This is to certify that the

dissertation entitled

INTEGRATING INFORMAL AND FORMAL TECHNIQUES TO REVERSE ENGINEER IMPERATIVE PROGRAMS

presented by

GERALD CATOLICO GANNOD

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Betty H-C Cheng
Major professor

Date 8/7/98

INTEGRA
RE

INTEGRATING INFORMAL AND FORMAL TECHNIQUES TO REVERSE ENGINEER IMPERATIVE PROGRAMS

By

Gerald Catolico Gannod

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Department of Computer Science

1998

INTEGRATIN

REVER

Many we

be constructe

these spaceca

be not were th

order.

The failure

Inadequate ev

be used for sp

considered

Reverse en

all compone

Documentation

section of a

have been

is applied to

ABSTRACT

INTEGRATING INFORMAL AND FORMAL TECHNIQUES TO REVERSE ENGINEER IMPERATIVE PROGRAMS

By

Gerald Catolico Gannod

Many well-documented computer failures have been attributed to software. Some of the most notable incidents include the catastrophic failures of the Therac-25 [1] and the Ariane 5 spacecraft [2]. A commonly overlooked aspect of these failures has been the fact that both were the result of an improper reengineering of software from one version to another.

The failure to correctly analyze software in both the Therac-25 and Ariane 5 resulted in catastrophic events that led to loss of life and property. These examples vividly illustrate the need for sophisticated and systematic methods for maintaining software in order to understand their functionality.

Reverse engineering of program code is the process of examining components and component interrelationships in order to construct a high-level abstraction of an implementation [3]. *Reengineering* is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [3]. Software reengineering is considered to be a better solution for handling legacy code as opposed to developing software from the original requirements. Since much of the

[illegible] of the e

[illegible] for many r

This research [illegible]

[illegible] and sc

[illegible] of a-

[illegible] predic

[illegible]

[illegible] This

[illegible]

functionality of the existing software has been achieved over a period of time, it must be preserved for many reasons, including providing continuity to current users of the software.

This research focuses on three primary contributions to the areas of software engineering and software maintenance. First, we have developed a technique for the construction of as-built formal specifications from program code using the strongest postcondition predicate transformer. Second, we have developed a formal technique for introducing abstraction into as-built specifications with the intent of obtaining design level specifications. Third, we have used the formal technique to support program understanding and software reuse activities.

To my parents

ACKNOWLEDGMENTS

This work was supported in part by the NASA Graduate Student Research Fellowship NGT-70376 and the National Science Foundation grants CCR-9633391, CCR-9407318, CCR-9209873, and CDA-9312389.

First, I would like to thank my family for always providing support and for reminding me that I will never be as "smart" as I think I am. Several people at the Jet Propulsion Laboratory provided assistance and valuable insight during my visits to the lab. Among these people are John Kelly, Ron Slusser, Martin Feather, Lan Tran, Richard Santiago, Rick Covington, Al Nikora, and Dorothy Huffman. In addition, many of the participants of the NASA Formal Methods Working Group provided advice and comments during group teleconferences including Robyn Lutz, Mike Lowry, Judy Crow, Jack Callahan, and Ben DiVito. I'm sure I have missed some people, and for that I apologize. However, be aware that I greatly appreciate the input provided by this group of people.

I would also like to thank the past and present members of the Software Engineering Network Systems (SENS) Research Group (formerly known as the Software Engineering Research Group). The long discussions and interaction will surely be missed. I am indebted to John Kelly, Bryan Pijanowski, Diane Rover, and Anthony Wojcik for their willingness

to serve as members of my dissertation committee, and to Linda Moore for always bailing me out of trouble when the university came calling.

I would be committing a crime if I did not mention my advisor, Betty Cheng. Thank you for all your support in the past several years. Words can barely express the gratitude and thanks I owe you. I can only hope that I can be as positive a force in the lives of students as you have been in mine.

Several friends need to be recognized for understanding when I couldn't break away from my schedule to see them. Specific names would be silly, so I'll stick to naming them by the groups that I have always known them. "SIGAWOTS", my thoughts and prayers will always be with you wherever it is you may be. To the "Koinonia" class, thanks for always feeding us when we needed to be fed. You've all helped me to set a new standard for my life here at the crossroads. Thanks to the "Llamas" for always providing a venue for chillin'. Finally, thanks to the "Dread Poets" for helping me to realize that while the cause may be just, its the experience that makes the difference.

Last, but not least, I would like to thank my wife, Barbie. Your love and support for everything that I do is unequaled by anyone I've ever known. Everyday I thank God that we found each other and it is clear to me that you, truly, "are the best".

LIST OF TABLES

LIST OF FIGURES

1 Introduction
1.1 Problem Descrip...
1.2 Contributions . . .
1.3 Organization of D...

2 Background
2.1 Software Mainten...
2.2 Formal Methods . ...
2.3 Informal Method...

3 Using Strongest Po...
3.1 Basic Constructs ...
3.2 Iterative and Proc...
3.3 Example . . . . .

4 Strongest Postcon...
4.1 Pointers . . . . .
4.2 Pointer Semantic...
4.3 Examples . . . .

5 Application of Str...
5.1 Assignment . . .
5.2 Alternation . . .
5.3 Current Express...
5.4 Sequence . . . .
5.5 Iteration . . . . .
5.6 Functions . . . .
5.7 Procedural Abst...

6 Design Abstractio...
6.1 Specification M...
6.2 Abstraction Ma...
6.3 Specification Ge...
6.4 Application to a...

TABLE OF CONTENTS

LIST OF TABLES xi

LIST OF FIGURES xii

1 Introduction 1
1.1 Problem Description and Motivation 1
1.2 Contributions 4
1.3 Organization of Dissertation 6

2 Background 7
2.1 Software Maintenance 7
2.2 Formal Methods 11
2.3 Informal Methods 17

3 Using Strongest Postcondition to Reverse Engineer Programs 20
3.1 Basic Constructs 20
3.2 Iterative and Procedural Constructs 27
3.3 Example 35

4 Strongest Postcondition Semantics of Pointers 39
4.1 Pointers 39
4.2 Pointer Semantics 40
4.3 Examples 53

5 Application of Strongest Postcondition to C Programs 59
5.1 Assignment 59
5.2 Alternation 62
5.3 Circuit Expressions 64
5.4 Sequence 66
5.5 Iteration 67
5.6 Functions 70
5.7 Procedural Abstractions 72

6 Design Abstractions 76
6.1 Specification Matching and Software Reuse 76
6.2 Abstraction Matching 78
6.3 Specification Generalization 85
6.4 Application to a JPL Ground-based Flight System 94

7 Reverse Engineering
7.1 Combining Informa
7.2 An Example . . . .

8 Tool Support
8.1 Overview . . . . .
8.2 AUTOSPEC . . . .
8.3 SPECGEN . . . .
8.4 SPECEDIT . . . . .
8.5 Theorem Prover . .
8.6 Formula Class Libra

9 Application of Reverse
9.1 Overview . . . . .
9.2 A Software Reverse E
9.3 Example . . . . . .

10 Related Work
10.1 Introduction . . . . .
10.2 Background . . . . .
10.3 Taxonomy . . . . .
10.4 Semantic Dimension
10.5 A Representative Tw
10.6 Comparison . . . .

11 Case Study
11.1 Overview . . . . .
11.2 Project-Specific Pr
11.3 High-Level Analy
11.4 Low-Level Analy
11.5 Formal Analysis .
11.6 Discussion . . . .
11.7 Lessons Learned

12 Conclusions and F
12.1 Summary of Cur
12.2 Future Investigat

A Semantics of C Exp
A.1 Assignment Operat
A.2 Logical Operators .
A.3 Bitwise Operators .
A.4 Equality and Relation
A.5 Shift Operators . . .
A.6 Additive and Multiplica

7 Reverse Engineering Framework **103**
7.1 Combining Informal and Formal Approaches 103
7.2 An Example 109

8 Tool Support **119**
8.1 Overview 119
8.2 AUTOSPEC 121
8.3 SPECGEN 129
8.4 SPECEDIT 133
8.5 Theorem Prover 136
8.6 Formula Class Library 139

9 Application of Reverse Engineering to Support Software Reuse **142**
9.1 Overview 142
9.2 A Software Reverse Engineering and Reuse Framework 143
9.3 Example 146

10 Related Work **159**
10.1 Introduction 159
10.2 Background 160
10.3 Taxonomy 163
10.4 Semantic Dimensions 168
10.5 A Representative Tools Survey 172
10.6 Comparison 189

11 Case Study **199**
11.1 Overview 199
11.2 Project-Specific Process 202
11.3 High-Level Analysis 204
11.4 Low-Level Analysis 210
11.5 Formal Analysis 217
11.6 Discussion 223
11.7 Lessons Learned 225

12 Conclusions and Future Investigations **227**
12.1 Summary of Contributions 227
12.2 Future Investigations 229

A Semantics of C Expressions **234**
A.1 Assignment Operators 234
A.2 Logical Operators 235
A.3 Bitwise Operators 235
A.4 Equality and Relational Operators 236
A.5 Shift Operators 237
A.6 Additive and Multiplicative Operators 237

B Partial Order Lemmas **239**
B.1 Lemma 1 239
B.2 Lemma 2 240
B.3 Lemma 3 242

C Application Program **243**

D Software Reuse Specifications **251**
D.1 As-built specification for the `Queue` source code 251
D.2 Circular Queue Library Specification 253

E `process_mnemonic_input` Source Code **254**

BIBLIOGRAPHY **258**

2.1 Properties of the
2.2 Properties of the
2.3 Pre/Post Match Cond

4.1 Pointer Assignment

5.1 Evaluation of A on
5.2 A Taxonomy of Pr

6.1 Weakening the pos

10.1 Tool Index . . . .
10.2 Comparison of Co
10.3 Comparison of Re
10.4 Comparison of Co
10.5 Comparison of Re
10.6 Comparison of Co
10.7 Comparison of Re

A.1 Bitwise Operative
A.2 Logical Operators
A.3 Bitwise Operator
A.4 Equality and Rel
A.5 Shift Operators
A.6 Additive and M

LIST OF TABLES

2.1 Properties of the *wp* predicate transformer 13
2.2 Properties of the *sp* predicate transformer 15
2.3 Pre/Post Match Criterion 17

4.1 Pointer Assignments 43

5.1 Evaluation of $\mathcal{A}$ on sample C assignment operators 61
5.2 A Taxonomy of Programming Language Functions 71

6.1 Weakening the postcondition 89

10.1 Tool Index 192
10.2 Comparison of Commercial Tools by informational criterion 193
10.3 Comparison of Research Tools by informational criterion 194
10.4 Comparison of Commercial Tools by By-products 195
10.5 Comparison of Research Tools by By-products 196
10.6 Comparison of Commercial Tools by evaluational criterion 197
10.7 Comparison of Research Tools by evaluational criterion 198

A.1 Bitwise Operative Assignment Operators 235
A.2 Logical Operators 236
A.3 Bitwise Operators 236
A.4 Equality and Relational Operators 237
A.5 Shift Operators 238
A.6 Additive and Multiplicative Operators 238

2.1 G: as an abstractio
2.2 Reverse Engineerin
2.3 Black box represen
2.4 OMT Summary

3.1 Annotated Source C
3.2 Strategy for annotat
3.3 Removal of pr ced
3.4 Code annotation fo
3.5 User Consultation

4.1 A simple pointer ex
4.2 Cell Memory Mod
4.3 Pointer Extensions
4.4 The points-to relat
4.5 The reset function
4.6 Three Sample Prog
4.7 Output of AUTOSP
4.8 AUTOSPEC applied
4.9 AUTOSPEC applied

5.1 An Assignment sta
5.2 Removal of proced
5.3 Code annotation fo

6.1 Syntax of Library
6.2 Square Root Speci
6.3 Square Root Libra
6.4 Specification Gene
6.5 Bubble Sort Progra
6.6 Bubble Sort Specif
6.7 Bubble Sort Specif
6.8 Bubble Sort Specif
6.9 Code Sequence Li
6.10 Annotation Abstra
6.11 Code annotation L

7.1 Translate Source C

LIST OF FIGURES

2.1 $G1$ as an abstraction of $G2$ 10
2.2 Reverse Engineering Process Model 11
2.3 Black box representation and differences between *wp* and *sp*: (a) *wp* (b) *sp* . . . 16
2.4 OMT Summary 18

3.1 Annotated Source Code for Unrolled Loop 30
3.2 Strategy for constructing a specification for an iteration statement 32
3.3 Removal of procedure call $p(\overline{a}, \overline{b}, \overline{c})$ abstraction 36
3.4 Code annotation for procedure call 37
3.5 User Consultation 38

4.1 A simple pointer example 42
4.2 Cell Memory Model 43
4.3 Pointer Extensions to the Memory Model 45
4.4 The *points-to* relation 46
4.5 The *coset* function 46
4.6 Three Sample Programs: (a) `alias` (b) `manyvars` (c) `maxThresh` 54
4.7 Output of AUTOSPEC applied to Figure 4.1 55
4.8 AUTOSPEC applied to the `manyvars` program 56
4.9 AUTOSPEC applied to the `maxThresh` program 58

5.1 An Assignment statement as a guard 62
5.2 Removal of procedure call abstraction: (a) before (b) after 74
5.3 Code annotation for procedure calls 75

6.1 Syntax of Library Specifications 81
6.2 Square Root Specification Library "Sqr" 82
6.3 Square Root Library as a partial order 83
6.4 Specification Generalization 86
6.5 Bubble Sort Program Annotated by AUTOSPEC 88
6.6 Bubble Sort Specification Brute Force Abstraction 91
6.7 Bubble Sort Specification Abstraction (postcondition) 92
6.8 Bubble Sort Specification after deletion of "a" and "b" 95
6.9 Code Sequence: Lines 108–135 99
6.10 Annotation Abstractions 100
6.11 Code annotation: Lines 404–420 101

7.1 Translate Source Code 110

72 Translate . . . . . .
73 Translate Source Co
74 Process Binary Out

81 Tool Suite . . . . . .
82 Level 0 AUTOSPEC
83 Level 1 Data Flow D
84 AUTOSPEC Main W
85 AUTOSPEC A Selec
86 Launching SPECEDI
87 Level 0 SPECGEN M
88 Level 1 SPECGEN M
89 SPECGEN Interface
810 Level 0 SPECEDIT M
811 Level 1 SPECEDIT M
812 SPECEDIT . . . . .
813 Level 0 TPROVER M
814 Level 1 TPROVER M
815 Example TPROVER
816 OMT model of the A

91 The Reverse Engine
92 Reverse Engineering
93 Software Reuse Fra
94 Queue Source Code
95 Circular Queue Diag
96 Output generated by
97 The enqueue ensu
98 SPECGEN Interface
99 The enqueue abo
910 Architecture of a so
911 Architecture specifi
912 Component Matchin
913 Wrappers generated

101 A Taxonomy of Rev
102 Precision Hierarchy
103 Example Plan . . .

111 Steps for Preparing
112 Communicator and P
113 Command Translato
114 Command Translato
115 Major Data Structure
116 Project Files Model f
117 Command Translato

7.2 Translate . . . 111
7.3 Translate Source Code . . . 111
7.4 Process Binary Output Source Code . . . 116

8.1 Tool Suite . . . 120
8.2 Level 0 AUTOSPEC Model . . . 122
8.3 Level 1 Data Flow Diagram of AUTOSPEC . . . 123
8.4 AUTOSPEC Main Window . . . 126
8.5 AUTOSPEC A Selection in the Main Window . . . 127
8.6 Launching SPECEDIT from AUTOSPEC . . . 128
8.7 Level 0 SPECGEN Model . . . 129
8.8 Level 1 SPECGEN Model . . . 130
8.9 SPECGEN Interface and Output . . . 132
8.10 Level 0 SPECEDIT Model . . . 134
8.11 Level 1 SPECEDIT Model . . . 135
8.12 SPECEDIT . . . 136
8.13 Level 0 TPROVER Model . . . 137
8.14 Level 1 TPROVER Model . . . 138
8.15 Example TPROVER Session . . . 140
8.16 OMT model of the *Formula* class library . . . 141

9.1 The Reverse Engineering and Reuse Framework . . . 144
9.2 Reverse Engineering Component . . . 145
9.3 Software Reuse Framework . . . 147
9.4 Queue Source Code . . . 148
9.5 Circular Queue Diagram . . . 148
9.6 Output generated by AUTOSPEC for the enQueue procedure . . . 150
9.7 The enQueue ensures clause in a conjunctive form . . . 151
9.8 SPECGEN Interface and Output . . . 152
9.9 The enQueue abstraction . . . 153
9.10 Architecture of a solution to Josephus problem . . . 155
9.11 Architecture specification . . . 156
9.12 Component Matching . . . 157
9.13 Wrappers generated by ABRIE for resolving naming conflicts . . . 158

10.1 A Taxonomy of Reverse Engineering Techniques . . . 165
10.2 Precision Hierarchy . . . 170
10.3 Example Plan . . . 173

11.1 Steps for Preparing, Transferring, and Radiating a Command file . . . 201
11.2 Communicator and Related Data Structures . . . 205
11.3 Command Translation Context Diagram . . . 206
11.4 Command Translation Data Flow Diagram . . . 207
11.5 Major Data Structures for Command Translation . . . 209
11.6 Project Files Model for Command Translation . . . 209
11.7 Command Translation: *Main* source model . . . 211

11.5 Translate source code
11.9 Translate source m
11.10Process Mnemonic s
11.11Alternative view of P
11.12Process.msg source
11.13Source code sequence
11.14Annotated source co
11.15The PDEW Macro
11.16Annotated source co

D1 ACTUSPEC output
D2 Circular Queue Libra

11.8 Translate source code 212
11.9 Translate source model 213
11.10 Process Mnemonic subgraph 214
11.11 Alternative view of Process Mnemonic subgraph 215
11.12 `process_msg` source code 216
11.13 Source code sequence for `process_mnemonic_input` 219
11.14 Annotated source code for `end_message_subroutine` 221
11.15 The POPM Macro 223
11.16 Annotated source code for `end_cmdxlt` 224

D.1 AUTOSPEC output of as-built specifications for queue source code 252
D.2 Circular Queue Library Specification 253

Chapter 1

Introduction

As the demands placed on software continue to grow, there is an increasing recognition that software can be error prone. Moreover, the rising cost of software development has resulted in software systems that are used for longer periods of time, for multiple purposes, and for increasingly larger customer bases. As a result, there is a need for more sophisticated and systematic approaches for maintaining software. Our research develops a new technique for reverse engineering that is mathematically rigorous and applicable to practical imperative programming languages such as C. As such, this research facilitates the systematic evolution of software by supporting software maintenance via program understanding and software reuse. This chapter discusses the motivation for the reverse engineering technique and gives the contributions of the work.

1.1 Problem Description and Motivation

Many well-documented computer failures have been attributed to software. Some of the most notable incidents include the catastrophic failures of the Therac-25 [1] and the Ariane

spacecraft [2]. A comm

hot were the result of an

Therac-25. The Therac-2

to the Therac-20 [1]. In

ensure that the radiation d

the development of the T

mistakes that were support

software by the Therac-2

software were to be reen

developing the new softw are

as a result several fatalities o

Ariane 5. The Ariane 5 is

follow-up to the highly succe

software it was determined th

below Ariane 4" [2]. The r

the Ariane 4 software that d

failed voyage of the Ariane

of the spacecraft.

The failure to correctly a

catastrophic events that led

tested for sophisticated a

affected their functionality.

5 spacecraft [2]. A commonly overlooked aspect of these failures has been the fact that both were the result of an improper reengineering of software from one version to another.

Therac-25. The Therac-25 is a radiation therapy system that was constructed as a follow-up to the Therac-20 [1]. In the Therac-20, many hardware safety interlocks were used to ensure that the radiation dosage was well within the prescribed limits for human exposure. In the development of the Therac-25 it was determined that many of the safety interlocking routines that were supported in hardware by the Therac-20 were instead to be supported in software by the Therac-25. Therefore, the combination of the Therac-20 software and hardware were to be reengineered to produce the Therac-25 software. In the course of developing the new software, many of the safety-critical properties were not preserved and as a result several fatalities occurred during the use of the Therac-25 [1].

Ariane 5. The Ariane 5 is a spacecraft developed by the European Space Agency as a follow-up to the highly successful Ariane 4 [2]. During the development of the Ariane 5 software it was determined that "it was not wise to make changes in software which worked well on Ariane 4" [2]. The result of this stance was that a requirement was retained from the Ariane 4 software that did not apply to the Ariane 5 software. Consequently, during the maiden voyage of the Ariane 5, a series of unfortunate events led to the eventual destruction of the spacecraft.

The failure to correctly analyze software in both the Therac-25 and Ariane 5 resulted in catastrophic events that led to loss of life and property. These examples vividly illustrate the need for sophisticated and systematic methods for maintaining software in order to understand their functionality.

Reverse engineering

al component interrelat

mplementation [3]. *Re*

tation of a system w

ftware reengineering

s opposed to develoning

tionality of the exist

pered for many reas

Current reverse en

presentations from pr

ligrammatic information

o recover functional info

ly on syntactic analys

ding program understan

ftware that is typically

Formal methods ar

nguages where a form

Addition, formal met

pecifications in order t

ong mathematical ba

The primary focus o

ngineering of program

ystem. By using for

Reverse engineering of program code is the process of examining components and component interrelationships in order to construct a high-level abstraction of an implementation [3]. *Reengineering* is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [3]. Software reengineering is considered to be a better solution for handling legacy code as opposed to developing software from the original requirements. Since much of the functionality of the existing software has been achieved over a period of time, it must be preserved for many reasons, including providing continuity to current users of the software.

Current reverse engineering techniques focus on the recovery of high-level design representations from program code. One class of approaches constructs structural (i.e., diagrammatic) information about software while another class of approaches has been used to recover functional information. These techniques have been largely informal since they rely on syntactic analysis and pattern matching. While the approaches are invaluable for aiding program understanding, they fail to provide a level of confidence in the reliability of software that is typically required for critical systems [4, 5].

Formal methods are techniques that incorporate the use of formal specification languages, where a formal specification language has a well-defined syntax and semantics. In addition, formal methods have associated calculation rules that can be used to analyze specifications in order to verify correctness and consistency. Since the notations have a formal mathematical basis, formal methods facilitate the use of automated processing.

The primary focus of our research is to apply the use of formal methods to the reverse engineering of program code in order to rigorously support maintenance and evolutionary activities. By using formal methods, our approach addresses the need for rigor in the

verse engineering and re

ical catastrophic events s

Thesis Statement:

Given imperative pr
specification using a
built specifications as
abstract, high-level
then be used for re
program understand

1.2 Contributions

This research makes three

specifically, software mai

is used for deriving as-bu

specification is said to be

is above the original imp

implementation bias. Wh

property that is importa

their representation, they

is the strongest postcon

whether the execution o

ne of the constructs of a

a program written in te

a formal specification co

ecifies such as rigorous

reverse engineering and reengineering of program code in order to minimize and perhaps avoid catastrophic events such as the Therac-25 and Ariane 5 cases [1, 2].

Thesis Statement:

> *Given imperative program code, it is possible to construct an as-built formal specification using a semi-automated translation process [6, 7]. When the as-built specifications are used in tandem with informal specification techniques, abstract, high-level design specifications can be derived. These designs can then be used for rigorous analysis and restructuring in order to facilitate program understanding and software reuse.*

1.2 Contributions

This research makes three major contributions to the area of software engineering and, specifically, software maintenance. First, the *strongest postcondition* predicate transformer is used for deriving as-built formal specifications from imperative program code [6]. A specification is said to be at the *as-built* level if the specification is at a level of abstraction just above the original implementation. As a result, an as-built specification may contain an implementation bias. While these detailed specifications provide a degree of traceability, a property that is important for facilitating confidence in the consistency and correctness of the representation, they may be difficult to read. Given a program S and a precondition Q, the strongest postcondition, denoted $sp(S, Q)$, is defined as the strongest condition that holds after the execution of S, given that S terminates. By defining the formal semantics of each of the constructs of a programming language, a formal specification of the behavior of a program written in terms of the given programming language can be constructed [6]. A formal specification constructed in this manner can then be used for a number of activities such as rigorous software analysis using theorem proving techniques. In order

demonstrate the applic

we have defined the form

plied its use in the ana

ommands for controlling

Second, we show how

sed to introduce abstrac

pecifications. Another pr

for introducing abstractio

To this end, we have dev

by constructing partially-o

ssing operations. Co

f this research can facil

ilding high-level progr

Third, we have develo

nerating specification

enerating an as-built sp

ate libraries with soft

ftware technology, this tech

ating code that may or

An important property

ecifications are well-defined

ncurrent processing [8]. T

this dissertation, we have de

to demonstrate the applicability of the strongest postcondition approach to a broad context, we have defined the formal semantics of a subset of the C programming language [7] and applied its use in the analysis of a mission control system that is used to translate user commands for controlling unmanned spacecraft [7].

Second, we show how a formal technique based on specification matching can be used to introduce abstractions in as-built specifications in order to produce high-level specifications. Another primary objective of the proposed research is to develop a technique for introducing *abstraction* into as-built specifications in order to facilitate their readability. To this end, we have developed a technique based on generalizing as-built specifications by constructing partially-ordered sets of specifications that are ordered using specification matching operations. Consequently, by generalizing as-built specifications, the results of this research can facilitate several reverse engineering and reengineering activities, including high-level program understanding.

Third, we have developed a technique for facilitating software reuse that is based on constructing specification libraries via reverse engineering. Specifically, the process of constructing an as-built specification and the subsequent generalizations can be used to populate libraries with software component specifications. When used along with software reuse technology, this technique can facilitate the construction of new applications using existing code that may or may not have been intended for reuse.

An important property of formal specification languages is that their syntax and semantics are well-defined. As such, formal specification languages are amenable to automated processing [8]. To support the formal reverse engineering technique described in this dissertation, we have developed several tools that provide assistance to a user during the

verse engineering proce

ongest postcondition c

om as-built specificatio

ne logic syntactic edi

1.3 Organization o

The remainder of this d

ntext for software re

ongest postcondition a

Chapter 4 extends the use

ranches. The applicatio

resented in Chapter 5.

pecifications is defined

It integrates the strongest

Chapter 8 provides a des

ngineering. In Chapter

acilitate software reuse.

he introduction of a ne

Chapter 11 presents the de

the NASA JPL applicatio

Delegations.

reverse engineering process. Specifically, we have developed tools that support the use of strongest postcondition to derive specifications from program code and derive abstractions from as-built specifications. In addition, we have developed a theorem prover and first-order logic syntactic editor that can be used throughout the specification process.

1.3 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides background material for software maintenance and formal methods. Our investigations into the use of *strongest postcondition* as a formal basis for reverse engineering is described in Chapter 3. Chapter 4 extends the use of strongest postcondition to include a formal treatment of pointer variables. The application of strongest postcondition to the C programming language is presented in Chapter 5. The approach for introducing design abstractions into as-built specifications is defined in Chapter 6. Chapter 7 presents a reverse engineering framework that integrates the strongest postcondition technique with the design abstraction technique. Chapter 8 provides a description of several tools that we have developed to support reverse engineering. In Chapter 9, we describe the use of our reverse engineering technique to facilitate software reuse. A survey of related work is presented in Chapter 10, including the introduction of a new taxonomy for comparing different techniques and their tools. Chapter 11 presents the details of a case study that applies the reverse engineering technique to a NASA JPL application. Finally, Chapter 12 draws conclusions and discusses remaining investigations.

Chapter 2

Background

This chapter provides background information for software maintenance and formal methods for software development. Included in this discussion is the formal model of program semantics used throughout this dissertation.

2.1 Software Maintenance

One of the most difficult aspects of software maintenance is the analysis of existing programs in order to determine functionality. This step in re-engineering is known as reverse engineering. Identifying design decisions, intended use, and domain specific details are often significant obstacles to successfully re-engineering a system.

Several terms are frequently used in the discussion of re-engineering [3]. *Forward Engineering* is the process of developing a system by moving from high-level abstract specifications to detailed, implementation-specific manifestations. The explicit use of the word "forward" is used to contrast the process with *Reverse Engineering*, the process of analyzing a system in order to identify system components, component relationships, and intended behavior. *Restructuring* is the process of creating a logically equivalent system

at the same level of abstr

of the system and is best

ial structured code. *Re-*

ansscribe it in a new fo

esign and implementation

There are four types o

ntenance [9]. Adaptive

ode to properly interface

ance is corrective mainte

xample of perfective ma

intainability or reliabil

ptive maintenance is

ic any form of software

x fix software when per

he current functionality

llow a programmer to

specially in legacy syste

requirements.

Refinement is the pr

owing that the new re

igh-level specification, c

ne each of these specific

sufficient amount of de

at the same level of abstraction. This process does not require semantic understanding of the system and is best characterized by the task of transforming unstructured code into structured code. *Re-Engineering* is the examination and alteration of a system to reconstitute it in a new form, which potentially involves changes at the requirements, design, and implementation levels.

There are four types of software maintenance: *adaptive, corrective, perfective, and preventive* [9]. Adaptive maintenance is the activity associated with changing code in order to properly interface with a changing environment. Changing code in order to fix errors is corrective maintenance. Adding new features in response to user needs is an example of perfective maintenance, and changing software in order to improve future maintainability or reliability is known as preventive maintenance. Pressman states that preventive maintenance is characterized by *reverse* and *re-engineering* [9], although, in fact, any form of software maintenance may involve both activities. For instance, in order to fix software when performing corrective maintenance, it is important to understand the current functionality of the program. Reverse engineering techniques can be used to allow a programmer to recover the design and functionality. Perfective maintenance, especially in legacy systems, may require a complete re-engineering in order to satisfy new requirements.

Refinement is the process of making a higher-level specification more concrete and showing that the new refined specification satisfies the higher-level specification. Given a high-level specification, called $s1$, and a refinement of the specification, called $s2$, we say that each of these specifications exist at different *levels of abstraction* since each provides a different amount of detail. In the context of a formal specification, a specification $s2$

satisfies a higher-level sp

all cases, s2 is stronger t

d satisfies a higher-leve

rfaces to s1, and s1 co

Abstraction is the p

and showing that the a

specification. In the conte

a lower-level specifica

That is, in all cases, s2

n abstracted specificatio

rfaces for s2 are con

ponents in s1.

From these descriptio

That is, given a high-lev

finement of s1 then s2

x > y". Let a refinemen

x > 0, the term $x = y -$

$- x > y$. In this ex

$x = y - c$, $(c > 0)$ is

ve the data flow diagra

ribes the specification

he dashed lines indicate

ce the implication is tha

satisfies a higher-level specification $s1$ if it can formally proven that $s2 \rightarrow s1$. That is, in all cases, $s2$ is stronger than $s1$. In the context of structural specifications, a specification $s2$ satisfies a higher-level specification $s1$ if the interfaces for $s2$ are consistent with the interfaces to $s1$, and $s1$ contains the elements of $s2$.

Abstraction is the process of making a low-level specification $s2$ less concrete and showing that the abstracted specification $s1$ is a generalization of the low-level specification. In the context of a formal specification, a specification $s1$ is a generalization of a lower-level specification $s2$ if it can formally proven that $s2 \rightarrow s1$ and that $s1 \not\rightarrow s2$. That is, in all cases, $s1$ is weaker than $s2$. In the context of a structural specification, an abstracted specification $s1$ is a generalization of a lower-level specification $s2$ if the interfaces for $s2$ are consistent with the interfaces to $s1$ and if the elements of $s2$ are contained in $s1$.

From these descriptions it follows that refinement and abstraction are dual concepts. That is, given a high-level specification $s1$ and a low-level specification $s2$, if $s2$ is a refinement of $s1$ then $s1$ is an abstraction of $s2$. For example, consider the specification "$x > y$". Let a refinement of this specification appear as "$(x = y + c) \wedge (c > 0)$". Since $c > 0$, the term $x = y + c$ always ensures that $x > y$. Therefore $((x = y + c) \wedge (c > 0)) \rightarrow (x > y)$. In this example, $(x > y)$ is an abstraction of $((x = y + c) \wedge (c > 0))$ and $((x = y + c) \wedge (c > 0))$ is a refinement of $(x > y)$. For example, consider Figure 2.1 where the two data flow diagrams depict $G1$ as an abstraction of $G2$, where the top diagram contains the specification $G1$ and the bottom diagram is the abstracted specification $G2$. The dashed lines indicate that the bottom diagram can be replaced by the top diagram. As such, the implication is that the behavior of $G1$ is refined by $G2$.

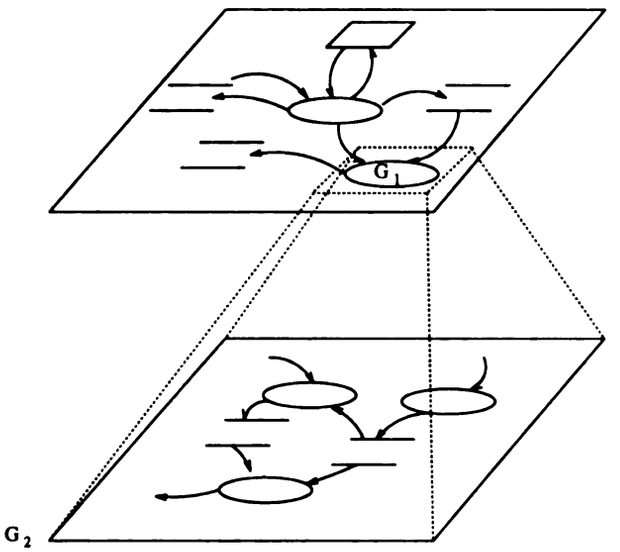


Figure 2.1: $G1$ as an abstraction of $G2$

Byrne described the re-engineering process using a graphical model similar to the one shown in Figure 2.2 [10, 11]. The process model appears in the form of two sectioned triangles, where each section in the triangles represents a different level of abstraction. The higher levels in the model are *concepts* and *requirements*. The lower levels include *designs* and *implementations*. The relative size of each of the sections is intended to represent the amount of information known about a system at a given level of abstraction. Entry into this re-engineering process model begins with system *A*, where *Abstraction* (or reverse engineering) is performed to an appropriate level of detail. The next step is *Alteration*, where the system is constituted into a new form at a different level of abstraction. Finally, *Refinement* of the new form into an implementation can be performed to create system *B*.

This dissertation describes an approach to reverse engineering that is applicable to the *implementation* and *design* levels. In Figure 2.2, the context for our approach is represented

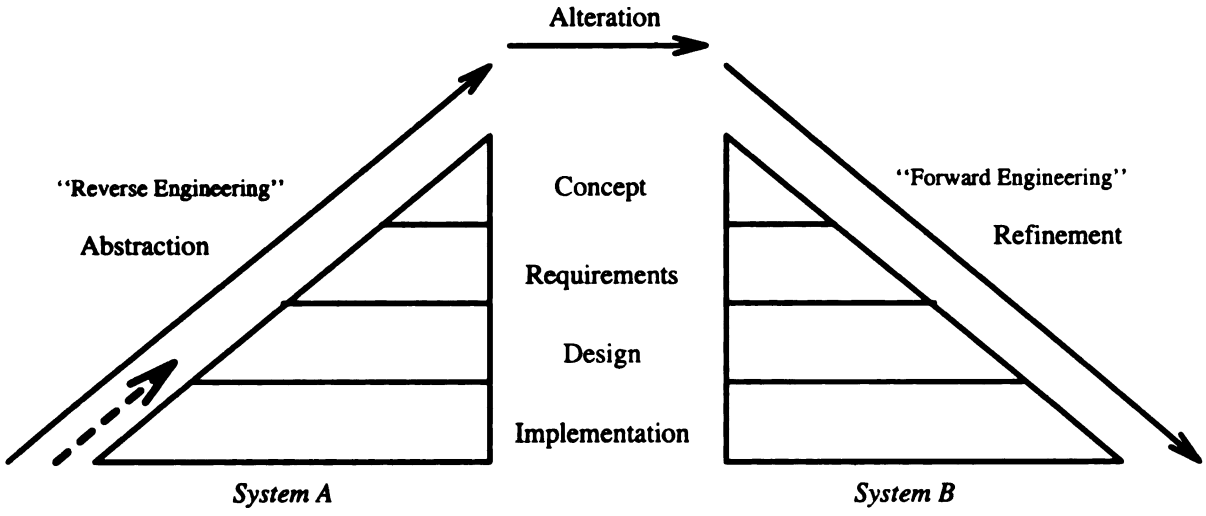


Figure 2.2: Reverse Engineering Process Model

by the dashed arrow. The motivation for deriving specifications at an implementation-bound level of abstraction is that it provides a means of traceability between the program source code and the formal specifications constructed using the techniques described in the chapters that follow. This traceability is necessary in order to facilitate technology transfer of formal methods [4, 5]. That is, currently existing development teams must be able to understand the relationship between the source code and the specifications.

