microsoft software engineer used the reflexion model technique and tools to help assess and plan an experimental reengineering of the Excel spreadsheet product.<sup>1</sup> The

---

<sup>1</sup>For background on the software development process at Microsoft Corporation, and in particular, the history of the Excel development, the reader is referred to the chronicle of Cusumano and Selby [Cusumano and Selby 1995].

experimental reengineering involved the identification and extraction of components from the existing system.

The Excel system comprises over 1.2 million lines of C code spread over more than 15,000 functions that are divided amongst almost 400 files. To help support the task, the Microsoft engineer was interested in understanding how Excel's source was divided into modules and in how those modules interacted at execution. In particular, the engineer was interested in calls made between functions belonging to different modules and in references by functions to global variables.

### 9.1.1 The Environment

Traditionally, an engineer at Microsoft would become familiar with these aspects of Excel's static structure by reading an "Internals" document, by oral tradition, and by studying the source code. This process is described by Jon De Vaan:

"Excel Internals...explains the philosophy of a few of the basic things in Excel, like the cell table formulas, memory allocation, a little bit about the layer [a special interface with the operating system that allows Microsoft to use the same Excel core on both Windows and Macintosh platforms; ...]. It's very sparse. We don't necessarily rely on that for people to learn things. I'd say we have a strong oral tradition, and the idea is that the mentor teaches people or people learn it themselves by reading code.... Over the course of a project, it goes from mostly truthful to less truthful, and then we have to fix it up. We don't fix it up as we go along on a project. We will give it some attention between projects." [Cusumano and Selby 1995, p. 109]

Although this approach may be effective for engineers during the development and evolution of a software product at Microsoft, it was not appropriate for the engineer performing the experimental reengineering activity for two reasons. First, the engineer performing the experimental reengineering activity belonged to a separate group from the Excel development group and thus could neither rely heavily on existing knowledge nor on consistent interaction with a mentor from the development group. Second, the experimental reengineering activity was to be performed within a specific time period.

The engineer—a developer with over ten years of experience at Microsoft—estimated that the use of existing approaches described above to gain the needed familiarity with Excel might take as much as two years; the reengineering activity, though, was to be performed within a few months.

From a presentation given at Microsoft, the engineer was introduced to the software reflexion model technique and he asked to apply it to try to reduce the time needed to perform the desired experimental reengineering activity. I installed the reflexion model tools at Microsoft and the engineer applied the technique, over a four week period, with only occasional e-mail, telephone, and personal contact. I gathered information about the engineer’s use of the technique in four ways: through e-mail exchanges, through phone conversations, through two site visits, and through snapshots of the input files to the reflexion model tools he sent periodically.

### **9.1.2 The Process**

The description of the process by which the Microsoft engineer used the technique is broken into two parts. First, the computation of the initial reflexion model is discussed. Then, the process used to refine a series of reflexion model for Excel is described.

#### **Computing an Initial Reflexion Model**

Based on some brief discussions with the Excel developers and on previous experience, the Microsoft engineer found it “natural” to state an initial high-level model that described some of the modules comprising the Excel system and the call dependences between those modules (Figure 9.1). This model consisted of 13 nodes and 19 interactions. To protect Microsoft proprietary information contained in this model, the names of all but three modules have been removed. The qualitative examples presented in this case study focus on the interactions between the three named modules; reported statistics consider the full reflexion models that were computed.

Since the high-level model of Excel captured calls between modules, the Microsoft

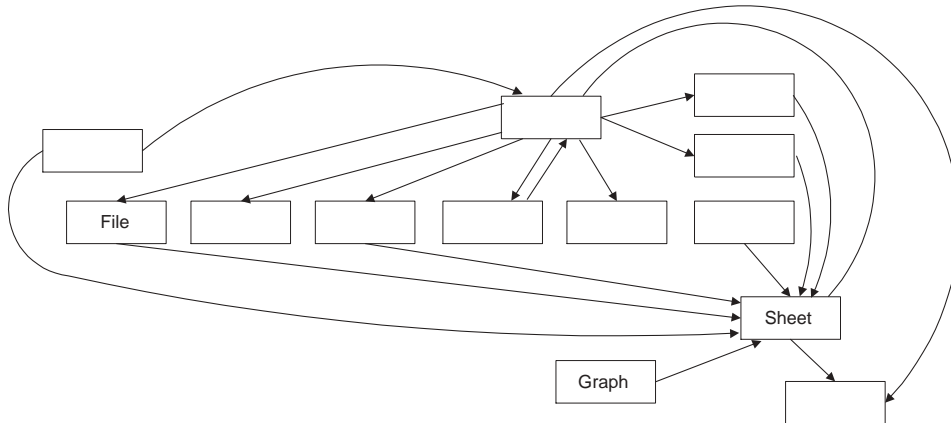


Figure 9.1: A High-Level Model of Excel. The boxes represent modules. The arcs represent dependencies between modules. The names of all but three of the modules have been removed to protect proprietary information.

engineer selected calls between functions as the basis for a compatible source model. An internal Microsoft tool was used to extract this source model from the Excel source code. The extracted source model consisted of 77,746 calls and included information about the directory and files in which the functions were physically placed. The calls information contained in this source model was approximate as the information included neither calls through function pointers, nor calls made in data initialization code.

The initial map defined by the Microsoft engineer was 170 lines in length. The map was created by perusing the files comprising the Excel source before the installation of the reflexion model tools. It took the Microsoft engineer a few hours to define this map. Each entry in the map associated one or more files with high-level model entities. For instance, the following is a snippet from the initial Excel map file.

```
[ file=~shtreal\.c mapTo=Sheet ]
[ file=~textfil[ez]\.c$ mapTo=File ]
```

In total, the engineer spent about a day defining the initial high-level model, setting up the extraction of the source model, and defining the initial map.

The initial reflexion model computed for Excel consisted of the 13 entities originally defined by the Microsoft engineer, 15 convergences, 83 divergences, and 4 absences. To give a sense of this reflexion model, a fragment is shown in Figure 9.2. This reflexion model summarized over 61% of the function calls in the source model. The remaining function calls did not match any line in the map.

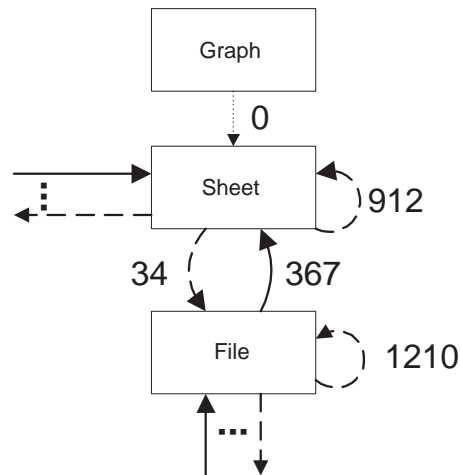


Figure 9.2: A Fragment of the Initial Reflexion Model for Excel.

The Microsoft engineer computed the initial reflexion model using a MS-DOS-based version of the tools running under Windows NT on a 90Mhz Pentium<sup>2</sup> with 40Mb of memory. This version of the tools had a simple textual command-line interface. Computation of the initial reflexion model for Excel took about 20 minutes. (The computation of this reflexion model with the version of the tools described in Chapter 3 takes approximately 4 minutes on a 100Mhz 486.) Although the engineer primarily used the textual output of the reflexion model tools, the engineer could and sometimes did choose to graphically display a computed reflexion model using the AT&T graphviz program[Gansner et al. 1988; Chen et al. 1995].

<sup>2</sup>Pentium is a registered trademark of Intel Corporation.

## Computing a Series of Refined Reflexion Models

Although the initial reflexion model provided insights about the structure of Excel to the Microsoft engineer, further detail about the structure was deemed necessary for the experimental reengineering task. To manage the computation and investigation of successive reflexion models and the exploration of the Excel source code, the engineer refined the iterative process of applying the technique. The process used by the engineer was supported by the reflexion model tools and by scripts developed by the engineer. These scripts operated on intermediate files produced as part of the computation of a reflexion model (Figure 9.3). (The reader is referred to Section 3.1 for a description of the intermediate files produced by the tools.)

*Selecting an Arc to Investigate.* A cycle of the engineer's investigation process began by choosing a reflexion model arc to investigate. The arc of interest was chosen in one of two ways. In the first week or so of applying the technique, the engineer considered the divergences in the reflexion model to determine if any represented an interaction that was missing from the high-level model. If so, the high-level model was updated to reflect the interaction. On further investigation, for instance, the engineer might have decided that the divergence from the `Sheet` to the `File` module represented a reasonable interaction and might have updated the high-level model to reflect the additional interaction. This type of investigation and update was less frequent in later refinements of the reflexion model as the high-level model of Excel used for the experimental reengineering task stabilized quickly.

More commonly, the engineer sorted the image arcs file, which contains both convergences and divergences, selecting the uninvestigated arc with the highest number of mapped calls. The engineer investigated the arc using a script that queried the mapped arcs file for all calls mapped to the arc (see Figure 9.3). The engineer then used an editor to visit and qualitatively assess a subset of the calls mapped to the arc in the source code. As part of the examination of the source, the engineer often used other scripts to

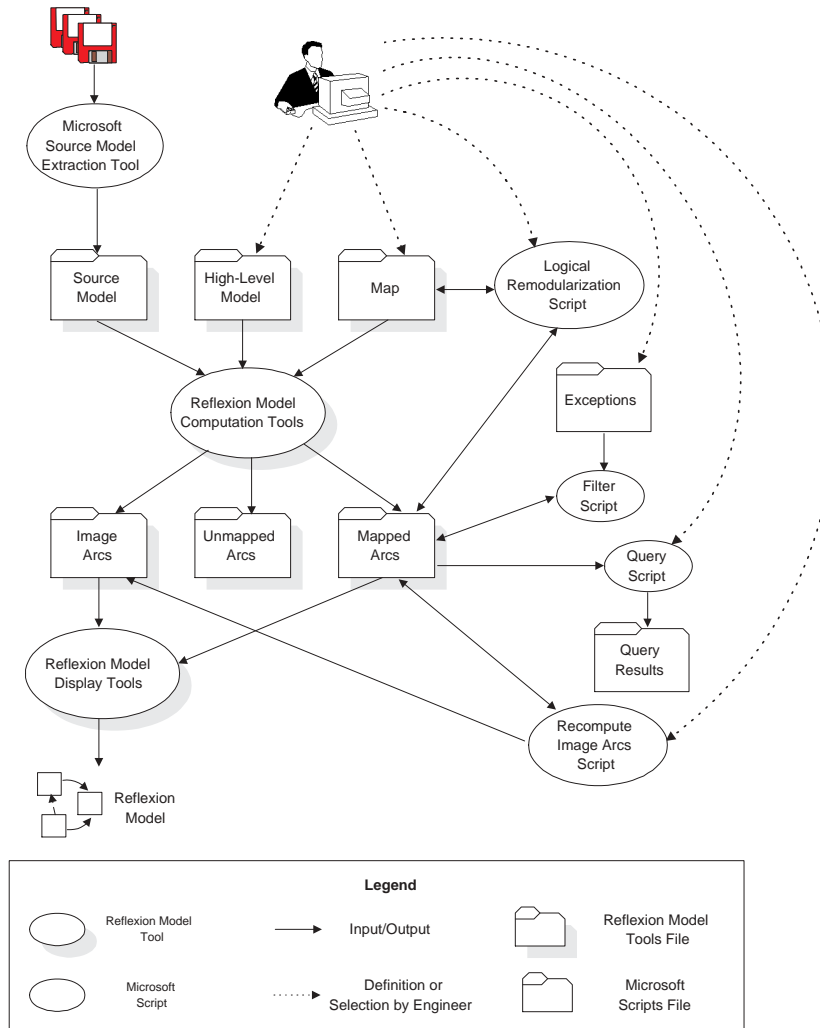


Figure 9.3: Applying the Reflexion Model Technique at Microsoft Corporation.

query the mapped arcs file for calls to specific functions and files.

For instance, sorting the image arcs file for the Excel reflexion model produces the following four lines.<sup>3</sup>

```
File File 1210
Sheet Sheet 912
File Sheet 367
Sheet File 34
```

These lines indicate that 1210 calls are mapped to the reflexion model arc from the **File** module to itself, and 912 calls are from the **Sheet** module to itself. These calls amongst functions assigned to the same module most often appear as self-divergences in the reflexion model since an engineer seldom defines a high-level model that includes calls from a module to itself. When investigating a reflexion model, the Microsoft engineer typically ignored these self-divergences. From this sorted list, then, the engineer would have chosen the arc from **File** to **Sheet** for further investigation. Querying the mapped arc file, he would have discovered calls of the form:

```
LoadPane    in loadutil.c calls CurPn  in common.c
LoadPane    in loadutil.c calls SavePn in common.c
```

and would have investigated the functions defined in `loadutil.c` and `common.c` as a result.

*Refining the Map.* During the investigation of a reflexion model arc, an engineer may discover that the map requires refinement. In some cases, the map requires modification because an entry in the map associates source model entities with the wrong high-level model entity. For instance, an engineer specifying the map for Excel may have inappropriately stated that the `textfile.c` file should be associated with the **Graph** module rather than the **File** module. More often, the map is not sufficiently specific to represent the engineer's intent. For instance, it may be that an engineer wants to

---

<sup>3</sup>These are selected sorted lines pertaining about the named modules.

associate most of the functions in the `fdefs.c` file with a user interface module, but a few of functions, like `ExplodeMergeCells`, should be associated with the `Sheet` module.

During the investigation of Excel, the Microsoft engineer encountered many cases of the second type, where functions were placed in files that were no longer representative of the module the file was intended to represent. As a result, the engineer spent a significant amount of time refining the map to *logically modularize* the Excel system prior to computing a reflexion model. Logically modularizing a function consisted of inserting an entry in the map that specifically associated a particular function with the appropriate module. Since the entries in the map are ordered, these logical modularizations could be inserted into the top of the map file with the existing map entry for the file remaining unchanged. The ordering of the map entries means that when a function from the source model is matched against the map, the matching process will stop when the first match is made against a map entry. For instance, logically modularizing the `ExplodeMergeCells` function in Excel's `fdefs.c` file to the `Sheet` module involved the addition of the entry:

```
[ function=^ExplodeMergeCells$ mapTo=Sheet ]
```

to the top of the map file.

Over the course of the experimental reengineering task, the Microsoft engineer refined the map to consist of over 1000 entries, focusing on refining specific areas of interest. Areas of the model that were deemed outside the activity were not refined, meaning that some parts of the reflexion model represented detailed summaries of the interactions between modules, while other interactions were left fuzzy. The engineer was thus able to manage the investigation process, making progress in areas of the source pertinent to the task while retaining the context of the overall reengineering activity.

*Documenting Exceptions.* A final step in the iterative process used by the Microsoft engineer was the maintenance of a file of *exceptions*. (Exceptions have since been generalized into the concept of annotations introduced in Chapter 4.) These exceptions were

regular expressions describing particular source model values mapped to particular arcs in the high-level model that had been investigated and categorized. For example, when investigating the reflexion model divergence from **Sheet** to **File**, the engineer found several calls related to an event-style interaction. The engineer categorized one or more of these calls using a regular expression. For instance, the following regular expression indicates that the call from **RemovePlyFromBook** to **CloseStmSh** is an event-style interaction.

```
# Event style
^Sheet File .*@RemovePlyFromBook .*@CloseStmSh$
```

The engineer used a simple script to remove the entries in the mapped arcs file corresponding to exceptions prior to the recomputation of a reflexion model. This filtering process helped the engineer focus on the uninvestigated and unexplained interactions summarized within a reflexion model. The file of exceptions also provided documentation on the meaning of some of the arcs in the reflexion model.

*Recomputing Reflexion Models.* Since the initial version of the tools used by the Microsoft engineer required 20 to 40 minutes to compute a reflexion model from the primary inputs, the engineer developed two scripts to support the faster recomputation of a reflexion model from some of the intermediate files. This faster recomputation was necessary to support the iterative use of the technique. The first script supported the logical remodularization process, updating both the map file and the mapped arcs file given the name of a function and the high-level model entity to which the function was to be associated (Figure 9.3). The second script recomputed the image arcs file from the mapped arcs file.

*Augmenting the Source Model.* Just over two weeks into the use of the technique, the Microsoft engineer decided that it would be useful to consider not only information about the calls between functions in the Excel source, but also references by functions

to global variables. Using the internal Microsoft tool, he extracted and incorporated this information into the source model. The combined calls between functions and references to global variables source model contained 119,637 arcs. Augmenting the source model in this way implicitly changed the meaning of the high-level model from a calls between modules model to a communicates-with model. The engineer had no problems in interpreting the subsequent reflexion models even with this change in semantics.

*A Refined Reflexion Model.* A portion of one of the later reflexion models computed for Excel that includes the additional source model information, changes to the map, and changes to the high-level model is shown in Figure 9.4. In this reflexion model snippet, there is evidence of a change to the high-level model with the introduction of a `Wks_File` module in addition to the more general `File` module. The number of interactions summarized between the modules has also increased as indicated by the larger numbers associated with the reflexion model arcs. This increase in the number of interactions summarized is a combination of the augmentation of the source model with global data reference information and the increased specificity of the map.

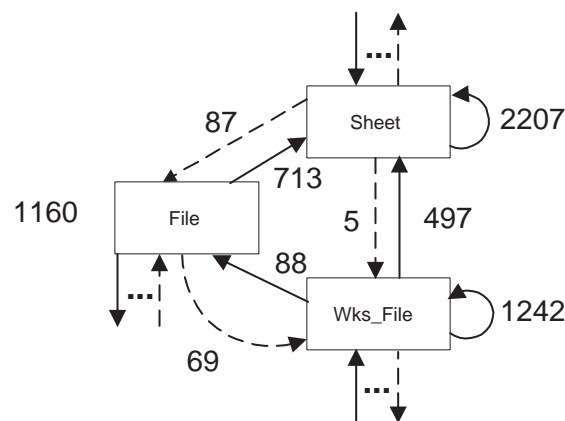


Figure 9.4: A Fragment of a Refined Reflexion Model for Excel.

### 9.1.3 Results

The Microsoft engineer judged the use of the reflexion model technique successful in helping to understand the system structure and source code, as well as to assess and plan the reengineering efforts.

Definitely confirmed suspicions about the structure of Excel. Further, it allowed me to pinpoint the deviations. It is very easy to ignore stuff that is not interesting and thereby focus on the part of Excel that I want to know more about.—Microsoft Engineer, e-mail communication.

More specifically, the technique helped the engineer refine an architectural view of the system, and investigate the connection between the architectural view and the system source code. This view, with the associated map file, was used as a basis for reasoning about the reengineering activity, for assessing the feasibility of various desired changes, and for automating portions of the reengineering activity itself. For example, the entries in the map corresponding to logical remodularizations were used to place conditional compilation statements into the source code as part of altering the Excel source to perform the desired extraction. These benefits were achieved by computing reflexion models based on approximate information.

It is difficult to quantify the exact improvement in performing the software engineering activity using the reflexion model technique. As described above, the Microsoft engineer believes that, in general, it might take as much as two years to gain the same level of familiarity with Excel that it took about one month to gain using the reflexion model technique.

There are several other indications that the technique provided value. First, the engineer chose to use the technique under the pressure of an *actual* engineering task. Second, since the original four week period of use of the reflexion model technique, the engineer has used the technique to refine additional parts of the reflexion model for Excel as well as to compute reflexion models for successive versions of the Excel product. The high-level model now consists of 16 entities and 114 interactions; the source model includes

131,042 call and data interactions; and the map has grown to comprise 1425 entries. The reflexion model computed from these inputs consists of 114 convergences, 77 divergences, and zero absences. The computation of this reflexion model on a DECStation Alpha 3000/300X takes under three minutes, summarizing over 99% of the source model interactions. Finally, the engineer has stated that slowdowns encountered in the performance of the reengineering activity itself were often the result of a lack of up-front understanding and planning. The engineer believes that these slowdowns could have been avoided with more use of the reflexion model technique.

## **9.2 Program Restructuring Tool: A Case Study in Design Conformance**

Engineers typically execute a planned change either manually or with the use of semi-automated methods. When a change is executed in this manner there can be unanticipated effects on the structure of the software system. One way to identify these effects is to check the conformance of the system structure subsequent to the change with the original plan for achieving the desired change. This check is referred to as a design conformance check.

I applied the reflexion model technique to investigate the structural conformance of a program restructuring tool implemented in CLOS with an architectural model of the system [Griswold and Notkin 1995].

### **9.2.1 The Environment**

I began using the reflexion model technique on this task with an understanding of the architectural model of the system [Griswold and Notkin 1995], but with no knowledge of the source code. During the application of the technique, I was in electronic mail contact with William Griswold, the author of the architectural model and the principal implementor of the system.

### 9.2.2 The Process

The previously prepared architectural model [Griswold and Notkin 1995] consisted of (essentially) eight submodules<sup>4</sup> with eleven call paths between those submodules. This model was easily translated into a high-level model for use with the reflexion model tools. One difference between the architectural and the high-level models was that the layer boundaries present in the architectural model were not represented in the high-level model.<sup>5</sup>

The source model extracted from the approximately 47,000 lines of source code consisted of calls between functions in the CLOS code. Also recorded in the source model was the class to which a function was associated. This source model was extracted using the Allegro Common Lisp environment. The source model included information about 5855 calls between 2505 functions. This version of the tool did not support the inclusion of physical software structure information (i.e., directory and file) information into the source model.

An initial map for the reflexion model computation was developed based on a perusal of the source code. Comments associated with functions and the names of functions were used to associate functions with entities in the high-level model. This map consisted of 162 entries. It would have likely been possible to create a map with fewer entries if more aggregate structural information, such as file information for each function, had been available.

The first reflexion model computed for this task mapped about 25% of the calls in the source model and included a number of absences and divergences. As I gained more understanding of the source, partly by investigating the reflexion model and partly from responses to questions by the author of the source, I refined the map and computed a series of reflexion models. In the end, eleven reflexion models were computed and the

---

<sup>4</sup>A submodule bundles a collection of operations for manipulating a data representation at a specific layer within a layered system.

<sup>5</sup>Since the layer boundaries were not represented, it was not necessary in the high-level model to represent versions of a submodule reexported at different layers.

map grew to 215 entries. The final reflexion model computed included only convergences and divergences; the absences had indicated missing entries in the map file which were added in later iterations.

### 9.2.3 Results

Although some divergences that appeared in the earlier computed reflexion models were indicative of misunderstandings on my part of how functions should be mapped to high-level model entities, other divergences indicated places in the code where restructurings had not been consistently or completely applied. Identifying these code sections was beneficial as the author was able to revisit those locations and update the code to ensure the implementation encoded the appropriate structure. The locations of inconsistently applied restructurings are often the means by which “structural drift” occurs in a software system and entropy begins [Lehman 1974]. Over time, if not attended to, these small drifts can result in increasing code complexity and increasing costs for change.

In the case of this design conformance task, divergences played the most important role because, by definition, divergences are more likely to be indicative of non-conforming structural interactions.

One concern that is often raised about the reflexion model technique is the level of knowledge an engineer must have of a system in order to compute a useful reflexion model. This case study and the Excel case study both provide evidence to quell this concern; an engineer can compute a useful reflexion model for at least some tasks with minimal knowledge of the system being studied.

## 9.3 SPIN: A Case Study in the Use of Approximate Information

In cooperation with another graduate student at the University of Washington, Marc Fiuczynski (who is referred to through this discussion as “the *SPIN* team member”), I applied both the reflexion model and the lexical source model extraction techniques to help developers determine if the software structure of the *SPIN* operating system was

being implemented to plan. The *SPIN* operating system is being developed as part of a research project at the University of Washington. The *SPIN* kernel, which is primarily implemented in Modula-3 [Cardelli et al. 1989], is layered on top of the DEC OSF/1 operating system, which is primarily implemented in C. Two aspects of the structure of the *SPIN* system were investigated: the calls between Modula-3 modules, and the calls from the Modula-3 *SPIN* code to the DEC OSF/1 C code.

### 9.3.1 The Environment

At the time this case study was undertaken in the Spring of 1995, to my knowledge, there were no commercial or public-domain tools available to support the extraction of call and reference information from Modula-3 source code. Based on an outline of requirements by the *SPIN* team member, I applied both the reflexion model technique and the lexical source model extraction technique to analyze the implementation of *SPIN*. The observations reported in this case study are based on notes taken during the study, on electronic mail exchanged with the *SPIN* team member, and on snapshots of input files to the tools.

### 9.3.2 The Process

The first step in applying the techniques was to use the lexical source model extraction technique to produce a source model with the structural information of interest to the *SPIN* developers. Then, a reflexion model was computed to summarize the extracted information. I describe the use of each of the techniques in turn.

#### Applying the Lexical Source Model Extraction Technique

Two lexical specifications were written to extract the desired structural information. The first specification extracted calls between Modula-3 procedures. The second specification extracted potential calls from a Modula-3 module to C functions based on the (required) external declaration of a C function within a module.

*Extracting Calls between Modula-3 Modules.* The calls between modules of interest to the developers were those of a particular syntactic form where the name of the called procedure was preceded by the name of the module and a period. For example, the construct `Word.LT` indicates a call to a procedure (or function) named `LT` located in the `Word` module. In addition to recording the module in which the procedure being called was located, we wanted the source model to contain directory information to ease the specification of the map.

Producing this source model required the extraction of multiple relations and the subsequent combination of those relations through a generated analyzer. One relation, named “indir”, described the location of modules in directories. The other relation, named “call”, described the calls between modules. During scanning, the name of a module was inferred from the name of the file. The two patterns used to extract the necessary information and to form the relations are shown in Figure 9.5.<sup>6</sup> The first pattern recognizes the start of a procedure definition, while the second pattern recognizes the desired call construct. From these patterns, an extractor was generated to scan the implementation files (i.e., `.m3` files). Only the implementation files for particular subsystems of interest were scanned.

```
PROCEDURE (proc_id) \( { (stuff) }+ \) [ : (stuff) ]
  [ RAISES \{ [ { (stuff) }+ ] \} ] = @
  directory := getArguments(1)
  file := getArguments(2)
  # record in "indir" relation the directory in which this module resides
  @

[ , ] (module) . (call) \( @
  # record in a "call" relation the caller and callee
  @
```

Figure 9.5: Patterns to Extract Calls between Modula-3 Procedures.

---

<sup>6</sup>The action code attached to these patterns has been slightly modified for presentation purposes. Specifically, the details of forming the relations have been suppressed.

*Extracting Calls from Modula-3 Modules to C Code.* Calls from Modula-3 modules to C code were recognized by the presence of an `EXTERNAL` declaration in the Modula-3 source code using the pattern shown in Figure 9.6. When the `EXTERNAL` declaration was seen, tuples were output for two relations: an “indir” relation that described the location of a module within a directory, and a “decl” relation that described the potential reference of the module to the specified C function. From this pattern, a scanner was generated and executed across the definition files (i.e., `.i3` files) comprising the implementation. The tuples resulting from the scan were appended to the file of tuples resulting from the scan for calls between Modula-3 modules.

```
\< * EXTERNAL [ { (junk) }+ ] * \> PROCEDURE (procName) \( @
  directory := getArguments(1)
  file := getArguments(2)
  # Record in "indir" relation the directory in which this module resides
  # Record in "decl" relation the definition of this external procedure
  @
```

Figure 9.6: A Pattern to Extract Calls to Externally Declared C Functions.

*Producing the Source Model.* The tuples resulting from the two scan sequences were processed by a generated analyzer that determined for each call described in the “call” relation if the call was to a procedure implemented in Modula-3 or in C. A query to the “decl” relation was used to determine the language of the callee. Each call was then written as a formatted entry to the source model. The generated source model consisted of 850 calls between 465 procedures and functions.

### Applying the Software Reflexion Model Technique

The reflexion model computed for investigating the structure of the *SPIN* implementation differs from those presented earlier in this dissertation because, in this case study, the high-level model consisted solely of entities with no interactions posited between the entities. The high-level model consisted of 17 entities whose names corresponded to the

names of the directories containing the source code. The entities were defined on the basis of the directory structure because this structure paralleled the desired architecture for the system.

The map was also defined using a different process than in the other examples presented. A shell script was used to automatically generate a map containing entries of the form shown below.

```
[ module=CPUState  mapTo=alpha ]
[ module=Pmap      mapTo=mm   ]
```

These entries were formed by using the names of the files as module names and the names of the directory in which a file was found as the high-level model entity.

The reflexion model computed from the high-level model, the map, and the source model produced using the lexical source model extraction technique consisted only of divergences since no interactions between the high-level model entities were posited. A total of 76 divergences were reported as a result of the computation.

### 9.3.3 Results

Looking at these divergences, the *SPIN* team member quickly identified a few suspect interactions. Using a graphical display of the reflexion model, the *SPIN* team member conferred with the *SPIN* developer responsible for that portion of the implementation. As a result of this interaction, the *SPIN* developer modified the Modula-3 implementation. Although the responsible developer had known that this undesirable interaction remained in the source code, the visualization of the interaction helped motivate the occurrence of the change.

This case study shows that although the source model produced with the lexical source model extraction technique was approximate, useful results were still obtained from summarizing that information in a reflexion model. Perhaps not surprisingly, this approximate information was useful for detecting an anomaly in the implementation with

respect to the design, rather than in directly building confidence that the implementation matched the design. Even so, it may be that as the anomalies detected decrease, an engineer's confidence in the implementation of the design as checked by a reflexion model rises even in the presence of an approximate source model. Boehm has also noted the usefulness of tools that support for anomaly detection in the evaluation of software products [Boehm 1976], although not specifically in the presence of approximate information.

#### 9.4 Additional Case Studies

In addition to the three case studies described in detail, there have been several other uses of each of the techniques. The other uses of the lexical source model extraction technique are outlined in Section 6.4. This discussion focuses on additional uses of the software reflexion model technique.

*Design Conformance.* An industrial partner used the software reflexion model technique as part of a design conformance task. Specifically, the partner applied the technique to check if a small, on the order of a few thousands of lines of source code, C++ [Stroustrup 1986] implementation of a subsystem matched previously prepared design documentation that consisted of a Booch object diagram [Booch 1993]. This case was unique in that the reflexion model was fully convergent with the source model. This result is not surprising given that the implementation was small, the same person who designed the system implemented the system, and the implementation had not undergone any evolutionary tasks.

*System Understanding.* I also applied the software reflexion model technique as part of a system understanding task to try to determine why a compiler used in undergraduate education at the University of Washington was difficult for the students to change. I first computed a reflexion model comparing an extracted Ada [GPO 1983] file imports

relation with a conventional model of a compiler [Perry and Wolf 1992]. The reflexion model contained meaningful divergences between almost all pairs of high-level entities. This high degree of coupling explains, in part, why the students had difficulty changing the system.

Later, I applied reflexion models to a newer version of the compiler, written in C++. The reflexion models for this compiler were far less cluttered than the Ada version. However, some unexpected interactions were identified using the reflexion model; these divergences may provide the basis for either minor restructuring of the compiler or at least additional warnings to the students.

## 9.5 Summary

This chapter has described, in varying levels of detail, five uses of the reflexion model technique and one use of the lexical source model extraction technique. Each case focused on a different combination of task and system. Together, these case studies provide support for the central claims of this dissertation. The use of the reflexion model technique by a Microsoft engineer, by choice and under time pressure, shows that the software reflexion model technique can provide benefits when assessing and planning a change task on a million line system. The use of the reflexion model technique to assess the conformance of a program restructuring tool's implementation to a published design shows that reflexion models may be helpful to an engineer manually or semi-automatically executing a change. Finally, the use of both the reflexion model and the lexical source model extraction techniques to investigate the structure of the *SPIN* operating system supports the claim that approximate structural information is useful for certain classes of tasks.

The use of the reflexion model technique in each of these cases also highlights the benefits of a technique that is partial and iterative. Users were able to compute an initial reflexion model that summarized structural information about a portion of a system and then could successively refine the computation to encompass as much of the system as was necessary for supporting the task at hand.

# Chapter 10

## Related Work

Many methods and techniques exist to help an engineer understand the structure of an existing software system. In this chapter, I compare the software reflexion model technique and the lexical source model extraction technique to these other approaches. I also discuss the use of summarization in other areas of computer science.