2.2 Formal Methods

Although the waterfall development life-cycle provides a structured process for developing software, the design methodologies that support the life-cycle (e.g., Structured Analysis and Design [12]) make use of informal techniques, thus increasing the potential for introducing ambiguity, inconsistency, and incompleteness in designs and implementations. In contrast, formal methods used in software development are rigorous techniques for

specifying, developing, an

a well-defined specificati

used to reason about a spec

are well-defined and thus,

2.2.1 Levels of Rigo

Easily defined *levels of r*

use[14]. These levels are

Level 0: No use of
diagrammatic

Level 1: Use of con
include projects
in data diction

Level 2: Use of for
support tools s
include projects
but do not use p

Level 3: Use of ful
support environ
checking. Exam
and proofs of th

The approach described in

hierarchy of rigor since we

support tools for theorem pr

2.2.2 Program Sema

The notation $Q\{S\}R$ [15]

reads: given that a logical c

the logical condition R will

specifying, developing, and verifying computer software [13]. A formal method consists of a well-defined specification language with a set of well-defined inference rules that can be used to reason about a specification [13]. A benefit of formal methods is that their notations are well-defined and thus, are amenable to automated processing [8].

2.2.1 Levels of Rigor

Rushby defined *levels of rigor* for describing the degree to which formal methods can be used [14]. These levels are summarized as follows.

> **Level 0:** No use of formal methods. Examples include all projects that use diagrammatic notations and no mathematical notation.
>
> **Level 1:** Use of concepts and notations from discrete mathematics. Examples include projects that include the use of formal languages to describe data in data dictionaries.
>
> **Level 2:** Use of formalized specification languages with some mechanized support tools such as syntax checkers and pretty printers. Examples include projects that use specifications languages to describe behavior, but do not use proof obligations to verify correctness.
>
> **Level 3:** Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking. Examples include any project that involves full specification and proofs of the specifications using automated tools.

The approach described in this dissertation can be considered to be at level 3 in the hierarchy of rigor since we are advocating the use of formal specification languages and support tools for theorem proving.

2.2.2 Program Semantics

The notation $Q \{ S \} R$ [15] is used to represent a *partial correctness* model of execution, where, given that a logical condition Q holds, if the execution of program S terminates, then logical condition R will hold. A rearrangement of the braces to produce $\{ Q \} S \{ R \}$,

in contrast, represents a *total correctness* model of execution. That is, if condition Q holds, then S is guaranteed to terminate with condition R true.

A *precondition* describes the initial state of a program, and a *postcondition* describes the final state. Given a statement S and a postcondition R, the *weakest precondition* predicate transformer $wp(S, R)$ describes the set of all states in which the statement S can begin execution and terminate with postcondition R true, and the *weakest liberal precondition* predicate transformer $wlp(S, R)$ is the set of all states in which the statement S can begin execution and establish R as *true* if S terminates. In this respect, $wp(S, R)$ establishes the total correctness of S, and $wlp(S, R)$ establishes the partial correctness of S. The *wp* and *wlp* are called predicate transformers because they take predicate R and, using the properties listed in Table 2.1, produce a new predicate.

$wp(S, \mathit{false})$	$\equiv$	false
$wp(S, A \wedge B)$	$\equiv$	$wp(S, A) \wedge wp(S, B)$
$wp(S, A \vee B)$	$\Rightarrow$	$wp(S, A) \vee wp(S, B)$
$wp(S, A \rightarrow B)$	$\Rightarrow$	$wp(S, A) \rightarrow wp(S, B)$

Table 2.1: Properties of the *wp* predicate transformer

The relationship between wp and wlp is the following.

$$wp(S, R) \equiv wp(S, \text{true}) \wedge wlp(S, R) \tag{2.1}$$

This states that the weakest precondition for establishing R as true given the program S is equivalent to the fact that if S terminates then $wlp(S, R)$ is true, and $wp(S, \text{true})$ holds. The conjunct $wlp(S, R)$ is used to establish correctness and the conjunct $wp(S, \text{true})$ is used to establish termination. The context for our investigations is that we are reverse engineering

systems that have desirable

Termination behavior is ty

the partial correctness mo

2.3 Strongest Pos

Consider the predicate –

execution of S that termin

in which satisfaction of R

of S, R, which, is the se

terminates with R true

An analogous characte

that describes initial con

transformer [16], which is

begins with Q true. That

terminates. As such,

of S. Finally, we make t

relationship between the t

The importance of this

existing programming

Section 3 provides a mea

formalizes the properties

systems that have desirable properties or functionality that should be preserved or extended. Termination behavior is typically determined by years of program observation. Therefore, the partial correctness model is sufficient.

2.2.3 Strongest Postcondition

Consider the predicate $\neg wlp(S, \neg R)$, which is the set of all states in which *there exists* an execution of S that terminates with R true. That is, we wish to describe the set of states in which satisfaction of R is possible [16]. The predicate $\neg wlp(S, \neg R)$ is contrasted to $wlp(S, R)$ which, is the set of states in which the computation of S either fails to terminate, or terminates with R true.

An analogous characterization can be made in terms of the computation state space that describes initial conditions using the *strongest postcondition* $sp(S, Q)$ predicate transformer [16], which is the set of all states in which *there exists* a computation of S that begins with Q true. That is, given that Q holds, execution of S results in $sp(S, Q)$ true, if S terminates. As such, $sp(S, Q)$ assumes partial correctness. Table 2.2 lists some properties of sp. Finally, we make the following observation about $sp(S, Q)$ and $wlp(S, R)$ and the relationship between the two predicate transformers, given the Hoare triple $Q \{ S \} R$ [16]:

$$\begin{aligned} Q &\Rightarrow wlp(S, R) \\ sp(S, Q) &\Rightarrow R \end{aligned}$$

The importance of this relationship is two-fold. First, it provides a formal basis for translating programming statements into formal specifications. Second, the symmetry of sp and wlp provides a method for verifying the correctness of a reverse engineering process that utilizes the properties of wlp and sp in tandem.

$sp(S, A \wedge B)$	$\equiv$	$sp(S, A) \wedge sp(S, B)$
$sp(S, A \vee B)$	$\Rightarrow$	$sp(S, A) \vee sp(S, B)$
$A \rightarrow B$	$\equiv$	$sp(S, A) \rightarrow sp(S, B)$
$sp(S, A \rightarrow B)$	$\equiv$	$sp(S, A) \rightarrow sp(S, B)$
$sp(S, \mathit{false})$	$\equiv$	false

Table 2.2: Properties of the *sp* predicate transformer

2.2.4 strongest postcondition vs. weakest precondition

Given a Hoare triple $Q \{ S \} R$, we note that wp is a backward rule, in that a derivation of a specification begins with R, and produces a predicate $wp(S, R)$. The predicate transformer wp assumes a total correctness model of computation, meaning that given S and R, if the computation of S begins in state $wp(S, R)$, the program S *will* halt with condition R true.

We contrast this model with the sp model, a forward derivation rule. That is, given a precondition Q and a program S, sp derives a predicate $sp(S, Q)$. The predicate transformer sp assumes a partial correctness model of computation meaning that if a program starts in state Q, then the execution of S will place the program in state $sp(S, Q)$ if S terminates. Figure 2.3 gives a graphical depiction of the differences between sp and wp, where the input to the predicate transformer produces the corresponding predicate. Figure 2.3(a) gives the case where the input to the predicate transformer is "S" and "R", and the output to the predicate transformer (given by the box and appropriately named "wp") is "wp(S,R)". The *sp* case (Figure 2.3(b)) is similar, where the input to the predicate transformer is "S" and "Q", and the output to the transformer is "sp(S,Q)".

Figure 2.3: Black box

The use of these

fications. Using

verse engineering.

a guideline for per

precondition Q is k

plication of p. As su

2.5 Formal Me

Software reuse is the

components. Jeng a

formal basis for ident

cking [19] describ

ase. In addition, Pe

ssing. Table 2.3

ect for software reu

In this chapter, we a

proaches on signature

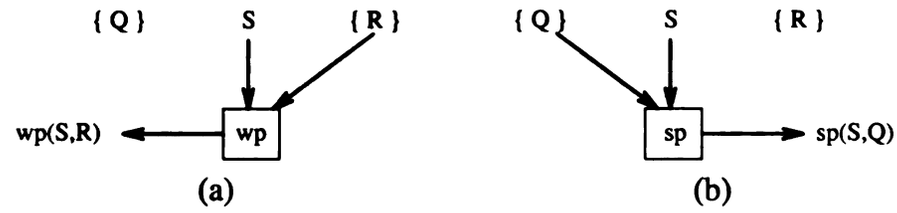


Figure 2.3: Black box representation and differences between *wp* and *sp*: (a) *wp* (b) *sp*

The use of these predicate transformers for reverse engineering have different implications. Using wp implies that a postcondition R is known. However, with respect to reverse engineering, determining R is the objective, therefore wp can only be used as a guideline for performing reverse engineering [17]. The use of *sp* assumes that a precondition Q is known and that a postcondition will be derived through the direct application of *sp*. As such, *sp* is better suited for reverse engineering.

2.2.5 Formal Methods Applied to Software Reuse

Software reuse is the process of constructing a software system using existing software components. Jeng and Cheng [18] describe the use of a generality operator as the formal basis for identifying reusable components via specification matching. Zaremski and Wing [19] describe several operators for matching queries to components for software reuse. In addition, Penix and Alexander [20] define the *satisfies* criterion for component matching. Table 2.3 lists several of the matching operators. [1] When these operators are used for software reuse, A is a query specification and R is a library specification.

[1] In this chapter, we assume that the signatures of a query specification and a library specification match. For details on signature matching, see [21].

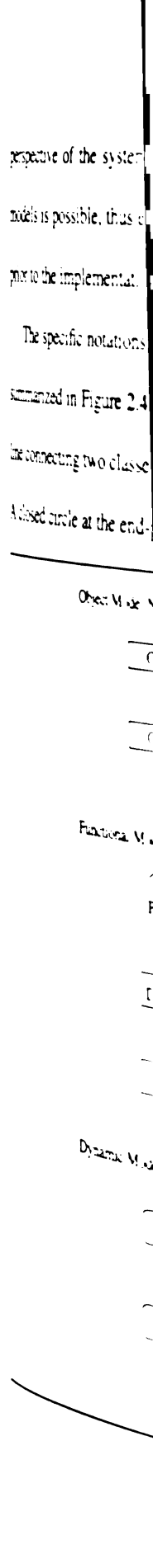
Match	Definition $R \preceq A$
Exact Pre/Post	$(A_{pre} \Leftrightarrow R_{pre}) \wedge (R_{post} \Leftrightarrow A_{post})$
Plug-in	$(A_{pre} \Rightarrow R_{pre}) \wedge (R_{post} \Rightarrow A_{post})$
Plug-in Post	$(R_{post} \Rightarrow A_{post})$
Weak Post	$R_{pre} \Rightarrow (R_{post} \Rightarrow A_{post})$
Guarded Plug-in	$(A_{pre} \Rightarrow R_{pre}) \wedge ((R_{pre} \wedge R_{post}) \Rightarrow A_{post})$
Guarded Post	$((R_{pre} \wedge R_{post}) \Rightarrow A_{post})$
Satisfies	$(A_{pre} \Rightarrow R_{pre}) \wedge ((A_{pre} \wedge R_{post}) \Rightarrow A_{post})$

Table 2.3: Pre/Post Match Criterion

2.3 Informal Methods

Informal (or semi-formal) methods are software development methods that are based on the use of techniques that lack the use of rigorous notation. One of the advantages of using an informal technique is that they are typically based on the use of graphical notations. As such, the techniques are amenable to high-level discussion, and are, in general, scalable to large systems. One of the disadvantages of using informal methods is that since the notations lack mathematical rigor, they are prone to ambiguity. An example of an informal method is the Object Modeling Technique (OMT) [22].

OMT is a modeling language that is commonly used in industry and academia. OMT comprises three complementary models, each of which are simple to use and understand. The *object model* describes the static, structural aspects of the system. The object model captures the objects of the system and the relationships between the objects. The *dynamic model* depicts the temporal and behavioral aspects of the system. Finally, the *functional model* describes the services provided by the system. Respectively, entity-relationship diagrams, state transition diagrams, and data flow diagrams are used to represent the object, dynamic, and functional models, and each model is only used to capture a specific



perspective of the system. With recent work [23, 24], rigorous analysis of each of the models is possible, thus enabling consistency and completeness checks at the model level prior to the implementation phase.

The specific notations for the object model, functional model, and dynamic model are summarized in Figure 2.4. In an object model, a rectangle is used to represent a class. A line connecting two classes indicates that some relationship exists between the two classes. A closed circle at the end-point of an association denotes a one-to-many relationship.

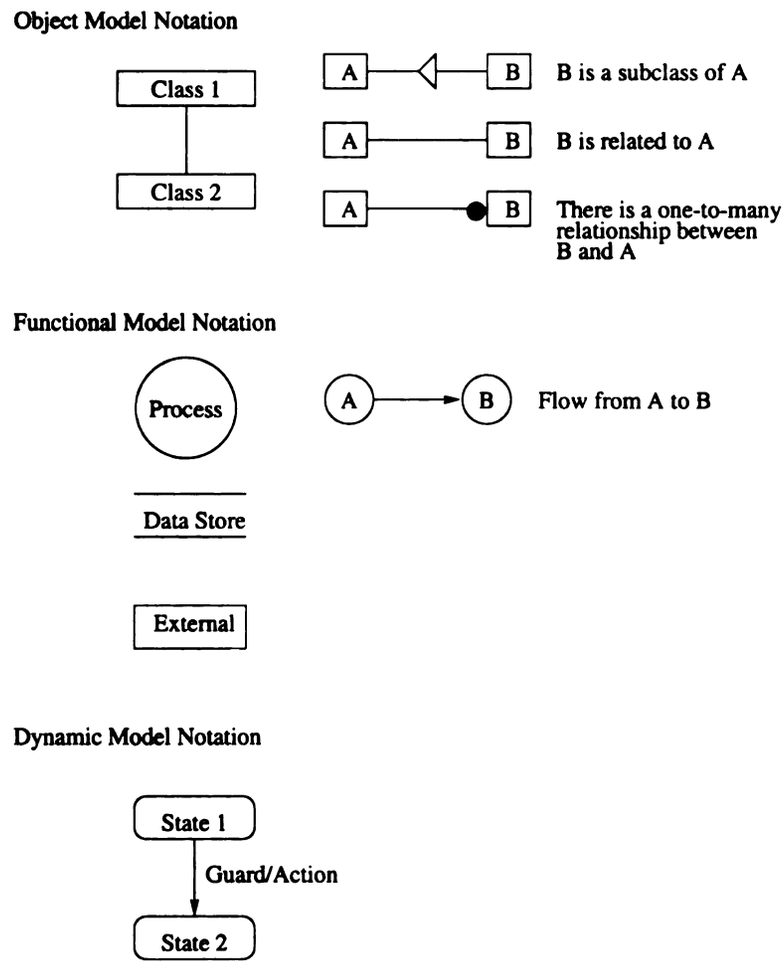


Figure 2.4: OMT Summary

In the functional mo

epresents a data store, a

Arcs are used to join the

low from one entity to a

In the dynamic mode

ransitions between state

tate". The text before

a transition out of a state

tate is generated by taki

In the functional model, a circle represents a process entity, a pair of parallel lines represents a data store, and a rectangle represents an entity external to the current model. Arcs are used to join the various entities of a functional model, with the arrow indicating a flow from one entity to another.

In the dynamic model, a rounded rectangle is used to denote a state and arcs indicate transitions between states. The transitions can be labeled with strings separated by a single slash '/'. The text before a slash defines a guarding condition that must be true in order for a transition out of a state to occur, and the text after the slash defines the event or action that is generated by taking the transition.

Chapter 3

Using Strongest Postcondition to Reverse Engineer Programs

Chapter 2 introduced the strongest postcondition predicate transformer $sp(S, Q)$. This chapter describes our investigations into the use of the strongest postcondition as the formal basis for reverse engineering [6] through the construction of formal specifications from programs written in terms of the Dijkstra guarded command language [25]. The primary result of this chapter is a demonstration of the use of the strongest postcondition to facilitate the construction of as-built formal specifications from program code for reverse engineering purposes.

3.1 Basic Constructs

This section describes the derivation of formal specifications from the primitive programming constructs of assignment, alternation, and sequences. The Dijkstra guarded command language [25] is used to represent each primitive construct but the techniques are applicable to the general class of imperative languages. For each primitive, we first describe the semantics of the predicate transformers wlp and sp as they apply to each primitive and then, for reverse engineering purposes, describe specification derivation in terms of Hoare

mples. Notationally,

e used to indicate a

3.1.1 Assignme

An assignment state

xpression. The

which represents the

xpression e. If x c

xpressions, then the

by E_i, respectively.

follows [16]

where Q is the prec

be quantified varia

Section 3.1.2 d

Expression (3.1). I

where C is a logica

Expression (3.1

$\ldots \wedge C_i \wedge \ldots y = x$

Hoad lemma, if x

the expression e, th

triples. Notationally, throughout the remainder of this paper, the notation $\{ Q \} S \{ R \}$ will be used to indicate a partial correctness interpretation.

3.1.1 Assignment

An assignment statement has the form `x:= e;` where `x` is a variable, and `e` is an expression. The *wlp* of an assignment statement is expressed as $wlp(\mathtt{x:=e}, R) = R^x_e$, which represents the postcondition R with every free occurrence of x replaced by the expression e. If x corresponds to a vector $\overline{y}$ of variables and e represents a vector $\overline{E}$ of expressions, then the *wlp* of the assignment is of the form $R^{\overline{y}}_{\overline{E}}$, where each y_i is replaced by E_i, respectively, in expression R. The *sp* of an assignment statement is expressed as follows [16]

$$sp(\mathtt{x:=e}, Q) = (\exists v :: Q^x_v \wedge x = e^x_v), \tag{3.1}$$

where Q is the precondition, v is the quantified variable, and '::' indicates that the range of the quantified variable v is not relevant in the current context.

Section 3.1.2 describes two lemmas for eliminating the existential quantification in Expression (3.1). In the first lemma, if the precondition Q is of the form $C \wedge (x = u)$, where C is a logical expression, then after the textual substitution of variable x with v in Q, Expression (3.1) reads as $(\exists v :: C^x_v \wedge (v = u) \wedge x = e^x_v)$. Since $(v = u)$, the expression $(\exists v :: C^x_v \wedge (v = u) \wedge x = e^x_v)$ is logically equivalent to $C^x_u \wedge (u = u) \wedge x = e^x_u$. In the second lemma, if x does not appear as a free variable in either the logical expression Q or the expression e, then $(\exists v :: Q^x_v \wedge x = e^x_v)$ is logically equivalent to $Q \wedge x = e$. Assuming

we can establish the

for assignment statem

where τ represents th

Q_0 is the precondition

3.1.2 Removin

This section descri

Expression 3.1 can b

The precondition is

derivation based on

One-point simplifi

In this section we p

The precondition ha

Lemma 3.1.1 (On
$:= \tau$, *where* U
statement "$x := e$

Proof. The sp de

we can establish the conditions for satisfying these lemmas, the Hoare triple formulation for assignment statements is as follows:

$\{Q\}$ /* precondition */
x := e;
$\{Q_v^x \wedge (x = e_v^x)\}$ /* postcondition */

where v represents the initial value of the variable x before execution of the statement and Q is the precondition.

3.1.2 Removing quantification from the specification of assignment

This section describes two lemmas that justify why the existential quantification in Expression 3.1 can be removed. The first lemma (Lemma 3.1.1) describes the case when the precondition is in a particular canonical form. The second lemma (Lemma 3.1.2) is a derivation based on the semantics for assignment described in [16].

One-point simplification

In this section we prove that the quantification in Expression (3.1) can be eliminated when the precondition has a particular canonical form.

Lemma 3.1.1 (One-Point) *Let the precondition Q in Expression (3.1) have the form $U \wedge (x = n)$ where U is a logical expression, n is a constant and x is the variable from the statement "*x := e*". Then*

$$sp(\mathtt{x := e}, Q) \equiv U_n^x \wedge x = e_n^x$$

Proof. The *sp* derivation is as follows.

$$\begin{aligned} sp(\mathtt{x := e}, Q) &\equiv (\exists v :: Q_v^x \wedge x = e_v^x) \\ &\quad \langle \text{Substitution of } Q \text{ with } U \wedge (x = n) \rangle \\ &\equiv (\exists v :: (U \wedge (x = n))_v^x \wedge x = e_v^x) \end{aligned}$$

$\langle$Textual Substitution$\rangle$
$\equiv (\exists v :: U_v^x \wedge (v = n) \wedge x = e_v^x)$
$\langle$Trading [16]$\rangle$
$\equiv (\exists v : v = n : U_v^x \wedge x = e_v^x)$
$\langle$One-point rule [16] with $v = n\rangle$
$\equiv U_n^x \wedge x = e_n^x$

□

The fact that the existential quantification can be removed when the precondition Q has the form $U \wedge (x = n)$ is convenient since the canonical form can be derived from parameter and variable declarations as described in Section 3.1.2. The logical formula U represents the part of the precondition Q that does not specify the value of x. The extension of the one-point rule to two points, three-points, or n points provide the more general expression given by Expression 3.1.

Substitution

In this section we prove that the quantification in Expression (3.1) can be eliminated when there is no free occurrence of x in expression e and precondition Q.

Lemma 3.1.2 (Substitution) *Assume that there are no free occurrences of* x *in precondition* Q *and expression* e *in the statement* x := e. *Then*

$$sp(\mathtt{x} := \mathtt{e}, Q) \equiv Q \wedge x = e.$$

Proof. The *sp* derivation is as follows.

$sp(\mathtt{x} := \mathtt{e}, Q) \equiv (\exists v :: Q_v^x \wedge x = e_v^x)$
$\langle$no substitutable y in U implies $U_v^y \equiv U\rangle$
$\equiv (\exists v :: Q \wedge x = e)$
$\langle$Predicate calculus$\rangle$
$\equiv Q \wedge x = e$

While the de

s less frequent a

Establishing the

Lemmas 3.1.1 a

specification of a

he conditions fo

wo properties of

pecifications and

An example p

Using the fact tha

parameter, it can b

and y have some

establishing th

assignment to x or

ecution, a declare

3.1.3 Alternat

n alternation staten

command language

□

While the derivation of $Q \wedge x = e$ is straightforward, the actual application of this case is less frequent and only occurs in the cases when variables have no initial value.

Establishing the conditions for removing quantification

Lemmas 3.1.1 and 3.1.2 identify the conditions for removing quantification from the specification of assignment statements. In order to take advantage of these lemmas, the conditions for removing quantification must be established. Fortunately, there are two properties of programs that allow for these conditions to be established: *parameter specifications* and *variable declarations*.

An example parameter specification might appear as follows:

`proc` p (**value** x; **value-result** y; **result** z);

Using the fact that x is defined as a **value** parameter and y is defined as a **value-result** parameter, it can be easily deduced that, upon entry into the program p, the parameters x and y have some initial value. As such, we can assert that $(x = X)$ and $(y = Y)$, thus establishing the conditions for removing the quantification in the specifications of any assignment to x or y. To establish the conditions for Lemma 3.1.2, note that during program execution, a declared variable has no initial value.

3.1.3 Alternation

An alternation statement (also known as a conditional statement) using the Dijkstra guarded command language is expressed as [25]

where $B_i \rightarrow S_i$ is a

guard B_i is true. Th

where IF represents

condition to satisfy A

S_i for each guarded

form [16]

The existential expres

of objects:

Expression (3.3) state

$S_i B_i * Q$ is true.

Si active nature of ab

terms of both the pre

and S_i, respectively.

```
if
        B1 → S1;
        ...
        || Bn → Sn;
fi;
```

where $\mathtt{B}_i \rightarrow \mathtt{S}_i$ is a guarded command such that S_i is only executed if logical expression (guard) $\mathtt{B}_i$ is true. The *wlp* for alternation statements is given by [16]:

$$wlp(\mathtt{IF}, R) \equiv (\forall i : \mathtt{B}_i : wlp(\mathtt{S}_i, R)),$$

where `IF` represents the alternation statement. The equation states that the necessary condition to satisfy *R*, if the alternation statement terminates, is that given $\mathtt{B}_i$ is *true*, the *wlp* for each guarded statement S_i with respect to R holds. The *sp* for alternation has the form [16]

$$sp(\mathtt{IF}, Q) \equiv (\exists i :: sp(\mathtt{S}_i, \mathtt{B}_i \wedge Q)). \tag{3.2}$$

The existential expression can be expanded into the following form comprising a sequence of disjuncts:

$$sp(\mathtt{IF}, Q) \equiv sp(\mathtt{S}_1, \mathtt{B}_1 \wedge Q) \vee \ldots \vee sp(\mathtt{S}_n, \mathtt{B}_n \wedge Q). \tag{3.3}$$

Expression (3.3) states that after execution of the `if-fi` statement, one of the disjuncts $sp(S_i, B_i \wedge Q)$ is true. The form of Expression 3.3 as a sequence of disjuncts illustrates the disjunctive nature of alternation statements where each disjunct describes the postcondition in terms of both the precondition Q and the guard and guarded command pairs, given by $\mathtt{B}_i$ and $\mathtt{S}_i$, respectively. This characterization follows the intuition that a statement $\mathtt{S}_i$ is

only executed if

based on the simul

alternation statem

follows

3.1.4 Sequen

For a given sequ

S_i is the precondi

follow accordingly

Likewise, the wp [1

In the case of w

the if the sequenc

sets defined by w

with respect to the

respect to a precondi

process is as follow

only executed if $\mathtt{B}_i$ is true. The translation of alternation statements to specifications is based on the similarity of the semantics of Expression (3.3) and the execution behavior for alternation statements. Using the Hoare triple notation, a specification is constructed as follows

$$
\begin{array}{l}
\{\, Q \,\} \\
\mathtt{if} \\
\qquad \mathtt{B}_1 \rightarrow \mathtt{S}_1; \\
\qquad \ldots \\
\qquad [\!] \; \mathtt{B}_n \rightarrow \mathtt{S}_n; \\
\mathtt{fi;} \\
\{\, sp(\mathtt{S}_1, B_1 \wedge Q) \vee \ldots \vee sp(\mathtt{S}_n, B_n \wedge Q) \,\}.
\end{array}
$$

3.1.4 Sequence

For a given sequence of statements $\mathtt{S}_1 ; \ \ldots \ ; \mathtt{S}_n$, the postcondition for some statement $\mathtt{S}_i$ is the precondition for some subsequent statement $\mathtt{S}_{i+1}$. The *wlp* and *sp* for sequences follow accordingly. The *wlp* for sequences is defined as follows [16]:

$$wlp(\mathtt{S}_1 ; \mathtt{S}_2, R) \equiv wlp(\mathtt{S}_1, wlp(\mathtt{S}_2, R)).$$

Likewise, the *sp* [16] is

$$sp(\mathtt{S}_1 ; \mathtt{S}_2, Q) \equiv sp(\mathtt{S}_2, sp(\mathtt{S}_1, Q)). \tag{3.4}$$

In the case of *wlp*, the set of states for which the sequence $\mathtt{S}_1 ; \mathtt{S}_2$ can execute with R true (if the sequence terminates) is equivalent to the *wlp* of $\mathtt{S}_1$ with respect to the set of states defined by $wlp(\mathtt{S}_2, R)$. For *sp*, the derived postcondition for the sequence $\mathtt{S}_1 ; \mathtt{S}_2$ with respect to the precondition Q is equivalent to the derived postcondition for $\mathtt{S}_2$ with respect to a precondition given by $sp(\mathtt{S}_1, Q)$. The Hoare triple formulation and construction process is as follows:

3.2 Iterative

The programming

produce straight-li

of iteration and re

Program develop

recursive programs

the formal specific

deviate from our pr

construct and use

approach is necessa

in terms of recursiv

3.2.1 Iteratio

Iteration enables th

language, has the fo

$\{ Q \}$
$S_1;$
$\{ sp(S_1, Q) \}$
$S_2;$
$\{ sp(S_2, sp(S_1, Q)) \}.$

3.2 Iterative and Procedural Constructs

The programming constructs of assignment, alternation, and sequence can be combined to produce straight-line programs (programs without iteration or recursion). The introduction of iteration and recursion into programs enables more compactness and abstraction in program development. However, constructing formal specifications of iterative and recursive programs can be problematic, even for the human specifier. This section discusses the formal specification of iteration and procedural abstractions without recursion. We deviate from our previous convention of providing the formalisms for wlp and sp for each construct and use an operational definition of how specifications are constructed. This approach is necessary because the formalisms for the wlp and sp for iteration are defined in terms of recursive functions [16, 26] that are, in general, difficult to practically apply.

3.2.1 Iteration

Iteration enables the repetitive application of a statement. Iteration, using the Dijkstra language, has the form

```
do
        B1 → S1;
        ...
      || Bn → Sn;
od;
```

In more general t

commands of the f

s true. A simplifie

In the context o

of iterations still to

before and after eac

of iteration statem

be determined. H

boundedness and te

Using the abbre

terms of the we

following [16]:

where the notation

Operationally, Exp

for the execution of

statement terminate

expression describ

The strongest p

formulation [16]:

In more general terms, the iteration statement may contain any number of guarded commands of the form $\mathtt{B}_i \rightarrow \mathtt{S}_i$, such that the loop is executed as long as any guard B_i is true. A simplified form of repetition is given by "`do B → S od`".

In the context of iteration, a *bound function* determines the upper bound on the number of iterations still to be performed on the loop. An *invariant* is a predicate that is true before and after each iteration of a loop. The problem of constructing formal specifications of iteration statements is difficult because the bound functions and the invariants must be determined. However, for a partial correctness model of execution, concerns of boundedness and termination fall outside of the interpretation, and thus can be relaxed.

Using the abbreviated form of repetition "`do B → S od`", the semantics for iteration in terms of the weakest liberal precondition predicate transformer wlp is given by the following [16]:

$$wlp(\mathtt{DO}, R) \equiv (\forall i : 0 \leq i : wlp(\mathtt{IF}^i, B \vee R)), \tag{3.5}$$

where the notation "$\mathtt{IF}^i$" is used to indicate the execution of "`if B → S fi`" i times. Operationally, Expression (3.5) states that the weakest condition that must hold in order for the execution of an iteration statement to result with R true, provided that the iteration statement terminates, is equivalent to a conjunctive expression where each conjunct is an expression describing the semantics of executing the loop i times, where $i \geq 0$.

The strongest postcondition semantics for repetition has a similar but notably distinct formulation [16]:

$$sp(\mathtt{DO}, Q) \equiv \neg B \wedge (\exists i : 0 \leq i : sp(\mathtt{IF}^i, Q)). \tag{3.6}$$

Expression (3.6

statement, given

is false $\neg B$, and

number of times

Although the

liberal precondit

recurrent nature

For instance, cons

application of the

$\{P\}$ do $i < n$

The closed form

$\{P'\}Q$ is true

for the unrolled ve

Application of

optimized code show

Expression (3.6) states that the strongest condition that holds after executing an iterative statement, given that condition Q holds, is equivalent to the condition where the loop guard is false ($\neg B$), and a disjunctive expression describing the effects of iterating the loop some number of times k, where $k \geq 0$.

Although the semantics for repetition in terms of strongest postcondition and weakest liberal precondition are less complex than that of the weakest precondition [16], the recurrent nature of the closed forms make the application of such semantics difficult. For instance, consider the counter program "`do i < n → i := i + 1 od`". The application of the *sp* semantics for repetition leads to the following specification:

$$sp(\texttt{do i < n} \rightarrow \texttt{i := i + 1 od}, Q) \equiv (i \geq n) \wedge (\exists j : 0 \leq j : sp(IF^j, Q)).$$

The closed form for iteration suggests that the loop be unrolled k times, such that $sp(IF^k, Q)$ is true. If k is set to $n - start$, where *start* is the initial value of variable i, then the unrolled version of the loop would have the following form:

```
1.    i:= start;
2.    if
3.       i < n --> i:= i + 1;
4.    fi
5.    if
6.       i < n --> i:= i + 1;
7.    fi
8.    ...
9.    if
10.      i < n --> i:= i + 1;
11.   fi
```

Application of the rule for alternation (Expression (3.2)) yields the sequence of annotated code shown in Figure 3.1, where the goal is to derive the specification given

by the expression:

sp do i

```
{i = I ∧
i := start;
{i = start
if i < n
{sp i := i -
    ∀
    (i >=
    ≡
    i = s
if i < n
{sp i := i -
    ∀
    (i >=
    ≡
    i = s
    ∀
    (i >=
...
{i = start
    ∀
    (i >=
    ≡
    (i = n
if i < n
{sp i := i -
    ∀
    (i >=
    ≡
    i = n
```

Fig

In the construct

produced by a hum

by the expression:

$$sp(\texttt{do i < n} \rightarrow \texttt{i := i + 1 od}, (start < n) \wedge (i = start)).$$

---

1. $\{ (i = I) \wedge (start < n) \}$
2. `i:= start;`
3. $\{ (i = start) \wedge (start < n) \}$
4. `if i < n -> i:= i + 1 fi`
5. $\{ sp(i := i + 1, (i < n) \wedge (i = start) \wedge (start < n))$
6. $\quad \vee$
7. $\quad ((i >= n) \wedge (i = start) \wedge (start < n))$
8. $\quad \equiv$
9. $\quad ((i = start + 1) \wedge (start < n)) \}$
10. `if i < n -> i:= i + 1 fi`
11. $\{ sp(i := i + 1, (i < n) \wedge (i = start + 1) \wedge (start < n))$
12. $\quad \vee$
13. $\quad ((i >= n) \wedge (i = start + 1) \wedge (start < n))$
14. $\quad \equiv$
15. $\quad ((i = start + 2) \wedge (start + 1 < n))$
16. $\quad \vee$
17. $\quad ((i >= n) \wedge (i = start + 1) \wedge (start < n)) \}$
18. ...
19. $\{ ((i = start + (n - start - 1)) \wedge (start + (n - start - 1) - 1 < n))$
20. $\quad \vee$
21. $\quad ((i >= n) \wedge (i = start + (n - start - 2)) \wedge (start + (n - start - 2) - 1 < n))$
22. $\quad \equiv$
23. $\quad ((i = n - 1) \wedge (n - 2 < n)) \}$
24. `if i < n -> i:= i + 1 fi`
25. $\{ sp(i := i + 1, (i < n) \wedge (i = n - 1) \wedge (n - 2 < n))$
26. $\quad \vee$
27. $\quad ((i >= n) \wedge (i = n - 1) \wedge (n - 2 < n))$
28. $\quad \equiv$
29. $\quad (i = n) \}$

Figure 3.1: Annotated Source Code for Unrolled Loop

In the construction of specifications of iteration statements, knowledge must be introduced by a human specifier. For instance, in line 3.2.1 of Figure 3.1 the inductive

assertion that "i =

providing the info

were unrolled at le

he derived specifi

For this simple

formal definition o

ided strategy for

repetition stateme

3.2.2 Procedu

This section descri

se of non-recursiv

sing the following

where $\bar{x}$, $\bar{y}$, and

procedure, respecti

for input to the proc

used only for ou

dicate that the pa

notation "body , rep

are Q_1 are the pre

appears as

assertion that "$i = start + (n - start - 1)$" is made. This assertion is based on a specifier providing the information that $(n - start - 1)$ additions have been performed if the loop were unrolled at least $(n - start - 1)$ times. As such, by using loop unrolling and induction, the derived specification for the code sequence is $((n - 1 < n) \wedge (i = n))$.

For this simple example, we find that the solution is non-trivial when applying the formal definition of $sp(\texttt{DO}, Q)$. As such, the specification process must rely on a user-guided strategy for constructing a specification. A strategy for obtaining a specification of a repetition statement is given in Figure 3.2.

3.2.2 Procedural Abstractions

This section describes the construction of formal specifications from code containing the use of non-recursive procedural abstractions. A procedure declaration can be represented using the following notation

$$\texttt{proc}\ p\ (\ \textbf{value}\ \overline{x};\ \textbf{value-result}\ \overline{y};\ \textbf{result}\ \overline{z}\ );$$
$$\{P\}\langle\ \text{body}\ \rangle\{Q\}$$

where $\overline{x}$, $\overline{y}$, and $\overline{z}$ represent the **value**, **value-result**, and **result** parameters for the procedure, respectively. A parameter of type **value** means that the parameter is used only for input to the procedure. Likewise, a parameter of type **result** indicates that the parameter is used only for output from the procedure. Parameters that are known as **value-result** indicate that the parameters can be used for both input and output to the procedure. The notation $\langle$ body $\rangle$ represents one or more statements making up the "procedure", while $\{P\}$ and $\{Q\}$ are the precondition and postcondition, respectively. The *signature* of a procedure appears as

1. The follow
specificatio

- *invaria*
exit of

- *guard*
of each
invaria

When n
of the l
and R i

2. Begin by init
loop.

3. Query the us
interaction a
the loop.

4. Apply the str
by Step 3.

5. Using the spe
loop invarian
construction o

6. Using the rela
the loop by tak

Figure 3.2: St

ere the Kleene s

denotes th

1. The following criteria are the main characteristics to be identified during the specification of the repetition statement:

   - *invariant (P):* an expression describing the conditions prior to entry and upon exit of the iterative structure.
   - *guards (B):* Boolean expressions that restrict the entry into the loop. Execution of each guarded command, $B_i \rightarrow S_i$ terminates with P *true*, so that P is an invariant of the loop.

     $$\{P \wedge B_i\} S_i \{P\}, \text{ for } 1 \leq i \leq n$$

     When none of the guards is *true* and the invariant is *true*, then the postcondition of the loop should be satisfied ($P \wedge \neg BB \rightarrow R$, where $BB = B_1 \vee \ldots \vee B_n$ and R is the postcondition).

2. Begin by introducing the assertion "$Q \wedge BB$" as the precondition to the body of the loop.

3. Query the user for modifications to the assertion made in Step 2. This guided interaction allows the user to provide generalizations about arbitrary iterations of the loop.

4. Apply the strongest postcondition to the loop body S_i using the precondition given by Step 3.

5. Using the specification obtained from Step 4 as a guideline, query the user for a loop invariant. Although this step is non-trivial, techniques exist that aid in the construction of loop invariants [27, 26].

6. Using the relationship stated above ($P \wedge \neg BB \rightarrow R$), construct the specification of the loop by taking the negation of the loop guard, and the loop invariant.

Figure 3.2: Strategy for constructing a specification for an iteration statement

$$\texttt{proc}\ p : (\mathit{input\_type})^* \rightarrow (\mathit{output\_type})^* \tag{3.7}$$

where the Kleene star (*) indicates zero or more repetitions of the preceding unit, *input_type* denotes the one or more names of input parameters to the procedure p, and

output_type denotes the one or more names of output parameters of procedure p. A specification of a procedure can be constructed to be of the form

$$
\begin{array}{l}
\{\ \textbf{P:}\ U\ \} \\
\texttt{proc p} : E_0 \rightarrow E_1 \\
\langle body \rangle \\
\{\ \textbf{Q:}\ sp(body, U) \wedge U\ \}
\end{array}
$$

where E_0 is one or more input parameter types with attribute **value** or **value-result**, and E_1 is one or more output parameter types with attribute **value-result** or **result**. The postcondition for the body of the procedure, *sp(body,U)*, is constructed using the previously defined guidelines for assignment, alternation, sequence, and iteration as applied to the statements of the procedure body.

Gries [26] defines a theorem for specifying the effects of a procedure call using a total correctness model of execution. Given a procedure declaration of the above form, the following condition holds [26]

$$\{PRT : P^{\overline{x},\overline{y}}_{\overline{a},\overline{b}} \wedge (\forall \overline{u}, \overline{v} :: Q^{\overline{y},\overline{z}}_{\overline{u},\overline{v}} \Rightarrow R^{\overline{b},\overline{c}}_{\overline{u},\overline{v}})\}\ p(\overline{a}, \overline{b}, \overline{c})\ \{R\} \tag{3.8}$$

for a procedure call $p(\overline{a}, \overline{b}, \overline{c})$, where $\overline{a}$, $\overline{b}$, and $\overline{c}$ represent the actual parameters of type **value**, **value-result**, and **result**, respectively. Local variables of procedure p used to compute **value-result** and **result** parameters are represented using $\overline{u}$ and $\overline{v}$, respectively. Informally, the condition states that *PRT* must hold before the execution of procedure p in order to satisfy R. In addition, *PRT* states that the precondition for procedure p must hold for the parameters passed to the procedure and that the postcondition for procedure p implies R for each **value-result** and **result** parameter. The formulation of Equation (3.8) in terms of a partial correctness model of execution is identical, assuming that the procedure

is straight-lir

n abstraction

procedure de

all can be pe

assignment.

A proced

Figure 3.3, w

$\{PR\}$ is the

program after

is the specific

the specificat

values of loca

parameters to

n this manner,

nd a postcond

a procedural ab

nclude non-rec

e can annotate

$\overline{x}$, $\overline{z}$, $\overline{s}$, $\overline{y}$, a

body), $\overline{y}$ (after e

n Section 3.1.2,

ould be remove

ad recognizing t

is straight-line, non-recursive, and terminates. Using this theorem for the procedure call, an abstraction of the effects of a procedure call can be derived using a specification of the procedure declaration. That is, the construction of a formal specification from a procedure call can be performed by inlining a procedure call and using the strongest postcondition for assignment.

A procedure call $p(\overline{a}, \overline{b}, \overline{c})$ can be represented by the program block [26] found in Figure 3.3, where $\langle body \rangle$ comprises the statements of the procedure declaration for p, $\{\ PR\ \}$ is the precondition for the call to procedure p, $\{\ P\ \}$ is the specification of the program after the formal parameters have been replaced by actual parameters, $\{\ Q\ \}$ is the specification of the program after the procedure has been executed, $\{\ QR\ \}$ is the specification of the program after formal parameters have been assigned with the values of local variables, and $\{\ R\ \}$ is the specification of the program after the actual parameters to the procedure call have been "returned". By representing a procedure call in this manner, parameter binding can be achieved through multiple assignment statements and a postcondition R can be established by using the *sp* for assignment. Removal of a procedural abstraction enables the extension of the notion of straight-line programs to include non-recursive straight-line procedures. Making the appropriate *sp* substitutions, we can annotate the code sequence from Figure 3.3 to appear as shown in Figure 3.4 where $\overline{\alpha}$, $\overline{\beta}$, $\overline{\gamma}$, $\overline{\zeta}$, $\overline{\vartheta}$, and $\overline{\varphi}$ are the initial values of $\overline{x}$, $\overline{y}$ (before execution of the procedure body), $\overline{y}$ (after execution of the procedure body), $\overline{z}$, $\overline{b}$, and $\overline{c}$, respectively. Recall that in Section 3.1.2, we described how the existential operators and the textual substitution could be removed from the calculation of the *sp*. Applying that technique to assignments and recognizing that formal and actual **result** parameters have no initial values, and that

local variabl

sequence can

annotated co

where Q is de

3.3 Examp

AUTOSPEC is

in the construct

we describe the

AUTOSPEC

this chapter, de

include extend

semantics of the

such as assignm

loops. AUTOSPE

For instance, con

given without po

local variables are used to compute the values of the **value-result** parameters, the above sequence can be simplified using the semantics of *sp* for assignments to obtain the following annotated code sequence:

$$
\begin{array}{l}
\{\, PR \,\} \\
\overline{x},\overline{y} := \overline{a},\overline{b}; \\
\{\, P\text{: } PR \wedge \overline{x} = \overline{a} \wedge \overline{y} = \overline{b} \,\} \\
\langle body \rangle \\
\{\, Q \,\} \\
\overline{y},\overline{z} := \overline{u},\overline{v}; \\
\{\, QR\text{: } Q \wedge \overline{y} = \overline{u}^{\overline{y}}_{\overline{y}} \wedge \overline{z} = \overline{v}^{\overline{y}}_{\overline{y}} \,\} \\
\overline{b},\overline{c} := \overline{y},\overline{z}; \\
\{\, R\text{: } QR \wedge \overline{b} = \overline{y} \wedge \overline{c} = \overline{z} \,\}
\end{array}
$$

where Q is derived using $sp(\langle body \rangle, P)$.

3.3 Example

AUTOSPEC is a tool that has been developed to support the use of strongest postcondition in the construction of formal specifications from existing program code [6]. In this section we describe the use of AUTOSPEC to facilitate the analysis of programs.

AUTOSPEC accepts programs as input and using rules such as the ones described in this chapter, derives a formal specification of the input program. Our current investigations include extending the AUTOSPEC tool to support the formal strongest postcondition semantics of the C programming language as described in Chapter 5 [7]. For statements such as assignments and conditionals, AUTOSPEC is fully automated. When processing loops, AUTOSPEC allows a user to provide appropriate preconditions and postconditions. For instance, consider Figure 3.5 which contains output from a session of the AUTOSPEC system without pointers. The input program

```
begin
   ...
   { PR }
   p(ā, b̄, c̄)                                    ⇒
   { R }
   ...
end
```

```
begin
   declare x̄, ȳ, z̄, ū, v̄;
   ...
   { PR }
   x̄,ȳ := ā,b̄;
   { P }
   ⟨body⟩
   { Q }
   ȳ,z̄ := ū,v̄;
   { QR }
   b̄,c̄ := ȳ,z̄;
   { R }
   ...
end
```

Figure 3.3: Removal of procedure call $p(\bar{a}, \bar{b}, \bar{c})$ abstraction

```
x := 0;
do
   (M > power(2,x)) -> x := x + 1;
od;
```

computes the value of the smallest integer x such that for an input value I, $I \leq 2^x$. The initial precondition to the loop is computed as `((I = I_0) & (x = 0))`. On encountering the loop statement the user is prompted by the string "`Enter Precondition:`" to enter a precondition for an arbitrary iteration to the loop. Figure 3.2 [6] discusses guidelines for specifying the effects of loops. Using these guidelines, the precondition `((I = I_0) & (x = i))` is input by the user and

$\{ PR \}$
$\overline{x},\overline{y} := \overline{a},\overline{b};$
$\{ P: (\exists \overline{\alpha}, \overline{\beta} :: PR^{\overline{x},\overline{y}}_{\overline{\alpha},\overline{\beta}} \wedge \overline{x} = \overline{a}^{\overline{x},\overline{y}}_{\overline{\alpha},\overline{\beta}} \wedge \overline{y} = \overline{b}^{\overline{x},\overline{y}}_{\overline{\alpha},\overline{\beta}}) \}$
$\langle body \rangle$
$\{ Q \}$
$\overline{y},\overline{z} := \overline{u},\overline{v};$
$\{ QR: (\exists \overline{\gamma}, \overline{\zeta} :: Q^{\overline{y},\overline{z}}_{\overline{\gamma},\overline{\zeta}} \wedge \overline{y} = \overline{u}^{\overline{y},\overline{z}}_{\overline{\gamma},\overline{\zeta}} \wedge \overline{z} = \overline{v}^{\overline{y},\overline{z}}_{\overline{\gamma},\overline{\zeta}}) \}$
$\overline{b},\overline{c} := \overline{y},\overline{z};$
$\{ R: (\exists \overline{\vartheta}, \overline{\varphi} :: QR^{\overline{b},\overline{c}}_{\overline{\vartheta},\overline{\varphi}} \wedge \overline{b} = \overline{y}^{\overline{b},\overline{c}}_{\overline{\vartheta},\overline{\varphi}} \wedge \overline{c} = \overline{z}^{\overline{b},\overline{c}}_{\overline{\vartheta},\overline{\varphi}}) \}$

Figure 3.4: Code annotation for procedure call

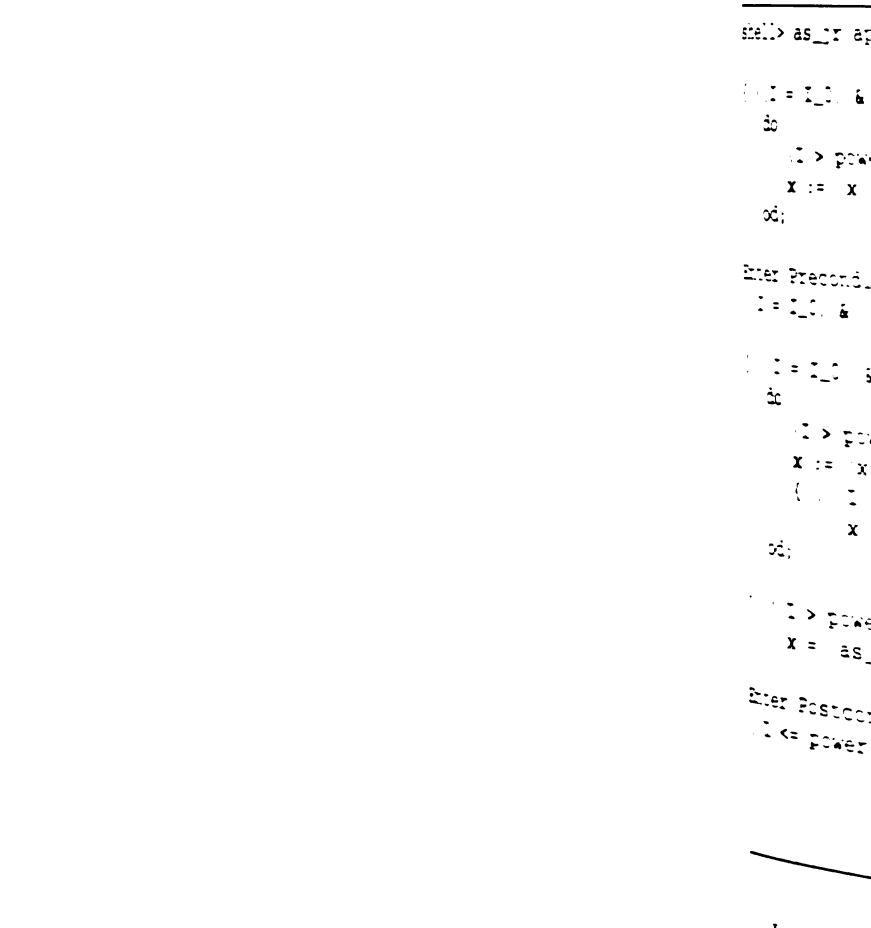
the term `(I > power(2,x))` is automatically generated and conjuncted to the precondition. The resulting preliminary specification of the postcondition for the loop that is generated by AUTOSPEC is as follows:

```
(((I > power(2,as_const2)) & ((I = I_0) & (as_const2 = i))) &
(x = (as_const2 + 1)))
```

where "&" is the logical connective "$\wedge$". The user is then prompted by the string "`Enter Postcondition:`" to enter a postcondition. From the preliminary specification we can deduce that while the guard is true, $(\forall i : 0 \leq i < x : (I > 2^i))$. Furthermore, after execution completes $I \leq 2^x$. Therefore, the final specification can be entered as

```
((I <= power(2,x)) & (forall i :  ((0 <= i) & (i < x)) :
(I > power(2,i))))
```

which states that after execution of the loop, $I \leq 2^x$ and that for every integer $0 \leq i < x$, $I > 2^i$.



In addition

specification p

mared by use

```
shell> as_jr approxI

{ ((I = I_0) & (x = 0)) }
   do
      (I > power( 2, x)) ->
      x := (x + 1);
   od;

Enter Precondition:
((I = I_0) & (x = i))

{ ((I = I_0) & (x = 0)) }
   do
      (I > power( 2, x)) ->
      x := (x + 1);
      { (((I > power(2,as_const2)) & ((I = I_0) & (as_const2 = i))) &
          (x = (as_const2 + 1))) }
   od;

{ (((I > power(2,as_const2)) & ((I = I_0) & (as_const2 = i))) &
    (x = (as_const2 + 1))) }

Enter Postcondition:
((I <= power(2,x)) & (forall i : ((0 <= i) & (i < x)) : (I > power(2,i))))
```

Figure 3.5: User Consultation

In addition to supporting the ability to have a user provide guidance during the specification process, AUTOSPEC supports syntactic and semantic verification of the input entered by users by using an integrated syntax checker and theorem prover.

Chapter 4

Strongest Postcondition Semantics of Pointers

Many modern programming languages support the use of pointer variables, including C and C++. This chapter describes how we extended the strongest postcondition predicate transformer to include the formal semantics of programs with pointers [28]. The semantics are defined for a modified Dijkstra language that has been extended to include pointer variables. The extension of the strongest postcondition semantics to include pointers facilitates the use of strongest postcondition for reverse engineering a more general set of programs.

4.1 Pointers

Using terminology of the C programming language [29], a *pointer* is a variable that contains the address of a variable. A common use of pointers is the creation of *aliases*, which refers to the fact that several names can be used to refer to a single data object. For instance, the statement "`x := @a`", where `x` is a pointer and `a` is some data variable, creates an alias, thus operations involving `x` and `a` are synonymous. The notation "`*p`" indicates a *dereference* of the pointer `p` in order to access the value of the referenced object. There are four different classes of alias detection: intraprocedural may-alias,

interprocedural may-alias, intraprocedural must-alias, and interprocedural must-alias. The term *may-alias* refers to the fact that given two variables, during *some execution* of a program, the variables are aliases for one another. The term *must-alias* means that during *all executions* of the program, the variables will be aliases for one another. The terms *interprocedural* and *intraprocedural* indicate the context of the aliasing, where interprocedural is global and intraprocedural is local. Compile-time analysis of programs to detect aliasing has long been recognized as difficult. In fact, it has been proven that static analysis to detect aliases is undecidable [30]. This research does not address may/must-aliasing problems directly although the intention in the development of the formal semantics for pointers is to provide a theoretically rich formalism that can be used to aid may/must-alias analysis.

In addition to having may/must-alias detection, alias detection techniques can be *flow-sensitive* or *flow-insensitive*. A technique is flow-sensitive if control structures are factored into the detection algorithm. The techniques that we suggest are flow-sensitive although, again, we do not directly address alias detection.

4.2 Pointer Semantics

A *pointer* is a variable that contains the address of some data object. Pointers can be assigned in a number of different ways including heap allocation and direct addressing of a variable. For instance, the C-like command "`p := @k`" assigns the address of the variable `k` to pointer variable `p`. As such, the pointer variable `p` *points-to* variable `k`. This section describes the strongest postcondition semantics of pointers.

4.2.1 Failu

The strongest p

statement "x :

hich states th

every free occu

wo lemmas for

emma if the p

hen after the t

$\exists x.\, C_t^x \ldots x =$

logically equival

free variable in e

is logically equiv

In a naive tre

statement to poi

ample in Figu

is a constant. G

and V is a consta

$:= {}^*p;\ {}^*p :=$

ntactics for assi

he lemmas in Se

4.2.1 Failure of Conventional Assignment Semantics

The strongest postcondition semantics of the assignment statement is as follows. Given a statement "`x := e`" and a precondition Q:

$$sp(\texttt{x := e}, Q) \equiv (\exists v :: Q_v^x \wedge x = e_v^x), \tag{4.1}$$

which states that after the execution of "`x := e`" there exists some variable v such that every free occurrence of x in Q is replaced with v and $x = e_v^x$. Section 3.1.2 describes two lemmas for eliminating the existential quantification in Expression (4.1). In the first lemma, if the precondition Q is of the form $C \wedge (x = u)$, where C is a logical expression, then after the textual substitution of variable x with v in Q, Expression (4.1) reads as $(\exists v :: C_v^x \wedge (v = u) \wedge x = e_v^x)$. Since $(v = u)$, the expression $(\exists v :: C_v^x \wedge (v = u) \wedge x = e_v^x)$ is logically equivalent to $C_u^x \wedge (u = u) \wedge x = e_u^x$. In the second lemma, if x does not appear as a free variable in either the logical expression Q or the expression e, then $(\exists v :: Q_v^x \wedge x = e_v^x)$ is logically equivalent to $Q \wedge x = e$.

In a naive treatment of pointers, we can attempt to apply the semantics of the assignment statement to pointer variables. However, doing so causes various problems. Consider the example in Figure 4.1 where `p` and `q` are pointer variables, `d` is a typed variable, and `e` is a constant. Given the precondition $\{*q = Y\}$, where $*q$ is a dereference of an object and Y is a constant, the strongest postcondition of the statement sequence "`p := q; d := *p; *p := e`" is $\{d = v \wedge p = q \wedge *q = Y \wedge *p = e\}$ when the conventional semantics for assignment is used for pointer assignments. This specification, derived using the lemmas in Section 3.1.2, states that after execution of the sequence, d has value v, p

and q point to

this specificat

is a contradict

{ q = Y }
p := q;
{ Sta
d := *p;
{ Sta
*p := e;
{ Sta
d = Y * p = q

The remain

of pointer ope

assignment ser

4.2.2 Mem

In the C progra

or stack. The m

memory consist

value. A diagra

labeled N indica

the values. In the

convention we

* In our exampl

and q point to the same object, and that the value of $*q = Y$ and $*p = e$. The problem with this specification is that while $p = q$ (i.e., pointers p and q refer to the same object), there is a contradiction in the conjuncts $*q = Y$ and $*p = e$.

$\{ *q = Y \}$
`p := q;`
$\{ (\exists v :: (*q = Y)^p_v \wedge p = q) \equiv (p = q \wedge *q = Y)$ ⟨Lemma 3.1.2⟩ $\}$
`d := *p;`
$\{ (\exists v :: (p = q \wedge *q = Y)^d_v \wedge d = *p) \equiv (d = *p \wedge p = q \wedge *q = Y)$ ⟨Lemma 3.1.2⟩ $\}$
`*p := e;`
$\{ (\exists v :: (d = *p \wedge p = q \wedge *q = Y)^{*p}_v \wedge *p = e) \equiv$
$(d = v \wedge p = q \wedge *q = Y \wedge *p = e)$ ⟨Lemma 3.1.2⟩ $\}$

Figure 4.1: A simple pointer example

The remainder of this section presents a model for describing the formal semantics of pointer operations that overcome the problems that occur when using conventional assignment semantics.

4.2.2 Memory Model

In the C programming language, variables can be allocated from heap storage, registers, or stack. The model used in this paper for representing memory is cell-based, where the memory consists of a large number of storage cells. Each cell is named and contains a value. A diagram of this model is shown in Figure 4.2, where the entries in the column labeled *N* indicate the names of the cells, and the entries in the column labeled *V* indicate the values. In the diagram, data objects x, y, and z have values a, b, and c, respectively. As a convention we use "$n.V$", where n is a cell name, to denote the value of the data object n. In our example, $x.V = a$, $y.V = b$, and $z.V = c$.

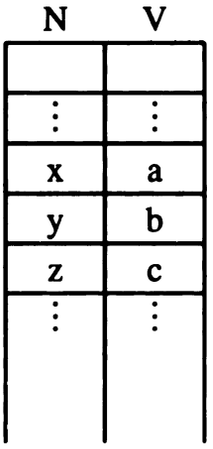


Figure 4.2: Cell Memory Model

4.2.3 Extending the Model for Pointers

A pointer can be assigned by heap allocation, pointer assignment, or alias assignment. Examples can be found in Table 4.1. Different alternatives for representing the use of pointers within the context of the cell memory model are available including the use of indirection where if pointer p points to some variable v, then the value of p is v.

Type	Example
heap allocation	`p := new T`
pointer assignment	`p := q`
alias assignment	`p := @x`

Table 4.1: Pointer Assignments

Consider the set of data objects N and the set of pointers M that are currently allocated at some step during the execution of a particular program. Assuming that all the pointers in M point to data objects (not necessarily distinct) in N, using the equivalence relation

"=" where point

M such that eac

of a particular e

any other memb

x and y and po

pointers s and

form one equiva

equivalence clas

The equivale

since the execut

each set as is th

pointer assignm

is an associated

assume that a da

equivalence clas

equivalence clas

M(

12.4 Points

In this section we

of which are used

Let M be the

the Figure 4.4

"=" where pointer $p = q$ if and only if p and q point to the same object, we can partition M such that each partition is an equivalence class. As such, any operation on a member of a particular equivalence class is behaviorally equivalent to performing an operation on any other member of the same equivalence class. For instance, suppose we have variables `x` and `y` and pointers `p`, `q`, `r`, `s`, and `t`. Let pointers `p`, `q`, and `r` point to variable `x`, and pointers `s` and `t` point to variable `y`. Since `p`, `q`, and `r` point to the same variable `x`, they form one equivalence class, and since `s` and `t` point to the other variable, they form another equivalence class.

The equivalence classes within the set of pointers M can be considered to be *dynamic* since the execution of a programming statement can possibly rearrange the members of each set as is the case when pointer variables are either reused in heap allocation or a pointer assignment. Figure 4.3 depicts the extension of the memory model where there is an associated equivalence class in M for each memory cell. For consistency sake we assume that a data object can reference itself and, as such, is a member of the associated equivalence class. For example, the data object `y` with pointers `s` and `t` has an associated equivalence class from M with members { `y`, `s`, `t` }. We refer to this equivalence class as $M[y]$.

4.2.4 Points-to and Coset

In this section we define the semantics of the *points-to* relation and the *coset* function, both of which are used to formally describe the behavior of pointer operations.

Let M be the set of pointers, N be the set of allocated data objects, and $\mathcal{B}$ be the Boolean type. Figure 4.4 defines the $\gtrdot$ (pronounced "*points-to*") relation. The primary use of the

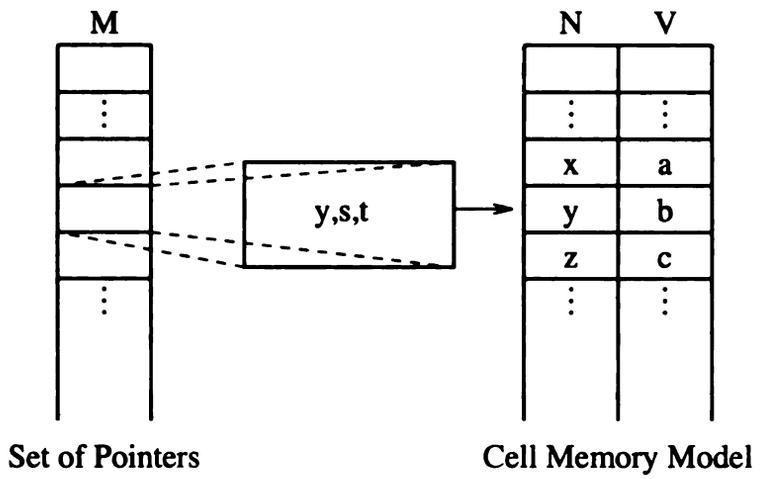


Figure 4.3: Pointer Extensions to the Memory Model

points-to relation is for making assertions about pointers and their relation to specific data objects. That is, it asserts that a pointer is in the equivalence class associated to a particular data object. Informally, the *points-to* relation is a heterogeneous relation on $\{M \cup N\} \times N$. The first axiom states that when data objects $o1$ and $o2$ are both in the set N that $o1 \gg o2$ is true if and only if $o1 = o2$, where "$o1 = o2$" when $o1$ and $o2$ are the same data object. As such, a data object can only reference itself and never references another data object. The second axiom states that for a pointer $p \in M$ and a data object $o \in N$, $p \gg o$ if and only if $p \in M[o]$. That is, p points-to o if and only if pointer p is an element of the equivalence class of o. Equivalently, a pointer p points to a data object o if it is in the equivalence class $M[o]$.

The *coset* function is defined in Figure 4.5. The primary use of the *coset* function is to identify a dereferenced object. Informally, the *coset* function maps pointers to data objects,

where a pointer

equivalence clas

4.2.5 Assig

Given the defin

model, we must

pointer) variable

precondition Q

postcondition fo

$$\_ \gg \_ : \{M \cup N\} \times N \mapsto \mathcal{B}$$

Axioms :

$$(\forall o1, o2 : o1, o2 \in N : o1 \gg o2 \Leftrightarrow o1 = o2)$$
$$(\forall p, o : p \in M \wedge o \in E : p \gg o \Leftrightarrow p \in M[o])$$

Figure 4.4: The *points-to* relation

where a pointer p maps to a data object o if $p \in M[o]$. If a pointer does not belong to any equivalence class then the *cost* function is undefined.

$$coset : M \mapsto \{N \cup \{undefined\}\}$$

$$coset(p) = \begin{cases} o & \textit{if and only if } p \gg o, \\ undefined & otherwise. \end{cases}$$

Figure 4.5: The *coset* function

4.2.5 Assignment Revisited

Given the definition of the *points-to* operator and the semantics of the equivalence class model, we must redefine the *sp* semantics of the assignment statement for simple (non-pointer) variables to be consistent with the model. Given a statement "x := e" and a precondition Q where x is a non-pointer variable and e is an expression, the strongest postcondition for assignment statements is:

$$sp(\mathtt{x} := \mathtt{e}, Q) \equiv (\exists v :: Q_v^{x.V} \wedge x.V = \tilde{e}_v^{x.V}),$$

which states t

every free occ

Informally, the

variable u that

a term in e is r

identified by c

referring to the

4.2.6 Heap

When a pointer

such that all th

method for ass

memory allocat

type. Informally

and the pointer p

by introducing a

equivalence class

removed from t

$\Psi p := r$

which states that after the execution of "x := e" there exists some variable v such that every free occurrence of $x.V$ in Q is replaced with v and $x.V = \tilde{e}^x_v$. Formally, $\tilde{e}$ means:

$$\tilde{e} \Leftrightarrow \forall u : (variable(u) \wedge term(u, e) \rightarrow e^u_{u.V}) \wedge \forall p : (pointer(p) \wedge term(*p, e)) \rightarrow e^{*p}_{coset(p).V}$$

Informally, the notation $\tilde{e}$ indicates that the expression e is transformed so that every simple variable u that is a term in e is replaced by $u.V$, and every pointer dereference *p that is a term in e is replaced by $coset(p).V$, where $coset(p).V$ refers to the value of the object identified by $coset(p)$. This formalization ensures that there is a consistent notation for referring to the values of data objects.

4.2.6 Heap Allocation

When a pointer is assigned a "value" then that pointer is placed into an equivalence class such that all the members of the equivalence class point to the same data object. One method for assigning a value to a pointer is through heap memory allocation. Heap memory allocation has the form "p := new T" where p is a pointer and T is a data type. Informally, upon allocation of heap memory, a new data object of type T is created and the pointer p is used to reference the object. In our model this action is represented by introducing a new entry o in N with an undefined value in V, and adding p to the equivalence class $M[o]$. In addition, if p was previously in some equivalence class $M[k]$, it is removed from that set. Formally we can state this condition as follows:

$$sp(\texttt{p := new T}, Q) \equiv (\exists c : c \in N : Q^p_c) \wedge p \gg o \wedge o.V = undefined, \qquad (4.2)$$

where o is a

with the term

previously be

Finally, the te

As an exampl

As such, a

some new objec

42.7 Pointe

Another way of a

"$p := \&x$". In

pointer p to the e

where o is a new data object. The textual substitution of every free occurrence of p in Q with the term $c \in N$ ensures that p is removed from any equivalence class that it may have previously been associated, and the assertion $p \gtrdot o$ places p into the equivalence class $M[o]$. Finally, the term $o.V = \mathit{undefined}$ asserts that the value of the new object o is undefined. As an example, let precondition Q be $\{q \gtrdot o1\}$ and statement S be "`q := new T`". Then

$$
\begin{aligned}
sp(S, Q) &\equiv sp(\texttt{q := new T}, q \gtrdot o1) \\
&\quad \langle \text{Expression (4.2)} \rangle \\
&\quad \equiv (\exists c : c \in N : (q \gtrdot o1)^q_c) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined} \\
&\quad \langle \text{Textual substitution of } q \text{ with } c \rangle \\
&\quad \equiv (\exists c : c \in N : c \gtrdot o1) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined} \\
&\quad \langle \text{Points-to axiom (Figure 4.4) applied to } c \gtrdot o1 \rangle \\
&\quad \equiv (\exists c : c \in N : c = o1) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined} \\
&\quad \langle \text{Trading [16]} \rangle \\
&\quad \equiv (\exists c : c = o1 : c \in N) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined} \\
&\quad \langle \text{One-point rule [16] with } c = o1,\ o1 \in N \equiv \mathit{true} \rangle \\
&\quad \equiv q \gtrdot o2 \wedge o2.V = \mathit{undefined}
\end{aligned}
$$

As such, after the execution of the statement "`q := new T`", the pointer q points to some new object $o2$.

4.2.7 Pointer Assignment

Another way of assigning a value to a pointer is via direct aliasing as in the C-like command "`p := @x`". In terms of the equivalence class model, the pointer alias assignment adds the pointer p to the equivalence class $M[x]$. The formal semantics of this command is similar

to the heap allocat

Expression (4.3) s

free occurrence of

placed into the eq

S be "p := ex"

Hence, the poi

The final way

form "p := q" is

class model, the po

to the heap allocation case. Formally the semantics is as follows:

$$sp(\texttt{p} := \texttt{@x}, Q) \equiv (\exists c : c \in N : Q_c^p) \wedge p \gg x. \tag{4.3}$$

Expression (4.3) states that after executing the statement p := @x that $p \gg x$ and that every free occurrence of p in Q is replaced with c. This relationship ensures that the pointer p is placed into the equivalence class associated to the variable x. As an example, let statement S be "p := @x" and let precondition Q be "$\{p \gg o1\}$". The sp derivation is as follows.

$$\begin{aligned}
sp(S, Q) &\equiv sp(\texttt{p} := \texttt{@x}, p \gg o1) \\
&\langle \text{Expression (4.3)} \rangle \\
&\equiv (\exists c : c \in N : (p \gg o1)_c^p) \wedge p \gg x \\
&\langle \text{Textual substitution of } p \text{ with } c \rangle \\
&\equiv (\exists c : c \in N : (c \gg o1)) \wedge p \gg x \\
&\langle \text{Points-to axiom applied to } c \gg o1 \rangle \\
&\equiv (\exists c : c \in N : c = o1) \wedge p \gg x \\
&\langle \text{Trading} \rangle \\
&\equiv (\exists c : c = o1 : c \in N) \wedge p \gg x \\
&\langle \text{One-point rule with } c = o1,\ o1 \in N \equiv \mathit{true} \rangle \\
&\equiv p \gg x
\end{aligned}$$

Hence, the pointer p points to the data object x.

The final way that a pointer can be assigned a value occurs when a statement of the form "p := q" is executed, where p and q are pointers. In the terms of the equivalence class model, the pointer assignment adds the pointer p to the class that contains pointer q.

In this case the

Expression (4.

set q. For e

$p > o2\}$". Ther

As such, afte

ject.

4.2.8 Value

In the C program

accessed using th

In this case the formal semantics is expressed as

$$sp(\mathtt{p} := \mathtt{q}, Q) \equiv (\exists c : c \in N : Q_c^p) \wedge p \gg coset(q). \tag{4.4}$$

Expression (4.4) states that after execution of the pointer assignment, p points to the object $coset(q)$. For example, let statement S be "p := q" and precondition Q be "$\{q \gg o1 \wedge p \gg o2\}$". Then

$$\begin{aligned}
sp(S, Q) &\equiv sp(\mathtt{p} := \mathtt{q}, q \gg o1 \wedge p \gg o2) \\
&\quad \langle \text{Expression (4.4)} \rangle \\
&\quad \equiv (\exists c : c \in N : (q \gg o1 \wedge p \gg o2)_c^p) \wedge p \gg coset(q) \\
&\quad \langle \text{Textual substitution of } p \text{ with } c \rangle \\
&\quad \equiv (\exists c : c \in N : (q \gg o1 \wedge c \gg o2)) \wedge p \gg coset(q) \\
&\quad \langle \text{Points-to axiom applied to } c \gg o2) \rangle \\
&\quad \equiv (\exists c : c \in N : (q \gg o1 \wedge c = o2)) \wedge p \gg coset(q) \\
&\quad \langle \text{Trading} \rangle \\
&\quad \equiv (\exists c : c = o2 : c \in N \wedge q \gg o1) \wedge p \gg coset(q) \\
&\quad \langle \text{One-point rule with } c = o2, o2 \in N \equiv true \rangle \\
&\quad \equiv q \gg o1 \wedge p \gg coset(q) \\
&\quad \langle \text{Definition of coset} \rangle \\
&\quad \equiv q \gg o1 \wedge p \gg o1
\end{aligned}$$

As such, after the execution of "p := q", the pointers p and q reference the same data object.