### 10.1 Software Reflexion Models

The software reflexion model technique provides an engineer with qualitative information about a system's structure at a higher-level than the source code. In this section, I both compare the reflexion model technique to other approaches with the same objective, and also discuss the relationship of the technique to other software engineering tools that are based on model comparison.

Approaches that provide structural information solely at the level of the source code are not included in this discussion. Similarly, metrics are not considered. The interested reader is directed to the brief discussion of these techniques provided in Chapter 1.

#### 10.1.1 Forward Engineering

One way to promote an understanding of software structure is to capture the desired information as part of the activity of system design. Most common design methods include some specification of one or more structural aspects of a system. In struc-

tured design [Yourdon and Constantine 1979], for instance, an engineer specifying structure charts details the calls between functions. Using the Booch object-oriented design method [Booch 1993], an engineer specifies in an object diagram the interactions between objects in the system.

To further support an engineer in specifying and reasoning about a system's structure, several research groups are developing software architecture description languages to formalize the often used box and arrow diagrams, including Allen and Garlan [Allen and Garlan 1994], Dean and Cordy [Dean and Cordy 1995], Shaw and colleagues [Shaw et al. 1995], and Moriconi and colleagues [Moriconi et al. 1995]. Formalizing these diagrams permits an engineer to analyze the intended structure for desired properties such as the absence of deadlock [Allen and Garlan 1994].

Specifying the intended structure before implementing the system has many benefits. For example, the preparation of an architectural specification using an architectural description language may help identify structural problems related to deadlock earlier in the development cycle when problems are less costly to solve. However, these specifications may be useless or even detrimental for an engineer trying to reason about and perform changes to an implemented system if they are not consistent with the "actual" structure embedded within the source. Even if accurate, the value of existing structural specifications depends on the appropriateness of the structural view(s) represented by those artifacts for the current task being performed. Since the task being performed may not have even been predicted during the initial design, there is a reasonable risk that the view(s) provide by the artifact will not be appropriate.

Three different approaches are available to address the consistency problem: computer-aided software engineering (CASE) tools that generate source from, in part, the structural specifications; traceability tools that maintain links between related parts of software artifacts; and checker tools that attempt to identify inconsistencies between the structural specification and the implementation. I consider each of these below.

*CASE Tools.* Several commercial CASE tools (e.g., Cayenne Software Inc.'s Teamwork/Ada product and Rational Software Corporation's Rose/C++ product) support the generation of code templates from design documents. Code templates in C++ [Stroustrup 1986], for example, can be generated from an OMT class diagram [Rumbaugh 1990]. Similar generation efforts have been suggested as a future direction for research on architecture description languages [Clements 1996]. However, except in specialized domains, no current CASE tool can produce a complete system from the design documents. As a result, engineers still need to write and later modify some of the source code. But, as soon as engineers begin to change the source, it is possible—and almost inevitable—that the structure of the system will begin to deviate from the intended structure in the design. For instance, Carmichael and several colleagues, while exploring the source code for a compiler that was developed at IBM using a traditional development process, found deviations between the structure as expressed in the source and the intended structure [Carmichael et al. 1995]. Thus, unless the CASE tool can be used in a way that removes the need for engineers to change the generated source code, the tool cannot, over time, ensure consistent high-level views of the structure of a software system.

*Traceability Tools.* Traceability tools help an engineer determine dependent items within and between the documents, the work products, and the source comprising a software system [Lindvall 1994]. These tools—for example, Pirnia and Hayek's requirements traceability tool [Pirnia and Hayek 1981] and Cayenne Software's RTM—may help an engineer maintain consistency between various analysis and design artifacts. However, traceability is rarely performed to the code level because of the number of items that exist in the code. One system that did perform traceability to the code level was the Pegasys system [Moriconi and Hare 1985]. Pegasys was developed to ease the use of graphical images as formal documentation and, amongst other features, permitted the expression of a program design as a set of hierarchically-refined pictures. Icons representing atomic items in a picture could be associated with text in the source code of

a system. However, even when tracing is performed to the code level, there will often be times in the development cycle that the tracing is not complete or is inconsistent. This is inevitable since as Balzer has noted, “[s]oftware systems, especially large ones, are rarely consistent” [Balzer 1991, p. 158]. Traceability tools, then, also do not solve the consistency problem.

*Checkers.* A third way to handle the consistency problem is to use a checking approach in which the structure as embedded in the software system is compared with the intended structure. Ossher built such a checker to support a comparison of the structure as expressed in a system’s source code with a GRID description of the intended structure [Ossher 1984]. A GRID is a formalism developed by Ossher to describe the structure of large, layered systems [Ossher 1987]. This kind of checker was also built as part of the Software Landscape environment that supports a visual module interconnection language [Penny 1993]. Similar to Ossher’s approach, in the Software Landscape, relations between program entities could be extracted and compared directly to relations between low-level design entities. Each of these checkers was particular to the kind of structural diagrams supported by the respective design methods. The software reflexion model technique provides a more general structural checker; fewer constraints are placed on the form of the high-level (design) model, and greater flexibility is supported for associating source model entities directly with high-level model entities of interest.

Sefika and colleagues describe a system called Pattern-Lint that supports compliance checking of implementations to a wide-range of high-level models [Sefika et al. 1996]. In Pattern-Lint, an engineer describes the high-level model as two sets of Prolog clauses: one set of clauses defines positive evidence that the source complies with the model, the second set defines violations that indicate non-compliance. An engineer may then compare structural information extracted from C++ source code with the pre-defined models. The system also supports compliance checking through a set of animations of dynamic structural information (e.g., calls reported when executing an instrumented

version of the program). The software reflexion model technique differs from Pattern-Lint in several ways. First, with the reflexion model technique, the engineer need only state what is expected for compliance with the high-level model, whereas in Pattern-Lint, the engineer must state both what is expected and what is not expected for compliance. If an engineer neglects to include a particular kind of violation in a Pattern-Lint model and the violation occurs, it will not be reported by Pattern-Lint, whereas it may be reported as a divergence using the reflexion model technique. Second, in Pattern-Lint, both the clauses defining the compliance rules for a model and the association of source model entities with higher-level entities are specified as Prolog clauses. It is unclear whether this heavier-weight description affects the scalability of the Pattern-Lint system. Finally, with the software reflexion model technique, structural information collected during the execution of a system may be summarized in terms of the same high-level model(s) in which information extracted statically is viewed. In Pattern-Lint, different views of the system are used to display this information.<sup>1</sup>

### 10.1.2 Reverse Engineering

When specifications do not exist as by-products of the activities of analysis or design, or when the system views created during those activities are inconsistent with the source code or are inappropriate for the task being performed, an engineer may apply a reverse engineering technique to create a high-level view of a system. In this section, I compare the software reflexion model technique to reverse engineering techniques that support the creation or derivation of high-level structural models; reverse engineering techniques that focus on the automatic identification of abstract data types, such as the technique of Canfora and colleagues [Canfora et al. 1993], or the reengineering of databases, such as the technique of Premerlani and Blaha [Premerlani and Blaha 1994], fall outside the

---

<sup>1</sup>It is difficult, if not impossible, to use some of Pattern-Lint's high-level models to check compliance with dynamically extracted information because the compliance rules refer to information only available statically. For instance, some rules refer to static structural information such as the `friends` relation between C++ classes.

scope of this comparison and are not discussed.

Techniques for deriving high-level structural models from a system's artifacts fall into two classes. First, there are a set of techniques that are based on clustering. Second, there are a set of techniques based around the use of transformations. The former class is discussed in this section, while the latter class is discussed in the knowledge-based section below.

*Clustering Reverse Engineering Techniques.* Lakhota characterizes twelve reverse engineering techniques based on clustering according to, amongst other things, the level of automation and the nature of the source information supported by the technique (i.e., control-flow graphs, data flow graphs, etc.) [Lakhota 1996]. Of the twelve techniques surveyed by Lakhota, the Rigi system [Müller and Uhl 1990] is the sole technique to operate, semi-automatically, on a generic set of source model relations similar to the software reflexion model technique. Hence, I focus in this section on the Rigi system and a similar system, called DMV, built at Microsoft Research.

The Rigi system “supports a method for identifying, building, and documenting layered subsystem hierarchies” [Wong et al. 1995, p. 47]. Similar to the software reflexion model technique, an engineer begins by extracting structural information from system artifacts and by representing that information as a set of relations. These relations are provided as input to the Rigi tool which displays them as a collection of overlapping graphs. The engineer may then manipulate the graph(s) to identify source model entities (nodes) that should be clustered. When a set of nodes to cluster is determined, the engineer may apply an operation that condenses those nodes into a higher-level abstract entity. This causes both the creation of a new node and also the appropriate replacement of links from nodes in the selected (now clustered) set to the rest of the graph with links from the new node to the rest of the graph.<sup>2</sup> An engineer identifies the set of entities to cluster by applying graph-theoretic algorithms (i.e., identify the strongly

---

<sup>2</sup>A variety of windowing support is provided to aid an engineer in developing and managing the hierarchy. This support is not pertinent to this discussion.

connected components), by applying quality metrics such as coupling and cohesion, or by defining procedural scripts to mimic user-interface operations such as searching for nodes conforming to particular naming conventions.

There are four significant differences between the software reflexion model technique and Rigi. First, Rigi uses abstraction rather than summarization to produce the desired high-level model. As a result, an engineer using Rigi produces an opaque model that does not let the details of the source model show through. For some change tasks, this opaque model may be appropriate, while for others, the details may be beneficial. For instance, in the estimation task described in Chapter 4, the values associated with arcs in a reflexion model that represent details of the underlying source model might be useful in producing a more accurate estimate of the difficulty of the change. The Microsoft engineer used this detail to iterate a series of reflexion models focusing on particular aspects of the Excel structure pertinent to the experimental reengineering activity (Section 9.1).

Second, when nodes of a graph are clustered based on source model information such as the names of entities, the clustering is performed either by using the interface or by using a procedural script. In contrast, the reflexion model technique clusters source model entities through the use of a declarative map. The benefits and limitations of a declarative as compared to a procedural map are similar to the benefits and limitations of a declarative programming language as compared to a procedural programming language. The declarative map is easy for the engineer to specify, the map is likely to be shorter than a procedural equivalent, and the map is likely simpler in format, improving the likelihood that the engineer specifies the desired mapping. The declarative map used in the reflexion model technique also has value in supporting the performance of changes. For example, the Microsoft engineer applying the reflexion model technique to aid with an experimental reengineering of Excel used the map as a basis for automating parts of the desired change. A limitation of a declarative map, though, is that an engineer might not be able to express as wide a range of mappings, or in the case of Rigi, clusterings,

than is possible with a procedural map.

Third, by employing a combined top-down and bottom-up process, the software reflexion model technique is better able to handle complexity and scale in the source model. The approach used in the reflexion model technique corresponds to the class of techniques categorized by Ossher as convergent approximation:

The technique of convergent approximation can be thought of as a combination of approximation, analogy and deviation. Assume an approximation of the truth is known. You describe a more accurate approximation by analogy with the known one, indicating just how the new one differs. Then repeat until acceptable accuracy is achieved. [Ossher 1987, p. 235]

Using the software reflexion model technique, an engineer can select a high-level model representing the truth, define a coarse map, and then quickly compute a reflexion model that summarizes either all or only a portion of the information in the source model. For instance, it took the Microsoft engineer only a few hours to compute an initial reflexion model for the Excel system. With Rigi, however, an engineer must express the desired clustering of source entities, either manually or programatically or both, based on the entire source model. If, for instance, the engineer clustered 95% of the entities in the Excel source model that is comprised of 18118 entities, the screen would still display 907 nodes even if all the selected entities had been clustered into one node. Significant effort by the engineer using Rigi may be necessary, then, before meaningful results can be produced.<sup>3</sup> Using Rigi, the engineer may also expend effort clustering entities that are uninteresting for the task being performed.

This leads to the fourth significant difference between the two techniques. Even when sufficient clustering has been performed to derive a high-level model with Rigi, the model that results is not necessarily a view of interest to the engineer since the Rigi method is driven from the bottom-up; this is described in Section 1.3.2. One way an engineer might

---

<sup>3</sup>The largest reported use of Rigi was as part of a redocumentation effort of a roughly 380,000 line subsystem of an IBM database product written in PL/AS [Tilley 1995; Wong et al. 1995]. Rigi was applied to cluster, based on naming conventions, a source model consisting of 23,389 arcs. This source model is about 20% of the size of the Excel source model.

improve the likelihood of generating a desired view with Rigi is to apply domain and system-specific knowledge during the bottom-up clustering process. Wong and colleagues describe how this kind of approach helped—in a later experiment from that described in Section 1.3.2—to produce a model of the IBM SQL/DS system that “the developers readily recognized” [Wong et al. 1995, p. 51]. With the software reflexion model technique, the engineer is guaranteed to see the system through the desired view (even if it is not a particularly appropriate representation of the true structure of the system) since the engineer specifies the high-level model. An advantage of the Rigi approach is that it may be applied even in the absence of any information about the structure of the source since the high-level view is derived and not stated. To date, there has not been a situation where an engineer applying the software reflexion model technique has not had any knowledge of the structure of the system; generally, engineers have found it easy to specify the necessary high-level model.

Weise’s DMV system, built at Microsoft Research, is similar to the Rigi system.<sup>4</sup> Like Rigi, DMV supports the construction of layered subsystem hierarchies. DMV differs from Rigi in the operations provided to create the subsystems. Rather than identifying groups of source model entities from which to create a cluster, in the DMV tool, an engineer posits the clusters and then performs move operations to associate source model entities with the appropriate cluster. The move operations are implemented through a point-and-click interface that allows an engineer to drag a source model entity from one cluster to another as the source is investigated. DMV, like Rigi, derives arcs between the clusters from the underlying source model information.

The development of the DMV tool was motivated by the use of the reflexion model tools on the experimental reengineering of Excel and was specifically designed to handle the logical remodularization operations performed repeatedly by the Microsoft engineer during the task (Section 9.1.2). Similar to the software reflexion model tool, the DMV tool requires an engineer to posit the desired clusters—high-level model entities in the termi-

---

<sup>4</sup>Personal communication, D. Weise, April 1996.

nology of reflexion models. An engineer using the DMV tool starts by specifying the same input files that are expected by the software reflexion model tools. Once input, however, the map becomes implicit and an engineer performs the desired move operations through the point-and-click interface. More investigation is needed to determine the benefits and limitations of the declarative map approach used in the reflexion model tools versus the direct manipulation of icons used in the DMV tool, particularly when large changes are made to the map. Unlike the reflexion model tools, DMV does not permit an engineer to posit interactions between high-level model entities and thus does not support a comparison between interactions in a high-level model and interactions in a source model. It is unclear how the lack of this comparison affects the early investigation of unrefined reflexion models; the Microsoft engineer, for instance, used this information to refine the high-level model in reflexion model computations performed at the beginning of the experimental reengineering task.

### 10.1.3 Knowledge-based Approaches

Another class of techniques to aid an engineer in understanding software structure use knowledge-bases. These techniques may be classified into two categories by the kind of knowledge comprising the knowledge-base. Techniques in the first category encode knowledge as pre-defined patterns or cliches [Rich and Waters 1990] that are generally domain-independent. Techniques in the second category encode knowledge about the domain and the system design.

Cliche-based techniques may be further subdivided into automated, semi-automated, and manual methods. Automated program understanding techniques, such as Quilici's technique [Quilici 1994], extract design information by matching recorded cliches to the source. These techniques differ in intent from the software reflexion model technique as they primarily focus on building a representation and understanding of the target system within a tool rather than on aiding the engineer in gaining the understanding. Semi-automated cliche-based techniques use the pre-coded patterns as a guide to aid an

engineer in producing a higher-level view of the system through the successive applications of transformations. An example of a semi-automated cliché-based technique is the technique of Ward and colleagues [Ward et al. 1989]. Since, like Rigi, these semi-automated techniques largely rely on a bottom-up process, they are likely to encounter problems associated with scale. An example of a manual cliché-based technique is the Synchronized Refinement approach [Ornburn and Rugaber 1992] in which an engineer states an expected design and then successively tries to use semantically-based clichés to rewrite the source code as higher-level pseudo-code to produce that design. When necessary, as driven by information from a code analysis, the engineer may also modify the expected design. This approach is similar in intent to the software reflexion model technique but does not use any automated comparison. Two limitations shared by all cliché-based techniques are the need to determine appropriate representations for clichés and the need to identify and populate the knowledge-bases. Compared to the software reflexion model technique, however, these approaches share the advantage that it may be possible to derive high-level models that contain both behavioural and structural information.

Closer in intent to the reflexion model technique are the category of techniques that use a knowledge-base of facts about a domain and a system to permit an engineer to perform queries at a higher-level of abstraction than the source. The LaSSIE system [Devanbu et al. 1991] is representative of this approach. In LaSSIE, the knowledge-base contains information about the domain (i.e., telecommunications), the architecture, the features, and the code of the system. An engineer may use the system to determine the answers to queries, such as which functions implementing a specific domain concept access global variables in a particular file. This type of technique shows promise for supporting an engineer trying to reason about desired changes. However, it is unclear if the effort of building the necessary knowledge base and maintaining its consistency is cost-effective.

#### 10.1.4 Program Visualization

As described in Chapter 1, program visualization techniques that provide direct graphic representations of the structure of a system face problems with scale. To combat the scale problem, some techniques provide limited support for clustering. Typically, this clustering is based on the physical structure of the source code. For instance, the Field system [Reiss 1990; Reiss 1995] can display information about the calls between C [Kernighan and Ritchie 1978] functions at the function, file, directory, or some combination of these levels. For many large systems, however, this does not sufficiently simplify the display of the structural information. The source for the GNU project's gcc compiler, for instance, has over 200 source files in one directory. Even a file-level representation of this information tends to be overwhelming for the engineer.

Another approach taken to deal with the scale problem is to produce displays in three dimensions; Reiss' PLUM system [Reiss 1993] and the Imagix Corporation's program visualization tool Imagix 4D fall into this category. Although this may permit the display of more information, it does not attack the essential problem of the invisibility of software. As Brooks has noted:

Whether one diagrams control flow, variable scope nesting, variable cross-references, data flow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the diagrams generated by the many relevant views, it is difficult to extract any global view. [Brooks 1986, p. 16]

Since it is this global view that is often of importance to an engineer reasoning about a change, it is not likely that more dimensions of visualizing source-level software structure will suffice.

#### 10.1.5 Model Comparison

Software reflexion models result from the comparison of two models at different levels of abstraction. An essential characteristic of the reflexion model technique is its use of

a declarative map to associate the two models. In this section, I compare the approach with three other software engineering tools: Jackson's Aspect system [Jackson 1995], Howden and Wieand's QDA analyzer [Howden and Wieand 1994], and Jackson and Ladd's semantic diff tool [Jackson and Ladd 1994].

The Aspect system supports the comparison of partial program specifications (high-level models) to data flow models extracted from the source. The system is able to detect bugs in the source that cannot be detected using static type checking. Aspect uses dependences between data stores as a model of the behaviour of a system. For abstract types, the system permits an engineer to express the dependences between data stores in terms of abstract components rather than in terms of the type's representation. An abstraction function is used to relate the concrete components of the type to the abstract components stated in the specification. When the Aspect checker is run, differences between the information extracted from the source and the specification are reported both in terms of the abstract and the concrete components.

Maps in the software reflexion model technique play a similar role to Aspect's abstraction functions. Both maps and abstraction functions permit many-to-many association between entities in models at different levels of abstraction. However, in Aspect's abstraction functions, there is no notion of approximation as is found in the reflexion model maps where coarse-grained maps are supported but may be refined over time. Aspect's abstraction functions also differ from software reflexion model maps in that they appear, to the engineer, to be bi-directional; the functions are used to translate source dependences to abstract dependences, and are also used to report discrepancies found at the abstract level in terms of the concrete level. Software reflexion model maps, on the other hand, are used only uni-directionally to push source model interactions through to the higher-level; the information about which source model values contribute to a reflexion model arc is independent of the map. A bi-directional treatment of the reflexion model map might extend the kinds of queries available to an engineer when successively refining a reflexion model; for instance, it might permit the determination of whether a reflex-

ion model arc is indicative of a relationship holding between all source model entities mapped to the respective high-level model entities or only some of the high-level model entities.

Howden and Wieand's Quick Defect Analysis (QDA) involves the introduction, by an engineer, of comments describing facts and hypothesis about a program's problem domain into the source code. Once commented, an engineer may use an analyzer tool to interpret the abstract program defined by the comments, and the control flow defined by the source code, to verify hypotheses. In this approach, a special kind of comment, a rule, may be used to associate program-oriented properties with properties of the abstract program. These rules thus play a similar role to the map of the software reflexion model technique. Similar to Aspect, the intent of the map in QDA is to enable automated analysis at the abstract level. To support this objective, the QDA map language, in contrast to a software reflexion model map, is richer and more precise, supporting an association between the conjunction of the states of one or more concrete properties to the state of a single abstract property.

Jackson and Ladd have developed a semantic difference approach for comparing the differences in input and output behaviour between two versions of a procedure. Given two versions of a procedure, this approach derives a model of the semantic effect of each procedure consisting of a binary relation that describes the dependence of variables at the exit point of the procedure with variables upon entry to the procedure. The semantic differencing tool compares the relations resulting from each version of the procedure. This tool thus differs from the reflexion model technique in supporting the comparison of relations at the same level of abstraction rather than the comparison of relations at different levels of abstraction.

## 10.2 Lexical Source Model Extraction

In this section, I compare various techniques for extracting structural information from system artifacts with the lexical source model extraction approach. The techniques in

the comparison are broken into two categories: tools based on regular expressions, and tools based on parse trees.

### 10.2.1 Tools Based on Regular Expressions

A number of tools and languages support the scanning of textual artifacts for specified regular expressions.

The grep family of tools (e.g., grep, fgrep, egrep, agrep [Wu and Manber 1992], and cgrep [Clarke and Cormack 1995]) support the identification of text in artifacts matching specified regular expressions. Most of these tools are restricted to searching and returning lines from the artifacts; exceptions are the agrep tool which permits the use of a separator pattern to delimit the records to be considered, and the cgrep tool, which permits the description of records, in which the grep is to be conducted, by a pair of regular expressions. None of these tools provide support for identifying the pieces of text matched to particular parts of the regular expression, and none support the execution of actions when matches are found, restricting their use for source model extraction.

These restrictions are relaxed in the awk text scanning language [Aho et al. 1979], the lex scanner generator [Lesk 1975], and the perl scripting language [Wall 1990], which support the execution of code written by the engineer when text is matched to specified regular expressions. The lex tool does not provide any support for unifying matched lexemes to parts of the regular expression, whereas the awk and perl languages support unification for a subclass of regular expressions. The use of these tools for source model extraction is complicated by the limited support for accessing the values of text matched to portions of the regular expressions, and a lack of support for specifying prioritized hierarchical collections of regular expressions.

The TLex tool [Kearns 1991], a pattern matching and parsing library for C++, is the lexical tool most similar to the lexical source model extraction technique, providing access when a match is made to a parse tree automatically constructed for the regular expression. TLex supports a broader class of regular expressions than the lexical source

model technique with some support for contextual regular expressions (i.e., match a regular expression only when another regular expression matches to the right). TLex, however, does not ease the specification of complex hierarchical regular expressions with associated heuristics for matching, nor does it provide the ability to control matching and backtracking within the action code.

As described earlier, the lexical source model extraction technique also differs from existing regular expression tools in its use of an implicitly defined lexer.

### 10.2.2 Tools Based on Parsing

To enable more precise matching of syntactic constructs, many approaches construct parse trees from structured artifacts, and provide support for traversing and performing actions on the parse trees.

Some, including Ladd and Ramming's A\* tool [Ladd and Ramming 1995], Arnon's Scrimshaw system [Arnon 1993], and Griswold's tawk system [Griswold et al. 1996], support regular expression matches over parse trees. Others, such as Refinery [Burson et al. 1990], TXL [Cordy et al. 1991], Scruple [Paul and Prakash 1992], GENOA [Devanbu 1992], Code Miner [Dunn and Knight 1993], and Ponder [Griswold and Atkinson 1995], take a variety of different approaches for querying and transforming parse trees. Scruple, for instance, provides a language with wildcards based on the concrete syntax of the artifacts for querying abstract syntax trees. Refinery, on the other hand, supports queries in first-order set-theoretic logic.

Regardless of the query language supported, the lexical source model extraction technique differs from all these systems in searching only for those code constructs that the engineer has specified rather than performing a parse. This searching approach makes the technique less sensitive to incomplete source and syntax errors. Approaches that use a parse tree, however, generally support the extraction of a wider range of (more precise) source models (e.g., program dependence graphs [Ferrante et al. 1987], def-use chains, etc.) than is possible using the lexical source model extraction technique.

The lexical source model extraction technique also does not require the specification of the parse tree or generation of a parser; a non-trivial exercise when a specification for an artifact of interest is not available. There have been several different approaches taken to address this problem. The Field [Reiss 1995] and Sniff [Bischofberger 1992] systems use approximate (fuzzy) parsers that perform a partial parse of the source code looking for particular syntactic constructs. Approximate parsers typically trade both the ability to create some source models, say def-use chains, and sometimes also the accuracy of extracted source models, for ease in specifying the parser. In contrast to my lexical technique, which makes similar trade-offs, the cost of building an approximate parser is generally more than a few hours.

Other systems, including Software Refinery's DIALECT, TXL, and the SOOP system [Gil and Lorenz 1994], have addressed the problem of creating a parse tree through parser generators. These approaches trade the cost of developing the necessary syntax and parse tree specifications for the ability to create more source models.

The GENOA system makes a similar tradeoff using a slightly different approach. A companion tool, called GENII, is provided to help interface an existing compiler front-end with the GENOA system. Although this simplifies the problem, the interface specifications are still fairly substantial:

From our experience, we estimate that interfacing to a well-documented front end for a fairly simple, typed algorithmic language like C or PASCAL would take a few days: far less time than it would take to implement a new front-end from scratch. [...] Generating an interface to a documented front-end for a complex language like C++, PL/1 or Ada would be proportionately more time-consuming. A rough rule-of-thumb would be that the time to write a GENII specification for an interface to a front end for a given language grows linearly with the size of the BNF specification of the grammar of the language. [Devanbu 1992, p. 311]

Whether the cost of creating appropriate specifications for generating a parse tree or interfacing to an existing front-end is warranted is dependent on the task, and anticipated future tasks, being performed by an engineer. The lexical source model extraction tech-

nique is not meant to replace these systems, but rather, to augment them by providing engineers with a means of evaluating whether a given type of source model information is useful for a particular kind of task, and to permit additional flexibility and tolerance for scanning new and different kinds of system artifacts, or artifacts that are not in the appropriate condition to parse.

### 10.3 Summarization Techniques

The use of the term “summarization” is also found in two other areas of computer science. In the artificial intelligence community, summarization refers to techniques that create narrative abstracts from textual artifacts, such as Maybury’s technique for technical articles [Maybury 1995]. Of the four properties of this kind of a summary proposed by Alterman [Alterman 1991], namely that a summary should reduce the workload for the understander over the text, should maintain coherence, should maintain coverage, and should include the important events of the story, the software reflexion model summarization technique maintains three: it reduces the workload for the software engineer over the source, it maintain coherence, and it maintains coverage by permitting the inclusion of all structural information extracted from the source. It is only the last property, which is specific to the summarization of text, that the software reflexion model technique does not uphold.

The term is also used in several program analysis techniques to refer to condensed information about program properties. In a program dependence graph, for instance, region nodes are used to “summarize the set of control conditions for a node and [to] group all nodes with the same set of control conditions together” [Ferrante et al. 1987, p. 326]. Another example is found in the program summary graph [Callahan 1988] in which interprocedural control flow is represented in a more compact way than previous approaches. These techniques use the term summarization as a synonym for abstraction rather than as the specific kind of process closely related to abstraction used in the software reflexion model technique and outlined in Chapter 1.

# Chapter 11

## Conclusion

Structure has been defined as “a set of interconnecting parts of any complex thing” [Oxford 1991]. Unlike a mechanical engineer who can often take a device apart and see how the parts fit together, a software engineer is faced with trying to comprehend systems that are “pure thought-stuff, [and] infinitely malleable” [Brooks 1986, p.12]. The motivation behind this dissertation has been to make it easier for a software engineer to take an existing software system apart to explore its structural features. In particular, this research has focused on helping an engineer explore the structure of large systems. It is anticipated that with the appropriate structural knowledge, an engineer can more effectively plan, assess, and execute changes to a software system.

My thesis has been that an approach based on summarization enables an engineer to flexibly explore the structure of large systems at reasonable cost, thus overcoming several limitations associated with existing approaches. Summarization involves the production of overviews of vast amounts of user-selected information in a timely manner. An overview produced by a summarization process lets detail of the underlying information show through as compared to an overview produced by an abstraction process that hides the underlying detail.

To support summarization in the context of software structure, I developed two techniques. The first technique, the *software reflexion model* technique, enables a software engineer to summarize, and later investigate, structural information about a software

system from the viewpoint of a selected high-level model. The second technique, the *lexical source model extraction* technique, supports the summarization process by facilitating access to structural information that is difficult or impossible to extract at low cost from system artifacts using existing tools. These techniques share two essential characteristics: each technique is *lightweight* and *iterative*. By lightweight, I mean that the effort required by the engineer to apply the technique as part of a task is low. By iterative, I mean that the engineer can initially produce partial and approximate results, and then may successively refine the inputs until the desired completeness and accuracy in the produced results is obtained. These characteristics allow the engineer to balance the cost of applying the techniques with the kind of information required to help perform a particular task. Being able to manage this trade-off permits an engineer to optimize the activities performed in the time available for a task.

I demonstrated the validity of my claims through several case studies spanning a variety of change tasks on several systems. In particular, the use of the software reflexion model technique by a software engineer at Microsoft Corporation to support an experimental reengineering of the million line Excel product indicates that the approach scales effectively.

## 11.1 Contributions

In addition to the development of the two techniques and the demonstration of the feasibility and usefulness of the summarization approach, this research makes three additional contributions.

First, the summarization approach as supported by the software reflexion model technique provides a means of *bridging the gap* between the high-level models commonly used by engineers to reason about a software system and the system artifacts that are the software system. In the context of this dissertation, closing this gap allows an engineer to produce a *task-specific* view of a software system to support the change process. More generally, closing this gap may permit engineers to produce documentation about

a system on-demand. The ability to produce documentation on-demand could have a long-term impact on the role that some forms of documentation play in the development process.

Second, the techniques and case studies presented in this dissertation demonstrate that *approximate* (non-conservative) structural information can be beneficial to an engineer planning, assessing, and executing change tasks on an existing system. The degree and kind of approximate information that is useful is dependent on the task that is being reasoned about. However, relaxing the requirement for conservative information may open up new approaches for accessing structural information from very large software systems.

Finally, the research has also confirmed the usefulness of *partial* approaches to accessing and understanding structural information. Permitting a software engineer to apply a structural understanding approach on parts of a large software system enables an engineer to focus on those pieces of a system that are pertinent to a task at hand. In addition, partial approaches help the engineer manage issues related to scale.

## 11.2 Future Work

Many questions remain to be answered about how to best support a software engineer in acquiring and using structural information as part of the change process. Several future research topics in this general direction are considered below, followed by a discussion of research topics related to the software reflexion model and the lexical source model extraction techniques. One central theme to these research directions is the importance of the software engineering task being performed in the development and assessment of techniques to aid a software engineer.

*Structural Information for Task.* Although many of the examples in this dissertation were based on structural information consisting of call interactions between software entities, software artifacts often contain richer forms of structural interaction such as data

flow and events. This data may be extracted either from scans of the static artifacts or by collecting information from executions of the system. Little is known about the kinds and styles of structural information that are pertinent and useful for reasoning about various classes of changes. More case studies are needed to improve the understanding of this correspondence. Additional data of this form would aid in developing guidance for software engineers trying to reason about a change task.