4.2.8 Value Assignment

In the C programming language, the value of the data object that a pointer references is accessed using the notation "*p", where p is a pointer variable. Using the same notation

convention, an

:= e", where

of "p sets the v

to a dereference

$sp\ ^{\prime\prime}p\ :$

where T is the t

type *integer*, the

dereferenced by

semantics state t

. Addi

and precondit

states that p point

derivation is as fo

$sp(S, Q) \equiv sp($

{Expres

$\equiv (\exists$

{Coset c

$\equiv (\exists$

{Trading

$\equiv (\exists$

{One-po

$\equiv p \geq o$

{Definitio

$\equiv p \geq o$

convention, an assignment to the data object is achieved using a command of the form "`*p := e`", where `e` is an expression. In terms of the equivalence class model, the assignment of `*p` sets the value of the data object $coset(p)$ to `e`. Formally, the semantics of assignment to a dereferenced data object is as follows:

$$sp(\texttt{*p := e}, Q) \equiv (\exists v : v \in T : Q_v^{coset(p).V} \wedge coset(p).V = \tilde{e}_v^{coset(p).V}) \qquad (4.5)$$

where T is the type of the data object, and v is a value of that type. For instance, if T is the type *integer*, then v is some integer. The variable v represents the value of the data object dereferenced by `*p` prior to the execution of the statement "`*p := e`". Informally, the semantics state that after execution of the statement "`*p := e`", `*p` will have the value $\tilde{e}_v^{coset(p).V}$. Additionally, $Q_v^{coset(p).V}$ will be true. For example, let statement S be "`*p := 5`" and precondition Q be $\{p \gtrdot o1 \wedge o1.V = n \wedge o1.V \leq k\}$. Informally the precondition states that `p` points to object $o1$, the value of $o1$ (denoted $o1.V$) is n, and $o1.V \leq k$. The sp derivation is as follows.

$sp(S, Q) \equiv sp(\texttt{*p := 5}, p \gtrdot o1 \wedge o1.V = n \wedge o1.V \leq k)$

⟨Expression (4.5)⟩

$\equiv (\exists v : v \in T : (p \gtrdot o1 \wedge o1.V = n \wedge o1.V \leq k)_v^{coset(p).V} \wedge coset(p).V = 5)$

⟨Coset definition and textual substitution of $coset(p).V$ with v⟩

$\equiv (\exists v : v \in T : p \gtrdot o1 \wedge v = n \wedge v \leq k \wedge coset(p).V = 5)$

⟨Trading⟩

$\equiv (\exists v : v \in T \wedge v = n : p \gtrdot o1 \wedge v \leq k \wedge coset(p).V = 5)$

⟨One-point rule with $v = n$⟩

$\equiv p \gtrdot o1 \wedge n \leq k \wedge coset(p).V = 5$

⟨Definition of coset⟩

$\equiv p \gtrdot o1 \wedge n \leq k \wedge o1.V = 5$

Hence, after the execution of "*p := 5", the data object pointed to by p has value 5.

4.2.9 Value Dereference

The command for observing the value of a pointer dereference has the form "x := *p", where x is a variable and p is a pointer. In terms of the equivalence class model, the value dereference *p refers to the value of the data object associated to the equivalence class containing p. That is, *p refers to $coset(\mathrm{p}).V$. The formal semantics of a value dereference is as follows:

$$sp(\mathrm{x} := \mathrm{*p}, Q) \equiv (\exists v : v \in T : Q_v^{x.V} \wedge x.V = coset(p).V) \tag{4.6}$$

where T is the type of the data object, and v is a value of that type. Informally, Expression (4.6) states that after the execution of a statement with *p on the right hand side of an assignment, the left hand side of the assignment takes on the value of the object that has been dereferenced. The term $Q_v^{x.V}$ states that every free occurrence of $x.V$ in Q is replaced with the value of x previous to executing the statement "x := *p". As an example, let statement S be "x := p" and let precondition Q be $\{p \geqslant o1 \wedge o1.V = n \wedge x.V = y\}$. The *sp* derivation proceeds as follows.

$$\begin{aligned}
sp(S, Q) &\equiv sp(\mathrm{x} := \mathrm{*p}, p \geqslant o1 \wedge o1.V = n \wedge x.V = y) \\
&\quad \langle \text{Expression (4.6)} \rangle \\
&\quad \equiv (\exists v : v \in T : (p \geqslant o1 \wedge o1.V = n \wedge x.V = y)_v^{x.V} \wedge x.V = coset(p).V) \\
&\quad \langle \text{Textual substitution of } x.V \text{ with } v \rangle \\
&\quad \equiv (\exists v : v \in T : p \geqslant o1 \wedge o1.V = n \wedge v = y \wedge x.V = coset(p).V)
\end{aligned}$$

T
≡
C
≡
D
≡

This states

4.3 Exam

Figure 4.6 co

chapter as we

pointers. Fig

demonstrating

shows a progr

pointer resolut

the AUTOSPEC

4.3.1 ali

The alias pr

of conventiona

and the failure

se. In this sec

AUTOSPEC tool

⟨Trading⟩
$\equiv (\exists v : v \in T \wedge v = y : p \gg o1 \wedge o1.V = n \wedge v = y \wedge x.V = coset(p).V)$
⟨One-point rule with $v = y$⟩
$\equiv p \gg o1 \wedge o1.V = n \wedge x.V = coset(p).V)$
⟨Definition of coset⟩
$\equiv p \gg o1 \wedge o1.V = n \wedge x.V = o1.V)$

This states that the new value of $x.V$ is equivalent to the value of the data object $o1$.

4.3 Examples

Figure 4.6 contains three programs for illustrating the pointer semantics described in this chapter as well as for showing the use of an automated tool for analyzing programs with pointers. Figure 4.6(a) is the program from Figure 4.1. Figure 4.6(b) is a program for demonstrating how aliases are resolved using the pointer semantics, and Figure 4.6(c) shows a program with a conditional statement and how the conditional statement impacts pointer resolution. The specifications in this section were all automatically generated by the AUTOSPEC tool.

4.3.1 `alias`

The `alias` program is shown in Figure 4.6(a). Figure 4.1 demonstrated the application of conventional strongest postcondition semantics for assignment to the `alias` program and the failure of those semantics to correctly specify the behavior in the context of pointer use. In this section we describe the semantics of the specification constructed using the AUTOSPEC tool with support for the pointer semantics presented in Section 4.2.

```
program alias(
   inputs: int e;
   int *q;)

decl
   int d;
   int *p;
lced
begin

   p := q;
   d := *p;
   *p := e;

end
```

(a)

```
program manyvars()

decl
   int z; int u; int *r;
   int *q;
lced

begin

   r := @u;
   z := 0;
   q := @z;
   *r := 1;
   *q := *r;

end
```

(b)

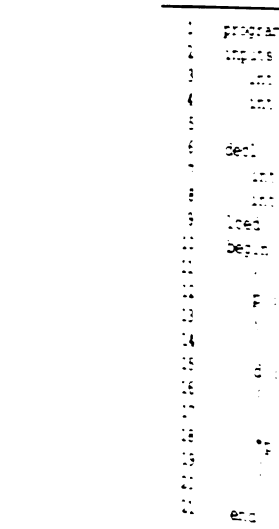
```
program maxThresh(
   inputs: int e; int x; int y;
   outputs: int *z; )

begin

   if ( x > y ) -> z := @x;
   || ( x <= y ) -> z := @y;
   fi;
   *z := *z + e;

end
```

(c)

Figure 4.6: Three Sample Programs: (a) `alias` (b) `manyvars` (c) `maxThresh`

Figure 4.7 contains the output of AUTOSPEC when executed using the `alias` program as input. The precondition appears as the logical formula enclosed within the curly braces "{" and "}" following the keyword `begin` at line `11`. It is derived from the parameter and variable declarations, `int e; int *q;`, and `int d; int *p;`, respectively. Informally, the precondition states that the declared variable `d` has initial value (`d.V`) equivalent to some constant `d_0`, parameter `e` has initial value (`e.V`) equivalent to some constant `e_0`, and parameter `*q` points to some object `obj_q`. Additionally, the initial value of `obj_q` (denoted `obj_q.V`) is equivalent to some constant `q_0`. After execution of the first statement of the program the pointer `p` points to the object identified by *coset*(`q`) which is specified by the conjunct `(p .> coset(q))` in the specification at lines `13-14`, where "`.>`" is the points-to relation.

The final specification of the `alias` program (lines `19-20`) is the following:



(((cons
e.V = e
obj_q.V

where "&" is th

alias progra

const4 such t

object obj_q.

4.3.2 many

The manyvar

difficulty in ur

uses two integ

used to create

value assignme

```
 1    program alias (
 2    inputs :
 3       int e;
 4       int *q;
 5    )
 6    decl
 7       int d;
 8       int *p;
 9    lced
10    begin
11       { ((d.V = d_0) & (((obj_q.V = q_0) & (q .> obj_q)) & (e.V = e_0))) }
12       p := q;
13       { (((d.V = d_0) & (((obj_q.V = q_0) & (q .> obj_q)) & (e.V = e_0))) &
14          (p .> coset( q ))) }
15       d := *p;
16       { (((((_cnst2 = d_0) & (((obj_q.V = q_0) & (q .> obj_q)) & (e.V = e_0))) &
17          (p .> coset( q ))) & (d.V = coset( p ).V)) }
18       *p := e;
19       { (((((_cnst2 = d_0) & (((_cnst4 = q_0) & (q .> obj_q)) & (e.V = e_0))) &
20          (p .> coset( q ))) & (d.V = _cnst4)) & (obj_q.V = e.V)) }
21    end
```

Figure 4.7: Output of AUTOSPEC applied to Figure 4.1

```
(((((_cnst2 = d_0) & (((_cnst4 = q_0) & (q .> obj_q)) &
(e.V = e_0))) & (p .> coset( q ))) & (d.V = _cnst4)) &
(obj_q.V = e.V))
```

where "&" is the logical connective "∧". The specification states that after executing the `alias` program, `obj_q.V` has value `e.V` such that `e.V = e_0` and `d.V` has value `_cnst4` such that `_cnst4 = q_0`. In addition, pointers `p` and `q` are aliases for the same object `obj_q`.

4.3.2 manyvars

The `manyvars` program is shown in Figure 4.6(b). This program demonstrates the difficulty in understanding programs that use a high degree of aliasing. The program uses two integer variables and two pointer variables, where the pointers `r` and `q` are used to create aliases of variables `u` and `z`, respectively. In addition, a number of value assignments are made to the primary variables (e.g., `z := 0;`) and aliases (e.g.,

'r := 1;). Fig

by the AUTOSPEC

an alias of variab

at line 12. The

"" to the data

... is the data v

lines 19-20.

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

The final sp

((((

`*r := 1;`). Figure 4.8 contains the specification of the `manyvars` program as generated by the AUTOSPEC system. The first statement of the program at line `11` (`r := @u`) creates an alias of variable u and is specified by the conjunct `(r .> u)` in the expression:

```
(((u.V = u_0) & (z.V = z_0)) & (r .> u))
```

at line `12`. The fourth statement in the program at line `18` (`*r := 1`) assigns the value "1" to the data object identified by *coset*(`r`) which, on account of the conjunct `(r .> u)`, is the data variable u. Hence, the conjunct `(u.V = 1)` appears in the specification at lines `19-20`.

```
 1        program manyvars (
 2        )
 3        decl
 4           int z;
 5           int u;
 6           int *r;
 7           int *q;
 8        lced
 9        begin
10           { ((u.V = u_0) & (z.V = z_0)) }
11           r := @u;
12           { (((u.V = u_0) & (z.V = z_0)) & (r .> u)) }
13           z := 0;
14           { ((((u.V = u_0) & (_cnst2 = z_0)) & (r .> u)) & (z.V = 0)) }
15           q := @z;
16           { (((((u.V = u_0) & (_cnst2 = z_0)) & (r .> u)) & (z.V = 0)) &
17                 (q .> z)) }
18           *r := 1;
19           { ((((((_cnst5 = u_0) & (_cnst2 = z_0)) & (r .> u)) & (z.V = 0)) &
20                  (q .> z)) & (u.V = 1)) }
21           *q := *r;
22           { (((((((_cnst5 = u_0) & (_cnst2 = z_0)) & (r .> u)) & (_cnst7 = 0)) &
23                  (q .> z)) & (u.V = 1)) & (z.V = coset( r ).V)) }
24        end
```

Figure 4.8: AUTOSPEC applied to the `manyvars` program

The final specification of the `manyvars` program (lines `22-23`) is the following:

```
((((((((_cnst5 = u_0) & (_cnst2 = z_0)) & (r .> u)) &
      (_cnst7 = 0)) & (q .> z)) & (u.V = 1)) &
      (z.V = coset( r ).V))
```

which states t

coset(x).

z.V. is 1.

4.3.3 max

The maxThr

demonstrate t

many objects

alias the max

program adds

Figure 4.9

the AUTOSPE

following (as

```
((x.V >
 z .> x) )
e.0))) & (
```

which states t

pointer z poin

of variable x a

The final s

```
((((cons
= e.0)))) &
cons5 =
cons5 = C
```

This specifica

```
os + e.y
```

which states that after the execution of the program, z.V has a value equivalent to that of coset(r).V. Since *coset*(r) = u, and (u.V = 1), the value of variable z, denoted z.V, is 1.

4.3.3 maxThresh

The maxThresh program is shown in Figure 4.6(c). The purpose of this program is to demonstrate the use of AUTOSPEC in specifying the cases where pointers may reference many objects rather than just a single object. The maxThresh program sets pointer z to alias the maximum of two input variables x and y. After determining the maximum, the program adds a threshold e to the maximum.

Figure 4.9 contains the specification of the maxThresh program as generated by the AUTOSPEC system. After the execution of the if-fi statement (lines 11-18), the following (as shown in lines 19-20) is true:

```
((((x.V > y.V) & ((y.V = y_0) & ((x.V = x_0) & (e.V = e_0)))) &
(z .> x)) | (((x.V <= y.V) & ((y.V = y_0) & ((x.V = x_0) & (e.V =
e_0)))) & (z .> y)))
```

which states that the value of variable x is greater than the value of variable y and the pointer z points to the variable x, or the value of variable y is greater or equal to the value of variable x and and the pointer z points to the variable y.

The final specification of the maxThresh program (lines 22-24) is as follows:

```
((((((((_cnst5 > _cnst4) & ((_cnst4 = y_0) & ((_cnst5 = x_0) & (e.V
= e_0)))) & (z .> x)) | (((_cnst5 <= _cnst4) & ((_cnst4 = y_0) &
((_cnst5 = x_0) & (e.V = e_0)))) & (z .> y))) & ((_cnst4 = CV3) |
(_cnst5 = CV3))) & (CV3 = _o6)) & (coset( z ).V = (_o6 + e.V)))
```

This specification states that the value of coset(z).V is equivalent to the expression (_o6 + e.V) where _o6 = CV3 and ((_cnst4 = CV3) | (_cnst5 = CV3)).

```
program m
inputs :
   int e
   int x
   int y
outputs :
   int *z
;
begin
   [
   if
      x
      z
      |
      x
      z
      |
   fi
   |
   *z :=
   [
      z
      e
   ]
end
```

Since the point

the statement a

the specificatio

AUTOSPEC sys

```
     program maxThresh (
 1   inputs :
 2      int e;
 3      int x;
 4      int y;
 5   outputs :
 6      int *z;
 7   )
 8   begin
 9      { ((y.V = y_0) & ((x.V = x_0) & (e.V = e_0))) }
10      if
11         (x > y) ->
12         z := @x;
13         { (((x.V > y.V) & ((y.V = y_0) & ((x.V = x_0) & (e.V = e_0)))) & (z .> x)) }
14      || (x <= y) ->
15         z := @y;
16         { (((x.V <= y.V) & ((y.V = y_0) & ((x.V = x_0) & (e.V = e_0)))) & (z .> y)) }
17      fi;
18      { ((((x.V > y.V) & ((y.V = y_0) & ((x.V = x_0) & (e.V = e_0)))) & (z .> x)) |
19         (((x.V <= y.V) & ((y.V = y_0) & ((x.V = x_0) & (e.V = e_0)))) & (z .> y))) }
20      *z := (*z + e);
21      { (((((((_cnst5 > _cnst4) & ((_cnst4 = y_0) & ((_cnst5 = x_0) & (e.V = e_0)))) &
22        (z .> x)) | (((_cnst5 <= _cnst4) & ((_cnst4 = y_0) & ((_cnst5 = x_0) &
23        (e.V = e_0)))) & (z .> y))) & ((_cnst4 = CV3) | (_cnst5 = CV3))) &
24        (CV3 = _o6)) & (coset( z ).V = (_o6 + e.V))) }
25   end
```

Figure 4.9: AUTOSPEC applied to the `maxThresh` program

Since the pointer `z` can point to either variable `x` or variable `y`, the value `(*z + e)` in the statement at line `21` is dependent on the result of the `if-fi` statement. As such, the specification of the conditional value for `CV3` is appended to the derivation by the AUTOSPEC system in order to preserve the one-point property of the specification.

Chapt

Applic
to C P

The C pro

Based on the

and semantic

including the

investigations

C programm

programs in C

C programmin

5.1 Assign

Let v be a varia

The C programm

=, ==, *=). Th

traditional assig

side-effect Boo

Chapter 5

Application of Strongest Postcondition to C Programs

The C programming language is one of the most popular programming languages [29]. Based on the imperative (procedural) programming style, C contains many syntactic and semantic elements that differentiate it from the Dijkstra guarded command language including the use of pointers and side-effect expressions. This chapter describes our investigations into the use of strongest postcondition to define the semantics of the C programming language [29] based on the semantics we developed for imperative programs in Chapter 3. The definition of the strongest postcondition semantics for the C programming language facilitates the reverse engineering of real industrial systems.

5.1 Assignment

Let v be a variable or an assignable expression and e be an expression. An assignment in the C programming language has the form $v \cong e$, where $\cong$ is an assignment operator (i.e., =, +=, *=). There are two roles that an assignment statement can have. The first is the traditional assignment of a variable with the value of an expression. The second role is as a side-effect Boolean expression.

In order to handle the dual role of an assignment statement, two functions are defined. First, in order to describe the semantics of the traditional use of assignment, an *evaluation* function $\mathcal{A}$: $S \rightarrow \mathcal{T}$ is defined, where S is the set of syntactically valid expressions, and $\mathcal{T}$ is the range of the result given by evaluating the expression e. If $s \in S$ is a non-assignment expression, then in general $\mathcal{A}(s) = s$. If, however, s is an assignment statement, such as "`x *= n`", the function $\mathcal{A}$ would be evaluated as $\mathcal{A}(\texttt{x *= n}) = \texttt{x} \times \texttt{n}$, where `n` is a variable of the same type as `x`. Table 5.1 defines the semantics of the function $\mathcal{A}$ on a few sample assignment operators. The left column of the table indicates which assignment operator is being performed and the right column indicates the value of the assignment as applied to v and e. For instance, for the operator $=$, the table states that $\mathcal{A}(v{=}e) = \mathcal{A}(e)$, and that $\mathcal{A}(v{+}{=}e) = v + \mathcal{A}(e)$.

A more general form of the function $\mathcal{A}$ can be defined as $\mathcal{A}(\texttt{b}) = \texttt{b}$, where `b` is a non-assignment expression. The interpretation is that the evaluation $\mathcal{A}$ on any expression has the value of the expression. For example, consider "$\mathcal{A}(\texttt{x + y + z})$". The expression "`x + y + z`" is a non-assignment expression, therefore $\mathcal{A}(\texttt{x + y + z}) = \texttt{x} + \texttt{y} + \texttt{z}$. For a discussion of the remaining expression constructs, see Appendix A. Using the definition of $\mathcal{A}$, we can define the strongest postcondition of an assignment in the following manner:

> **Definition 5.1 (Assignment Semantics)**
> *Let Q be the precondition, v be the quantified variable, and '::' indicate that the range of the quantified variable v is* true. *Then the strongest postcondition of an assignment is*
>
> $$sp(\texttt{x} \cong \texttt{e}, Q) \equiv (\exists v :: Q_v^x \wedge x = \mathcal{A}(\texttt{x} \cong \texttt{e}_v^x)).$$

Definition 5.1 states that after the execution of an assignment statement, there exists some value v such that the textual substitution of every free occurrence of x with v in Q keeps

Operation $v \cong e$	Evaluation $\mathcal{A}$
=	$\mathcal{A}(e)$
*=	$v \times \mathcal{A}(e)$
/=	$\frac{v}{\mathcal{A}(e)}$
+=	$v + \mathcal{A}(e)$
-=	$v - \mathcal{A}(e)$
%=	$v \bmod \mathcal{A}(e)$

Table 5.1: Evaluation of $\mathcal{A}$ on sample C assignment operators

Q true, and x takes the value of the evaluation $\mathcal{A}$ on $\mathtt{x} \cong \mathtt{e}_v^x$. This means that after the execution of an assignment statement, the precondition Q must still be true with respect to the value that the variable x had before the assignment, and the assignment must be valid.

The second function that is used to define the effects of an assignment statement is the *logical valuation* function $\mathcal{V} : S \rightarrow \mathcal{B}$, where S is the set of valid expressions, and $\mathcal{B}$ is the Boolean type. Note that S includes general expressions and assignment expressions. The purpose of $\mathcal{V}$ is best motivated by an example. Consider the sequence of code given in Figure 5.1. Informally, the semantics of this code sequence is that if the guard is true, execute `S1`, otherwise execute `S2`. However, the guard is worth noting since the expression is not a logical one, but rather an assignment expression. The semantics in this case are dependent on the side-effect of executing the statement `v = e`. Using the evaluation function $\mathcal{A}$, function $\mathcal{V}$ is defined as follows:

$$\mathcal{V}(v \cong e) = \begin{cases} T & \text{if } \mathcal{A}(v \cong e) \neq 0 \\ F & \text{if } \mathcal{A}(v \cong e) = 0 \end{cases},$$

where T and F are Boolean constants *true* and *false*, respectively. In general, for some arbitrary expression b, $\mathcal{V}$ is defined as:

$$\mathcal{V}(b) = \begin{cases} T & \text{if } \mathcal{A}(b) \neq 0 \\ F & \text{if } \mathcal{A}(b) = 0\,. \end{cases}$$

Although the side-effects of an assignment statement have no effect on the assignment itself, the side-effects do impact other operations as was shown in the example in Figure 5.1. The use of $\mathcal{V}$ will be important for defining the semantics of alternation statements with side-effects in Sections 5.2 and 5.3.

```
if (v = e) {
   S1
} else {
   S2
}
```

Figure 5.1: An Assignment statement as a guard

5.2 Alternation

The alternation statement for C programs can take two forms:

```
if B {
   S
}
```

and

```
if B {
   S1
} else
   S2
```

We refer to these statements as `C-IF1` and `C-IF2`, respectively. If the guard of an alternation statement has no side-effects, then the semantics of the alternation statement is as follows:

Definition 5.2 (Conditional Semantics without Side-effects)
Let Q be the precondition for the conditional statement, and B *be the guard. Then* C-IF1 *and* C-IF2*, have the following semantics, respectively.*

$$\begin{aligned} sp(\texttt{C-IF1}, Q) &\equiv sp(\texttt{S}, \texttt{B} \wedge Q) \vee sp(\texttt{skip}, \neg\texttt{B} \wedge Q) \\ &\equiv sp(\texttt{S}, \texttt{B} \wedge Q) \vee (\neg\texttt{B} \wedge Q) \end{aligned} \tag{5.1}$$

$$sp(\texttt{C-IF2}, Q) \equiv sp(\texttt{S}_1, \texttt{B} \wedge Q) \vee sp(\texttt{S}_2, \neg\texttt{B} \wedge Q). \tag{5.2}$$

The specification of $sp(\texttt{C-IF1}, Q)$ states that after execution of C-IF1 either $sp(\texttt{S}, \texttt{B} \wedge Q)$ is true (i.e., S was executed) or $(\neg\texttt{B} \wedge Q)$ is true (guard B was false). Similarly, the specification of $sp(\texttt{C-IF1}, Q)$ states that after execution of C-IF2 either $sp(\texttt{S}_1, \texttt{B} \wedge Q)$ is true (i.e., $\texttt{S}_1$ was executed) or $sp(\texttt{S}_2, \neg\texttt{B} \wedge Q)$ is true (guard B was false and statement $\texttt{S}_2$ was executed).

If the restriction of having alternation statements without side-effects in the guards is removed, then the semantics of the alternation statement has a different meaning. Informally, if there is a side-effect in the guard B, then the execution of an alternation is analogous to "executing" B, followed by the execution of the alternation using the evaluation of B. More formally, let B be a guard of an alternation statement (C-IF1 for instance) such that the evaluation of B causes a side-effect, and let $\mathcal{V}(B)$ represent the truth value of B. Execution of the alternation statement is equivalent to the execution of the following, respectively:

```
B;
if V(B) {
   S
}
```

```
B;
if V(B) {
   S1
} else
   S2
```

We refer to the alternation statements (the `if` statement with the replacement of `B` by $\mathcal{V}(B)$) as $\texttt{C-IF1}_s$ and $\texttt{C-IF2}_s$, respectively. The semantics of $\texttt{C-IF}_s$ are as follows:

Definition 5.3 (Conditional Semantics with Side-effects)
Let Q be the precondition for the conditional statement, and `B` *be a guard with side-effects. Then* $\texttt{C-IF1}_s$ *and* $\texttt{C-IF2}_s$ *have the following semantics, respectively.*

$$\begin{aligned} sp(\texttt{C-IF1}_s, Q) &\equiv sp(\texttt{C-IF1}, sp(\texttt{B}, Q)) \\ &\equiv sp(\texttt{S}, \mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)) \vee (\neg\mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)) \end{aligned} \tag{5.3}$$

$$\begin{aligned} sp(\texttt{C-IF2}_s, Q) &\equiv sp(\texttt{C-IF2}, sp(\texttt{B}, Q)) \\ &\equiv sp(\texttt{S}_1, \mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)) \vee sp(\texttt{S}_2, \neg\mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)). \end{aligned} \tag{5.4}$$

Expression (5.3) states that after execution of $\texttt{C-IF1}_s$ either $sp(\texttt{S}, \texttt{B} \wedge Q)$ is true (i.e., `S` was executed) or the valuation of $(\neg\texttt{B} \wedge Q)$ is true (the valuation of the guard $\mathcal{V}(\texttt{B})$ was false). Similarly, the Expression (5.4) states that after execution of `C-IF2` either $sp(\texttt{S}_1, \texttt{B} \wedge Q)$ is true (i.e., $\texttt{S}_1$ was executed) or $sp(\texttt{S}_2, \neg\texttt{B} \wedge Q)$ is true (the valuation of `B` was false and statement $\texttt{S}_2$ was executed).

5.3 Circuit Expressions

Expressions in the C programming language have a *circuit* property that cause a logical expression to be true or false before the entire expression has been completely evaluated. For instance, suppose the expression `(v == 5) && (n == 10)` is to be evaluated, and at the time of execution `v` has the value `3`. According to the definition of C, the logical value of the expression `(v == 5) && (n == 10)` is determined to be false immediately after the evaluation of the subexpression `(v == 5)`.

In the instances when the evaluation of an expression has no side-effects, the circuit property has no impact on the semantics of a program. That is, for instance, the semantics of the alternative structure `C-IF1` has the form given by Expression (5.1) (as opposed to the form given by Expression (5.3)). The existence of a side-effect requires that we define the semantics of expression evaluation in the presence of the circuit property. The following definitions define the syntax and semantics of logical expressions in the presence of side-effects

Definition 5.4 (Atomic Expression)
Any variable, constant, or function is an atomic expression. Let α and β be atomic expressions. Then the following are atomic expressions

Expression	*Meaning*
$(\alpha$ == $\beta)$	α *equals* β
$(\alpha$!= $\beta)$	α *does not equal* β
$(\alpha$ < $\beta)$	α *is less than* β
$(\alpha$ <= $\beta)$	α *is less than or equal to* β
$(\alpha$ > $\beta)$	α *is greater than* β
$(\alpha$ >= $\beta)$	α *is greater than or equal to* β

Definition 5.5 (Logical Expression)
An atomic expression is a logical expression. Let α and β be logical expressions. Then the following are logical expressions

Expression	*Meaning*
$(\alpha$ && $\beta)$	α *and* β
$(\alpha$ || $\beta)$	α *or* β
$(!\alpha)$	*not* α

Definition 5.6 (Circuit Expression Semantics)
Let α and β be logical expressions such that one or both of α and β have side-effects. The evaluation of $(!\alpha)$ have the usual semantics, respectively. The evaluation of $(\alpha$ && $\beta)$ and $(\alpha$ \| \| $\beta)$ has the following semantics.

$$sp(\alpha \text{ \&\& } \beta, Q) \equiv sp(\alpha; \beta, \mathcal{V}(\alpha) \wedge Q) \vee sp(\alpha, \neg\mathcal{V}(\alpha) \wedge Q) \quad (5.5)$$

$$sp(\alpha \text{ || } \beta, Q) \equiv sp(\alpha; \beta, \neg\mathcal{V}(\alpha) \wedge Q) \vee sp(\alpha, \mathcal{V}(\alpha) \wedge Q) \quad (5.6)$$

Definition 5.4 describes the syntax for atomic logical expressions in the C programming language and Definition 5.5 describes the general form for logical expressions. Definition 5.6 describes the semantics of two kinds of logical expressions: those formed by conjunction and those formed by disjunction. Expression (5.5) states that the semantics of evaluating an expression of the form $(\alpha \;\&\&\; \beta)$ is equivalent to either *executing* the sequence $\alpha;\beta$, given that $\mathcal{V}(\alpha) \wedge Q$ is true, or *executing* α given that $\neg\mathcal{V}(\alpha) \wedge Q$ holds. Informally, this means that either both subexpressions α and β are evaluated if $\mathcal{V}(\alpha)$ is true, or only α is evaluated since the falsity of $\mathcal{V}(\alpha)$ forces the entire expression $(\alpha \;\&\&\; \beta)$ to be false.

Expression (5.6) states that the semantics of evaluating an expression of the form $(\alpha \;||\; \beta)$ is equivalent to either *executing* the sequence $\alpha;\beta$, given that $\neg\mathcal{V}(\alpha) \wedge Q$ is true, or *executing* α given that $\mathcal{V}(\alpha) \wedge Q$ holds. Informally, this means that either both subexpressions α and β are evaluated if $\neg\mathcal{V}(\alpha)$ is true, or only α is evaluated since $\mathcal{V}(\alpha)$ true forces the entire expression $\alpha \;||\; \beta$ to be true.

5.4 Sequence

Sequences of statements in the C programming language have the form $\mathtt{S}_1\,; \;\ldots\; ; \mathtt{S}_n$. The appropriate semantics using *sp* is as follows:

$$sp(\mathtt{S}_1\,;\mathtt{S}_2, Q) \equiv sp(\mathtt{S}_2, sp(\mathtt{S}_1, Q)). \tag{5.7}$$

This formulation is identical to the semantics for sequences in the Dijkstra language [6]. Additionally, since the impact of side-effects are specified by the corresponding *sp*

formalisms for assignment, alternation, and iteration, this characterization of the semantics of sequence is sufficient.

5.5 Iteration

In the C programming language, the iteration construct can take one of the following forms:

```
while (B) {          do {              for (expr1;expr2;expr3) {
   S;                   S;                S;
}                    } while (B)       }
```

where B is the guard expression and $expr_i$ represent `for` iteration expressions. This section describes the strongest postcondition semantics for the `while`, `do-while`, and `for` iteration constructs of the C programming language. For the `do-while` and `for` constructs, transformations using the `while` semantics are provided.

5.5.1 `while`

When no side-effects are present, the `while` iteration construct has the following semantics:

Definition 5.7 (While Semantics without Side-effects)
Let Q be the precondition for the `while` *statement and* B *be the guard. Then the semantics of the* `while` *statement is as follows:*

$$sp(\texttt{while}, Q) = \neg \texttt{B} \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^{\texttt{i}}, Q)),$$

Definition (5.7) states that if the execution of the `while` statement terminates then the guard B is false and the result of applying the rule $sp(\texttt{C-IF1}, Q)$ i times is true. This construction is used given that an iteration statement can be considered a series of alternation statements, where the guard for the alternation is given by the guard of the iteration and the number of alternation statements that are included in the series is

determined by the guard. Clearly, it is not decidable to determine how many alternation statements to include in the series. Notationally, $sp(\texttt{C-IF1}^i, Q)$, where i is the number of iterations, means that *sp* is recursively applied to the result of $sp(\texttt{C-IF1}, Q)$. For instance, $sp(\texttt{C-IF1}^j, Q)$ has the following derivation:

$$\begin{aligned} sp(\texttt{C-IF1}^j, Q) &\equiv sp(\texttt{C-IF1}, sp(\texttt{C-IF1}^{j-1}, Q)) \\ &\equiv sp(\texttt{C-IF1}, sp(\texttt{C-IF1}, sp(\texttt{C-IF1}^{j-2}, Q))) \\ &\vdots \end{aligned}$$

In the case when the guard of the `while` statement has a side-effect, the semantics are similar to executing the following construct:

```
B;
while (V(B)) {
   S;
   B;
}
```

where $\mathcal{V}$ is the valuation function described previously. The corresponding *sp* semantics of the `while` statement with side-effects (denoted $\texttt{while}_s$) is

Definition 5.8 (While Semantics with Side-effects)
Let the body of the statement `C-IF1` *consist of "*`S; B;`*" as given by the transformation of the* `while` *statement to account for the side-effect, and let Q be the precondition. Then the semantics of the* `while` *statement with side-effects is*

$$sp(\textbf{while}_s, Q) = \neg\mathcal{V}(\texttt{B}) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(\texttt{B}, Q))).$$

Definition (5.8) states that if the execution of the `while` statement terminates then the valuation of the guard `B` is false and the result of applying the rule $sp(\texttt{C-IF1}, Q)$ i times is true.

5.5.2 do-while

The semantics of the do-while statement are similar to the while statement, where the guarding condition appears after the loop body. Using the while construct, do-while can be written as the following:

```
S;
B;
while (V(B)) {
   S;
   B;
}
```

The corresponding formal specification of the semantics of the do-while statement is given by Definition (5.9)

Definition 5.9 (Do-While Semantics with Side-effects)
Let the body of the statement C-IF1 *consist of "*S; B*", and the effects of executing "*S; B*" before entering the loop be given by the precondition argument of* $sp(\texttt{C-IF1}^i, sp(\texttt{B}, sp(S, Q)))$.

$$\begin{aligned} sp(\texttt{do-while}_s, Q) &\equiv sp(\texttt{while}_s, sp(S, Q)) \\ &\equiv \neg\mathcal{V}(\texttt{B}) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(\texttt{B}, sp(S, Q)))). \end{aligned}$$

This specification states that after the execution of a *do-while* statement, the valuation of B is false, and the body of the loop is executed i times.

5.5.3 for

Recall that the for construct in C has the form

```
for (expr1;expr2;expr3) {
   S;
}
```

The semantics of the `for` iteration statement is that the first expression (*expr1*) is executed (evaluated) once, the second expression (*expr2*) is evaluated before each iteration, and the third expression (*expr3*) is evaluated after each iteration. These semantics, defined in terms of the `while` construct, are represented by the following:

```
expr1;
expr2;
while (V(expr2)) {
    S;
    expr3;
    expr2;
}
```

The resulting formal specification of the semantics of the `for` command using the *sp* semantics for `while` is the following:

Definition 5.10 (For Semantics with Side-effects)
Let the body of the statement `C-IF1` *consist of "*`S; expr3; expr2;`*". Then the semantics of the* `for` *statement is as follows*

$$\begin{aligned} sp(\texttt{for}_s, Q) &\equiv sp(\texttt{while}_s, sp(expr1, Q)) \\ &\equiv \neg\mathcal{V}(expr2) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(expr2, Q))). \end{aligned}$$

This definition states that after the execution of the `for` loop, the logical valuation of *expr2* is false, and the loop body is executed i times where the initial precondition to the loop is given by $sp(expr2, Q)$.

5.6 Functions

Functions in the C programming language can serve two basic purposes. A function can be a *pure value function*, where the purpose is to compute and return a simple value based on the parameters. Alternatively, a function can be a *procedure*, where the purpose is to

perform a number of encapsulated tasks. Table 5.2 contains a taxonomy of functions based on the properties of *variables*, *side-effects*, *values returned*, and *parameters*.