*Structural Extraction and Collection Techniques.* Conducting further case studies into the correspondence between structure and change, and supporting engineers in performing change, requires tools to extract a variety of kinds of structural information. The lexical source model extraction tool provides support for a new point in the design space of static analysis extractors. More experience is needed with this technique to understand the forms and accuracy of structural information that may be extracted in a timely manner. In addition, algorithms for some of the structural information of interest, such as data flow, do not currently scale for large systems written in common programming languages like C [Kernighan and Ritchie 1978]. One interesting research direction is to develop approximate, non-conservative, data flow algorithms and to determine if the algorithms provide information useful to engineers during the performance of a task. Finally, an analysis of the viability of existing monitoring tools is needed to assess the usefulness of structural information collected dynamically from an executing system to different parts of the change process.

*Assessing Approximation.* Both the software reflexion model and the lexical source model extraction techniques may produce approximate, non-conservative, results. As discussed in Section 8.2, several other software engineering techniques for extracting structure are also approximate, although this is not always evident. Approximation is also used in other areas of computer science such as in the development of practical algorithms for NP-complete problems [Garey and Johnson 1979]. For many of these

algorithms, theoreticians have been able to prove upper and lower bounds on the approximation of the results returned by the algorithm. Although bounds may be difficult to prove given the variation in systems and tasks, it would be helpful to provide methods and tools to enable an engineer to more easily assess the degree of approximation when using the software reflexion model and lexical source model extraction techniques for a particular task.

*Visualization.* There has been a tacit assumption in the software engineering research community that graphical displays are an integral part of program visualization and reverse engineering approaches. The experience of the Microsoft engineer in applying the software reflexion model technique raises questions about which portions of a summarization approach require this kind of visual support. This experience with summarization casts doubt on the validity of the graphical display assumption inherent in many existing techniques. Further experience with both textual and graphical forms of structural information, whether directly represented, abstracted, or summarized, is needed.

*What-If Tools for Planning.* The research undertaken in this dissertation is only a small step towards understanding the nature of the tools and information necessary to support an engineer in assessing and planning change. An interesting direction to explore in this area is the development of approximate what-if tools as a means for software engineers to investigate the impact of various ways of implementing proposed changes. Similar to the role approximation played in the techniques described in this dissertation to ease access to structural information, approximation might also be able to overcome limitations associated with access and scale that plague the use of existing what-if techniques.

### 11.2.1 Software Reflexion Models

Areas of future research involving the software reflexion model technique fall into two categories: extensions to the technique itself, and the use of the technique to investigate

issues related to the structure of systems.

*Extensions to the Technique.* There are several possible extensions to the technique that may broaden the applicability of the approach. For instance, would support for exceptions in the mapping—where an engineer characterizes source model entities to exclude from a general description of entities—significantly reduce the size of the map required for large systems? What kinds of software engineering tasks may benefit from comparisons based on a different form for the map, perhaps a map that permits the association of arcs to entities? Are there different mapping functions that would be more effective for other tasks? For instance, would a mapping function that has knowledge of the kind of structural information, say data-flow information, be able to more effectively summarize information in the context of a high-level model, perhaps by computing the summary with respect to the transitive closure? What are effective means for naming dynamic source model entities, like objects, with entities in a high-level model?

Additional investigation of the usability of the technique is also required to determine the appropriate kinds of interfaces for novice and expert users. In particular, more experience with the technique is necessary to understand the manipulations an engineer may wish to perform directly on a visualized reflexion model such as making small adjustments to the map by dragging entities. As with other extensions to the technique, usability issues are also likely related to the tasks that are being performed.

*Investigating System Structure with Reflexion Models.* The case studies reported in this dissertation demonstrate the benefits of applying the software reflexion model technique. In at least two of the cases reported, though, the map files created were non-trivial in size, ranging from a few hundred lines to about a thousand lines. One way to mitigate the cost of creating these files is to automatically produce the map file as was the case in analyzing *SPIN*. However, this file cannot often be produced automatically. Although the Microsoft case study shows that producing the map manually is not only feasible

but sometimes cost-effective, further benefits from the technique might be accrued if the maps were shown to be stable across versions of the system. Stable maps would allow the cost of initially developing large maps to be amortized over many uses of the software reflexion model technique. Some initial feedback from the Microsoft case study, in which reflexion models were computed for more than one version of the Excel source, indicate the map was reasonably stable. More experience is needed to assess the longer-term stability of the map's form and content.

Stability of the map would also enable the software reflexion model technique to be used to qualitatively assess the evolution of the structure of a system. Studying the evolution of a family of systems could provide the basis for characterizations of the change process and for the analysis of the benefits and limitations of different architectural approaches to system design.

### **11.2.2 Lexical Source Model Extraction**

The lexical source model extraction technique represents a useful point in the design space between existing lexical and syntactic approaches. This design point makes particular trade-offs between accuracy and the efficiency of developing a desired extractor. A more precise characterization of the design space for source model extractors is needed to understand other trade-offs that may prove beneficial to software engineers.

Using this characterization, it would be possible to determine if a more accurate version of the lexical source model extraction technique is warranted. A more accurate version could be developed by adding some recursive capabilities to the technique. Another way of achieving this goal might be to blend the lexical and syntactic approaches by using lexical patterns to help derive a more complete—perhaps abstract syntax form—of a program that may then be queried.

# Bibliography

- AHO, A., KERNIGHAN, B., AND WEINBERGER, P. 1979. Awk – a pattern scanning and processing language. *Software—Practice and Experience* 9, 4, 267–280.
- AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.
- AJILA, S. 1995. Software maintenance: An approach to impact analysis of objects change. *Software—Practice and Experience* 25, 10, 1155–1181.
- ALLEN, R. AND GARLAN, D. 1994. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 71–80.
- ALTERMAN, R. 1991. Understanding and summarization. *Artificial Intelligence Review* 5, 4, 239–254.
- ARNON, D. 1993. Scrimshaw: A language for document queries and transformations. In *EP '94: Proceedings of the Fifth International Conference on Electronic Publishing, Document Manipulation and Typography*. John Wiley & Sons Ltd., Chichester, U.K., 385–396.
- BAECKER, R. AND MARCUS, A. 1986. Design principles for the enhanced presentation of computer program source text. In *Proceedings of the Human Factors in Computing Systems (CHI '86)*. ACM, New York, NY, 51–58.
- BAECKER, R. AND MARCUS, A. 1990. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, MA.
- BALL, T. AND EICK, S. 1996. Software visualization in the large. *Computer* 29, 4, 33–43.
- BALZER, R. 1991. Tolerating inconsistency (software development). In *Proceedings of the 13th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 158–165.

BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety and performance in the SPIN operating system. *Operating Systems Review* 29, 5, 267–284.

BISCHOFBERGER, W. 1992. Sniff—a pragmatic approach to a C++ programming environment. In *Proceedings of the 1992 USENIX C++ Conference*. USENIX, Berkeley, CA, 67–81.

BLUM, B. 1989. Improving software maintenance by learning from the past: A case study. *Proceedings of the IEEE* 77, 4, 596–606.

BOBROW, D., DEMICHIEL, L., GABRIEL, R., KEENE, S., AND KICZALES, G. 1989. Common lisp object system specification. *Lisp and Symbolic Computation* 1, 3/4, 245–394. Also appears as X3J13 Document 88-002R.

BOEHM, B. 1976. Software engineering. *IEEE Transactions on Computers C-25*, 1226–1241.

BOEHM, B., BROWN, J., AND LIPOW, M. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 592–605.

BOOCH, G. 1993. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, Redwood City, CA.

BRACCI, G., PADINI, P., AND PELAGATTI, G. 1976. Binary logical associations in data modeling. In *Modelling in Data Base Management Systems*, G. NIJSSEN Ed., Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems. North-Holland, Amsterdam, Netherlands, 125–148.

BROCKSCHMIDT, K. 1995. *Inside OLE, Second Edition*. Microsoft Press, Redmond, WA.

BROOKS, F. 1986. No silver bullet—essence and accidents of software engineering. In *Information Processing 86*. Elsevier Science Publishers, Amsterdam, Netherlands, 1069–1076. Also appeared in *Computer* 20 4, 10–19. Page numbers quoted refer to the *Computer* article.

BURSON, S., KOTIK, G., AND MARKOSIAN, L. 1990. A program transformation approach to automating software re-engineering. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference*. IEEE Computer Society Press, Los Alamitos, CA, 314–322.

- CALLAHAN, D. 1988. The program summary graph and flow-sensitive interprocedural data flow analysis. *SIGPLAN Notices* 23, 7, 47–56.
- CANFORA, G., CIMITILE, A., AND MUNRO, M. 1993. A reverse engineering method for identifying reusable abstract data types. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 73–82.
- CARDELLI, L., DONAHUE, J., JORDAN, M., KALSOW, B., AND NELSON, G. 1989. The Modula-3 type system. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 202–212.
- CARMICHAEL, I., TZERPOS, V., AND HOLT, R. 1995. Design maintenance: Unexpected architectural interactions (experience report). In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, 134–137.
- CHEN, Y. 1995. Reverse engineering. In *Practical Reusable UNIX Software*, B. KRISHNAMURTHY Ed. John Wiley & Sons, Chichester, U.K., chap. 6.
- CHEN, Y.-F., FOWLER, G., KORN, D., KOUTSOFIOS, E., NORTH, S., ROSENBLUM, D., AND VO, K.-P. 1995. Intertool connections. In *Practical Reusable Unix Software*, B. KRISHNAMURTHY Ed. John Wiley & Sons, Chichester, U.K., chap. 11.
- CHEN, Y.-F., NISHIMOTO, M., AND RAMAMOORTHY, C. 1990. The C information abstraction system. *IEEE Transactions on Software Engineering SE-16*, 3, 325–334.
- CHIKOFSKY, E. AND CROSS II, J. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7, 1, 13–17.
- CLARKE, C. AND CORMACK, G. 1995. Context grep. Technical Report MT-95-02, Multitext Project, Department of Computer Science, University of Waterloo, Waterloo, Canada.
- CLEMENTS, P. 1996. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE Computer Society Press, Los Alamitos, CA, 16–25.
- CONSENS, M., MENDELZON, A., AND RYMAN, A. 1992. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*. ACM, New York, NY, 138–156.
- CORDY, J., HALPERN-HAMU, C., AND PROMISLOW, E. 1991. TXL: A rapid prototyping system for programming language dialects. *Computer Languages* 16, 1, 97–107.

CUSUMANO, M. AND SELBY, R. 1995. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, New York, NY.

DEAN, T. AND CORDY, J. 1995. A syntactic theory of software architecture. *IEEE Transactions on Software Engineering* 21, 4, 302–313.

DEVANBU, P. 1992. Genoa—a customizable, language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*. ACM, New York, NY, 307–317.

DEVANBU, P., BALLARD, B., BRACHMAN, R., AND SELFRIDGE, P. 1991. *LaSSIE: A Knowledge-Based Software Information System*, Chapter 2, 25–40. AAAI Press/The MIT Press, Cambridge, MA.

DEVANBU, P., BRACHMAN, R., SELFRIDGE, P., AND BALLARD, B. 1991. Lassie: A knowledge-based software information system. *Communications of the ACM* 34, 5, 34–49.

DUNN, M. AND KNIGHT, J. 1993. Automating the detection of reusable parts in existing software. In *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 381–390.

FELDMAN, S. I. 1979. Make — A program for maintaining computer programs. *Software—Practice and Experience* 9, 3, 255–265.

FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3, 319–349.

GANSNER, E., NORTH, S., AND VO, K. 1988. Dag—a program that draws directed graphs. *Software—Practice and Experience* 18, 11, 1047–1062.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability*. W.H. Freeman and Company, New York, N.Y.

GARLAN, D. AND NOTKIN, D. 1991. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the Fourth International Symposium of VDM Europe*, Volume 551 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 31–44.

GIL, J. AND LORENZ, D. 1994. SOOP—a synthesizer of an object-oriented parser. Technical Report 9404, Technion—Israel Institute of Technology, Technion City, Israel.

- GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.
- GPO. 1983. *Reference manual for the Ada programming language*. MIL STD 1815A. United States Government Printing Office, Washington, D.C.
- GRAHAM, S., KESSLER, P., AND MCKUSICK, M. 1982. Gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '92 Symposium on Compiler Construction*. ACM, New York, NY, 120–126.
- GRISWOLD, R. AND GRISWOLD, M. 1983. *The Icon Programming Language*. Prentice Hall, Englewood Cliffs, N.J.
- GRISWOLD, W. AND ATKINSON, D. 1995. Managing the design tradeoffs for a program understanding and transformation tool. *Journal of Systems and Software* 30, 1-2, 99–116.
- GRISWOLD, W., ATKINSON, D., AND MCCURDY, C. 1996. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the Fourth Workshop on Program Comprehension*. IEEE Computer Society Press, Los Alamitos, CA, 144–153.
- GRISWOLD, W. AND NOTKIN, D. 1995. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Transactions on Software Engineering* 21, 4, 257–275.
- HARRIS, D., REUBENSTEIN, H., AND YEH, A. 1995. Reverse engineering to the architectural level. In *Proceedings of the 17th International Conference on Software Engineering*. ACM, New York, NY, 186–195.
- HEIMANN, D. 1995. Using complexity-tracking in software development. In *Proceedings of the Annual Reliability and Maintainability Symposium*. IEEE, New York, NY, 433–438.
- HENRY, S. AND KAFURA, D. 1981. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering SE-7*, 5, 509–518.
- HOWDEN, W. AND WIEAND, B. 1994. QDA—a method for systematic informal program analysis. *IEEE Transactions on Software Engineering* 20, 6, 445–462.
- HUTCHENS, D. AND BASILI, V. 1985. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering SE-11*, 8, 749–757.
- JACKSON, D. 1995. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology* 4, 2, 109–145.

- JACKSON, D. AND LADD, D. 1994. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, 243–25.
- JAVEY, S., MITSUI, K., NAKAMURA, H., OHIRA, T., YASUKA, K., KUSE, K., KAMIMURA, T., AND HELM, R. 1992. Architecture of the XL C++ browser. In *Proceedings of the 1992 CAS Conference*. IBM Canada Ltd. Laboratory, Center for Advanced Studies, Toronto, Canada, 369–379.
- JOHNSON, S. 1975. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J.
- KAFURA, D. AND REDDY, G. 1987. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering SE-13*, 3, 335–343.
- KEARNS, S. 1991. TLex. *Software—Practice and Experience* 21, 8, 805–821.
- KELLER, R., CAMERON, M., TAYLOR, R., AND TROUP, D. 1991. User interface development and software environments: The Chiron-1 system. In *Proceedings of the 13th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 208–218.
- KERNIGHAN, B. AND RITCHIE, D. 1978. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J.
- LADD, D. AND RAMMING, J. 1995. A\*: A language for implementing language processors. *IEEE Transactions on Software Engineering* 21, 11, 894–901.
- LAKHOTIA, A. 1996. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*. To appear.
- LEHMAN, M. 1974. Programs, cities, students—limits to growth. Inaugural lecture series volume 9, Imperial College of Science and Technology, London, U.K. Reprinted in *Programming Methodology*, D. Gries (ed), Springer Verlag, New York, 1978 and in M.M. Lehman, L.A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 7, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.
- LEHMAN, M. AND BELADY, L. 1978. Laws of program evolution—rules and tools for programming management. In *Infotech State of the Art Conference*. Pergamon Press. Reprinted in M.M. Lehman, L.A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 12, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

- LEHMAN, M. AND BELADY, L. 1980. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE Special Issue on Software Engineering 68*, 9, 1060–1076. Reprinted in M.M. Lehman, L.A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 19, APIC Studies in Data Processing No. 27. Academic Press, London, 1985. Page number in citation refers to reprinted version.
- LESK, M. 1975. Lex—a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J.
- LINDVALL, M. 1994. A study of traceability in object-oriented systems development. Ph. D. thesis, Linköping University, Department of Computer and Information Science, Linköping, Sweden.
- LINTON, M. 1983. Queries and views of programs using a relational database. Ph. D. thesis, University of California, Berkeley, Berkeley, CA.
- LINTON, M. 1984. Implementing relational views of programs. *SIGPLAN Notices 19*, 5, 132–140. Also appeared in the Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.
- MAYBURY, M. 1995. Generating summaries from event data. *Information Processing & Management 31*, 5, 735–751.
- MCCABE, T. 1976. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4, 308–320.
- MCCLURE, C. 1978. A model for program complexity analysis. In *Proceedings of the 3rd International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 149–157.
- MEYER, B. 1985. The software knowledge base. In *Proceedings of the 8th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 158–165.
- MEYER, B. 1992. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, U.K.
- MORICONI, M. AND HARE, D. 1985. Visualizing program designs through PegaSys. *Computer 18*, 8, 72–85.
- MORICONI, M., QIAN, X., AND RIEMENSCHNEIDER, R. 1995. Correct architecture refinement. *IEEE Transactions on Software Engineering 21*, 4, 356–372.
- MÜLLER, H. AND KLASHINSKY, K. 1989. A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 80–86.

- MÜLLER, H. A. AND UHL, J. S. 1990. Composing subsystem structures using (k,2)-partite graphs. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, 12–19.
- MURPHY, G., NOTKIN, D., AND LAN, E.-C. 1996. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 90–99.
- MURPHY, G., NOTKIN, D., AND SULLIVAN, K. 1995. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 18–28.
- MURPHY, G. C. AND NOTKIN, D. 1995. Lightweight Source Model Extraction. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 116–127.
- ORNBURN, S. AND RUGABER, S. 1992. Reverse engineering: Resolving conflicts between expected and actual software designs. In *Proceedings of the 1992 IEEE Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, 32–40.
- OSSHER, H. 1987. A Mechanism for Specifying the Structure of Large, Layered Systems. In *Research Directions in Object-Oriented Programming*, B. SHRIVER AND P. WEGNER Eds. The MIT Press, Cambridge, MA, 219–252.
- OSSHER, H. L. 1984. A new program structuring mechanism based on layered graphs. Ph. D. thesis, Stanford University, Palo Alto, CA.
- OUSTERHOUT, J. 1994. *TCL and the TK Toolkit*. Addison-Wesley, Reading, MA.
- Oxford. 1991 *Concise Oxford Dictionary—Eighth Edition*. On-line version. Oxford University Press, Oxford, U.K.
- PARNAS, D. 1994. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 279–287.
- PARNAS, D. L. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15, 12, 1053–1058.
- PAUL, S. AND PRAKASH, A. 1992. Source code retrieval using program patterns. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 95–105.

- PENNY, D. 1993. The software landscape: A visual formalism for programming-in-the-large. Ph. D. thesis, University of Toronto, Toronto, Canada.
- PERRY, D. E. AND WOLF, A. 1992. Foundations for the study of software architecture. *Software Engineering Notes* 17, 4, 40–52.
- PFLEEGER, S. AND BOHNER, S. 1990. A framework for software maintenance metrics. In *Proceedings of the IEEE Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, 320–327.
- PIRNIA, S. AND HAYEK, M. 1981. Requirements definition approach for an automated requirements traceability tool. In *Proceedings of the IEEE 1981 National Aerospace and Electronics Conference (NAECON '81)*, Volume 1. IEEE, New York, NY, 389–394.
- PREMERLANI, W. AND BLAHA, M. 1994. An approach for reverse engineering of relational databases. *Communications of the ACM* 37, 5, 42–49.
- PRICE, B., BAECKER, R., AND SMALL, I. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing* 4, 3, 211–266.
- QUILICI, A. 1994. A memory-based approach to recognizing programming plans. *Communications of the ACM* 37, 5, 84–93.
- RABIN, M. AND SCOTT, D. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 3, 2, 114–125.
- REISS, S. 1990. Connecting tools using message passing in the field program development environment. *IEEE Software* 7, 4, 57–66.
- REISS, S. 1995. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Amsterdam, Netherlands.
- REISS, S. P. 1993. A framework for abstract 3d visualization. In *IEEE Symposium on Visual Languages*. IEEE Computer Society Press, Los Alamitos, CA, 108–115.
- REUBENSTEIN, H., PIAZZA, R., AND ROBERTS, S. 1993. Separating parsing and analysis in reverse engineering tools. In *Proceedings of Working Conference on Reverse Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 117–125.
- RICH, C. AND WATERS, R. 1990. *The Programmer's Apprentice*. Addison-Wesley, Reading, MA.
- RISHE, N. 1985. A relational database design methodology using binary conceptual schema. Technical report, University of California, Santa Barbara, Santa Barbara, CA.

- RUMBAUGH, J. 1990. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, N.J.
- RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, N.J.
- SCHEIFLER, R. AND GETTYS, J. 1986. The x window system. *ACM Transactions on Graphics* 5, 2, 79–109.
- SCHNEIDEWIND, N. 1989. Software maintenance: The need for standardization. *Proceedings of the IEEE* 77, 4, 618–624.
- SCHWANKE, R., ALTUCHER, R., AND PLATOFF, M. 1989. Discovering, visualizing, and controlling software structure. In *Proceedings of Fifth International Workshop on Software Specification and Design*. IEEE Computer Society Press, Washington, D.C., 147–150. Also appeared in *Tagungsband Kuenstliche Intelligenz in der Praxis*, Deimens AG, December 1988.
- SEFIKA, M., SANE, A., AND CAMPBELL, R. 1996. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 387–396.
- SELBERG, E. AND ETZIONI, O. 1995. Multi-Service Search and Comparison Using the MetaCrawler. In *Proc. 4th World Wide Web Conference*. Boston, MA USA.
- SHAW, M., DELINE, R., KLEIN, D. V., ROSS, T. L., YOUNG, D. M., AND ZELESNIK, G. 1995. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering* 21, 4, 314–335.
- SPIVEY, J. 1992. *The Z Notation* (second ed.). Prentice Hall, Hemel Hempstead, U.K.
- STEFFEN, J. 1985. Interactive Examination of a C Program with Cscope. In *Proceedings of the USENIX Winter Conference*. USENIX, Berkeley, CA, 170–175.
- STEVENS, W., MYERS, G., AND L., C. 1974. Structured design. *IBM Systems Journal* 13, 2, 115–139.
- STROUSTRUP, B. 1986. *C++ Programming Language*. Addison-Wesley, Reading, MA.
- SULLIVAN, K. J. 1994. Mediators: Easing the design and evolution of integrated software systems. Ph. D. thesis, University of Washington, Seattle, WA.
- TEITELMAN, W. AND MASINTER, L. 1981. The Interlisp Programming Environment. *IEEE Computer* 14, 4, 25–33.

- TILLEY, S. 1995. Domain-retargetable reverse engineering. Ph. D. thesis, University of Victoria, Victoria, Canada.
- VON MAYRHAUSER, A. AND MANS, A. 1996. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering* 22, 6, 424–437.
- VON MAYRHAUSER, A. AND VANS, A. 1995. Industrial experience with an integrated code comprehension model. *Software Engineering Journal* 10, 5, 171–182.
- WALL, L. 1990. *Programming Perl*. O'Reilly & Associates, Sebastopol, CA.
- WARD, M., CALLIS, F., AND MUNRO, M. 1989. The maintainer's assistant. In *Proceedings of the 1989 Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, 307–315.
- WONG, K., TILLEY, S., MÜLLER, H., AND STOREY, M. 1995. Structural redocumentation: A case study. *IEEE Software* 12, 1, 46–54.
- WU, S. AND MANBER, U. 1992. Agrep—a fast approximate pattern-matching tool. In *Proceedings of the USENIX Winter 1992 Technical Conference*. USENIX, Berkeley, CA, 153–162.
- YAU, S. AND COLLOFELLO, J. 1980. Some stability measures for software maintenance. *IEEE Transactions on Software Engineering SE-6*, 545–552.
- YIN, R. 1984. *Case Study Research*. Sage Publications, Beverly Hills, CA.
- YOURDON, E. AND CONSTANTINE, L. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice Hall, Englewood Cliffs, N.J.

# Appendix A

## A Short Overview of Z

This appendix provides a synopsis of the Z notation [Spivey 1992] used in this dissertation. The intent of this appendix is not to completely describe the notation, but to provide enough background for a reader to understand the Z descriptions of the reflexion model technique. For a full description of the notation, the reader is directed to the language's reference manual [Spivey 1992] or any of a number of publications on the language.

### A.1 Data Types and Schemas

A Z specification models a software system as a set of mathematical data types on which operations are performed. A basic data type is introduced in square brackets. For example,

$$[NAME]$$

introduces a basic data type called NAME.

Constructed data types are introduced using abbreviations. An abbreviation defines a global constant. For instance,

$$NameRelation == NAME \leftrightarrow NAME$$

introduces the name *NameRelation* as an abbreviation for the set of relations between values of type NAME.

Global functions are defined using axiomatic descriptions. An axiomatic description introduces a global variable and, optionally, provides constraints on the values of the variable. For instance, the axiomatic description

$$\left| \text{square} : \mathbb{N} \rightarrow \mathbb{N} \right.$$

declares a function named *square* with a particular type signature. When constraints are provided, they follow the declaration as shown below.

$$\left| \begin{array}{l} \text{square} : \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall x : \mathbb{N} \bullet \text{square}(x) = x * x \end{array} \right.$$

Here, the constraint—the predicate after the dividing line—defines the value of the function for all possible inputs; given a natural number, the *square* function returns the square of that value.

Basic types, abbreviations, and axiomatic descriptions often help in the definition of schemas. The core of a *Z* specification consists of several schemas linked by a natural language commentary. A number of different syntactic conventions are available for defining schemas. In this dissertation, all schemas are defined using the open box notation as shown below.

$$\boxed{\begin{array}{l} \text{SampleSystem} \\ \text{friends} : \text{NameRelation} \\ \hline \text{dom friends} \neq \emptyset \end{array}}$$

The above construct defines a static schema named *SampleSystem*. Static schemas define the states a system can occupy and the invariants that must be maintained as the system transitions between states. The schema above defines the state space for the system as consisting of one relation named *friends* of type *NameRelation* with the invariant that the domain of the *friends* relation is never the empty set.

Dynamic schemas define operations that may be performed on the system, the relationship between the inputs and the outputs of the operation, and the changes of state.

In a dynamic schema, the part after the dividing line defines preconditions and postconditions on the operation. An example of a dynamic schema named *SampleOperation* is given below.

<i>SampleOperation</i>	_____
$\Xi$ <i>SampleSystem</i>	
<i>inputVar?</i> : <i>NAME</i>	
<i>outputVar!</i> : <i>NAME</i>	
_____	
<i>inputVar?</i> $\in$ <i>dom friends</i>	
<i>outputVar!</i> = <i>inputVar?</i>	

In this schema, the first predicate after the dividing line is a precondition stating that the value of *inputVar?* must be in the domain of the *friends* relation, and the second predicate states a postcondition that the value of *outputVar!* is the value of *inputVar?*. By convention, variables terminating in a ? are input variables and variables terminating in a ! are output variables. Note the  $\Xi$  *SampleSystem* construct in the first line of the schema. This indicates that the operation does not affect the state of the system. The  $\Delta$  *SampleSystem* construct is used to indicate that an operation does affect the state of the system.

## A.2 The Meaning of Symbols

Many symbols are used in the Z schemas in this dissertation. This section summarizes the symbols used and provides a short description of the meaning of each symbol.

*Set Notation.* Table A.1 summarizes the set notation used in the schemas in this dissertation. Most of this notation is likely familiar to readers. One notation that may not be familiar is the set comprehension expression. To clarify this notation, consider the following simple example.

$$\{x : \mathbb{N} \mid x \bmod 2 = 0 \bullet x * x\}$$

This expression is interpreted as follows. The part before the | symbol declares variables; for instance, x is declared to be a variable of the natural number type. The part after the

| symbol introduces predicates constraining the values of declared variables; for instance,  $x$  must be an even number. Finally, the part after the  $\bullet$  symbol defines an expression; here, the expression is the square of the value of  $x$ . The value of the set comprehension expression is thus the set of the squares of all positive natural numbers.

Table A.1: Z Set Notation Used in the Dissertation.

$\mathbb{P}$	Power Set
$\times$	Cartesian Product
$\emptyset$	Empty Set
$\in$ and $\notin$	Membership and Non-membership
$\subseteq$	Subset Relation
$\setminus$	Set Difference
$\cup$ and $\cap$	Set Union and Set Intersection
$\bigcup$	Generalized Union
$ \bullet$	Set Comprehension

*Relation Notation.* Table A.2 summarizes the relational notation used in this dissertation.

Table A.2: Z Relation Notation Used in the Dissertation.

$\leftrightarrow$	A Binary Relation
dom and ran	The Domain and Range of a Relation
$(\mid)$	Relational Image

*Function, Sequence, and Bag Notation.* A function is defined in Z using the  $\rightarrow$  notation.

For example,

$$| \text{square} : \mathbb{N} \rightarrow \mathbb{N}$$

defines a function named square that takes a natural number and returns another natural number.

A sequence, an ordered list of items, is defined using  $\text{seq}$ . For example,

$$\mid x : \text{seq } \mathbb{N}$$

defines  $x$  to be a sequence of natural numbers.

A bag, a collection of items in which the number of times an item occurs is significant, is defined using  $\text{bag}$ . For example,

$$\mid x : \text{bag } \mathbb{N}$$

defines  $x$  to be a bag of natural numbers. The notation “in” tests an item for membership in a given bag. For example, the phrase “3 in  $x$ ” tests if 3 is a member of the bag  $x$ . A bag is formally defined as shown below.

$$\text{bag } X == X \rightarrow \mathbb{N}_1$$

The domain of a bag thus refers to all elements placed into the bag. (Note that an element is not stored in a bag if there are no occurrences of the element in the bag.)

*Other Notation.* Table A.3 summarizes a variety of other notation used in the dissertation.

Table A.3: Other Z Notation Used in the Dissertation.

=	Equality
$\wedge$ and $\vee$	Conjunction and Disjunction
$\mathbb{N}$	Natural Numbers
<b>let</b> •	Local Definition. The scope of variables declared after the <b>let</b> extends to the predicate defined after •. In essence, the construct defines a predicate whose value is true only if the predicate introduced after the • is true for all values of the local variables.
if then else	Conditional Expression
$\forall$ •	Universal quantifier. The value after the • is a predicate.
$\forall$   •	Universal Schema Quantifier. Variables are introduced before the   symbol. Constraints on variables are defined between the   and • symbols. A schema expression follows the • symbol.
$\exists$   •	Universal Existential Quantifier. Variables are introduced before the   symbol. Constraints on variables are defined between the   and • symbols. A schema expression follows the • symbol.

## Appendix B

# A Formal Characterization of Typed Software Reflexion Models

This appendix provides a Z specification of the typed software reflexion model technique described in Chapter 4. This specification supersedes the untyped reflexion model specification provided in Section 2.2 since an untyped reflexion model may be treated as a typed reflexion model where an arbitrary, but uniform, type is assigned to every arc in the source and high-level models. This specification may be helpful to readers implementing software reflexion model tools because it makes precise the computation of a reflexion model and the operations supported by the technique for the various combinations of typed and untyped high-level and source models.

### B.1 Typed Reflexion Model

Any system implementing the typed reflexion model technique must maintain the state components and invariants described in the *TypedReflexionModel* schema presented below.

The *TypedReflexionModel* schema uses three basic types. As in the specification for an untyped reflexion model (Section 2.2.1), *HLEMENTITY* represents the type of a high-level

model entity, and *SMENTITY* represents the type of a source model entity. *ARCTYPE* represents the type of an interaction between source model entities or high-level model entities and is assumed to describe a set of values that includes all user-defined types plus a special value indicating the lack of a user-defined value (i.e., the “untyped” value).

[*HLEMENTITY*, *SMENTITY*, *ARCTYPE*]

Nine type synonyms are also used. As in the untyped reflexion model specification, the two synonyms, *HLMTuple* and *SMTuple*, represent the type of an interaction between high-level model entities and source model entities respectively.

*HLMTuple* == *HLEMENTITY* × *HLEMENTITY*  
*SMTuple* == *SMENTITY* × *SMENTITY*

The synonyms, *TypedHLMRelation* and *TypedSMRelation*, define relations over high-level model entities and source model entities that include, for each tuple, an indication of the type of the interaction represented by the tuple. The *TypedHLMTuple* and *TypedSMTuple* synonyms define the types of the tuples in the *TypedHLMRelation* and in the *TypedSMRelation* respectively.