	Function Class	
Property	Procedural	Pure valued
variables	global, local	local
side-effects	yes	no
parameters	value, value-result, result	value
values returned	multiple	single

Table 5.2: A Taxonomy of Programming Language Functions

The *variables* property describes the kinds of variables that are used by a function. The *side-effects* property indicates whether the class of functions produces side-effects. The types of parameters and the number of values that are returned by a function are described by the *parameter* and *values returned* properties, respectively. *Pure valued* functions are characterized by the use of local variables, in that the functions produce no side-effects, the parameters are value parameters, and the functions return a single value. Note that a procedural function can effectively serve the role of a pure valued function if it can be ensured that the functions produce no side-effects. This property implies that the number of values must be singular.

A function in the C programming language has a signature (or prototype) of the form $\mathcal{R}$ `f(`$\mathcal{D}$`)`, where $\mathcal{R}$ is the return type, and $\mathcal{D}$ is the input type of function `f`. For example, a function `max` could have a signature "`int max(int,int);`". Given a variable "`x`" of type $\mathcal{R}$, a parameter "`a`" of type $\mathcal{D}$, and an assignment operator $\cong$, a call to the function f has the form "`x` $\cong$ `f(a)`".

Let f be a pure valued function. The effect of calling the function is that a value is returned and assigned to the variable x. The corresponding *sp* semantics for the function call is given by the following definition

Definition 5.11 (Function Call Semantics)
Let Q be the precondition. The semantics of the function call is the following:

$$sp(\mathtt{x} \cong \mathtt{f(a)}, Q) = (\exists v :: Q_v^x \wedge x = \mathcal{A}(\mathtt{x} \cong \mathtt{f}(\mathtt{a}_v^x)).$$

This definition states that after the execution of an assignment statement using a function call, there exists some value v such that the textual substitution of x with v in Q is true, and x takes the value of the evaluation $\mathcal{A}$ on $\mathtt{x} \cong \mathtt{f}(\mathtt{a}_v^x)$. Note that in the case where a pure valued function is called but not assigned that $sp(\mathtt{f(a)}, Q) = Q$.

5.7 Procedural Abstractions

This section describes the construction of formal specifications from code containing the use of procedural abstractions. In C, a signature for a procedure has the form $\mathcal{R}$ $\mathtt{p}(\mathcal{D})$, where $\mathcal{R}$ is the type returned by the `return` statement in the procedure p, and $\mathcal{D}$ is the list of input types to p. The code for a procedure has the following form:

$\mathcal{R}$ $\mathtt{p}(\mathcal{D}_p)$ {
  DL
  S_p
}

where $\mathcal{D}_p$ is a list of comma delimited type-parameter pairs, *DL* is a list of declarations, and S_p is a sequence of programming statements.

5.7.1 Parameters

Parameters for a procedure call, as stated in Table 5.2, can be **value** or **value-result** parameters, respectively. A parameter of type **value** means that the parameter is used only for input to the procedure. Parameters that are known as **value-result** indicate that the parameters can be used for both input and output to the procedure. A **value** parameter declaration has the general form "`datatype q`", where `datatype` is the type of the parameter, and `q` is the name of the parameter. A parameter declared in this manner is visible in the scope local to the procedure being called. A **value-result** parameter declaration has the general form "`datatype *q`", where `*q` is a pointer variable. Any operation performed in a parameter declared in this manner has scope that is beyond the local procedure.

5.7.2 Procedure Call Semantics

In Section 3.2.2 we described the *sp* semantics for procedure call. In C, a procedure call `p`($\bar{a}$, $\bar{b}$) can be represented by the sequence of statements found in the righthand column of Figure 5.2, where S_p is the body of the procedure `p`, Q is the precondition, and R is the postcondition to the procedure call to `p`. In addition, annotations Q_p, R_p, and R_pR represent the precondition to the procedure call after the binding of actual parameters to formal parameters, the postcondition to the procedure, and the postcondition after the procedure "returns", respectively. By representing a procedure call in this manner, parameter binding can be achieved through assignment statements and a postcondition R can be established by using the *sp* for assignment.

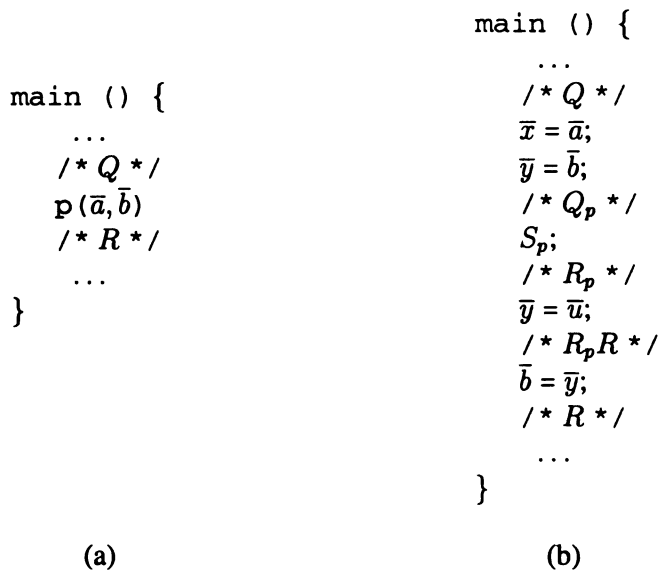


Figure 5.2: Removal of procedure call abstraction: (a) before (b) after

Figure 5.3 depicts the code annotations for a procedure call given the use of inlining to achieve parameter binding. Since $\overline{x}$ and $\overline{y}$ are formal parameters there are no occurrences of $\overline{x}$ and $\overline{y}$ in Q, and as such we can apply Lemma 3.1.2. The final annotation is summarized by the following definition.

Definition 5.12 (Procedure call Semantics)
Let Q be the precondition, $\overline{x}$ and $\overline{y}$ be formal parameters, $\overline{a}$ and $\overline{b}$ be actual parameters, $\overline{u}$ be the variables local to p *used to compute the results of the* **value-result** *parameters, and S_p be the body of the procedure. Then the semantics of the procedure call are:*

$$sp(\mathrm{p}(\overline{a}, \overline{b}), Q) \equiv sp(S_p, Q \wedge \overline{x} = \overline{a} \wedge \overline{y} = \overline{b}) \wedge \overline{b} = \overline{u}$$

This definition states that the semantics of a procedure call is a conjunction of the application of sp on the precondition $Q \wedge \overline{x} = \overline{a} \wedge \overline{y} = \overline{b}$ and the result of the binding of $\overline{b}$ to a value, $\overline{u}$, computed within the procedure.

```
main () {
    ...
    /* Q */
    x̄ = ā;
    ȳ = b̄;
    /* Q ∧ x̄ = ā ∧ ȳ = b̄ */
    S_p;
    /* sp(S_p, Q ∧ x̄ = ā ∧ ȳ = b̄) */
    ȳ = ū;
    /* sp(S_p, Q ∧ x̄ = ā ∧ ȳ = b̄) ∧ ȳ = ū */
    b̄ = ȳ;
    /* sp(S_p, Q ∧ x̄ = ā ∧ ȳ = b̄) ∧ b̄ = ū */
    ...
}
```

Figure 5.3: Code annotation for procedure calls

Chapter 6

Design Abstractions

The derivation of abstract specifications from as-built specifications facilitates the construction of a description of a system at a level of abstraction that is higher than both source code and as-built specifications. As such, high-level reasoning and understanding of specifications can be enabled by the existence of abstract specifications. In this chapter, we describe a technique for identifying abstract behavior in specifications. Specifically, we define an approach for deriving abstract specifications from as-built specifications by requiring that the derived abstraction and the as-built specification satisfy a matching relation.

6.1 Specification Matching and Software Reuse

Many approaches have been suggested for the retrieval of components from reusable component libraries, ranging from classification of search criteria [19, 20] to retrieval [31, 32, 33] and library structuring [34]. Jeng and Cheng describe the use of *analogy* and *generality* [18, 35] as the basis for matching functions. Zaremski and Wing have proposed a technique for signature [21] and specification matching [19]. Fischer *et al.* have described

an approach for retrieval of reusable components using filters to narrow the search spaces [32, 33]. The approach described in this chapter differs from these approaches in that we are using specification matching for reverse engineering as opposed to retrieval of reusable components.

Mili *et al.* describe an approach for structuring component libraries using refinement orderings [34]. Their approach uses relational specifications as the formalism for describing software components, and structures libraries using relational definitions of refinement. Our approach incorporates their ideas on the structure of libraries using partial order relations although our focus is on axiomatic specifications, rather than relational specifications. In addition, our primary goal is to use partial order matching operators to generalize specifications for purposes of reverse engineering.

Other approaches to reverse engineering focus on the construction of specifications, both informal and formal, and are based on the identification of *plans* [36], the construction of high-level *structural* specifications such as data flow and call diagrams [37], or transformation of programs into specifications [38, 39]. Of these techniques, the approach proposed by Baxter and Mechlich [39] is the most closely related. They suggest an approach to reverse engineering using "backward transformation" where a series of transformations (semantic preserving rewrite rules), similar to those used in forward transformation, are used in an inverse manner. The use of a library is extensive in this approach where the contents of the library are semantic preserving transformations. In the remainder of this chapter we describe an approach that derives abstract behavior by preserving a match relation between generalized and as-built specifications. As such, we do not rely on the existence of a domain library to provide specification matches.

6.2 Abstraction Matching

In this section we describe an approach for software reverse engineering that is based on the use of specification matching.

6.2.1 Approach

Given a library of axiomatic (pre- and postcondition) specifications describing software components, these approaches use a *plug-in* or *generality* criteria [18, 19] to identify components in the library that match a *query* specification. The plug-in match is defined as follows:

> **Definition 6.1 (Generality (Plug-in) Match [31])** *Let q be a query specification with precondition q_{pre} and postcondition q_{post} and l be a library specification with precondition l_{pre} and postcondition l_{post}. Specifications q and l* **match** *(denoted by $l \preceq q$) if*
>
> $$(q_{pre} \rightarrow l_{pre}) \wedge (l_{post} \rightarrow q_{post}).$$

Informally, this definition means that the library component l is a refinement (i.e., more specific) than q, or conversely, that q is an abstraction of l. In both interpretations, any program whose behavior is described by q will be satisfied by l and as such, l can be used as an implementation for the query given by q. Many different criteria for matching query specifications with software components have been identified [20, 19] and all vary in the degree of component modification required to use a library component as an implementation for a given query.

In Chapters 3, 4, and 5 we described the use of strongest postcondition to construct formal as-built specifications from program code. Although as-built specifications facilitate traceability between code and specifications, they may be difficult to use for high-level

reasoning since they contain an implementation bias. Therefore, a rigorous technique for deriving a more abstract functional specification is desired.

Let $\mathcal{I}$ be a program with specification i such that the precondition is i_{pre}. The corresponding postcondition (denoted i_{post}) can be derived using the strongest postcondition (e.g., by using $sp(\mathcal{I}, i_{\text{pre}})$) [6]. Let l be a specification in a *specification library* with precondition l_{pre} and postcondition l_{post}. Suppose $i \preceq l$, then l is a generalization or abstraction of i. Conversely, i is a refinement of l. This means that any behavior described by l is satisfied by i and as such, program $\mathcal{I}$ can be used as an implementation for the specification given by l. The following definition summarizes this idea.

> **Definition 6.2 (Abstraction Match)** *Let $\mathcal{I}$ be a program with specification i such that the corresponding precondition and postcondition are i_{pre} and i_{post}, respectively, and let l be an axiomatic specification with precondition l_{pre} and postcondition l_{post}. A match is an* **abstraction match** *if $i \preceq l$, so that*
>
> $$(l_{pre} \rightarrow i_{pre}) \wedge (i_{post} \rightarrow l_{post}).$$

The importance of the abstraction match is that if a specification l exists that is well-understood in terms of its abstract high-level behavior, then any specification i that can be shown to satisfy an abstraction match relationship, where $i \preceq l$, has the same abstract behavior as the specification l. In terms of reverse engineering, this fact provides a means for introducing abstraction into a specification i via the identification of specification l.

6.2.2 Specification Libraries

Specification libraries have been used in the area of automated program construction to describe theories about specific problem domains [40]. In addition, specification libraries have been used as a means for collecting components into reuse libraries [20, 19]. This

section describes the use of partial order relations to organize specification libraries and describes several properties that facilitate analysis of specifications based on semantic commonality and difference.

Partial Order Relations

In some cases, the matching criteria given in Table 2.3 define a partial order relationship between specifications A and R. The following lemmas and definitions reinforce this idea. For the proofs of these lemmas, please refer to Section B.

> **Lemma 6.2.1 (Equivalence)** *The exact pre/post match is reflexive, symmetric, and transitive (i.e., the exact pre/post match is an equivalence relation).*

Using exact pre/post match as the equivalence relation for anti-symmetry, the following lemma holds.

> **Lemma 6.2.2 (Plug-In)** *The plug-in match is reflexive, anti-symmetric, and transitive (i.e., the plug-in match is a partial order relation).*

> **Definition 6.3 (Weak Equivalence)** *Let A and R be axiomatic specifications. Then define the relation:*
>
> $$A_{post} \Leftrightarrow R_{post}$$
>
> *to be* **weak equivalence**, *where* $\Leftrightarrow$ *is logical equivalence.*

The intuition behind weak equivalence is that specifications A and R are equivalent if their postconditions are logically equivalent. As such, their output behaviors are the same while the relationship of their input behaviors is unknown. Using weak equivalence as the equivalence relation for anti-symmetry, the following lemma holds.

> **Lemma 6.2.3 (Plug-In Post)** *The plug-in post match is a weak partial order relation.*

Library Structure

Mili *et al.* [34] have suggested that libraries be structured based on refinement orderings. Furthermore, they describe a number of properties and measures for managing and retrieving components from libraries that are structured using refinement orderings [34, 41]. Since the plug-in and the plug-in post matches are (weak) partial order relations, specification libraries can be structured as partially ordered sets with the matching operators serving as the partial order (refinement) relation.

The convention used in this chapter for library specifications, given in Figure 6.1, is based on the Larch interface language [42] syntax. In this convention, *domainsort* and *rangesort* are the input and output types of a given function, respectively. The **locals** keyword lists the variables defined within the scope of the specification, if applicable. The **requires** keyword is used to indicate the precondition of the given function. The **ensures** keyword describes the postcondition of a given function. Finally, the **modifies** keyword lists the variables that are modified by the function.

---

spec *name* ((*var*: *domainsort*)*) ⟶ *var*: *rangesort*
locals (var: domainsort)*
requires *precondition*
modifies *variables*
ensures *postcondition*

Figure 6.1: Syntax of Library Specifications

Figure 6.2 shows the set of "Sqr" specifications that describe the square root function. The specification *Sqr0* allows negative roots as output whereas *Sqr1* ensures that the positive roots are returned. The specifications *Sqr2* and *Sqr3* return undefined values when

the input value is less than zero. These two specifications differ in that they allow (*Sqr2*) or disallow (*Sqr3*) negative roots. The specification *Sqr4* returns $root = 0$ when the input is a negative number, and a positive root for positive inputs.

---

spec *Sqr0* $(x : real) \longrightarrow r : real$
requires $x \geq 0$
ensures $r^2 = x$

spec *Sqr1* $(x : real) \longrightarrow r : real$
requires $x \geq 0$
ensures $r \geq 0 \wedge r^2 = x$

spec *Sqr2* $(x : real) \longrightarrow r : real$
requires *true*
ensures $(x \geq 0 \wedge r^2 = x) \vee$
$(x < 0 \wedge r = undefined)$

spec *Sqr3* $(x : real) \longrightarrow r : real$
requires *true*
ensures $(x \geq 0 \wedge (r \geq 0 \wedge r^2 = x)) \vee$
$(x < 0 \wedge r = undefined)$

spec *Sqr4* $(x : real) \longrightarrow r : real$
requires *true*
ensures $(x \geq 0 \wedge r^2 = x) \vee$
$(x < 0 \wedge r = 0)$

spec *SqrPos* $(x : real) \longrightarrow r : real$
requires $x \geq 0$
ensures $r \geq 0$

Figure 6.2: Square Root Specification Library "Sqr"

As a partially ordered set on the plug-in relation, the library in Figure 6.2 has the structure given by the Hasse diagram of Figure 6.3, where the specification at the head of the arc is more general than the specification at the tail of the arc. As such, *Sqr0* is more general than *Sqr1*, and *Sqr2* is more general than *Sqr3*. The structure of this library suggests that there are three different ways to construct a square root function. The first way requires that the inputs to the function be a positive real number. The second way to construct a square root function is to produce an undefined value when the input is a negative real number. The final way to construct a square root function is to return the value zero when a negative real is used as input.

Structuring a library as a partially ordered set has many applications including the fact that it provides a means for partitioning libraries based on behavioral differences as

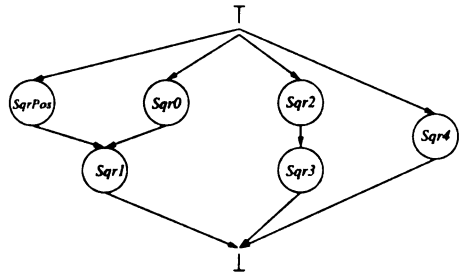


Figure 6.3: Square Root Library as a partial order

in the example above. In addition, the partial order structure facilitates inserting new specifications into a library and helps increase the efficiency of the retrieval process [31]. An interesting activity for analyzing libraries that are structured using partial order relations is to determine if the library has certain lattice-like properties [34, 41]. In particular, it is of interest to determine the least upper bound (lub) and greatest lower bound (glb) for given specifications, if they exist, since the lub can be used to identify common behavior and the glb can be used to identify compositional behavior. Given a partially ordered set, where an ordered pair $(L, \preceq)$ indicates the set and the ordering operator, respectively, the following interpretations for the lub and glb of two specifications $S_1 \in L$ and $S_2 \in L$ can be made, where we denote the lub for S_1 and S_2 as $S_1 \sqcap S_2$, and the glb as $S_1 \sqcup S_2$:

> **Definition 6.4 (Behavioral Commonality** [41]) *Let $T = S_1 \sqcap S_2$, if it exists. Then T captures the behavior common to S_1 and S_2.*

> **Definition 6.5 (Behavioral Composition** [41]) *Let $B = S_1 \sqcup S_2$, if it exists. Then B captures the composition of behavior for S_1 and S_2.*

If specifications S_1 and S_2 are related such that $S_1 \preceq S_2$ or $S_2 \preceq S_1$, then the following definition is of particular interest:

Definition 6.6 (Semantic Difference [41]) *Let S_1 and S_2 be specifications such that $S_1 \preceq S_2$. Then the semantic difference between S_1 and S_2 (denoted $S_1 \ominus S_2$) is the most general specification E such that*

$$S_2 \sqcup E \preceq S_1.$$

Definition 6.6 states that some specification E, is the semantic difference between S_1 and S_2 in the case that the meet of S_2 and E is more specific than S_1. As an example, consider the specifications for *Sqr0* and *Sqr1*, where *Sqr0* $\preceq$ *Sqr1* when using the plug-in post relation. Figure 6.3 shows the Hasse diagram for $(Sqr, \preceq_{pip})$, where $\preceq_{pip}$ is the plug-in post relation. The semantic difference is the specification E such that

$$(Sqr0 \sqcup E) \preceq Sqr1$$

When we substitute the specification names with the corresponding postconditions, we get the following expression:

$$(root^2 = r) \sqcup E \preceq (root \geq 0 \wedge root^2 = r).$$

Using the Hasse diagram in Figure 6.3 we find that *SqrPos* satisfies the conditions for E such that

$$(root^2 = r) \sqcup (root \geq 0) \preceq (root \geq 0 \wedge root^2 = r),$$

In fact,

$$(root^2 = r) \sqcup (root \geq 0) \equiv (root \geq 0 \wedge root^2 = r).$$

That is, the meet of $root^2 = r$ and $root \geq 0$ is equivalent to $(root \geq 0 \wedge root^2 = r)$. Therefore, *SqrPos* is the semantic difference between *Sqr0* and *Sqr1*.

6.3 Specification Generalization

Many of the techniques that utilize formal specifications to specify and retrieve reusable components from component libraries attempt to identify candidate components by searching the library for those components that satisfy specific match criterion. Similarly, as stated by Definition 6.2, if we have a library specification that is an abstraction match for an as-built specification, then the library specification is a generalization of the as-built specification. However, it is possible that an abstraction match does not exist for a given as-built specification. In this case, some other technique must be used to derive abstractions of the as-built specification. In this section we describe an approach to reverse engineering based on preserving the partial order relationship between an as-built specification and a derived abstraction of that as-built specification.

6.3.1 Basic Approach

Consider an axiomatic specification I that consists of precondition I_{pre} and postcondition I_{post}. Assuming that the relation $\preceq$ is a partially ordered matching operator, we would like to identify an axiomatic specification A such that $I \preceq A$. That is, we would like to identify a specification A that is an abstraction of I in a manner that does not involve matching specifications in a library. In fact, we can identify such a specification by modifying I so that we have a specification I' that satisfies the relationship that $I \preceq I'$. If, for instance, $\preceq$ is a plug-in match operator, then by either strengthening the precondition I_{pre}, weakening the postcondition I_{post}, or both, we produce a specification I' that satisfies the property that $I \preceq I'$. A modification of I' to produce a specification I'' that satisfies the property $I' \preceq I''$ provides another level of abstraction such that $I \preceq I' \preceq I''$.

A likely situation is shown in Figure 6.4, where a specification I has been decomposed into several different specifications, each describing a different behavior such that the composition of their behaviors is the original specification I. In addition, several of these specifications can be decomposed into other specifications, each at a different level of abstraction.

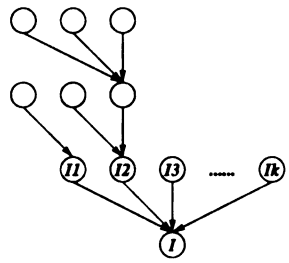


Figure 6.4: Specification Generalization

Using a brute force approach for specification generalization can result in the construction of an exponential number of specifications, all of which satisfy the partial order constraints of the original specification. For instance, the program in Figure 6.5 shows a typical bubble sort program with logical annotations contained within the curly braces '{' and '}'. The annotations, constructed using the strongest postcondition semantics by a prototype system called AUTOSPEC [43], use the notation '&' to indicate a logical and ('∧'), '`exists`' to indicate an existential quantification, and '`forall`' to indicate a general quantification. In addition, the notation $v.V$ represents the value of a variable v. The axiomatic specification for the program is as follows:

$$
\begin{array}{l}
\textbf{spec}\ \textit{BubbleSort}\ (a[] : \textit{int}, n : \textit{int}) \longrightarrow \textit{root} : \textit{real} \\
\quad \textbf{locals}\ \ i, j, t : \textit{int} \\
\quad \textbf{requires}\ n = |a| \\
\quad \textbf{modifies}\ a \\
\quad \textbf{ensures}\ \ (i \geq n)\ \wedge\ (j \leq n)\ \wedge\ \textit{perm}(a\_1, a)\ \wedge \\
\qquad\qquad (\exists u : 1 \leq u \leq n : (t = a\_1[u]))\ \wedge \\
\qquad\qquad (\forall k : 1 \leq k < n : \\
\qquad\qquad (\forall r : k + 1 < r \leq n : a\_1[r] \geq a\_1[r - 1])),
\end{array}
\tag{6.1}
$$

where the **ensures** clause (postcondition) states that after the execution of the program, the variable `i` is greater than or equal to the size of the array `a`, the variable `j` is less than or equal to the size of the array `a`, the variable `t` has some value equivalent to some element of the array, all the elements of the array are ordered in ascending fashion, and the final array is a permutation of the original array. Given the five conjuncts in Specification (6.1), it is possible to construct at least thirty-one different specifications that satisfy the partial order property of the abstraction match operator.

In order to handle the complexity of this situation we make the assumption that the reverse engineering programmer is guiding the abstraction process. In order to support this process we are developing a support tool called SPECGEN that visually displays the partially ordered sets of specifications that are constructed using the specification generalization technique. In the following sections, we describe several guidelines that can be used to construct abstractions from a specification. The remainder of this section discusses the guidelines from the point of view of weakening the postcondition and strengthening a precondition.

```
program BubbleSort (inputs :  int a[]; int n;
                    outputs :  int a[]; )
decl
   int i; int j; int y; int x; int t;
lced
begin
   {  (((t.V = t_0) ∧ ((x.V = x_0) ∧ ((y.V = y_0) ∧
      ((j.V = j_0) ∧ (i.V = i_0))))) ∧ ((n.V = n_0) ∧ (a.V = a_0))) }
   i := 1;
   {  ((((t.V = t_0) ∧ ((x.V = x_0) ∧ ((y.V = y_0) ∧ ((j.V = j_0) ∧
      (_cnst1 = i_0))))) ∧ ((n.V = n_0) ∧ (a.V = a_0))) ∧ (i.V = 1)) }
   do
      (i < n) ->
      j := n;
      {  (((i.V < n.V) ∧ ((i.V = k) ∧ (n.V = n_0))) ∧ (j.V = n.V)) }
      y := (i + 1);
      {  ((((i.V < n.V) ∧ ((i.V = k) ∧ (n.V = n_0))) ∧ (j.V = n.V)) ∧ (y.V = (i.V + 1))) }
      do
         (j > y) ->
         x := (j - 1);
            {  ((((j.V > y.V) ∧ (((i.V = k) ∧ (n.V = n_0)) ∧ (j.V = M))) ∧ (x.V = (j.V - 1)) ∧
               (a[j.V] = a_j0)) ∧ (a[x.V] = a_x0)) }
         if
            (a[j.V] < a[x.V]) ->
            t := a[j];
            {  (((((a[j.V] < a[x.V]) ∧ (((j.V > y.V) ∧ (((i.V = k) ∧
               (n.V = n_0)) ∧ (j.V = M))) ∧ (x.V = (j.V - 1)))) ∧
               (t.V = a[j.V])) ∧ (a[j.V] = a_j0)) ∧ (a[x.V] = a_x0)) }
            a[j] := a[x];
            {  (((((_cnst2 < a[x.V]) ∧ (((j.V > y.V) ∧ (((i.V = k) ∧
               (n.V = n_0)) ∧ (j.V = M))) ∧ (x.V = (j.V - 1)))) ∧
               (t.V = _cnst2)) ∧ (a[j.V] = a[x.V])) ∧ (_cnst2 = a_j0)) ∧
               (a[x.V] = a_x0)) }
            a[x] := t;
            {  (((((((_cnst2 < _cnst3) ∧ (((j.V > y.V) ∧ (((i.V = k) ∧
               (n.V = n_0)) ∧ (j.V = M))) ∧ (x.V = (j.V - 1)))) ∧
               (t.V = _cnst2)) ∧ (a[j.V] = _cnst3)) ∧ (a[x.V] = _cnst2)) ∧
               (_cnst2 = a_j0)) ∧ (_cnst3 = a_x0))}
         fi;
         {  (((((((_cnst2 < _cnst3) ∧ (((j.V > y.V) ∧ (((i.V = k) ∧
            (n.V = n_0)) ∧ (j.V = M))) ∧ (x.V = (j.V - 1)))) ∧
            (t.V = _cnst2)) ∧ (a[j.V] = _cnst3)) ∧ (a[x.V] = _cnst2)) ∧
            (_cnst2 = a_j0)) ∧ (_cnst3 = a_x0)) }
         j := (j - 1);
         {  ((((((((_cnst2 < _cnst3) ∧ (((j.V > y.V) ∧ (((i.V = k) ∧
            (n.V = n_0)) ∧ (j.V = M))) ∧ (x.V = (j.V - 1)))) ∧ (t.V = _cnst2))
            ∧ (a[j.V] = _cnst3)) ∧ (a[x.V] = _cnst2)) ∧
            (_cnst2 = a_j0)) ∧ (_cnst3 = a_x0)) ∧ (j.V = (M - 1))) }
      od;
      {  ((((j.V = (k + 1)) ∧ (x.V = (j.V - 1))) ∧
         (∀ r : (((k + 1) < r) ∧ (r ≤ n_0)) : (a_1[r] ≥ a_1[r-1]))) ∧
         (∃ u : (((k + 1) < u) ∧ (u ≤ n_0)) : (t = a_1[u]))) }
      i := (i + 1);
      {  ((((j = (k + 1)) ∧ (x.V = (j.V - 1))) ∧
         (∀ r : (((k + 1) < r) ∧ (r ≤ n_0)) : (a_1[r] ≥ a_1[r-1]))) ∧
         (∃ u : (((k + 1) < u) ∧ (u ≤ n_0)) : (t = a_1[u]))) ∧
         (i.V = (_cnst7 + 1))) }
   od;
   {  ((¬(i.V < n.V)) ∧ ((((¬(j.V > n_0)) ∧ (∀ v : ((1 ≤ v) ∧ (v < n_0)) :
      (∀ r : (((v + 1) < r) ∧ (r ≤ n_0)) : (a_1[r] ≥ a_1[r-1])))) ∧
      (∃ u : (((k + 1) < u) ∧ (u ≤ n_0)) : (t = a_1[u]))) ∧ perm( a_1, a ))) }
end
```

Figure 6.5: Bubble Sort Program Annotated by AUTOSPEC

Weakening the postcondition

Let I be a specification with precondition I_{pre} and postcondition I_{post} and let I' be a specification such that $I'_{pre} \leftrightarrow I_{pre}$ and $I_{post} \rightarrow I'_{post}$. As such, $I \preceq I'$, since

$$\begin{array}{l} ((I'_{pre} \leftrightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})) \\ \Rightarrow \\ ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})). \end{array} \tag{6.2}$$

Expression (6.2) provides a basis for deriving abstractions from a specification by weakening a postcondition I_{post} to produce a postcondition I'_{post}. Several options are available for weakening the postcondition including those listed in Table 6.1, which includes *delete a conjunct*, *add a disjunct*, $\wedge$ *to* $\vee$ *transformation*, and $\wedge$ *to* $\rightarrow$ *transformation*.

Operation	I_{post}	I'_{post}
Delete a conjunct	$A \wedge B \wedge C$	$A \wedge C$
Add a disjunct	$A \wedge B$	$(A \wedge B) \vee C$
$\wedge$ to $\rightarrow$	$A \wedge B$	$A \rightarrow B$
$\wedge$ to $\vee$	$A \wedge B$	$A \vee B$

Table 6.1: Weakening the postcondition

Delete a conjunct. Given a specification in conjunctive form (not necessarily a normal form), deletion of a conjunct weakens a specification by removing additional or constraining conditions. For example, consider Figure 6.6, where the specification *abcde* represents the **ensures** clause of the specification in Expression (6.1). In the Hasse diagram, the vertex label '*xy*' represents the logical conjunction $x \wedge y$. Each successive level of abstraction is derived by deleting a conjunct from the lower levels of abstraction. Below are guidelines that can be used to identify the appropriate conjunct for deletion.

Local Scope: If a conjunct specifies behavior that is local to a procedure and has no impact on the output variables of the system, then that conjunct is a candidate for deletion. Examples include specifications of the value of a loop index or temporary variables.

Independence: If a conjunct specifies some behavior that is logically independent of the remaining conjuncts, then that conjunct is a candidate for deletion. As an example, consider the expression $(x = c) \wedge (c = y) \wedge (z = n)$. The conjunct $(z = n)$ is independent of the conjuncts $(x = c)$ and $(c = y)$.

Preservation: If a conjunct captures some behavior that must be expressed in the higher level specification, the remaining conjuncts are candidates for deletion. Refer again to Figure 6.7 where the conditions a and c have been selected as behaviors to be preserved. The remaining specifications in the partial order indicate refinements between the as-built specification and the specification *ce*.

These guidelines are by no means comprehensive. Ultimately, a maintenance engineer using this approach must decide whether to delete a specific conjunct in a specification

Add a disjunct. Given a specification in any form, adding a disjunct weakens a specification by generalizing or increasing the scope of the specification. The addition of a disjunct should be used in very few instances since the new disjunct potentially introduces superfluous behavior that may not be reflected in the original system.

Conjunction to implication or disjunction transformations. Given a specification in conjunctive form (not necessarily a normal form), transformation of the conjunction to an implication or disjunction provides a logical weakening of the specification and facilitates manipulation of the specification using several standard equivalence transformations. Our ongoing investigations include determining the usefulness of these transformation techniques to derive specification abstractions.

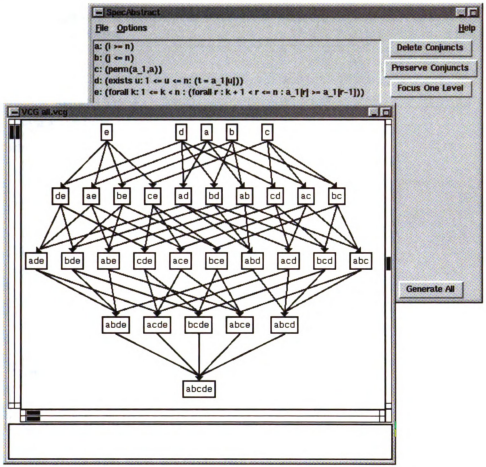


Figure 6.6: Bubble Sort Specification Brute Force Abstraction

Strengthening the precondition

Let I be a specification with precondition I_{pre} and postcondition I_{post} and let I' be a specification such that $I'_{pre} \rightarrow I_{pre}$ and $I_{post} \leftrightarrow I'_{post}$. As such, $I \preceq I'$, since

$$\begin{array}{l} ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \leftrightarrow I'_{post})) \\ \Rightarrow \\ ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})). \end{array} \tag{6.3}$$

Expression (6.3) provides a basis for deriving abstractions from a specification by strengthening a precondition I_{pre} to produce a precondition I'_{pre}. Weakening a postcondition has many advantages over strengthening a precondition in the context of

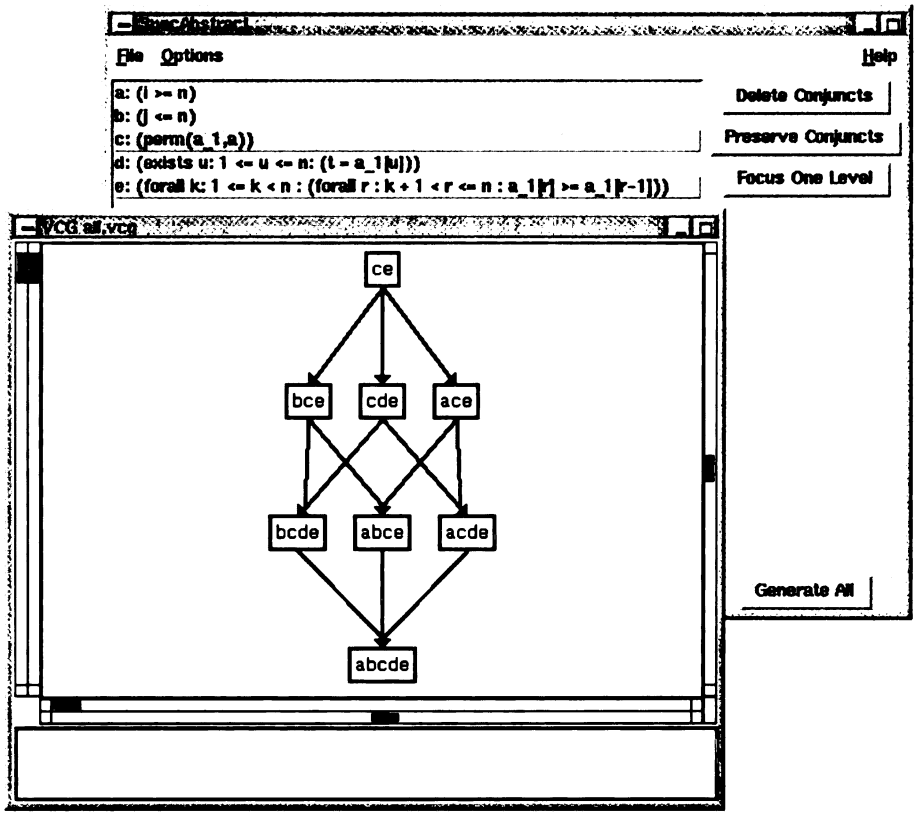


Figure 6.7: Bubble Sort Specification Abstraction (postcondition)

deriving abstractions from specifications. The primary advantage is a consequence of the reverse engineering activity in that we are interested in deriving a specification of the behavior of a program. This behavior is captured in the specification of the postcondition rather than the precondition. The utility of strengthening the precondition is that it provides a mechanism for identifying a narrower set of conditions that can be used to constrain the domain of input. The available techniques for strengthening the precondition include *adding a conjunct* and *deleting a disjunct*.

Add a conjunct. Given a specification in a conjunctive (not necessarily normal) form, adding a conjunct to the precondition provides further conditions that are required in order for the specification to achieve the desired behavior.