*TypedHLMTuple* == *ARCTYPE* × *HLMTuple*  
*TypedHLMRelation* == *ARCTYPE* ↔ *HLMTuple*  
*TypedSMTuple* == *ARCTYPE* × *SMTuple*  
*TypedSMRelation* == *ARCTYPE* ↔ *SMTuple*

The *TypedMappedSM* synonym defines a relation that describes which source model values contribute to an arc within a computed reflexion model, including an indication of the type of each source model value. This synonym is based on the *MappedTuple* synonym that defines the form of a tuple relating a source model value to a high-level model interaction, and the *TypedMappedTuple* synonym that associates a type with a mapped tuple.

*MappedTuple* == *HLMTuple* × *SMTuple*  
*TypedMappedTuple* == *ARCTYPE* × *MappedTuple*  
*TypedMappedSM* == *ARCTYPE* ↔ *MappedTuple*

To ease the comparison of values of the different types described above in predicates within the schemas, several helper functions are also defined. The first three functions

provide access to the type information for values of the *TypedSMTuple*, *TypedHLMTuple*, and *TypedMappedTuple* types respectively.

$$\frac{\text{typeOfSMTuple} : \text{TypedSMTuple} \rightarrow \text{ARCTYPE}}{\forall s : \text{TypedSMTuple} \bullet \text{typeOfSMTuple}(s) = (\text{first } s)}$$

$$\frac{\text{typeOfHLMTuple} : \text{TypedHLMTuple} \rightarrow \text{ARCTYPE}}{\forall h : \text{TypedHLMTuple} \bullet \text{typeOfHLMTuple}(h) = (\text{first } h)}$$

$$\frac{\text{typeOfMappedTuple} : \text{TypedMappedTuple} \rightarrow \text{ARCTYPE}}{\forall m : \text{TypedMappedTuple} \bullet \text{typeOfMappedTuple}(m) = (\text{first } m)}$$

The next two functions provide access to the tuple information stored within a typed tuple. The first function accesses the source model interaction information from a value of type *TypedSMTuple*. The second function accesses the information concerning an interaction between high-level model entities from a value of type *TypedHLMTuple*.

$$\frac{\text{tupleOfSMTuple} : \text{TypedSMTuple} \rightarrow \text{SMTuple}}{\forall s : \text{TypedSMTuple} \bullet \text{tupleOfSMTuple}(s) = (\text{second } s)}$$

$$\frac{\text{tupleOfHLMTuple} : \text{TypedHLMTuple} \rightarrow \text{HLMTuple}}{\forall h : \text{TypedHLMTuple} \bullet \text{tupleOfHLMTuple}(h) = (\text{second } h)}$$

The last two functions provide access to the source model and high-level model interaction information within a value of type *TypedMappedTuple*.

$$\frac{\text{smTupleOfMappedTuple} : \text{TypedMappedTuple} \rightarrow \text{SMTuple}}{\forall m : \text{TypedMappedTuple} \bullet \text{smTupleOfMappedTuple}(m) = (\text{second } (\text{second } m))}$$

$$\frac{\text{hlmTupleOfMappedTuple} : \text{TypedMappedTuple} \rightarrow \text{HLMTuple}}{\forall m : \text{TypedMappedTuple} \bullet \text{hlmTupleOfMappedTuple}(m) = (\text{first } (\text{second } m))}$$

Similar to the untyped schema presented in Section 2.2.1, the *TypedReflexionModel* schema defines six variables that must be maintained by a typed reflexion model system. The *rmEntities* variable, as in the untyped reflexion model schema, describes the entities

in a computed reflexion model. As before, the *convergences* relation defines where a high-level model agrees within a given source model; the *divergences* relation defines where a source model differs from a high-level model; and the *absences* relation defines where a high-level model differs from a source model. Also as before, *mappedSourceModel* is a relation describing which source model values contribute to a convergent or divergent arc in the reflexion model, and *smBag* is a bag that describes the number of occurrences of each source model value. In contrast to the untyped reflexion model specification, each of these variables also includes type information.

<i>TypedReflexionModel</i>	
<i>rmEntities</i> : $\mathbb{P} \text{HLMENTITY}$	
<i>convergences</i> : <i>TypedHLMRelation</i>	
<i>divergences</i> : <i>TypedHLMRelation</i>	
<i>absences</i> : <i>TypedHLMRelation</i>	
<i>mappedSourceModel</i> : <i>TypedMappedSM</i>	
<i>smBag</i> : bag <i>TypedSMTuple</i>	
<i>convergences</i> $\cap$ <i>divergences</i> = $\emptyset$	
<i>divergences</i> $\cap$ <i>absences</i> = $\emptyset$	
<i>convergences</i> $\cap$ <i>absences</i> = $\emptyset$	
$\forall m : \text{TypedMappedTuple} \mid m \in \text{mappedSourceModel} \bullet$	(4)
$\exists s : \text{TypedSMTuple} \mid s \text{ in } \text{smBag} \bullet$	
$\text{typeOfMappedTuple}(m) = \text{typeOfSMTuple}(s) \wedge$	
$\text{smTupleOfMappedTuple}(m) = \text{tupleOfSMTuple}(s);$	
$\forall h : \text{TypedHLMTuple} \mid (h \in (\text{convergences} \cup \text{divergences})) \bullet$	(5)
$\exists m : \text{TypedMappedTuple} \mid m \in \text{mappedSourceModel} \bullet$	
$\text{typeOfMappedTuple}(m) = \text{typeOfHLMTuple}(h) \wedge$	
$\text{hlmTupleOfMappedTuple}(m) = \text{tupleOfHLMTuple}(h)$	
$\text{dom}(\text{ran}(\text{convergences} \cup \text{divergences} \cup \text{absences})) \subseteq \text{rmEntities}$	(6)
$\text{ran}(\text{ran}(\text{convergences} \cup \text{divergences} \cup \text{absences})) \subseteq \text{rmEntities}$	(7)

Seven constraints are also defined on the schema.<sup>1</sup> The first three constraints state that the values of the *convergences*, *divergences*, and *absences* relations are disjoint. The fourth constraint ensures that all source model values named in the *mappedSourceModel* relation are defined in the source model (specified by *smBag*). The fifth constraint ensures that any tuple in the *convergences* and *divergences* relations has associated

<sup>1</sup>As in Chapter 2, some predicates are numbered on the right hand side of the schema for ease of reference. This is a slight deviation from the standard Z schema layout.

source model values defined within the *mappedSourceModel* relation by checking that for each tuple in the *convergences* and *divergences* relations, there exists a tuple in the *mappedSourceModel* relation that names the same interaction between high-level model entities. This ensures that the investigation of convergent and divergent arcs within a computed typed reflexion model is defined. The sixth and seventh predicates ensure that all arcs in a typed reflexion model are between entities defined in the *rmEntities* set.

## B.2 Computing a Typed Reflexion Model

The dynamic schema, *ComputeTypedReflexionModel*, presented below describes the computation of a typed reflexion model from a high-level model, a source model, and a mapping from the source to the high-level model. The high-level model is described by two variables: the *hlmEntities?* variable describes the set of entities in the high-level model, and the *hlm?* variable describes a relation over high-level model entities including associated type information. Splitting the description of the high-level model across these two variables permits the definition of high-level models with entities that are not part of any interaction. The source model (*sm?*) is described as a bag of typed source model tuples. As before, the mapping (*map?*) is specified as an ordered list of map entries. Each map entry names zero or more source model entities and associates them with one or more high-level model entities. *SMENTITYDESC* again represents the type of a description naming zero or more source model entities, and a function, *mapFunc*, is again required that matches entities from the source model to the specified map, producing a set of associated high-level model entities. The definition of *SMENTITYDESC* and *mapFunc* are unchanged from the specification given for untyped reflexion models in Chapter 2.

[*SMENTITYDESC*]

*MapEntry* == *SMENTITYDESC* × ( $\mathbb{P}$  *HLMENTITY*)

| *mapFunc* : (seq *MapEntry* × *SMENTITY*) → ( $\mathbb{P}$  *HLMENTITY*)

The first predicate after the dividing line is a pre-condition relating the two variables describing the high-level model. Specifically, the predicate states that the high-level

model entities set (*hlmEntities?*) may be a superset of the entities that are part of the given high-level model relation (*hlm?*), but all entities named in the high-level model relation must be part of the *hlmEntities?* set. In addition, the high-level model entities named in the *map* are constrained by the second pre-condition to be a subset of the *hlmEntities?* set.

<i>Compute Typed Reflexion Model</i>	
$\Delta$ <i>Typed Reflexion Model</i>	
<i>hlm?</i> : <i>Typed HLM Relation</i>	
<i>hlmEntities?</i> : $\mathbb{P}$ <i>HLM ENTITY</i>	
<i>sm?</i> : <i>bag SM Typed Tuple</i>	
<i>map?</i> : <i>seq Map Entry</i>	
$(\text{dom}(\text{ran } hlm?) \cup \text{ran}(\text{ran } hlm?)) \subseteq hlmEntities?$	(1)
$\bigcup(\text{ran}(\text{ran } map?)) \subseteq hlmEntities?$	(2)
$\forall s : \text{Typed SM Tuple} \mid s \text{ in } sm? \bullet \text{unTyped}(\text{typeOf SM Tuple}(s)) = 0$	(3)
$\forall h : \text{Typed HLM Tuple} \mid h \in \text{extractUntypedHLM}(hlm?) \bullet$ $\neg (\exists i : \text{Typed HLM Tuple} \mid i \in (hlm? \setminus \{h\}) \bullet$ $\text{tupleOfHLM Tuple}(h) = \text{tupleOfHLM Tuple}(i))$	(4)
<i>rmEntities'</i> = <i>hlmEntities?</i>	(5)
<i>smBag'</i> = <i>sm?</i>	(6)
<i>mappedSourceModel'</i> = $\bigcup(\{s : \text{Typed SM Tuple} \mid s \text{ in } smBag' \bullet$ $\{\text{typeOf SM Tuple}(s)\} \times$ $\text{mapFunc}(map?, \text{fromEntityOf SM Tuple}(s)) \times$ $\text{mapFunc}(map?, \text{toEntityOf SM Tuple}(s)) \times$ $\{\text{tupleOf SM Tuple}(s)\})$	(7)
<i>convergences'</i> = $\{h : \text{Typed HLM Tuple}; m : \text{Typed Mapped Tuple} \mid$ $h \in hlm? \wedge m \in \text{mappedSourceModel}' \wedge$ $\text{typeMatch}(\text{typeOfHLM Tuple}(h), \text{typeOfMapped Tuple}(m)) > 0 \wedge$ $\text{tupleOfHLM Tuple}(h) = \text{hlm TupleOfMapped Tuple}(m) \bullet$ $(\text{typeOfMapped Tuple}(m), \text{tupleOfHLM Tuple}(h))\}$	(8)
<i>divergences'</i> = $\{m : \text{Typed Mapped Tuple} \mid$ $m \in \text{mappedSourceModel}' \wedge$ $(\text{hlm TupleOfMapped Tuple}(m) \notin \text{ran } hlm?) \vee$ $(\text{hlm TupleOfMapped Tuple}(m) \in \text{ran } hlm?) \wedge$ $(\forall c : \text{Typed HLM Tuple} \mid c \in \text{convergences}' \bullet$ $(\text{typeMatch}(\text{typeOfMapped Tuple}(m),$ $\text{typeOfHLM Tuple}(c)) = 0))\} \bullet$ $(\text{typeOfMapped Tuple}(m), \text{hlm TupleOfMapped Tuple}(m))\}$	(9)
<i>absences'</i> = $\{h : \text{Typed HLM Tuple} \mid$ $h \in hlm? \setminus \text{convergences}' \wedge$ $((\text{unTyped}(\text{typeOfHLM Tuple}(h)) = 0) \vee$ $((\text{unTyped}(\text{typeOfHLM Tuple}(h)) > 0) \wedge$ $\neg (\exists c : \text{Typed HLM Tuple} \mid c \in \text{convergences}' \bullet$ $\text{tupleOfHLM Tuple}(h) = \text{tupleOfHLM Tuple}(c)))\} \bullet$ $h\}$	(10)
$h\}$	(11)

Another pre-condition on the computation, given by the third predicate after the dividing line, ensures that all source model relation values have a user-defined type. This predicate uses the *unTyped* function that takes a type value and returns a positive

integer if the value is the special value that indicates the lack of a user-defined type.

$$\left| \text{unTyped} : \text{TYPE} \rightarrow \mathbb{N} \right.$$

The final pre-condition, given by the fourth predicate, ensures that untyped interactions are only specified between high-level model entities that are not related by any typed interaction (see Section 4.1.2). This pre-condition uses the *extractUntypedHLM* function that returns the subset of a given high-level model relation that is untyped.

$$\left| \begin{array}{l} \text{extractUntypedHLM} : \text{TypedHLMRelation} \rightarrow \text{TypedHLMRelation} \\ \hline \forall hlm : \text{TypedHLMRelation} \bullet \text{extractUntypedHLM}(hlm) = \{h : \text{TypedHLMTuple} \mid \\ h \in hlm \wedge \text{unTyped}(\text{typeOfHLMTuple}(h)) > 0 \bullet h\} \end{array} \right.$$

As before, the fifth predicate sets the entities of the computed reflexion model to be the entities of the high-level model, and the sixth predicate sets the value of *smBag'* to the input source model *sm?*.

The value of *mappedSourceModel* (the seventh predicate) may then be computed by pushing the elements of each source model tuple through the map, resulting in two sets of high-level model entities. The cross-product of these sets of high-level model entities, the type of the source model tuple, and the source model tuple itself is then computed, and the resultant tuples are added to the *mappedSourceModel* relation. Two functions, *fromEntityOfSMTuple* and *toEntityOfSMTuple*, are used in this computation to provide access to the elements of each source model tuple.

$$\left| \begin{array}{l} \text{fromEntityOfSMTuple} : \text{TypedSMTuple} \rightarrow \text{SMENTITY} \\ \hline \forall s : \text{TypedSMTuple} \bullet \text{fromEntityOfSMTuple}(s) = (\text{first}(\text{second } s)) \end{array} \right.$$

$$\left| \begin{array}{l} \text{toEntityOfSMTuple} : \text{TypedSMTuple} \rightarrow \text{SMENTITY} \\ \hline \forall s : \text{TypedSMTuple} \bullet \text{toEntityOfSMTuple}(s) = (\text{second}(\text{second } s)) \end{array} \right.$$

The value of the *convergences* relation (the eighth predicate) is computed by determining, for each arc in the high-level model, if an appropriately typed similar arc appears in the computed *mappedSourceModel* relation. If the high-level model arc is typed, an appropriately typed similar arc is one with the same type between the same high-level

model entities. If the high-level model arc is untyped, any arc regardless of type between the same high-level model entities in the *mappedSourceModel* relation is considered appropriate. The types of the arcs from the high-level model and the *mappedSourceModel* relation are compared using the *typeMatch* function. This function takes two types as arguments, returning a positive integer if the types are compatible, and zero if the two types are not compatible. Two type values are defined to be compatible if they are equal, if one of the two values is the special value indicating the lack of a defined type, or if both values are the special undefined type value.

$$\mid \text{typeMatch} : (\text{ARCTYPE} \times \text{ARCTYPE}) \rightarrow \mathbb{N}$$

When computing the *convergences* relation, if the type of the high-level model arc is the special value indicating an untyped arc, the type associated with the arc will be the type from the *mappedSourceModel* value. Since the type of the *mappedSourceModel* value is induced by types in the source model, this value cannot be the special undefined type value.

A mapped arc between two high-level model entities (i.e., the arc is in the domain of the range of *mappedSourceModel*) is added to the *divergences* relation (the ninth predicate) if either the arc, ignoring types, is not part of the high-level model, or the arc is part of the *convergences'* relation but no arc between the same high-level model entities in the *convergences'* relation has the same type as the type of the mapped arc.

The value of the *absences* relation (the final predicate) includes all typed arcs that are in the high-level model but not in the computed *convergences'* relation, and all untyped arcs in the high-level model for which there is no computed typed convergent arc between the same high-level model entities.

### B.3 Displaying a Typed Reflexion Model

The dynamic schema, *ComputeTypedReflexionModelArcValue*, describes an operation to determine the numeric label of an arc in a computed typed reflexion model. Similar

to the same operation for an untyped reflexion model, this operation takes one input, the arc in the reflexion model ( $rmArc?$ ), and produces one numeric output value in the  $arcVal!$  variable. This operation may be invoked with each computed reflexion model arc in turn to determine the information needed to visually display a typed reflexion model.

As before, the operation also requires a function,  $addArcCounts$ . This function is slightly modified from before; given a set of typed source model tuples and the bag, the function returns the sum of the number of times each tuple appears in the source model. If the specified set of source model tuples is empty,  $addArcCounts$  returns zero.

|  $addArcCounts : (TypedSMRelation \times (bag\ TypedSMTuple)) \rightarrow \mathbb{N}$

$\frac{\text{Compute Typed Reflexion Model Arc Value}}{\exists Typed Reflexion Model}$ $rmArc? : TypedHLM Tuple$ $arcVal! : \mathbb{N}$ <hr style="border: 0.5px solid black;"/> $rmArc? \in (convergences \cup divergences \cup absences)$ $\mathbf{let\ currentArc\ Values} == \{m : TypedMapped Tuple \mid$ $m \in mappedSourceModel \wedge$ $typeMatch(typeOfMapped Tuple(m), typeOfHLM Tuple(rmArc?)) > 0 \wedge$ $hlm TupleOfMapped Tuple(m) = tupleOfHLM Tuple(rmArc?) \bullet$ $(typeOfMapped Tuple(m), sm TupleOfMapped Tuple(m))\} \bullet$ $arcVal! = addArcCounts(currentArc Values, smBag?)$
---

This operation has one pre-condition. The arc for which the value is desired must be an arc in the computed reflexion model (the first predicate after the dividing line).

The operation returns the result of applying the  $addArcCounts$  function to the set of typed source model values mapped to the specified arc by the  $mappedSourceModel$  relation.

#### B.4 Tagging a Reflexion Model Arc

The tagging operation is described by two dynamic schemas. The first schema,  $TagArc$ , describes an operation that records, amongst other information, the source model values contributing to an arc “tagged” by an engineer. This operation is invoked when an

engineer selects an arc in a displayed reflexion model for tagging. The second schema, *ApplyTagsToArc*, applies previously recorded tagged information for an arc to the display of a recomputed reflexion model.

#### B.4.1 Tagging an Arc

The *TagArc* schema takes one input, *rmArc?*, the arc to tag, and produces four outputs. The first output, *taggedArcValues!*, is the set of typed source model values contributing to the specified arc. The second through four outputs are copies of the *convergences*, *divergences*, and *absences* relations respectively.

The only pre-condition to the operation is that the arc to tag is an arc in the reflexion model. The value of *taggedArcValues!* is computed from the *mappedSourceModel* relation by extracting the source model values and associated types mapped to the specified arc.

<i>TagArc</i>
$\exists$ <i>TypedReflexionModel</i> <i>rmArc?</i> : <i>TypedHLM Tuple</i> <i>taggedArcValues!</i> : $\mathbb{P}$ <i>TypedSM Tuple</i> <i>previousConvergences!</i> : <i>TypedHLM Relation</i> <i>previousDivergences!</i> : <i>TypedHLM Relation</i> <i>previousAbsences!</i> : <i>TypedHLM Relation</i>
<i>rmArc?</i> $\in$ ( <i>convergences</i> $\cup$ <i>divergences</i> $\cup$ <i>absences</i> ) <i>previousConvergences!</i> = <i>convergences</i> <i>previousDivergences!</i> = <i>divergences</i> <i>previousAbsences!</i> = <i>absences</i> <i>taggedArcValues!</i> = { <i>m</i> : <i>TypedMapped Tuple</i>   <i>m</i> $\in$ <i>mappedSourceModel</i> $\wedge$ <i>typeMatch</i> ( <i>typeOfMapped Tuple</i> ( <i>m</i> ), <i>typeOfHLM Tuple</i> ( <i>rmArc?</i> )) $>$ 0 $\wedge$ <i>hlm TupleOfMapped Tuple</i> ( <i>m</i> ) = <i>tupleOfHLM Tuple</i> ( <i>rmArc?</i> ) $\bullet$ ( <i>typeOfMapped Tuple</i> ( <i>m</i> ), <i>sm TupleOfMapped Tuple</i> ( <i>m</i> )) }

#### B.4.2 Applying Tags to a Reflexion Model Arc

The *ApplyTagsToArc* operation takes five input values. The *rmArc?* variable defines the arc in the reflexion model that has been previously tagged. The *taggedArcValues?* variable is a relation output from the *TagArc* operation that defines the source model

values associated with the reflexion model interaction in the previously computed reflexion model. The *previousConvergences?* relation, *previousDivergences?* relation, and *previousAbsences?* relation describe the previously computed reflexion model; these relations were recorded when the specified arc was tagged.

<i>ApplyTagsToArc</i>	
$\Delta$ <i>TypedReflexionModel</i>	
<i>rmArc?</i> : <i>TypedHLM Tuple</i>	
<i>taggedArcValues?</i> : $\mathbb{P}$ <i>TypedSMTuple</i>	
<i>previousConvergences?</i> : <i>TypedHLMRelation</i>	
<i>previousDivergences?</i> : <i>TypedHLMRelation</i>	
<i>previousAbsences?</i> : <i>TypedHLMRelation</i>	
<i>rmArc?</i> $\in$ ( <i>convergences</i> $\cup$ <i>divergences</i> $\cup$ <i>absences</i> )	(1)
<i>previousConvergences?</i> $\cap$ <i>previousDivergences?</i> = $\emptyset$	(2)
<i>previousDivergences?</i> $\cap$ <i>previousAbsences?</i> = $\emptyset$	(3)
<i>previousConvergences?</i> $\cap$ <i>previousAbsences?</i> = $\emptyset$	(4)
<b>let</b> <i>currentArcValues</i> == { <i>m</i> : <i>TypedMapped Tuple</i>	
<i>typeMatch</i> ( <i>typeOfMapped Tuple</i> ( <i>m</i> ), <i>typeOfHLM Tuple</i> ( <i>rmArc?</i> )) > 0	
$\wedge$ <i>hlm TupleOfMapped Tuple</i> ( <i>m</i> ) = <i>tupleOfHLM Tuple</i> ( <i>rmArc?</i> ) •	
( <i>typeOfMapped Tuple</i> ( <i>m</i> ), <i>sm TupleOfMapped Tuple</i> ( <i>m</i> )) } •	
<i>convergences'</i> =	
if <i>taggedArcValues?</i> = <i>currentArcValues</i> $\wedge$ <i>rmArc?</i> $\in$ <i>convergences</i> $\wedge$	
<i>rmArc?</i> $\in$ <i>previousConvergences?</i> then	
<i>convergences</i> $\setminus$ { <i>rmArc?</i> }	
else <i>convergences</i> ) $\wedge$	
<i>divergences'</i> =	
if <i>taggedArcValues?</i> = <i>currentArcValues</i> $\wedge$ <i>rmArc?</i> $\in$ <i>divergences</i> $\wedge$	
<i>rmArc?</i> $\in$ <i>previousDivergences?</i> then	
<i>divergences</i> $\setminus$ { <i>rmArc?</i> }	
else <i>divergences</i> ) $\wedge$	
<i>absences'</i> =	
if <i>currentArcValues</i> = $\emptyset$ $\wedge$ <i>rmArc?</i> $\in$ <i>absences</i> $\wedge$	
<i>rmArc?</i> $\in$ <i>previousAbsences?</i> then	
<i>absences</i> $\setminus$ { <i>rmArc?</i> }	
else <i>absences</i> )	(5)

The first pre-condition defined on the operation ensures that the interaction is still defined in the current reflexion model. The next three predicates ensure the description of the previous reflexion model is well-defined.

The last (fifth) predicate computes new values for the *convergences*, *divergences*, and *absences* relations of the reflexion model. The local variable, *currentArcValues*, is a

relation that includes the source model values contributing to the arc in the newly computed reflexion model. This set is compared to the specified source model values in *taggedArcValues?* to determine if the specified interaction should be removed from the appropriate reflexion model relation. An interaction may only be removed if it is still of the same designation as in the previous reflexion model (i.e., if the interaction was previously a convergence, it must still be a convergence, etc.) and if the source model values associated with the arc have not changed.

### B.5 Annotating a Reflexion Model Arc

The *AnnotateTypedReflexionModelArc* dynamic Z schema defines the annotation operation. The operation uses a new basic type, *ANNOTATIONDESC*, which describes a set of source model values contributing to a particular reflexion model arc together with a textual description.

[*ANNOTATIONDESC*]

The operation requires one input, *annotation?*, which is an annotation description for one particular reflexion model arc. The operation produces an output value in the *annotateVal!* variable which is the annotation value to display for the reflexion model arc described in the annotation. Applying the operation does not affect the state of the reflexion model. This operation may be invoked once for each annotation description.

The operation requires two functions. The *applyAnnotation* function, given an annotation and a mapped source model, computes the contributing source model values described by the annotation. The second function, *addArcCounts*, is the same function required by the *ComputeTypedReflexionModelArcValue* schema. This function, when given a set of typed source model tuples and the source model, sums the number of times each tuple appears in the source model and produces the result.

$\begin{array}{l} \text{Annotate TypedReflexionModelArc} \\ \exists \text{ TypedReflexionModel} \\ \text{annotation?} : \text{ANNOTATIONDESC} \\ \text{annotateVal!} : \mathbb{N} \\ \text{applyAnnotation} : (\text{ANNOTATIONDESC} \times \text{TypedMappedSM}) \rightarrow \text{TypedSMRelation} \\ \text{addArcCounts} : (\text{TypedSMRelation} \times (\text{bag TypedSMTuple})) \rightarrow \mathbb{N} \end{array}$	
$\text{annotateVal!} = \text{if } \text{applyAnnotation}(\text{annotation?}, \text{mappedSourceModel}) = \emptyset \text{ then } 0 \\ \text{else } \text{addArcCounts}(\text{applyAnnotation}(\text{annotation?}, \text{mappedSourceModel}), \text{smBag?})$	

The predicate (after the dividing line) describes that the output value of the operation is zero if the annotation does not describe any values in the *mappedSourceModel*, otherwise, the value is the sum of the number of times the contributing source model values described by the annotation appear in the source model.

## B.6 Families of Typed Reflexion Models

This Z specification, similar to the Z specification for the untyped reflexion models, defines a family of reflexion model systems. As before, different kinds of systems result depending on the choices made for representing the source model entity descriptions in a map (*SMENTITYDESC*) and for defining the mapping function (*mapFunc*). For instance, *SMENTITYDESC* and *mapFunc* could be defined to consider type information. The specification also defines a family of annotation operations that depend on the representation of the ANNOTATIONDESC type.

## Appendix C

# Lexical Source Model Extraction Patterns

This appendix contains several patterns for the lexical source model extraction technique described in Chapter 6 and Chapter 7.

### C.1 A Pattern for Identifying Functions in C

In Chapter 6, a lexical source model extraction pattern was defined to find function definitions within K&R C [Kernighan and Ritchie 1978] source code. Figure C.1 includes a refinement of the pattern defined in Section 6.2.1. This pattern was used to find all function definitions from the 18,000 lines of C, yacc and lex code comprising the cross-reference tool of the Field software system that were also reported by running the Field tool over the same software.

The scanner generated from the pattern in Figure C.1 extracted, from the post-processed source code, 278 function definitions also reported by the Field tool. The scanner also extracted one extraneous function definition consisting of empty brackets; this definition is easily identified and filtered from the final source model.

When the pre-processed source code is provided as input to the same generated scanner, 99% (276 of 278 function definitions) of all functions definitions are extracted. In this case, the scanner misses the definition of a `yyparse` function because it is surrounded by

```

[ (type) ] (functionName) @
  if functionName == "if" | functionName == "switch" |
    functionName == "while" | functionName == "forin" then
    fail
  write ( functionName )
@
\ ( [ (formalArg) [ { , (formalArg) }+ ] ] \ )
[ { (type) [ { (mod) }+ ] (argDecl) [ { , (argDecl) }+ ] ; }+ ] \ {

```

Figure C.1: C Function Definition Pattern.

preprocessor symbols and the definition of `XRDB_scan_string`, a function defined with variable arguments. Three extraneous function definitions were also reported.

## C.2 Patterns for Identifying Calls in Non-Preprocessed C Source

The patterns in Figure C.2 were used to extract calls from non-preprocessed C source code comprising the Field [Reiss 1990; Reiss 1995] software system. The performance of an extractor generated from these patterns is described in Table 7.2 as “LSME (2 patterns, no cpp)”. The `writeCall` function referenced in the action code writes out the function call information to the output stream.

## C.3 Patterns for Identifying Calls in Preprocessed C Source

The patterns in Figure C.3 were used to extract calls from postprocessed C source code comprising the Field [Reiss 1990; Reiss 1995] software system. The performance of an extractor generated from these patterns is described in Table 7.2 as “LSME (3 patterns, cpp)”. The `writeCall` function referenced in the action code writes out the function call information to the output stream. The `keywordq` function returns true if the given value is one of `if`, `while`, `switch`, `for`, or `typedef`. The `operatorq` function returns true if the given value is a common operator such as `<<=` or an arithmetic operator.

- (1) [ <type> ] <functionName> @  
     if functionName == "if" | functionName == "switch" |  
     functionName == "while" | functionName == "forin" |  
     functionName == "for" then  
         fail @  
 \([ [ <formalArg> [ { , <formalArg> }+ ] ] \)  
 [ { <type> [ { <mod> }+ ] <decl> [ { , <decl> }+ ] ; }+ ] \{
- (2)     <calledFuncName> @  
        writeCall ( functionName, calledFuncName ) @  
 \([ ( <arg> | { <embeddedCall> } @  
     writeCall ( functionName, embeddedCall ) @  
        \([ [ <arg> [ { , <arg> }+ ] ] \) } )  
     [ { , ( <arg> | { <embeddedCall> } @  
        writeCall ( functionName, embeddedCall ) @  
        \([ [ <arg> [ { , <arg> }+ ] ] \) } ) }+ ] ] \)

Figure C.2: Patterns for Extracting Calls from Non-Preprocessed C Source. The second pattern is a child of the first pattern.

- (1) [ <type> ] <functionName> @  
     if keywordq(functionName) | operatorq(functionName) then  
         fail  
 \([ [ { <formalArg> }+ ] \) [ { { <type> }+ ; }+ ] \{
- (2) <calledFuncName> @  
     if keywordq(calledFuncName) | operatorq(calledFuncName) then  
         fail @  
     writeCall ( functionName, calledFuncName )  
     fail @  
 \([ [ {  
 ( <arg> | { <embeddedCall> } @  
     writeCall ( functionName, embeddedCall )  
     fail @  
 \([ [ { ( <arg> ) | { <embeddedCall> } @  
     writeCall ( functionName, embeddedCall )  
     fail @  
 \([ [ { ( <arg> ) [ , ] }+ ] \) } )  
 [ , ] }+ ] \) } )  
 [ , ] }+ ] \)
- (3) <calledFuncName> @  
     if keywordq(calledFuncName) | operatorq(calledFuncName) then  
         fail  
     writeCall ( functionName, calledFuncName )  
     fail @  
 \([ [ { ( <arg> ) | { \([ [ { <arg> }+ ] \) } ) [ , ] }+ ] \)

Figure C.3: Patterns for Extracting Calls from Postprocessed C Source. The second and third patterns are each children of the first pattern.

## Vita

Gail Cecile Murphy was born in Edmonton, Alberta, Canada on May 12, 1965. Gail received her B.Sc.(Honors) with First Class Standing in Computing Science from the University of Alberta in Edmonton in 1987. From 1987 to 1992, she worked as a software designer at MPR Teltech in Burnaby, B.C., Canada. In 1992, she started graduate studies at the University of Washington in Seattle, completing a Master of Science degree in 1994. After completing her doctoral studies at the University of Washington in 1996, she joined the University of British Columbia in Vancouver B.C. as an Assistant Professor in the Department of Computer Science.