Delete a disjunct. Given a specification in any form, deleting a disjunct will make the precondition more specialized (e.g., less general) in the initial conditions required to satisfy a behavior.

6.3.2 Example

Consider once again the example in Figure 6.5 and the corresponding postcondition specification in Expression (6.1). In this section we focus on constructing an abstraction of the specification by weakening the postcondition. Since the specification is in a conjunctive normal form, it is appropriate to use the *delete a conjunct* strategy to construct an abstraction. In a completely brute force approach we would derive four abstractions for each of the five produced in the first step. However, we advocate a user-driven process that relies on a user to decide the direction of the abstraction steps. Figure 6.6 depicts the brute force application of the delete a conjunct strategy, where the expression "*abcde*" at the bottom of the graph represents the specification of Expression (6.1), where "*a*" represents the conjunct $(i \geq n)$, "*b*" represents the conjunct $(j \leq n)$, "*c*" represents the conjunct $perm(a\_1, a)$, "*d*" represents the conjunct $(\exists u : 1 \leq u \leq n : (t = a\_1[u]))$, and "*e*" represents the conjunct $(\forall k : 1 \leq k < n : (\forall r : k+1 < r \leq n : a\_1[r] \geq a\_1[r-1]))$.

In the first step, the "*a*" conjunct can been deleted since the "*a*" conjunct involves a specification of the value of an iteration variable. Deleting one conjunct from the specification "*bcde*" results in four different specifications. Using the same reasoning as

in the previous step, we consider only the specification that excludes the conjunct "b". Figure 6.8 shows the partially ordered set of specifications that result from deleting "*a*" and "*b*" where the resulting specification is "*cde*" which states that the output array is a permutation of the input array, that the variable *t* takes the value of some element of the array, and the array is ordered in increasing value. At this point, three abstractions are possible. However, the conjunct $(\exists u : 1 \leq u \leq n : (t = a\_1[u]))$ specifies information about the temporary variable t, and as such we consider only the specification *ce*, which is equivalent to:

$$perm(a\_1, a) \wedge$$
$$(\forall k : 1 \leq k < n : (\forall r : k + 1 < r \leq n : a\_1[r] \geq a\_1[r - 1])).$$

This specification states that after execution of the program, the output array is a permutation of the input array, and that the array is ordered in increasing value.

By focusing attention on a few conjuncts, the complexity of the task of constructing specification abstractions can be reduced since many of the other possible abstractions for the original as-built specification can be removed from consideration. In addition, by using a few simple support tools, the difficulty of deriving the abstractions can be greatly reduced.

6.4 Application to a JPL Ground-based Flight System

In our previous investigations we described a technique for analyzing C programs using the strongest postcondition predicate transformer [7]. In addition, we have defined the semantics of pointers and pointer operations in terms of *sp* [43]. In this section we present a case study that applies the *sp* technique for C programs to a module from a ground-

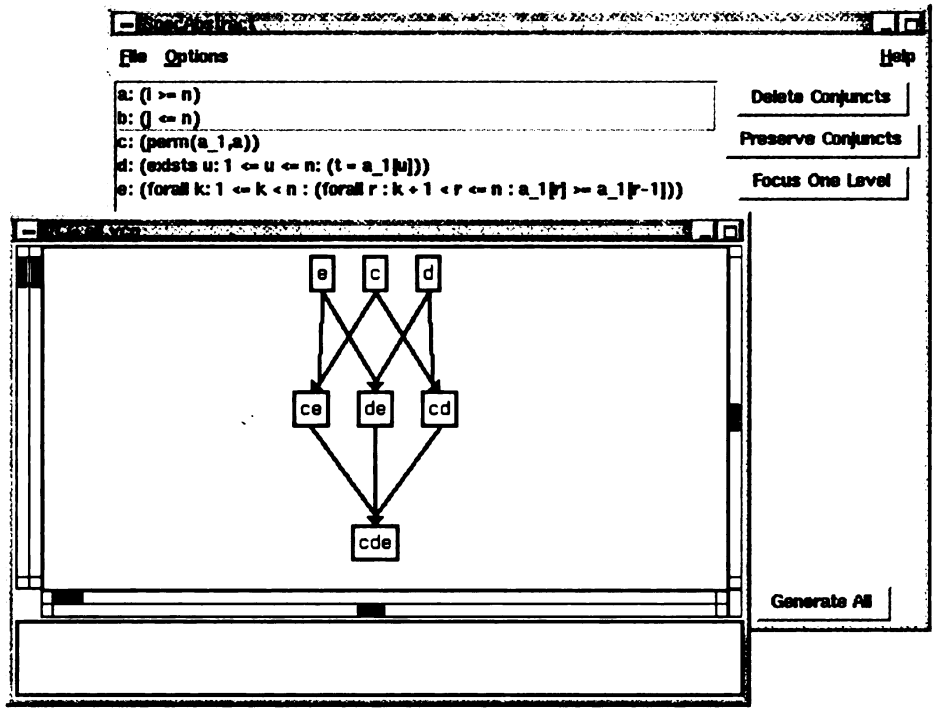


Figure 6.8: Bubble Sort Specification after deletion of "a" and "b"

based mission control system used by the NASA Jet Propulsion Laboratory. The system is responsible for the translation of user commands into appropriate spacecraft mnemonics, enabling users to modify spacecraft mission operations. This particular module takes a sequence of elements from a file and returns an index to a subsequence of elements specified by begin and end indices. In our previous investigations, we described the *sp* semantics for C [7] and pointers [43]. Those semantics were used to construct the `/*AS AS*/` annotations for the code contained in this section.

6.4.1 Code Analysis

First Code Sequence. Appendix C contains a program listing of a module that takes a sequence of elements from a file and returns an index to a subsequence of elements

specified by begin and end indices. These elements correspond to message fragments used for spacecraft control. The annotations for the code in Appendix C were constructed using the *sp* semantic rules for the C programming language.

One code sequence of interest is the code for lines 48-82, which appears as follows:

```
if (!skip_gcmd_sfdu(fd, L2))
{
   inform_user(
     "line %d: copy failed: bad SFDU header (%s)",
     body_lineno, file);
   dontoutput = 1;
   close(fd);
   if (params->cmdcntl) master_unlock();
   return(NULL);
}
```

The purpose for this code sequence is to abort processing if the file header is corrupted. The precondition for this block is

```
(fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0),
```

which makes assertions about the initial values of several variables and pointers, where the & is the logical connective '$\wedge$'. The specification states that `fd` has the initial value `FH0`, and that the value is greater than or equal to 0. The specification also states that the variables `begin` and `end` have the values `B0` and `E0`, respectively. Finally, the specification states that the pointer `file` points to some object `F0`.

The following annotation describes the behavior of the code when the conditional path is taken in the case that `skip_gcmd_sfdu` evaluates to zero:

```
(params->cmdcntl != 0 & master_unlocked() &
     closed(fd) & dontoutput = 1 &
     skip_gcmd_sfdu(fd, L2) = 0 &
     fd >= 0 & fd = FH0 & begin = B0 &
     end = E0 & file .>  F0) |
(params->cmdcntl = 0 & closed(fd) &
     dontoutput = 1 & fd >= 0 &
     skip_gcmd_sfdu(fd, L2) = 0 & fd = FH0 &
     begin = B0 & end = E0 & file .> F0),
```

which is equivalent to

```
((params->cmdcntl != 0 & master_unlocked()) |
     params->cmdcntl = 0 ) &
closed(fd) & dontoutput = 1 &
     skip_gcmd_sfdu(fd, L2) = 0
     & fd >= 0 & fd = FH0 & begin = B0
     & end = E0 & file .>  F0 .
```

This specification states that in addition to the precondition being true, the file `FH0` is closed, the variable `dontoutput` is set to 1, and depending on whether the `params->cmdcntl` has the value 0, the master key is unlocked. In this system, processing is regarded as having failed whenever the variable `dontoutput` is set to a non-zero value. This specification recurs throughout this code when certain failure conditions are met.

The postcondition annotation at lines `84-85` asserts the following:

```
skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 &
fd = FH0 & begin = B0 & end = E0 & file .> F0 ,
```

which states that in addition to the precondition being true, that the function `skip_gcmd_sfdu` evaluates to a non-zero value. This specification is reasonable since the body of the statement in question ends with a `return` statement. As such, the program only proceeds past the conditional statement if `skip_gcmd_sfdu` evaluates to a non-zero value.

Due to space constraints, some annotations were omitted due to the similarity of some blocks of code. For instance, annotations for the code sequence from lines 48-82 are very similar to the annotations that would appear for the code blocks at lines 87-93, 98-106, and 115-123.

Second Code Sequence. Another interesting sequence of code appears in Figure 6.9 and occurs at lines 113-123. One of the activities that can be performed is to analyze the postcondition at lines 125-134 using the specification generalization technique described in Section 6.3. First, we can rewrite the specification into an equivalent form by factoring terms so that the specification appears as follows:

```
((E0 = -1 & end = gcmd_hdr.elem_count) |
   (end <= gcmd_hdr.elem_count & end != -1 &
   end = E0)) & params->sc = gcmd_hdr.SC &
   get_gcmd_hdr(fd, gcmd_hdr) != 0 &
   skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 &
   fd = FH0 & begin = B0 & file .> F0 .
```
(6.4)

This specification states that the constant E0 is equal to -1 and end = gcmd_hdr.elem_count or that end = E0, end <= gcmd_hdr.elem_count, and end != -1. In addition, several conditions regarding the input header are true as well as conditions that describe the input file. At this point the specification is in a form suitable to apply the **delete a conjunct** strategy. Figure 6.10 shows the possible abstractions for the specification when we delete the file related conjuncts. Successive application of the strategy leads to the abstraction of the behavior described by Expression 6.4. This specification corresponds to the specification '*a*' in the Hasse diagram in Figure 6.10 and appears as follows:

```
( (E0 = -1 & end = gcmd_hdr.elem_count) |
  (end <= gcmd_hdr.elem_count & end != -1 & end = E0)),
```

which states that `E0 = -1` and the variable end has the value `gcmd_hdr.elem_count`, or, `end = E0`, `end != -1` and `end <= gcmd_hdr.elem_count`. Essentially, this states that if this point in the program has been reached, the variable `end` has a value that is less than or equal to `gcmd_hdr.elem_count` and not equal to `-1`. This behavior creates an issue that must be addressed since behavior for the case when `end < -1` may not be what is expected.

```
108. /*AS (params->sc = gcmd_hdr.SC &
109.       get_gcmd_hdr(fd, gcmd_hdr) != 0 & file .> F0 &
110.       fd >= 0 & fd = FH0 & begin = B0 & end = E0 &
111.       skip_gcmd_sfdu(fd, L2) != 0 )AS*/
112. /* make sure the file has enough elements */
113. if (end == -1)
114.   end = gcmd_hdr.elem_count;
115. else if (end > gcmd_hdr.elem_count)
116. {
117.   inform_user("line %d: copy: not enough elements \
118.                in GCMD file (%s)",body_lineno, file);
119.   dontoutput = 1;
120.   close(fd);
121.   if (params->cmdcntl) master_unlock();
122.   return(NULL);
123. }
124.
125. /*AS
126.    (E0 = -1 & end = gcmd_hdr.elem_count &
127.     params->sc = gcmd_hdr.SC & E0 = E0 & file .> F0 &
128.     get_gcmd_hdr(fd, gcmd_hdr) != 0 & fd = FH0 &
129.     skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 & begin = B0 )
130.    |
131.    (end <= gcmd_hdr.elem_count & end != -1 & begin = B0 &
132.     params->sc = gcmd_hdr.SC & end = E0 & file .> F0 &
133.     get_gcmd_hdr(fd, gcmd_hdr) != 0 & fd = FH0 &
134.     skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 ) AS*/
135.
```

Figure 6.9: Code Sequence: Lines 108–135

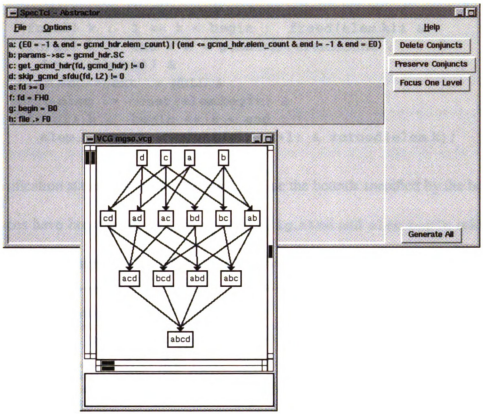


Figure 6.10: Annotation Abstractions

Third Code Sequence. The final annotation that is of interest is found on lines 404-420 of the program in Appendix C. The annotation, also found in Figure 6.11, shows the annotation for the program after a simplification step that factors conjuncts from a disjunction.

Informally, this specification makes assertions about a chain of elements and the relationship between the requested subsequence of elements and the elements read from a file. The following specification abstraction can be derived by applying the *delete a conjunct* strategy to the annotation, where we focus specifically on the conjuncts that contain a reference to the variables `begin` and `end`:

```
(forall k :  1 <= k < begin :  freed(elem_k)) &
(forall k :  end < k < gcmd_hdr.elem_count :
  freed(elem_k)) &
elem_end->next .> NULL &
orig_elem .> coset(elem_begin) &
(forall k :  begin <= k < end :
  elem_k->next .> coset(elem_k+1) & zeroed(elem_k))
```
(6.5)

The specification states that all the elements outside the bounds specified by the begin and end indices have been freed, that the pointers `orig_elem` and `elem_begin` refer to the same object, and that all the elements within the begin and end bounds form a chain.

```
closed(fd) &
(forall k :  end < k < gcmd_hdr.elem_count :  freed(elem_k)) &
ep .> coset(elem_gcmd_hdr.elem_count) &
elem .> coset(ep) &
elem_end->next .> NULL &
orig_elem .> coset(elem_begin) &
checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
elem_gcmd_hdr.elem_count .> NULL &
(forall k :  1 <= k < begin :  freed(elem_k)) &
(forall k :  begin <= k < end :
  elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
((E0 = -1 & end = gcmd_hdr.elem_count) |
  (end <= gcmd_hdr.elem_count & end != -1 & end = E0)) &
params->sc = gcmd_hdr.SC &
get_gcmd_hdr(fd, gcmd_hdr) != 0 &
skip_gcmd_sfdu(fd, L2) != 0 &
fd >= 0 &
fd = FH0 &
begin = B0 &
file .> F0 .
```

Figure 6.11: Code annotation: Lines 404–420

6.4.2 Discussion

The analysis of the code in Appendix C has led to several observations that empirically validate the appropriateness of the *delete a conjunct* strategy for specification generalization of *sp* specifications. First, the specifications that are constructed using *sp* are conjunctive in nature due to the semantics of the assignment statement. As such,

application of the *delete a conjunct* strategy facilitates the analysis of specifications of program code by decomposing those specifications into smaller, more manageable pieces. Second, analysis of annotations that occur within the code (as opposed to only analyzing the final postcondition) is an important activity for understanding the behavior of programs. As an example, our analysis of the code from lines `108-135` and the corresponding specification in Expression 6.4 facilitated the identification of behavior in the program that must be analyzed further in order to determine the impact of inconsistent inputs. Finally, although the reverse engineered specifications describe logical abstractions, some mechanism must be provided in order to describe the abstractions using natural language. For instance, instead of providing the specification in Expression (6.5) to a user, it would be desirable to state that since the procedure returns a subsequence of a list of elements, that the abstract behavior corresponds to a list subsequence *cliché* or *plan* [44], where a *plan* describes common or canonical program behavior.

Chapter

Reverse

The previous chap
to support reverse
technique with th
addition, we descr
and how our form
understanding by f

7.1 Combinir

Due to the mathem
been perceived as
defined, formal me
formal methods ar
hierarchical decomp
thus facilitating eas
is that the notations

Chapter 7

Reverse Engineering Framework

The previous chapters have discussed several techniques that we have developed in order to support reverse engineering. In this chapter we integrate the strongest postcondition technique with the specification generalization technique to form a single process. In addition, we describe how informal methods can be used for high-level concept discovery and how our formal reverse engineering technique can be used to supplement program understanding by facilitating formal reasoning.

7.1 Combining Informal and Formal Approaches

Due to the mathematical nature of formal specification languages, formal methods have been perceived as time consuming and tedious. However, since the languages are well-defined, formal methods have been found to be amenable to automated processing. Semi-formal methods are techniques for specifying system requirements and design using hierarchical decomposition. Many semi-formal methods use notations that are graphical, thus facilitating ease of use in their application. The drawback to semi-formal methods is that the notations are typically imprecise and ambiguous. This section describes an

approach to rev

in order to bene

7.1.1 Struc

Although the re

oriented techno

programming la

of these langua

Analysis and De

function. The a

the system. Dur

descriptions of f

incorporate imp

and procedures i

When using s

is abstracted int

structure charts.

Further analysis

functions by cor

graphical descrip

descriptions as a g

In general, the

the first phase, a

approach to reverse engineering that combines the use of semi-formal and formal methods in order to benefit from the complementary advantages of both approaches.

7.1.1 Structured Analysis

Although the recent trend in software development has been to build systems using object-oriented technology, a majority of existing systems has been developed using imperative programming languages, such as C, FORTRAN, and COBOL. The procedural structure of these languages makes them amenable to the techniques offered by the *Structured Analysis and Design Technique* (SADT) [12]. In SADT, the focal point is the procedure or function. The analysis stage centers around high-level descriptions of the functionality of the system. During the design phase, the refinement and decomposition of the high-level descriptions of functions yields more detailed descriptions of functions and procedures that incorporate implementation details. Finally, during the implementation phase, functions and procedures identified during design are decomposed into more specific functions.

When using SADT for reverse engineering activities, the structure of an implementation is abstracted into low-level graphical descriptions functions known as call graphs or structure charts. These graphs depict the calling hierarchy of functions within a system. Further analysis of source code involves analyzing the data that flows to and from various functions by constructing data flow diagrams. Our approach is to construct various graphical descriptions of a program, in most cases automatically, and then use those descriptions as a guide for constructing formal specifications.

In general, the construction of the graphical descriptions proceeds in two phases. In the first phase, a high-level model is constructed that is based on information gathered

from user manuals or high-level design descriptions. In the case that these documents do not exist, it is appropriate to incorporate user interviews, and if possible, empirical testing to determine high-level behavior. In the second phase, a low-level model in the form of call graphs and/or control-flow graphs is constructed. Several tools exist that support the construction of such models as is described in Chapter 10.

7.1.2 A note about formal techniques and large systems

One of the limitations of our technique is that the specifications that are constructed can grow to be exponential in size with respect to the input program. Given this limitation, the appropriateness of using formal methods in the context of large systems must be well-understood. That is, the use of formal methods for reverse engineering, as is the case with all applications of formal methods, must be targeted to those contexts where it has the highest payoff; namely critical systems [4, 5]. However, in the reverse engineering of software, we can extend the context a bit further to include the parts of a software system that are deemed "critical". In order to determine those critical portions of the software, several factors must be taken into account, including call graph and flow graph complexity.

7.1.3 Applying formal techniques

The motivation of using both semi-formal and formal methods is two-fold. First, it is desirable to take advantage of the benefits of the complementary techniques. Second, by using a semi-formal technique to guide the formal technique, organization of the formal specifications will be based on the structure of an implementation. As such, in the case where formal specifications are warranted, the specifications can be directly associated with

a graphical e

can be left u

semi-formali

We propo

1. Local

2. Use A

3. Global

During th

skeletal form

the *sp* predic

unevaluated.

complexity o

three phases

phase is chara

are determine

the semantics

predicate can

semantics of

to represent th

the final step,

to obtain a glo

form of the sk

upon the sema

a graphical entity, while those parts of a module that do not require rigorous descriptions can be left unspecified (formally), with the descriptions of these modules being left to the semi-formalisms.

We propose that three phases be followed when formally specifying a module:

1. Local Analysis
2. Use Analysis
3. Global Analysis

During the *local analysis* phase, the calling hierarchy of a module is constructed and a skeletal formal specification is built using the rules presented in Chapters 3, 4, and 5, with the *sp* predicates left as parameterized transforms, that is, the transformations for *sp* are unevaluated. The objective at this stage is to gain a high-level understanding of the logical complexity of the given code. The second step, *use analysis*, is a recursive step where the three phases are applied to the functions and procedures *used* by the original module. This phase is characterized by the fact that the semantics of the *used* functions and procedures are determined before they are used by the original module. However, in many cases, where the semantics are either well-defined or the semantics are not critical, an unevaluated *sp* predicate can be used. For example, given a statement S and a precondition Q, where the semantics of S are well-defined, instead of evaluating the transformation, we use $sp(S, Q)$ to represent the logical expression describing the semantics. In the *global analysis* phase, the final step, the *use analysis* information is combined with the *local analysis* information to obtain a global description of the original module. The global description, an expanded form of the skeleton formal specification constructed during the first phase, elaborates upon the semantics of a module by integrating the specifications constructed during the

use analysi

provided by

described a

Defin

Let

the se

M. A

M ca

Metho

1.

2.

3.

7.1.4 Ab

After constr

Section 7.1.3

described in

steps during i

the logical co

method to be

Definiti

Let M

the set o

M. A st

M calls

use analysis into the skeleton. This activity corresponds to removing the encapsulation provided by a procedure or function call. The following definition summarizes the method described above.

Definition 7.1 (Structure Based Analysis Method)
Let M be a program with statements $m_1, \ldots, m_k$, and $P = \{P_1 \ldots P_n\}$ be the set of procedures called by M. In addition, let Q be the precondition for M. A statement m_i is in P if there is a procedure p_j in P such that program M calls p_j at line i of M.

Method $R(M, Q)$

1. *(a) Apply $sp(m_1; \ldots; m_k, Q)$*
   (b) For each i such that $m_i \in \{P_1, \ldots, P_n\}$
   set $sp(m_i, sp(m_1; \ldots; m_{i-1}, Q))$:= "$sp(m_i, sp(m_1; \ldots; m_{i-1}, Q))$".
   (We refer to the right hand side as the skeleton.*)*
2. *For each $p \in \{P_1, \ldots, P_n\}$, apply $R(p, Q_p)$, where Q_p is the precondition to the procedure p.*
3. *Replace skeletons from Step 1b with results of Step 2.*

7.1.4 Abstraction

After constructing an as-built formal specification using the process described in Section 7.1.3, an abstraction of the specification can be constructed using the approach described in Chapter 6. In addition, we have found it appropriate to apply the abstraction steps during intermediate steps of the method in Definition 7.1 in order to aid in reducing the logical complexity of the specifications. As such, we can modify the structure based method to be the following.

Definition 7.2 (Structure Based Analysis Method (Abstraction))
Let M be a program with statements $m_1, \ldots, m_k$, and $P = \{P_1 \ldots P_n\}$ be the set of procedures called by M. In addition, let Q be the precondition for M. A statement m_i is in P if there is a procedure p_j in P such that program M calls p_j at line i of M.

Met

1.

2.

3.

4.

7.1.5 Pr

The entire c

as follows:

Defini

1. C
2. C
3. A
   cr

One of th

can be used fo

procedures by

procedures is t

the in-degree a

data structure us

Method $R'(M, Q)$

1. *(a) Apply $sp(m_1; \ldots; m_k, Q)$. Abstraction method may be used after each application of sp.*
   (b) For each i such that $m_i \in \{P_1, \ldots, P_n\}$ set $sp(m_i, sp(m_1; \ldots; m_{i-1}, Q))$:= "$sp(m_i, sp(m_1; \ldots; m_{i-1}, Q))$". (We refer to the right hand side as the skeleton.*)*
2. *For each $p \in \{P_1, \ldots, P_n\}$, apply $R(p, Q_p)$, where Q_p is the precondition to the procedure p.*
3. *Replace skeletons from Step 1b with results of Step 2.*
4. *Apply the abstraction method to the final specification.*

7.1.5 Process Summary

The entire combined process for the reverse engineering of programs can be summarized as follows:

Definition 7.3 (Informal and Formal Reverse Engineering Method)

1. *Construct an informal high-level model of the software*
2. *Construct an informal low-level model of the software*
3. *Apply R' to a module M, where M is chosen using some selection criteria.*

One of the primary difficulties in the process is the determination of the criteria that can be used for Step 3. The criteria that we have used include the identification of critical procedures by examining the call graph constructed in Step 2. Our selection of critical procedures is typically based on choosing those vertices with a large difference between the in-degree and out-degree. Other criteria that can be used include keyword search and data structure usage.

7.2 A

In this se

to reverse

Jet Propu

spacecraft

However,

This exam

specificati

7.2.1 L

Figure 7.1

semi-forma

where the

given by t

that the `tra`

`process_b`

and `proce`

translate, co

translate fun

control argu

`INIT`, `CARG`,

are left with

been attached

7.2 An Example

In this section, we demonstrate the use of the combined formal and informal approach to reverse engineer modules from a mission control ground-based system at the NASA Jet Propulsion Laboratory. The purpose of the code is to translate user commands into spacecraft commands. The entire system consists of several thousand lines of code. However, in many instances, it is more appropriate to analyze more critical sections. This example focuses on a sequence of code in order to illustrate the derivation of the specifications of modules that contain representative logic and programming constructs.

7.2.1 Local Analysis

Figure 7.1 gives the code for the translate procedure. Using AUTOSPEC, an initial semi-formal analysis of the translate code yields a call graph as depicted in Figure 7.2, where the rectangles indicate functions, and the labels correspond to the function names given by the index to the right of the graph. From this initial analysis, we find that the `translate` function uses five functions including `initialize_interpreter`, `process_binary_output`, `inform_user`, `process_mnemonic_input`, `end_cmdxlt`, and `process_carg`. The `translate` function has four different modes: initialize, translate, control argument assignment, and error. For this analysis, we focus on the translate function in the translate mode (`XLT`). Thus, we are ignoring the initialization, control argument, and default modes in this analysis. These modes correspond to the `INIT`, `CARG`, and `default` cases of the `switch` statement, respectively. Therefore, we are left with specifying the `while` statement depicted in Figure 7.3, where labels have been attached to the programming constructs for convenience in the following discussion.


struct ms
{
exter
stati
struc
Switc
{
ca
ca
Cas
def
}
return(
}

```
struct msg *translate (int op, char *args)
{
    extern int dontoutput;
    static struct project_parameters *pp;
    struct msg *mp = NULL;

    switch (op)
    {
        case INIT:              /* initialize the interpreter */
           pp = initialize_interpreter();
           break;

        case XLT:               /* interpret a message */
           while (args[0] != '\0')
           {
              if (process_mnemonic_input(&args, pp))
              {
                 if (mp == NULL)
                    mp = process_binary_output(pp);
                 else
                 {
                    mp->next = process_binary_output(pp);
                    mp = mp->next;
                 }
              }
              else
                 dontoutput = 1;
           }
           break;

        case CARG:   /* set a value for a control argument */
           process_carg(&args, pp);
           break;

        default:
           inform_user("internal error: bad op in translate");
           end_cmdxlt(CMD_ERROR);
    }

    return(mp);
}
```

Figure 7.1: Translate Source Code

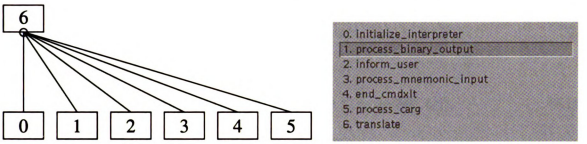


Figure 7.2: Translate

Informally, the `translate` function in the translate mode is responsible for building a list of spacecraft instructions corresponding to interpreted commands by calling a function called `process_binary_output`.

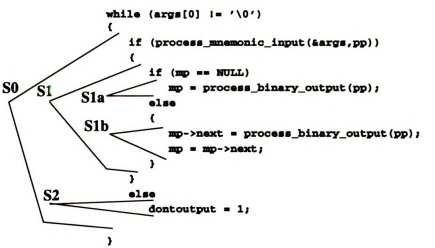


Figure 7.3: Translate Source Code

A loca

statement

where the

to the stat

executed. t

some numb

outside of

of *spi* **S0**. *Q*

Using t

where B c

states that af

was execute

and $\neg\mathcal{V}(B)$

executed or

precondition

contain a sid

A local analysis (first step) of the code in Figure 7.3 using the *sp* rule for the `while` statement yields the following specification:

$$\neg(\texttt{args[0] != '\0'}) \wedge (\exists i : 0 \leq i : sp(\mathbf{S0}^i, Q)), \tag{7.1}$$

where the expression (`args[0] != '\0'`) has no side-effects, and Q is the precondition to the statement **S0**. This specification states that after the `while` statement has been executed, the `args` array has a '\0' as the first entry, and the statement **S0** has been executed some number of iterations. Unfortunately, the specification in (7.1) is not very informative outside of identifying that the program uses an iterative construct. As such, an expansion of $sp(\mathbf{S0}, Q)$ is warranted.

Using the labels shown in Figure 7.3, a specification of $sp(\mathbf{S0}, Q)$ is given by

$$\begin{aligned} sp(\mathbf{S0}, Q) \;=\; & sp(\mathbf{S1}, \mathcal{V}(B) \wedge sp(B, Q)) \vee \\ & sp(\mathbf{S2}, \neg\mathcal{V}(B) \wedge sp(B, Q)) \end{aligned} \tag{7.2}$$

where B corresponds to `process_mnemonic_input(&args,pp)`. This specification states that after executing the statement **S0**, it will be true that either **S1** was executed or **S2** was executed, where the semantics are determined by the preconditions $\mathcal{V}(B) \wedge sp(B, Q)$ and $\neg\mathcal{V}(B) \wedge sp(B, Q)$, respectively. So, in this case, either the `if` statement (**S1**) was executed or the assignment statement (**S2**) was executed. This specification states that the precondition $sp(\texttt{process\_mnemonic\_input(\&args,pp)}, Q)$ to the statement **S0** may contain a side-effect. This fact is made explicit by the use of the valuation function $\mathcal{V}$.

Note that

Further ex

sp S

and

$sp($S2. –

respectively

states that g

executed or

$mp = NUL$

that given th

assignment o

The preli

be constructe

Expression (7

Note that if the function `process_mnemonic_input` has no side-effect then

$$sp(\texttt{process\_mnemonic\_input(\&args,pp)}, Q) = Q.$$

Further expansion of $sp(\mathbf{S1}, \mathcal{V}(B) \wedge sp(B, Q))$ and $sp(\mathbf{S2}, \neg\mathcal{V}(B) \wedge sp(B, Q))$ yield

$$\begin{aligned} sp(\mathbf{S1}, \mathcal{V}(B) \wedge sp(B, Q)) \;=\; & sp(\mathbf{S1a}, (mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B, Q)) \vee \\ & sp(\mathbf{S1b}, (mp \neq NULL) \wedge \mathcal{V}(B) \wedge sp(B, Q)), \end{aligned} \tag{7.3}$$

and

$$\begin{aligned} sp(\mathbf{S2}, \neg\mathcal{V}(B) \wedge sp(B, Q)) \;&=\; sp(\texttt{dontoutput = 1}, \neg\mathcal{V}(B) \wedge sp(B, Q)) \\ &=\; (dontoutput = 1) \wedge (\neg\mathcal{V}(B) \wedge sp(B, Q))_v^{dontoutput} \end{aligned} \tag{7.4}$$

respectively, where v is the value of `dontoutput` before executing **S2**. Expression (7.3) states that given that the expression '$\mathcal{V}(B) \wedge sp(B, Q)$' is true, either **S1a** has been executed or **S1b** has been executed, each depending on the added condition that either $(mp = NULL)$, or $(mp \neq NULL)$, respectively. On the other hand, Expression (7.4) states that given that the expression '$\neg\mathcal{V}(B) \wedge sp(B, Q)$' is true, execution of **S2** results in the assignment of '`1`' to the variable '`dontoutput`'.

The preliminary skeleton of the logical specification of the translation module can be constructed by substituting the Expressions (7.3) and (7.4) back into the original Expression (7.2) such that

which stat

S1b, or S2

At this

specificati

begin a *us*

function p

In sum

of the func

function.

encapsulati

7.2.2 U

Use analys

In our exa

involves sp

function pr

Figure

analysis fo

followed fo

$$
\begin{aligned}
sp(\mathbf{S0}, Q) \;=\; & sp(\mathbf{S1a}, (mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B, Q)) \vee \\
& sp(\mathbf{S1b}, \neg(mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B, Q)) \vee \\
& (dontoutput = 1) \wedge (\neg\mathcal{V}(B) \wedge sp(B, Q))_{v}^{dontoutput}
\end{aligned} \tag{7.5}
$$

which states that in every iteration, one of three actions is executed, namely one of **S1a**, **S1b**, or **S2** (`dontoutput = 1`).

At this point in the analysis, since **S1a** and **S1b** are statements that depend on the specification of functions and procedures that are used by `translate`, it is appropriate to begin a *use analysis* (second stage) for the `translate` function, where in this case, the function `process_binary_output` is analyzed.

In summary, during the local analysis phase for `translate`, a graphical representation of the function was created with the intention of determining the calling hierarchy for the function. Next, a logical analysis was performed using a top-down approach that uses encapsulation with the intent of determining the logical complexity.

7.2.2 Use Analysis

Use analysis involves the specification of functions that are used by a given object of study. In our example, given that the object of study is the `translate` function, use analysis involves specifying the functions used by `translate`. In this section we describe the function `process_binary_output`.

Figure 7.4 contains the source code for `process_binary_output`. The *use* analysis for this function involves three steps, each corresponding to the steps followed for `translate`. That is, we perform *local*, *use*, and *global* analyses on

process.

similar to

simplifyin

process_

strict appli

Here, we

informatio

analysis is

to describe

of the entir

chapter is t

these constr

show how a

translate

Consider

determine w

respectively.

for the return

object due to

Therefore, we

process_binary_output. The remaining analysis of process_binary_output is similar to the process used to analyze translate. However, in the interest of simplifying the analysis we shall ignore many of the details involved with analyzing process_binary_output and focus primarily on the output characteristics. Note that the strict application of the rules for *sp* requires a line by line construction of a specification. Here, we informally construct the specification with the understanding that all of the information can and should be constructed rigorously. Our main objective in this example analysis is to provide enough information about process_binary_output to be able to describe translate in a sufficient manner without having to perform a full analysis of the entire command translation system. Again, we note that the code used in this chapter is taken out of context. Therefore, it is unreasonable to specify this code without these constraints. Therefore, the specification given in this section is used primarily to show how a true specification of process_binary_output might be used to describe the translate function.

Consider the code given in Figure 7.4 for process_binary_output. Three statements determine whether or not the output of the function is defined, labeled by I, J, and K, respectively. Line I, for instance, has the interpretation that if space could not be allocated for the return object, then the routine aborts, while line J forces the routine to return a NULL object due to some other error. Finally, the line K indicates a successful return of an object. Therefore, we can construct the following specification for process_binary_output:

```
struc
{
    e
    U
    s

    W
    S

    if
    {

    }
    /*
    PU

    ep
    P
    do
    {

    } w
    mp-
    if
    if
    {

    }
    mp-
    cop
    cop
    cop
    cop

    mp->
    retu
}
```

```
   struct msg *process_binary_output (struct project_parameters *pp)
   {
       extern U16 *stem_entry;
       U16 code;
       U16 *ep;
       struct msg *mp;

       Q = control_list;
       W = (U16 *)stack_base;
       S = (U32 *)min_S;

       mp = (struct msg *)malloc(sizeof(struct msg) + MAX_MSG_BYTES);
I:     if (mp == NULL)
       {
           warn("process_binary_output: \
                 out of memory (malloc failed)\n");
           end_cmdxlt(-1);
       }
       /* -1 for length field, written over later */
       PUSHL(mp->msg_bits - 1);

       ep = get_entry(get_U32_Q());
       P = ep + 1;
       do
       {
           code = *P++;
           if ((code < 1) || (code > 32))
           {
               warn("bad code");
               end_cmdxlt(-1);
           }
           (*output_rtn[code])();
       } while (code != RFMS);
       mp->next = NULL;
       mp->msg_len = *(mp->msg_bits - 1);
J:     if (mp->msg_len > pp->max_msg_bits)
       {
           fail(TOO_MANY_BITS, NULL, NULL);
           free(mp);
           return(NULL);
       }
       mp->msg_num = 0;
       copy_space_filled("", mp->start, sizeof(mp->start));
       copy_space_filled("", mp->open, sizeof(mp->open));
       copy_space_filled("", mp->close, sizeof(mp->close));
       copy_space_filled(get_stem_and_title(stem_entry), mp->comment,
                         sizeof(mp->comment));
       mp->chksum = chksum(mp->msg_bits, FLD_LEN_OF(mp->msg_len)*2);
K:     return(mp);
   }
```

Figure 7.4: Process Binary Output Source Code

sp(warn,
sp(fail, fr
sp(return

which stat

were execu

Again, we

functionali

7.2.3 G

The final st

back into th

sp(**S0**, *Q*

where u is so

has either the

or points to s

cases holds, i

the behavior

construct. As

is constructed

make the assu

$$sp(\textit{warn; end\_cmdxlt}, (mp = NULL) \wedge Q) \;\vee$$
$$sp(\textit{fail; free; return}(NULL), (\texttt{mp->msg\_len} > \texttt{pp->max\_msg\_bits}) \wedge (mp \neq NULL) \wedge Q)) \;\vee$$
$$sp(\textit{return}(mp), (\texttt{mp->msg\_len} \leq \texttt{pp->max\_msg\_bits}) \wedge (mp \neq NULL) \wedge Q) \tag{7.6}$$

which states that after executing `process_binary_output` either `warn` and `end_cmdxlt` were executed, the routine returned a `NULL` object, or the routine returned a valid object. Again, we stress that this specification is incomplete and only specifies a small slice of the functionality of the routine.

7.2.3 Global Analysis

The final step in the analysis is to take the specification in Expression (7.6) and integrate it back into the skeleton specification of Expression (7.5). This specification is as follows

$$\begin{aligned} sp(\mathbf{S0}, Q) \;=\; & ((mp = NULL) \vee (mp = u)) \vee \\ & (((mp\text{->}next = NULL) \vee (mp\text{->}next = u)) \wedge (mp = mp\text{->}next)) \vee \\ & (dontoutput = 1) \wedge (\neg \mathcal{V}(B) \wedge sp(B, Q))_{v}^{dontoutput} \end{aligned} \tag{7.7}$$

where u is some new object. This specification states that after executing **S0**, the variable `mp` has either the value `NULL` or points to some new object, or `mp->next` has the value `NULL` or points to some new object with `mp` pointing to `mp->next`. Finally, if neither of those cases holds, it must be that `dontoutput = 1`. Recall that Expression (7.1) describes the behavior of the `XLT` mode for `translate` and that the `XLT` mode uses an iterative construct. As such, Expression (7.7) states that after each iteration, a chain of messages is constructed or the `dontoutput` flag is set to '1'. Note that in this specification we make the assumption that the pointer assignment behaves like a variable assignment. In

this case.

the alloca

7.2.4 I

At a high

mode, ea

condition.

to Express

a specifica

to derive s

of softwar

facilitate fo

imperative.

understand

still reflect

a mechanis

level repres

it is import

specificatior

this case, this assumption has no impact on the specification since no reference is made to the allocated data objects outside of simple assignments.

7.2.4 Discussion

At a high-level, the specification in Expression (7.7) states that while in the XLT mode, each iteration of the loop adds to a chain of messages or results in an error condition. The refinement of the specification of the XLT mode from Expression (7.1) to Expression (7.7) represents just a small portion of what would be required to obtain a specification of an entire system. As described in Section 7.1.2, it is not feasible to derive specifications for the entire system. Formal specifications of critical sections of software, however, are merited and having such formal, concise specifications can facilitate formal program understanding since the specifications are behavioral rather than imperative. The specification in Expression (7.7) does provide a somewhat higher-level of understanding of the corresponding program code. Nonetheless, the resulting specifications still reflect significant implementation bias. However, the "as-built" specifications provide a mechanism for traceability of the reverse engineering process, particularly as the higher-level representations become more abstract. That is, for technology transfer purposes, it is important for system maintainers to understand the starting point for the formal specification process for reverse engineering.

Chapter 8

Tool Support

One of the attractive properties of formal methods is that formal languages with well-defined syntax and semantics facilitate the use of automated support tools. In this chapter, we describe the development of several tools that have been designed to support the formal reverse engineering techniques presented in this dissertation.

8.1 Overview

Chapter 10 describes several tools that have been developed to support reverse engineering activities. In this chapter, we describe the development of a suite of reverse engineering tools that have been designed to support the formal reverse engineering techniques presented in this dissertation. The suite consists of four tools:

AUTOSPEC: AUTOSPEC is a tool that is used to support the construction of specifications using the semantics of the strongest postcondition predicate transformer.

SPECGEN: SPECGEN is a tool that is used to support the derivation of abstract specifications from as-built specifications.

SPECEDIT: SPECEDIT is a specification editor that is used to support the construction of syntactically correct specifications.

TP

Figur

in the for

the tools

during th

circles.

The over

the SUIF Co

programming

TPROVER: TPROVER is a tableau theorem prover that is used to verify the consistency of specifications that are modified by a user.

Figure 8.1 shows the inter-relationships that exist between the various tools in the suite in the form of a data flow diagram. In addition, the diagram shows the relationship between the tools in the suite and external tools that we have used to aid in the analysis of software during the reverse engineering process. In the diagram, external tools are shown as dashed circles.

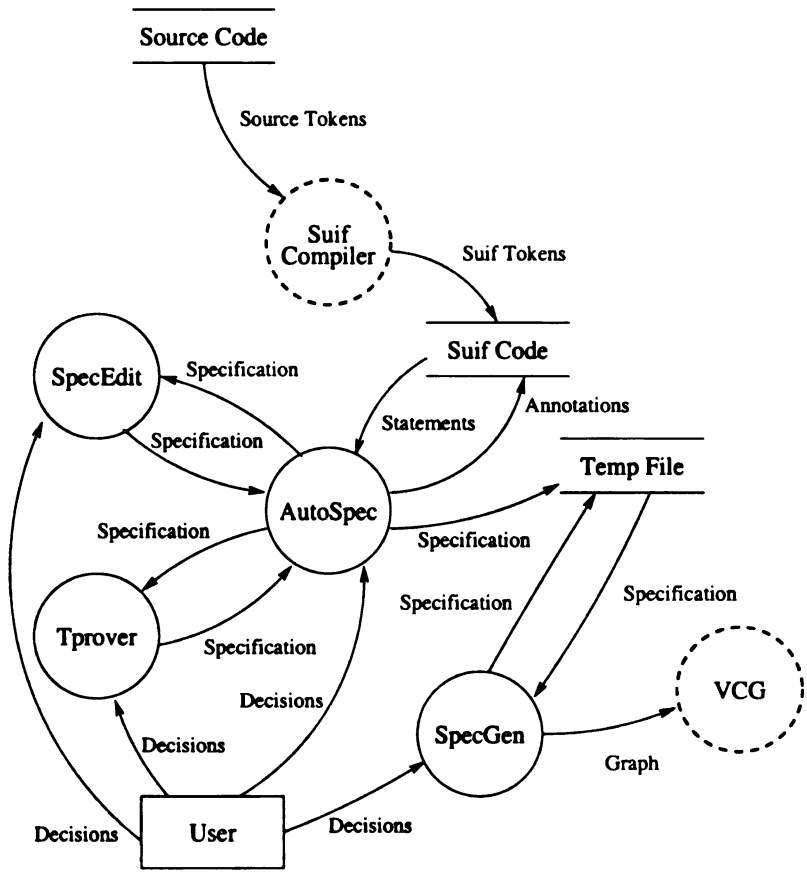


Figure 8.1: Tool Suite

The overall process for using the tools begins with a pre-processing step whereby the SUIF Compiler [45] is used to generate an intermediate format based on the C programming language. The AUTOSPEC system takes the SUIF generated code as input

and ba

can pro

editor.

to the g

using th

be visua

8.2 A

The (Ser

in order

engineeri

prototype

specificati

different v

with the in

including t

analysis of

AUTOSPEC

languages a

8.2.1 De

The high-lev

system intera

and based on user input, generates source code annotations. During the analysis, the user can provide assistance to the AUTOSPEC system via the use of the SPECEDIT specification editor. In addition, the TPROVER theorem prover can be used to verify user modifications to the generated specifications. Finally, after as-built specifications have been constructed using the AUTOSPEC tool, the SPECGEN tool can be used to generate abstractions that can be visualized using the Visualization of Compiler Graphs (*VCG*) tool [46].

8.2 AUTOSPEC

The (Semi-)***Auto***mated ***Spec***ification system, or AUTOSPEC, was originally developed in order to demonstrate the feasibility of our initial investigations into reverse engineering [17]. Written in an object-oriented variant of Prolog, the original prototype facilitated the application of user-directed heuristics to construct predicate logic specifications from Dijkstra guarded command language programs [17]. Since then, several different variations and refinements of the AUTOSPEC system have been developed, each with the intention of investigating some aspect of the research described in this dissertation, including the analysis of programs using strongest postcondition semantics [6], and the analysis of pointer semantics [43]. In this section, we describe the most recent version of AUTOSPEC that has been refined from previous versions in order to handle more complex languages and a wider variety of programs.

8.2.1 Design

The high-level design of the AUTOSPEC system is shown in Figure 8.2. The AUTOSPEC system interacts with three different environmental entities: the *User*, a specification editor

called SP

theorem p

reads a fil

formal spe

source co

comes in

user also

and TPROV

The desi

compiler an

consists of a

tree, an *ana*

and an *outpu*

contains the

called SPECEDIT, and a theorem prover called TPROVER. The specification editor and theorem prover are described in Sections 8.4 and 8.5, respectively. The AUTOSPEC system reads a file, and based on various interactions with the user and external tools, generates formal specifications based on the use of strongest postcondition, and annotates the original source code with those specifications. Direct user interaction with the AUTOSPEC system comes in the form of decisions about how a source file analysis should proceed. The user also interacts with the AUTOSPEC system indirectly via the use of the SPECEDIT and TPROVER systems.

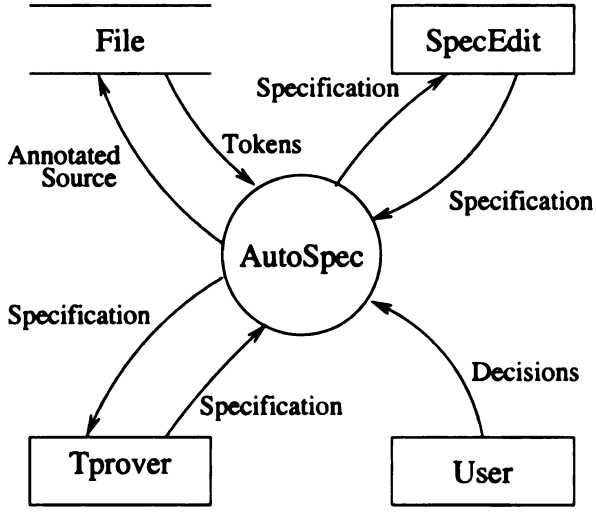


Figure 8.2: Level 0 AUTOSPEC Model

The design of the AUTOSPEC system follows the same general architecture of many compiler and static analysis systems [47]. That is, the design of the AUTOSPEC system consists of a *parsing* component that reads a source file and creates an *abstract syntax tree*, an *analysis component* that is used to construct specifications from the program, and an *output* component that writes the results to an appropriate output file. Figure 8.3 contains the level 1 data flow diagram of the AUTOSPEC system. The *Parse*, *SP*, and

Output pr

In additio

componer

system is

addition t

and *annot*

statements

T

The prima

The *SP* comp

specifications

Output processes correspond to the *parsing*, *analysis* and *output* components, respectively. In addition to the standard compiler-oriented components, the AUTOSPEC system has a component for interacting with the user (i.e., a *user interface*). Data in the AUTOSPEC system is centered primarily around the *Abstract Syntax Tree* and *program statements*. In addition to statements, flow of data in the AUTOSPEC system consists of *specifications* and *annotations*, where annotations are specifications that are tied to specific program statements.

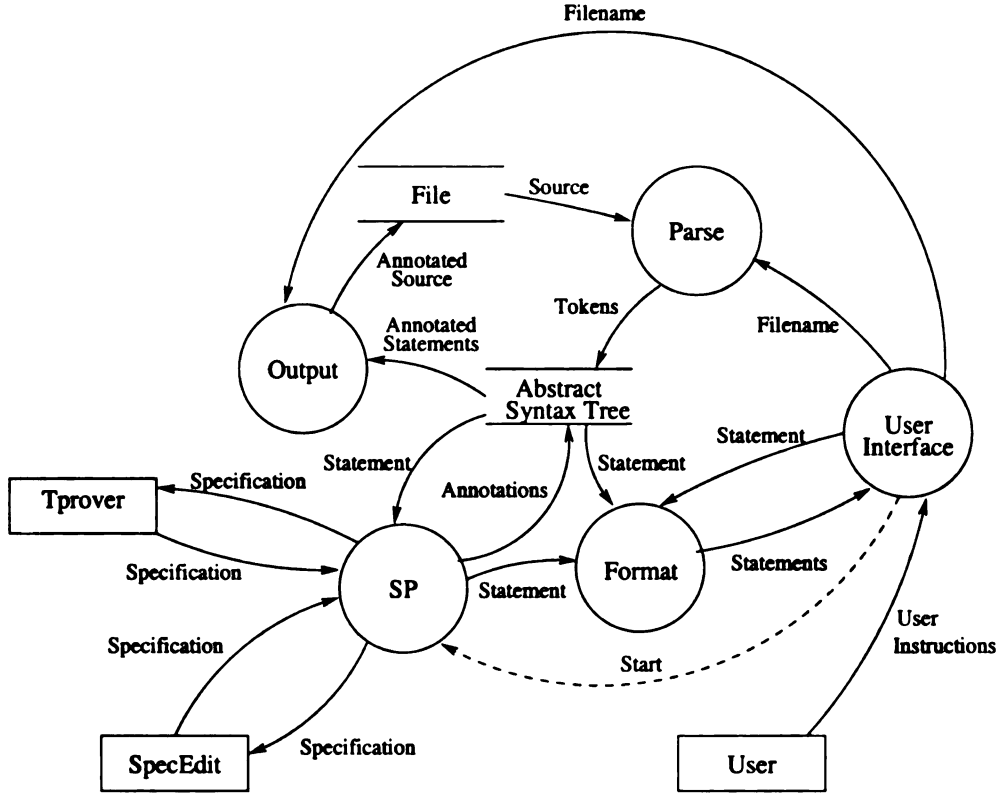


Figure 8.3: Level 1 Data Flow Diagram of AUTOSPEC

The primary component of the AUTOSPEC system is the analysis, or *SP* component. The *SP* component consists of several procedures that are responsible for constructing specifications from programming constructs. The formal specifications that are generated

by the *SI*

dissertati

launching

8.2.2

The AUT

of source

developec

primarily

the applic

must be ir

section de

SUIF Cor

The Stanf

were deve

Developed

is to prov

investigatic

The mo

compiler pr

via the use

SUIF comp

use of *anno*

by the *SP* component correspond directly to the semantic definitions given throughout this dissertation. In addition to constructing specifications, the *SP* component is responsible for launching the TPROVER and SPECEDIT applications when user input is required.

8.2.2 Implementation

The AUTOSPEC system was developed primarily as a means for supporting the analysis of source code using strongest postcondition. The AUTOSPEC system was originally developed to support the Dijkstra guarded command language. Since that language is used primarily for theoretical development and analysis, it was decided that in order to show the applicability of our approaches to real systems, support for a commonly-used language must be included in the subsequent implementations of AUTOSPEC. The remainder of this section describes the implementation of a C variant of the AUTOSPEC system.

SUIF Compiler

The Stanford University Intermediate Format (SUIF) library is a suite of routines that were developed to support research for optimizing and parallelizing compilers [45]. Developed by the Stanford University Compiler Group, the objective of the SUIF compiler is to provide an extensible support system for a wide variety of compiler-oriented investigations [45].

The motivation for using the SUIF compiler suite of tools is as follows. First, the SUIF compiler provides a library of routines for parsing and accessing source code information via the use of an abstract syntax tree representation of SUIF code. Second, by using the SUIF compiler, we are able to take advantage of several built-in features including the use of *annotations* to document programs, and source code iterators for traversing the

abstract sy

Finally, by

communi

The S

based on

of Fortra

conditiona

that is dec

libraries su

to traverse

SUIF

AUTOSPEC

specificati

annotation

In the A

scc is used

SUIF librar

intermediat

to access th

specificatior

facility is us

final, a tool

code with an

abstract syntax trees. In addition, the SUIF library has extensive support for symbol tables. Finally, by using the SUIF compiler we are able to leverage the experience of an established community of users.

The SUIF library focuses on the organization of input files into abstract syntax trees based on the structure of the C programming language. SUIF also supports the analysis of Fortran programs and contains support for programming constructs, such as loops, conditionals, and assignments. These constructs are translated into an intermediate format that is decomposed into semantically equivalent SUIF constructs. In addition, the SUIF libraries support the use of symbol tables as well as convenience functions that can be used to traverse source code.

SUIF also supports the use of source code annotations. In the context of the AUTOSPEC system, these SUIF code annotations are used to attach strongest postcondition specifications to particular programming statements. After analyzing the source code, these annotations can be translated into comments and annotated to the original source code.

In the AUTOSPEC system we use the SUIF tools as follows. First, the SUIF compiler *scc* is used to generate a SUIF intermediate file from the original source code. Second, the SUIF library of tools is used by the AUTOSPEC system in order to manipulate the SUIF intermediate file. Third, the symbol table and source code traversal functions are used to access the abstract syntax trees for the input source code in order to generate formal specifications based on the semantics of the strongest postcondition. Fourth, the annotation facility is used to associate formal specifications with specific source statements. Fifth and final, a tool called *s2c* is used to translate the SUIF source code into equivalent C source code with annotations.

Interface

The interf

In additio

the SUIF

main winc

procedure

The an

on allowin

for indicati

Interface

The interface for the AUTOSPEC system was constructed using the Tcl/Tk language [48]. In addition, we used the C++ language to provide the interconnection between Tcl/Tk and the SUIF libraries. The interface is organized primarily around the input source code. The main window, as shown in Figure 8.4, is used to display the source code for a user-selected procedure.

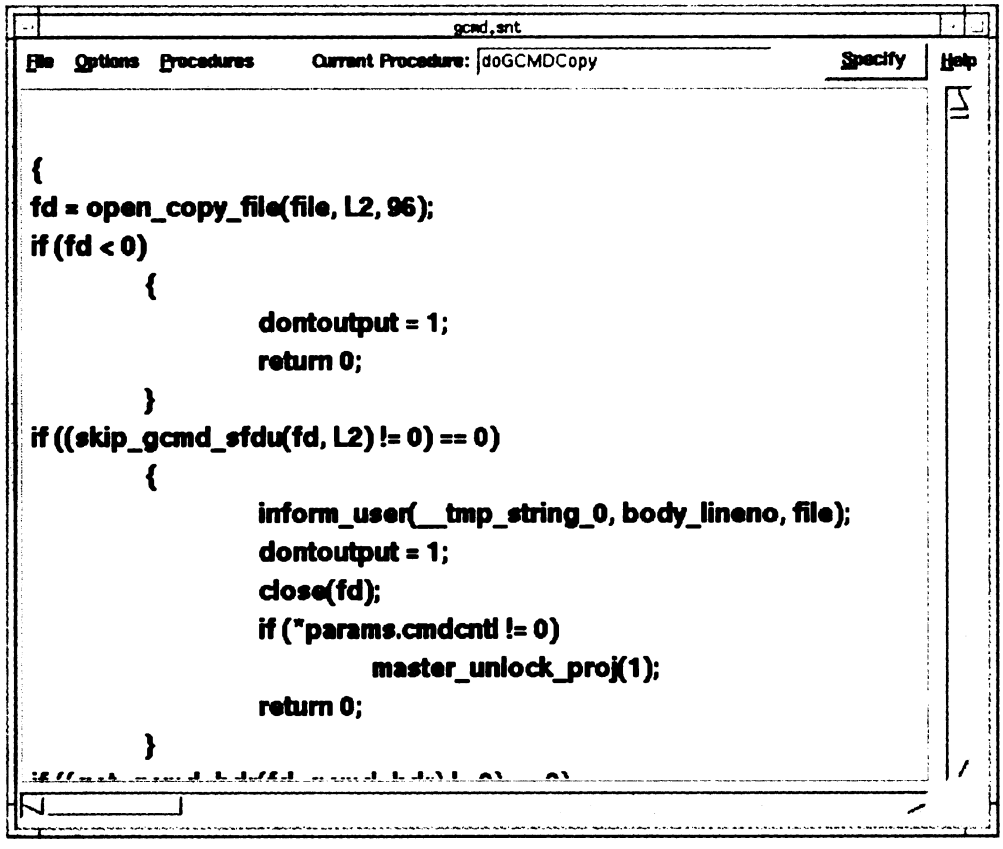


Figure 8.4: AUTOSPEC Main Window

The analysis of the source code includes three distinct phases. The first phase focuses on allowing a user to select *analysis breakpoints*, thus providing the user with a means for indicating where they prefer to provide input to the analysis. Selecting (i.e., "double-

clicking

procedu

shows t

system.

The se

constructs

semantics

AUTOSPEC

annotation

specificatio

in Section

clicking") a particular programming statement indicates that the analysis of the current procedure should be interrupted just before processing of the selected statement. Figure 8.5 shows the interface with a selected statement shown in italics. During an execution of the system, the selected statement appears italicized in blue.

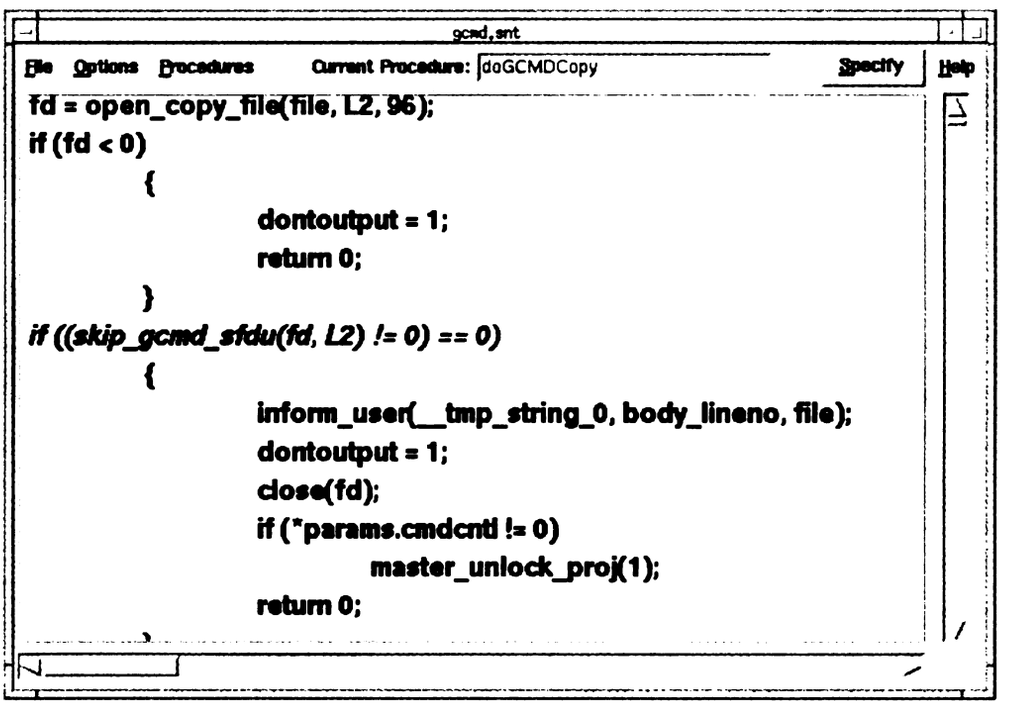


Figure 8.5: AUTOSPEC A Selection in the Main Window

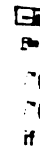
The second phase of the analysis is the *specification phase*. In this phase, AUTOSPEC constructs a formal specification of the procedure by using the strongest postcondition semantics described in this dissertation. As each analysis breakpoint is encountered, AUTOSPEC pauses, as depicted in Figure 8.6, and allows the user to modify the current annotation (i.e., the precondition for the next statement). The modification of the specification is performed by using the SPECEDIT specification editor, which is described in Section 8.4. Using the SPECEDIT system, the user modifies the precondition and can

optionally

user-defin

prover to s

implemen



The fina

user is free

In addition,

specification,

optionally launch a theorem prover that can be used to check the consistency between the user-defined specification and the system-defined specification. We developed the theorem prover to support the tableau proof method [49]. In Section 8.5 we describe the design and implementation of this prover.

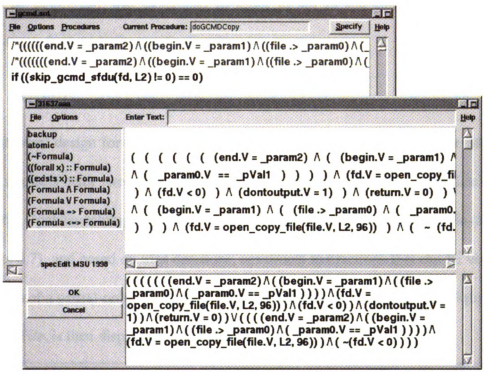


Figure 8.6: Launching SPECEDIT from AUTOSPEC

The final phase of the analysis is the *post-specification* phase. During this phase, the user is free to select and modify any annotation that is displayed in the main window. In addition, we are planning on providing a mechanism where the user can modify a specification, replay the analysis, and incorporate the changes into the analysis.

8.3 S

The Au

the speci

need to b

system.

specifica

8.3.1

The high

system in

VCG [46]

in Figure

in the form

the *Graph*

8.3 SPECGEN

The AUTOSPEC system focuses on the construction of as-built specifications. Once the specifications have been constructed using the AUTOSPEC system, the specifications need to be generalized into higher-levels of abstraction. The *Spec*ification *Gen*eralization system, or SPECGEN, was developed in order to aid in the construction of abstract specifications from as-built specifications.

8.3.1 Design

The high-level design for the SPECGEN system is shown in Figure 8.7. The SPECGEN system interacts with two environmental entities: the *User*, and the visualization tool *VCG* [46]. Upon launching, the SPECGEN system reads a specification from a file (*SpecFile* in Figure 8.7), and based on user decisions, constructs abstractions that can be visualized in the form of a partial-order diagram. The partial-order diagram, depicted in Figure 8.7 as the *GraphFile*, is then displayed using the *VCG* tool.

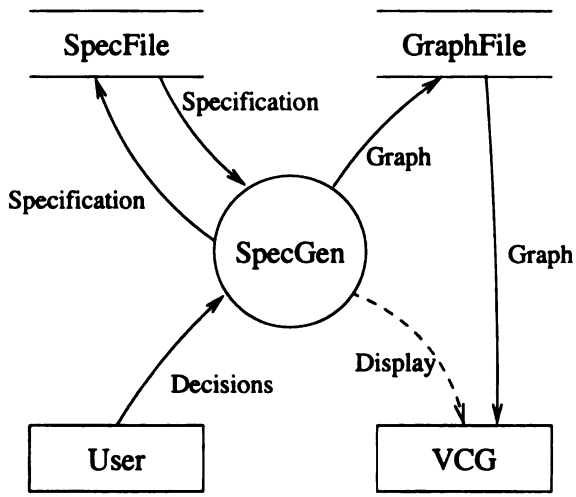


Figure 8.7: Level 0 SPECGEN Model

Figur

internal s

a *user in*

parser tha

Section 8.

for media

engine is

technique.

Spe

8.3.2 Im

The implem

component i

Figure 8.8 shows the level 1 data flow diagram for the SPECGEN system. The internal structure of the SPECGEN system consists of three major components: a *parser*, a *user interface*, and an *abstraction engine*. The parser is a standard *Lex* and *Yacc* [50] parser that has been constructed for checking the syntax of first-order logic specifications. Section 8.6 describes the parser in more detail. The user interface is the primary mechanism for mediating interaction between the user and the abstraction engine. The abstraction engine is responsible for constructing high-level specification generalizations based on the techniques described in Chapter 6.

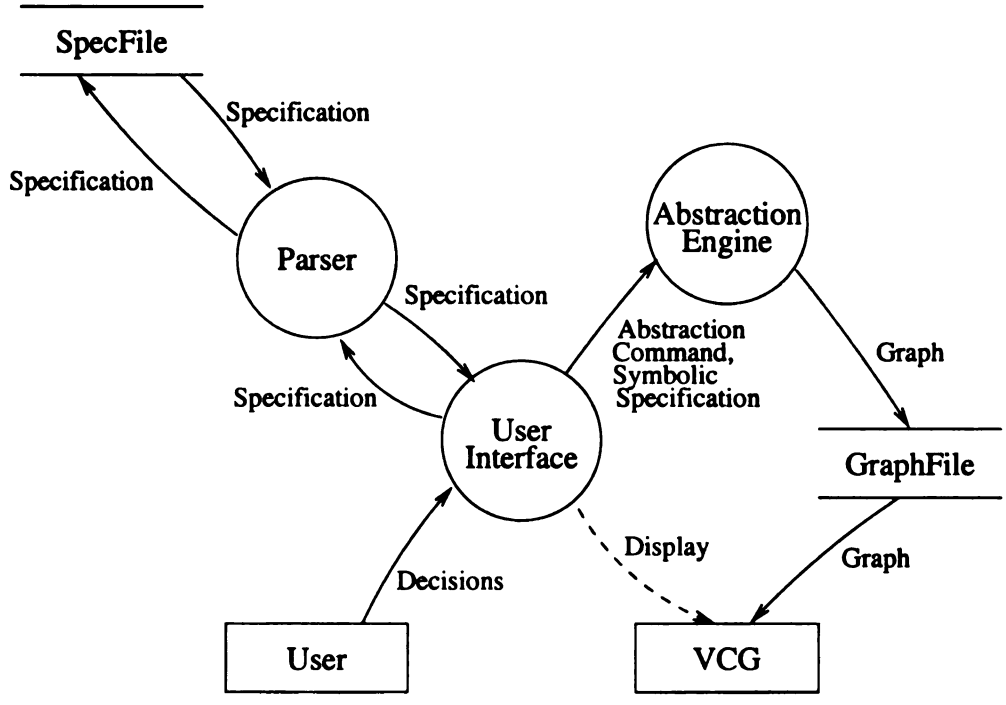


Figure 8.8: Level 1 SPECGEN Model

8.3.2 Implementation

The implementation of the SPECGEN system has two major components. The first component is the *abstraction engine*, which is responsible for deriving symbolic

abstractio

which pro

this sectio

Prolog

The abstra

C routines

and backtr

size (10-1

The P

specificati

using a lil

straightfor

facilitated

Interface

The user in

the direct u

specificatio

facilitates sp

of interest,

all possible

the top port

buttons labe

abstractions from an input specification. The second component is the *user interface*, which provides a mechanism for facilitating user-driven specification generalization. In this section we describe each of these components in detail.

Prolog

The abstraction engine was written using SWI-Prolog [51] interconnected with a number of C routines. The primary motivation for using Prolog was to take advantage its inferencing and backtracking capabilities. The small number of routines (approximately 20) and their size (10-15 lines each), easily justified our choice of language in this case.

The Prolog routines are primarily responsible for deriving partial-orderings of specifications as well as pruning and expanding the orderings based on user input. By using a library of C routines that support integration of C and Prolog, it became a straightforward process to build the system along with a graphical user interface that facilitated visualization and manipulation of the specification generalizations by a user.

Interface

The user interface for SPECGEN includes two different components. The first component is the direct user manipulation component that allows a user to load, manipulate, and analyze specifications using the abstraction engine. Written using Tcl/Tk [48], this component facilitates specification generalization by allowing a user to select specification components of interest, to exclude or *mask* out certain specification components, and to generate all possible abstractions. For instance, Figure 8.9(a) depicts the SPECGEN system with the top portion containing the conjuncts of a conjunctive normal form expression. The buttons labeled "Delete Conjuncts", "Preserve Conjuncts", "Focus", and "Generate All"

allow a u

specificat

been gen

called VC

The sec

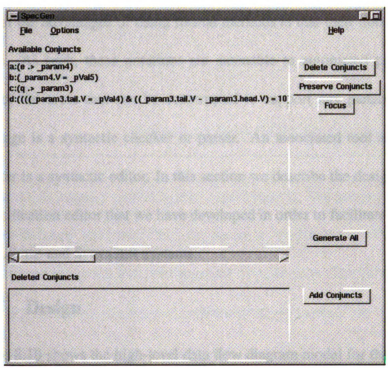
component

the visualiz

issues that

of using VC

allow a user to derive different levels of abstraction with differing levels of detail from a specification. Figure 8.9(b), shows the results of an analysis, where a specification has been generalized using SPECGEN and the resulting partial-ordering visualized using a tool called *VCG*.



(a) SpecGen

(b) Focus Graph

Figure 8.9: SPECGEN Interface and Output

The second component of the user interface is the visualization component. In this component we take advantage of an existing system called *VCG* [46], a system that supports the visualization of graphs. VCG provides many functions for graph layout and placement, issues that are well beyond the scope of our investigations. One of the shortcomings of using VCG, however, is that it lacks a mechanism for providing feedback to external

systems s

we plan o

facility.

8.4 SP

One of th

based. A

One of th

language i

checker is

a specifica

AUTOSPEC

8.4.1 D

Figure 8.10

SPECEDIT

and allows

incorporatio

Figure 8

design of the

The user inte

ways. First,

specifications

systems such as the main SPECGEN system. As such, as part of our future investigations, we plan on extending the functionality of SPECGEN to incorporate an internal visualization facility.

8.4 SPECEDIT

One of the advantages of using formal methods is that their notations are mathematically based. As such, these notations are amenable to automated processing and reasoning. One of the tools that can be constructed to support any particular formal specification language is a syntactic checker or parser. An associated tool along the same lines of a checker is a syntactic editor. In this section we describe the design and implementation of a specification editor that we have developed in order to facilitate user interaction with the AUTOSPEC and SPECGEN systems.

8.4.1 Design

Figure 8.10 shows the high-level data flow diagram model for the SPECEDIT system. The SPECEDIT system interacts primarily with the user to construct or modify specifications, and allows the user to save the specifications to files for later modification or for incorporation into other tools that use first-order logic as an input language.

Figure 8.11 contains the data flow diagram for the SPECEDIT system. The internal design of the SPECEDIT system has two primary components: a *parser* and a *user interface*. The user interface facilitates the construction of a syntactically correct specification in two ways. First, the user interface has a graphical interface that allows users to construct specifications using a point and click method. The user interface also has a text-based

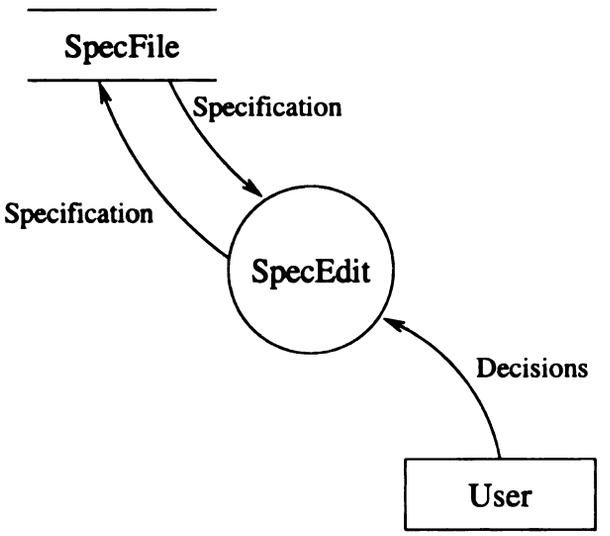


Figure 8.10: Level 0 SPECEDIT Model

interface that allows a user to type in a specification. The parsing component is responsible for two different activities. First, the parser is responsible for checking the syntax of pre-existing specifications that are contained in input files. The parser is also responsible for checking the syntax of user modifications that are made using the text-based interface for the system.

8.4.2 Implementation

The *Spec*ification *Edit*ing system, or SPECEDIT, was developed in order to facilitate specification modification during the analysis phase of the AUTOSPEC system. The main objective in constructing the SPECEDIT system was to provide a way of ensuring syntactic correctness during the modification of a specification. This correctness is ensured in one of two ways: by construction, or by verification. Correctness of construction is facilitated by providing a mechanism whereby a user can click on various syntactic elements and replace them with valid substitutions. For instance, Figure 8.12, shows the conjunctive formula

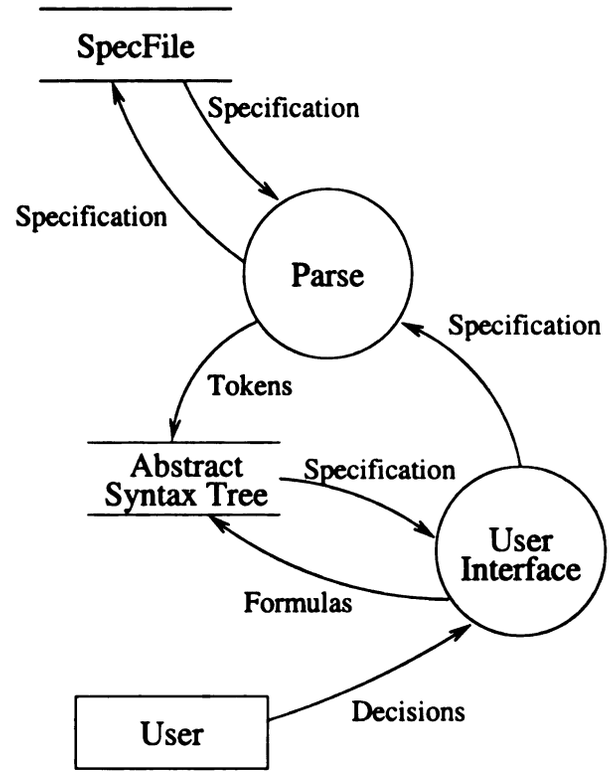


Figure 8.11: Level 1 SPECEDIT Model

"$p(x) \wedge$ Formula". The italicized font for $p(x)$ indicates that the term has been selected using a mouse click. By double-clicking on the "Formula" conjunct in the upper window of SPECEDIT, the user can choose to substitute the conjunct with either an atomic formula, a conjunction, disjunction, implication, or quantification. Since substitutions can only be made with valid substitutions, the final specification is syntactically correct by construction.

The second way of ensuring syntactic correctness that is supported by SPECEDIT is to verify correctness using a syntactic checker or parser. In the lower window of the SPECEDIT interface there is a specification modification window that can be used by a user to type in a desired specification. By clicking on the "OK" button, a user directs the SPECEDIT system to run the syntactic checker on the specification in the lower window. If

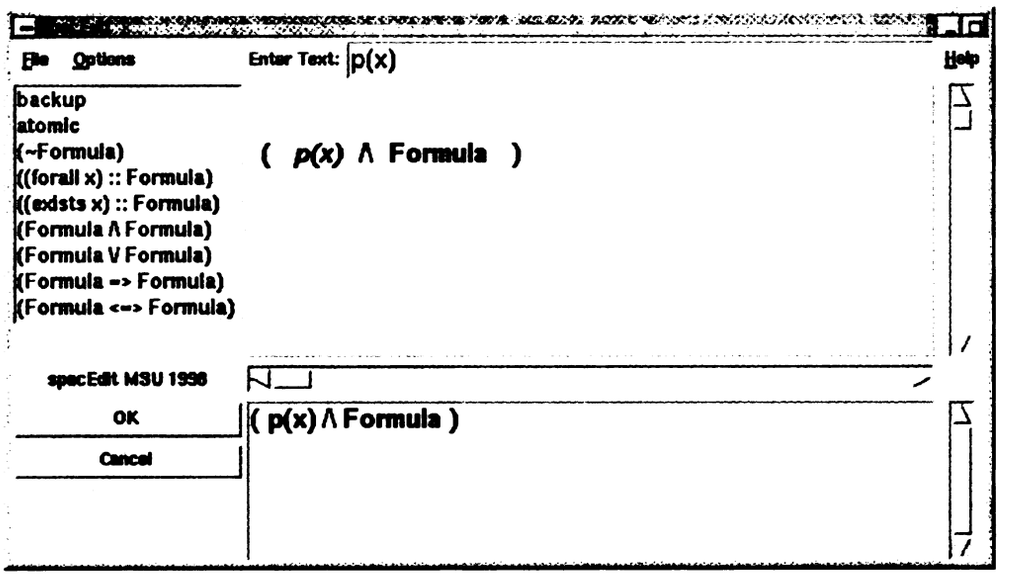


Figure 8.12: SPECEDIT

the specification is syntactically correct, the specification is loaded into the upper window and the user is free to modify the specification in either window.

8.5 Theorem Prover

Many of the interactions between a user and the AUTOSPEC system involves the modification of a specification by a user and the re-introduction of the modified specifications into the current analysis or program annotation. In order to verify that the specification modifications made by a user are logically consistent with the system-generated specifications, we have constructed a simple theorem prover. In this section we describe the design and implementation of the theorem prover TPROVER.

8.5.1 Design

Figure 8.13 depicts the high-level design of the TPROVER system. Except for file interactions and user guidance, the TPROVER system is entirely self-contained. The

TPROVE

be prove

the TPR

Figure

component

consult com

translate firs

uses the info

proof techni

identified in

used to intera

method.

TPROVER system takes as input a source file containing a first-order logic specification to be proved. Using guidance provided by a user for reasoning about quantified expressions, the TPROVER system determines whether or not the specification is valid.

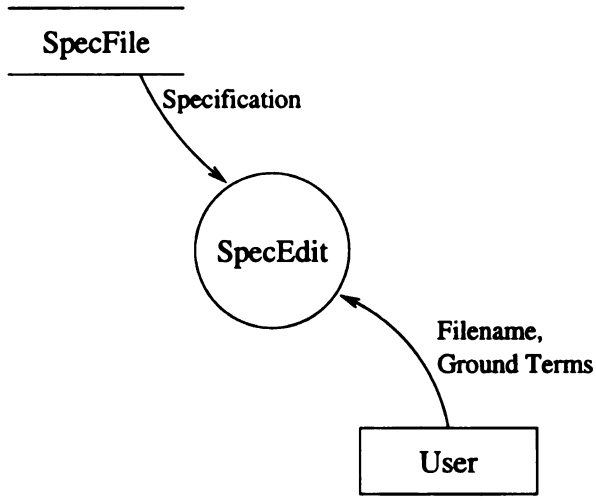


Figure 8.13: Level 0 TPROVER Model

Figure 8.14 shows the internal structure of the TPROVER system. The primary components of the system are: the *parse* component, the *prover* component, and the user *consult* component. The *parse* component is a *Lex* and *Yacc* generated parser that is used to translate first-order logic specifications into an *abstract syntax tree*. The *prover* component uses the information in the abstract syntax tree to generate a proof tree based on the tableau proof technique. For certain proof rules in the tableau method, ground terms must be identified in order to continue processing. In these instances, the *consult* component is used to interact with the user in order to identify an appropriate ground term for the proof method.

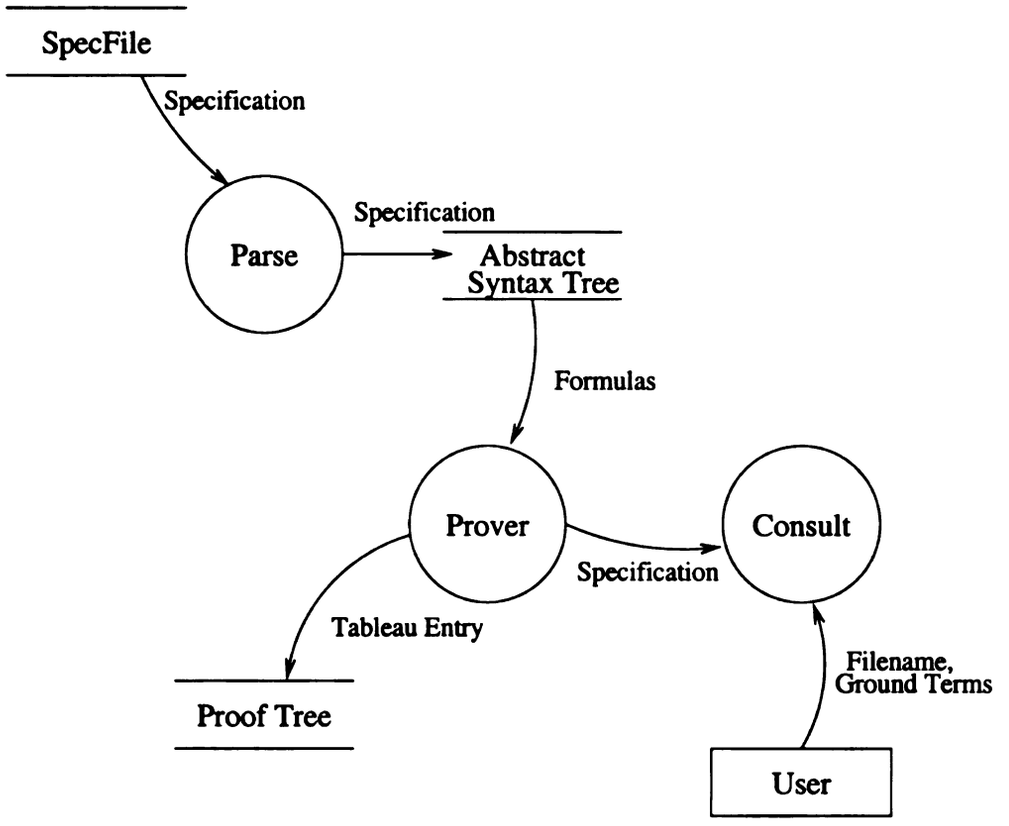


Figure 8.14: Level 1 TPROVER Model

8.5.2 Implementation

The main proof engine in the TPROVER system was constructed using C++. The proof engine construction was facilitated by the existence of the formula class library described in Section 8.6. The graphical user interface was constructed using Tcl/Tk interconnected with C++.

The TPROVER system takes as input the name of a file containing a logical expression. Based on the logical rules described in Section 8.5.1, the TPROVER system will generate a proof tree. For propositional logic and certain expressions of the first order logic, the theorem prover is automatic. For the remaining classes of valid input, user direction is

required

former c

Figu

the main

user. The

provided

the proof

in proof c

each indic

8.6 Fo

Each of th

library that

This *Formu*

according t

Figure 8

"*Formula*" i

NegatedForm

elements of

(i.e., *BinaryA*

The atomi

are aggregatio

required. In the latter case, the TPROVER system acts as a tableau proof editor. In the former case, the TPROVER system acts as a theorem prover.

Figure 8.15 shows the main window of the TPROVER system. The upper sub-window is the main prover window. In this window the proof tree is constructed and displayed to the user. The lower sub-window is the proof information window. In this window the user is provided information about the current proof. Specifically, during times of user interaction, the proof information window contains data about the current entry in the proof tree. To aid in proof comprehension, the vertices in the proof trees are displayed with different colors, each indicating a different state of processing.

8.6 Formula Class Library

Each of the support tools described in this chapter rely heavily upon the use of a class library that we have developed to facilitate the manipulation of first-order logic expressions. This *Formula* class library is a collection of classes that organize logical expressions according to an inductive definition of first-order logic [49].

Figure 8.16 shows the object model for the Formula class library. The superclass "*Formula*" is the base class for the entire library. Each of the subclasses *BinaryFormula*, *NegatedFormula*, *QuantifiedFormula*, and *AtomicFormula* represent the different syntactic elements of first-order logic. In addition, the *AtomicFormula* class has three subclasses (i.e., *BinaryAtomic*, *UnaryAtomic* and *Literal*).

The atomic classes in the *Formula* class library, with the exception of the *Literal* class, are aggregations of *Term* objects. The *Term* objects correspond to terms in first-order logic.

The implem

constants, ar

The *For*

In addition,

construction

means for che

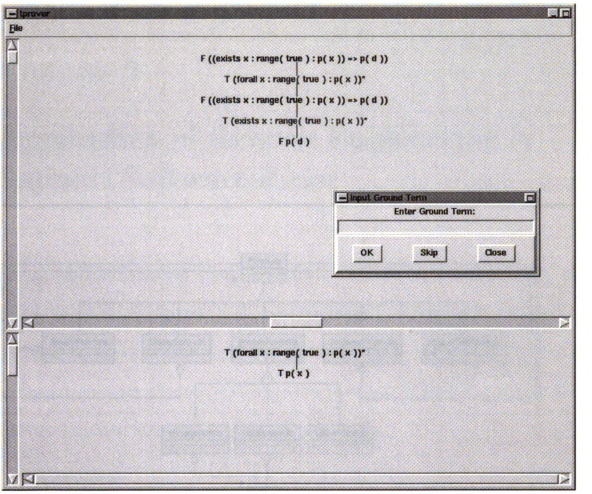


Figure 8.15: Example TPROVER Session

The implementation of the *Term* classes include specific classes to represent variables, constants, and functions.

The *Formula* class library was implemented using the C++ programming language. In addition, a standard formula parser was constructed using the *Lex* and *Yacc* parser construction system. This parser is used by every tool in the AUTOSPEC tool suite as a means for checking the syntax of file and user input.

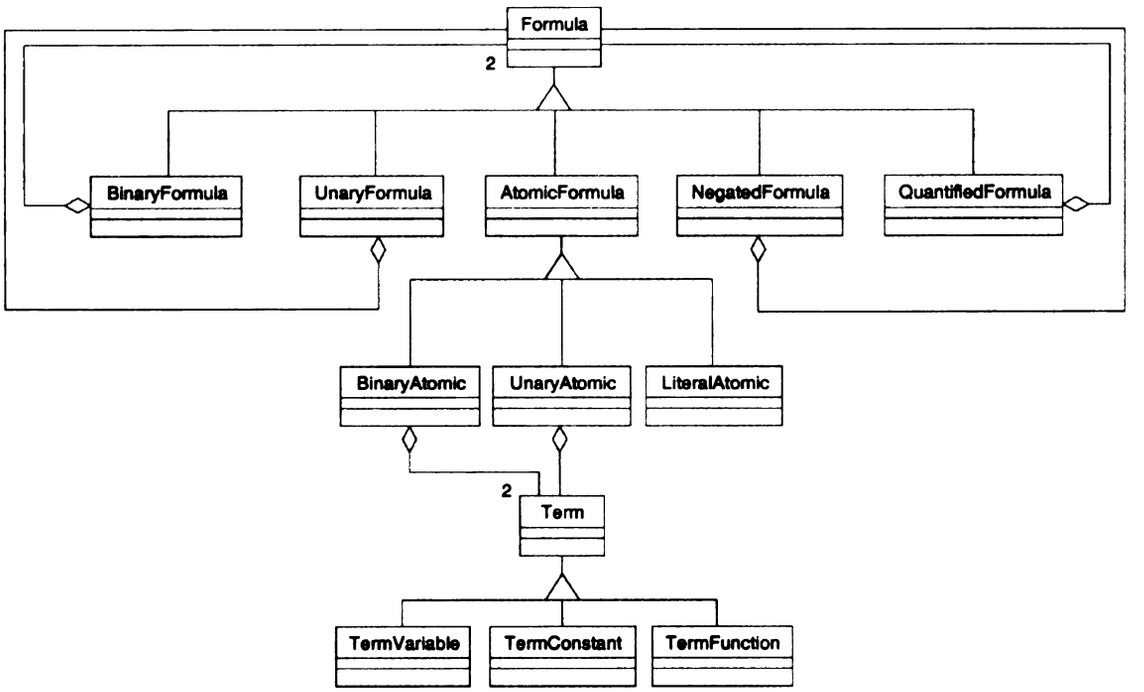


Figure 8.16: OMT model of the *Formula* class library

Cha

App
Sup

One

software

was reus

of reverse

software

9.1 O

Historical

complete

and other

grained ur

same man

technique,

software co

matching c

library to id

Chapter 9

Application of Reverse Engineering to Support Software Reuse

One of the methods that is used to explicitly reduce the effort needed to develop software is to reuse existing code. In the case of the Ariane 5, the software for the Ariane 4 was reused and resulted in catastrophic loss [2]. In this chapter we discuss the application of reverse engineering to the area of software reuse in order to facilitate the construction of software component libraries.

9.1 Overview

Historically, the use of software components-off-the-shelf (COTS) has been limited to complete applications. The introduction of object-oriented programming, design patterns, and other new development techniques have focused on the creation and reuse of finer grained units such as software COTS, but the wide-scale use of such components in the same manner as hardware integrated components has been limited. As a development technique, *software reuse* is a process of constructing a software system using existing software components. Formal approaches to software reuse rely heavily upon specification matching criterion, where a search query using formal specifications is used to search a library to identify components that can be used for reuse purposes. Jeng and Cheng [35, 31]

addresse

Chen an

specifica

that mak

indices is

In thi

softw are

reuse. In

tools to su

a softw are

9.2 A

Many

componen

with reusa

placing the

Foundation

mathematic

repositories

packaging t

for indexing

This cha

with a formal

addressed the use of formal methods and component libraries to support software reuse, and Chen and Cheng [52, 53] investigated the construction of software based on architectural specifications. One of the primary difficulties of using a formal approach for software reuse that makes use of formally specified components is that creation of the formal specification indices is not explicitly addressed.

In this chapter, we present an approach for combining software reverse engineering and software reuse to support populating specification libraries for the purposes of software reuse. In addition, we discuss the results of our preliminary investigations into the use of tools to support an entire process of populating and using a specification library to construct a software application.

9.2 A Software Reverse Engineering and Reuse Framework

Many software reuse approaches depend on the assumption that a library of reusable components is available for use. There are two techniques for populating these libraries with reusable code. The first technique is to construct components with the intention of placing them into code repositories. Examples of these repositories are the Microsoft Foundation Classes [54] as well as libraries for standard problem domains such as mathematics, graphics, and networking. The second technique for populating code repositories is by identifying existing code as potential candidates for reuse, and then packaging that code into a library. In either case, a primary concern is the mechanisms used for indexing, identifying, and retrieving the components from the libraries [31, 19, 20, 32].

This chapter describes how a formal approach to reverse engineering can be integrated with a formal approach to software reuse in order to support after-the-fact construction and

use of re

software

distinct c

by the pr

process c

the linka

is require

library is

Suite.

Within

specificati

engineerin

the search,

use of reusable code libraries. Figure 9.1 gives an overview of the reverse engineering and software reuse framework in the form of a data flow diagram. The diagram shows two distinct components within the framework: the *Reverse Engineering* component, indicated by the process circle labeled "RevEgr Suite", and the *Reuse* component, indicated by the process circle labeled "Reuse Suite". Two integrating factors within this framework provide the linkage between the two components; the *User* and the *Specification Library*. The user is required to direct the reverse engineering and the reuse processes, and the specification library is the common medium and repository between the RevEgr Suite and the Reuse Suite.

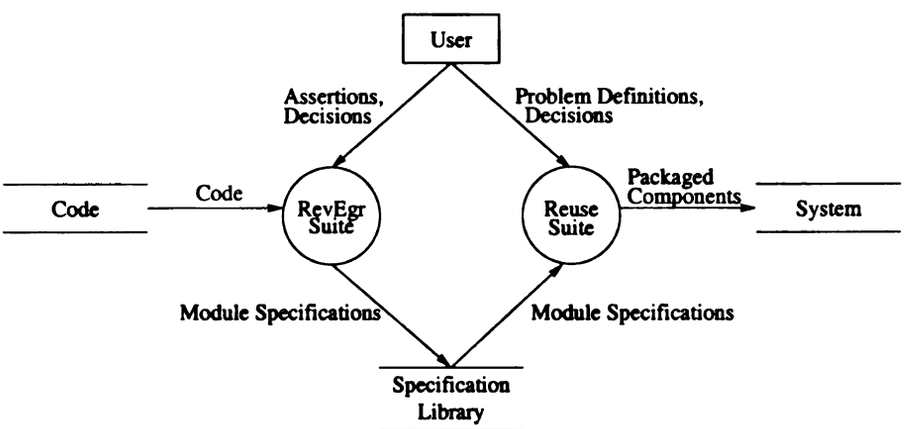


Figure 9.1: The Reverse Engineering and Reuse Framework

Within this framework, a user can analyze source code and construct formal specifications that can be used to index the source for reuse purposes. This reverse engineering and population activity facilitates the reuse part of the framework, namely the search, identification, and packaging of components into new systems. We next discuss

the spec

aspects

9.2.1

Figure

enginee

investig

investig

decision

Within

means for

9.2.2 S

A *software*

56, 57]. So

terms of its

the specific techniques and tools that are used to support the reverse engineering and reuse aspects of the framework, respectively.

9.2.1 Reverse Engineering

Figure 9.2 contains a data flow diagram that depicts our investigations into reverse engineering as a two-stage process, where the AUTOSPEC process bubble represents our investigations with strongest postcondition, and SPECGEN process bubble represents our investigations into abstraction. During the entire process the *User* provides guidance and decisions in order to reduce the complexity of the specifications and abstractions.

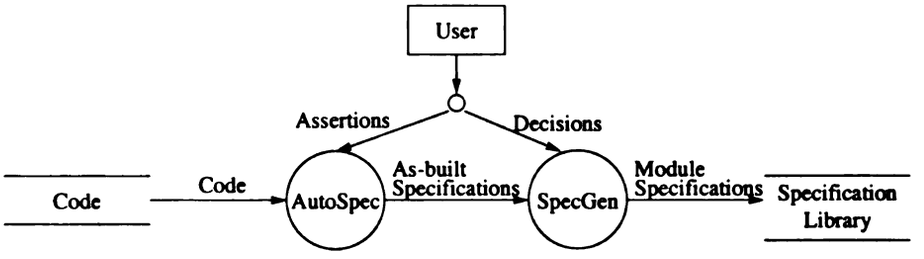


Figure 9.2: Reverse Engineering Component

Within the context of software reuse, our reverse engineering technique provides a means for constructing module specifications using pre- and postconditions.

9.2.2 Software Reuse

A *software architecture* is defined to be a configuration of components and connectors [55, 56, 57]. Software architectures describe the overall organization of a software system in terms of its constituent elements, including computational units and their interrelationships.

Chen a

on the use

a softw are

system. D

describe th

a library o

Using speci

target syste

assumption

order to vali

to populate

Figure 9

Cheng [52,

the *Arch De*

store represe

specification

dissertation.

consists of a

and Integrati

9.3 Exan

In this secti

engineering

Chen and Cheng [52, 53] have developed an approach to software reuse that is based on the use of *software architectures*. The approach involves a three-stage process where a software system is *specified*, components are *selected*, and then *packaged* into the final system. During the specification phase, a software architecture specification is used to describe the target system. Given this specification, a user can select components from a library of components that potentially satisfy the requirements of the target system. Using specification and component matching, components that meet the constraints of the target system are then validated and packaged to form the final target system. One of the assumptions that is made by the approach is that a library of components is available. In order to validate this assumption, we advocate the use of our reverse engineering approach to populate component libraries with specifications of existing program code.

Figure 9.3 depicts the framework for the reuse investigations described by Chen and Cheng [52, 53]. In the diagram, each of the stages described above is represented by the *Arch Design*, *Select/Match*, and *Package* processes. The *Specification Library* data store represents the combination of specifications and associated program code. These specifications can be constructed using the reverse engineering technique described in this dissertation. The *System* data store represents the final output of the reuse framework and consists of a packaged component. To support this framework, Architecture Based Reuse and Integration Environment (ABRIE) system has been developed [53].

9.3 Example

In this section we discuss an example that illustrates the use of the integrated reverse engineering and reuse framework. First, we populate a library, using reverse engineering

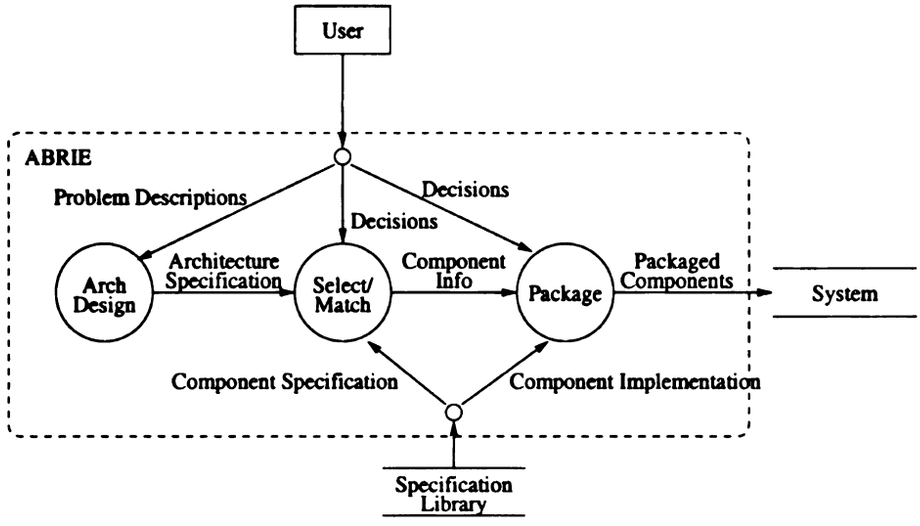


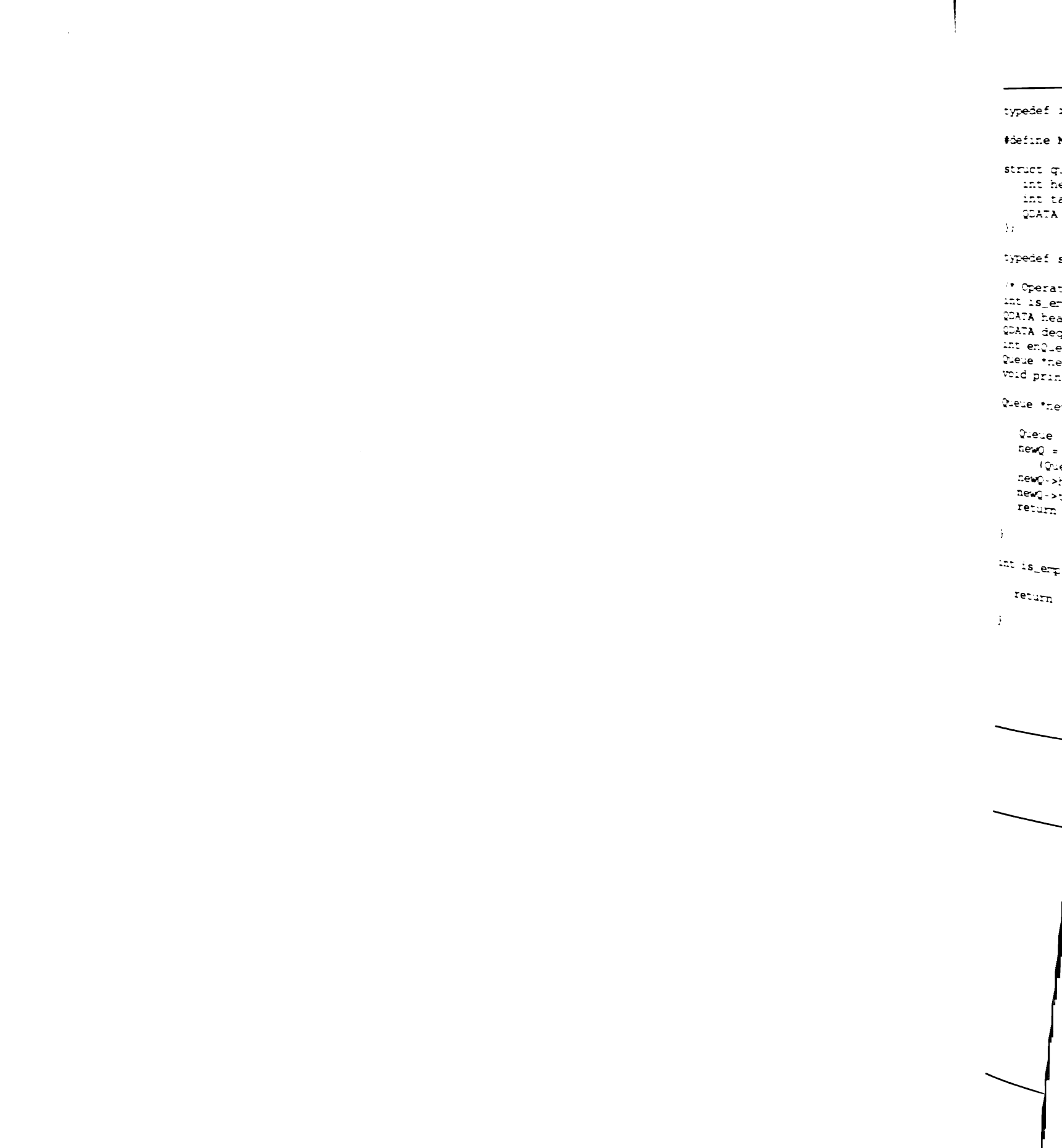
Figure 9.3: Software Reuse Framework

techniques, with component specifications. Then we demonstrate the process of specifying an application and searching the specification and component library for suitable reusable code. Finally, we show the process of packaging components into the final application.

9.3.1 Populating the Library

Figure 9.4 shows the source code for an array implementation for a queue abstract data type. The source code, written using the C programming language, implements a circular queue so that the head and tail of the queue can "wrap" around the lowest and highest indices of the array, as shown in Figure 9.5.

The queue data structure consists of three parts: an index to the front, or *head* of the queue, an index to the end, or *tail* of the queue, and an array that is used to store the elements of the queue. The queue source code contains several functions that correspond to the operations typically associated with queues including `enqueue`, `dequeue`, `new_queue`, `head`, and `is_empty`. The abstract behavior of these operations is as expected, where



```
typedef int QDATA;

#define MAXSIZE 100

struct queue {
   int head;
   int tail;
   QDATA data[MAXSIZE];
};

typedef struct queue Queue;

/* Operations */
int is_empty(const Queue);
QDATA head(const Queue);
QDATA dequeue(Queue *);
int enQueue(Queue *, QDATA *);
Queue *new_queue();
void printQ(Queue);

Queue *new_queue(){

   Queue *newQ;
   newQ =
      (Queue *)malloc(sizeof(Queue));
   newQ->head = 0;
   newQ->tail = 0;
   return newQ;

}

int is_empty(const Queue q){

   return (q.head == q.tail);

}
```

```
int enQueue(Queue *q, QDATA *e){
   int tail;
   int head;

   if ((q->tail - q->head) == MAXSIZE)
   {
      printf("Full\n");
      return 0;
   } else {
      q->data[q->tail % MAXSIZE] = *e;
      q->tail = q->tail + 1;
      return 1;
   }

}

QDATA dequeue(Queue *q){

   int temp;
   if (!is_empty(*q)){
      temp = q->head % MAXSIZE;;
      q->head = (q->head + 1);
      return q->data[temp];
   } else {
      return 0;
   }

}

QDATA head(const Queue q){

   return q.data[q.head % MAXSIZE];

}
```

Figure 9.4: Queue Source Code

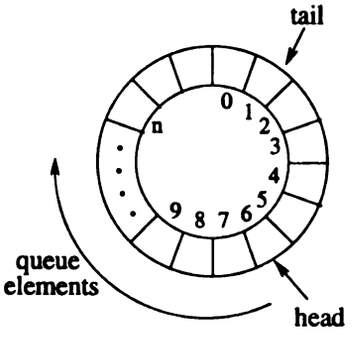


Figure 9.5: Circular Queue Diagram

enqueue

the front o

front of th

As des

populating

of an as-bu

semantics c

source code

the symbols

respectively

pointers and

For the p

of the enQu

procedure as

enQueue pr

to an object

object_para

addition, the

is true. The

difference be

maximum s

failure. The

enqueue adds a new element to the end of the queue, dequeue removes the element at the front of the queue, new_queue creates a new queue, head returns the element at the front of the queue, and is_empty checks to see if the queue contains any elements.

As described in Section 9.2.1, construction of a specification that is suitable for populating a component specification library involves two primary steps: 1) construction of an as-built specification, and 2) derivation of a high-level abstraction. Using the *sp* semantics of the C programming language [58], the complete as-built specifications of the source code in Figure 9.4 can be constructed as shown in Appendix D (Figure D.1), where the symbols '&&' and '||' are used to indicate the logical connectives and ($\wedge$) and or ($\vee$), respectively. In addition, the symbol '.>' is the points-to operator [43] for reasoning about pointers and pointer operations.

For the purposes of illustration, the remainder of this section will focus on the analysis of the enQueue procedure. Figure 9.6 shows the as-built specification of the enqueue procedure as derived by the AUTOSPEC tool. Informally, the as-built specification of the enQueue procedure states that prior to the execution of the procedure, the pointer e points to an object _param4, the value of _param4 is _pVal5, and the pointer q points to an object _param3 such that the tail component of _param3 has the value _pVal4. In addition, the specification states that after execution of the procedure, one of two conditions is true. The first condition describes the behavior when the queue is full in which case the difference between the values _param3.tail.V and _param3.head.V is equal to the maximum size of the queue. Here, the return value of the procedure is 0, indicating a failure. The second condition describes the behavior when there is room to place an item

on the q

increme

```
s
r
(

m

e
(

( (
```

Fi

While

program, t

technique t

the postcon

must first p

form, the e

object -par

the disjunct

on the queue. In this case, the return value of the procedure is 1, the index to the tail is incremented, and the data element is added to the queue data array.

```
spec int enQueue(Queue *q, QDATA *e)
requires
(((e .> _param4) &&
   (_param4.V == _pVal5)) &&
   ((q .> _param3) &&
   (_param3.tail.V == _pVal4)))
modifies
   q (_param3)
ensures
((((((e .> _param4) && (_param4.V == _pVal5)) &&
   ((q .> _param3) && (_param3.tail.V == _pVal4))) &&
   ((_param3.tail.V - _param3.head.V) == MAXSIZE)) &&
   (return.V = 0)) ||
(((((((e .> _param4) && (_param4.V == _pVal5)) &&
   ((q .> _param3) && (_param3.tail.V == _pVal4))) &&
   (!((_pVal4 - _param3.head.V) == MAXSIZE))) &&
   (_param3.data[(_pVal4 % MAXSIZE)].V = _param4.V)) &&
   (_param3.tail.V = (_pVal4 + 1))) &&
   (return.V = 1)))
```

Figure 9.6: Output generated by AUTOSPEC for the enQueue procedure

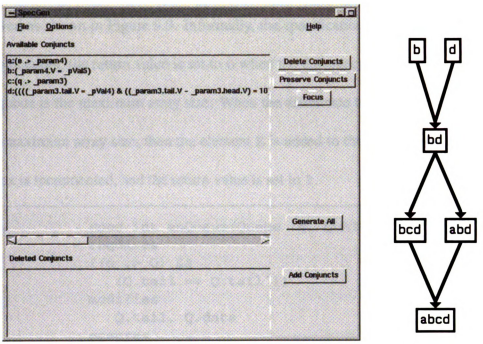
While the specification in Figure 9.6 is accurate with respect to the original source program, the level of detail can inhibit high-level reasoning. As described in Section 6, one technique that can be used to derive an abstraction of an as-built specification is to weaken the postcondition by using the *delete a conjunct* strategy. For the enQueue example, we must first put the **ensures** clause into a conjunctive form, as shown in Figure 9.7. In this form, the enQueue specification has four conjuncts that specify that (a) e points to the object _param4, (b) _param4.V has value _pVal5, (c) q points to _param3, and (d) the disjunctive statement:

```
((((_param3.tail.V = _pVal4) &
   ((_param3.tail.V - _param3.head.V) = MAXSIZE)) &
   (return.V = 0)) ||
(((((as_const9 = _pVal4) &
   (!((_pVal4 - _param3.head.V) = MAXSIZE))) &
   (_param3.data[as_const9 % MAXSIZE].V = _param4.V)) &
   (_param3.tail.V = (as_const9 + 1))) &
   (return.V = 1))).
```

Figure 9.8(a) shows the results generated by the SPECGEN tool when applied to the specification in Figure 9.7. One of the operations that can be performed by the tool is to generate all the possible abstractions of a specification based on preserving one or more of the conjuncts in the specification. The representation of the specifications that are generated by preserving conjuncts is called a *focus graph.* In our example, we are interested in preserving the conjuncts (b) and (d) and deleting conjuncts (a) and (c) due to the independence property stated in Section 6. Figure 9.8(b) shows the focus graph for the example, where the vertex labeled "abcd" indicates that conjuncts (a), (b), (c), and (d) are conjuncted. This vertex corresponds to the original specification in Figure 9.7. The remaining vertices in the graph represent the possible abstractions that are formed by deleting conjunct (a), (c), or both.

```
(e .> _param4) && (_param4.V = _pVal5) && (q .> _param3) &&
((((_param3.tail.V = _pVal4) &
   ((_param3.tail.V - _param3.head.V) = MAXSIZE)) &
   (return.V = 0)) ||
(((((as_const9 = _pVal4) &
   (!((_pVal4 - _param3.head.V) = MAXSIZE))) &
   (_param3.data[as_const9 % MAXSIZE].V = _param4.V)) &
   (_param3.tail.V = (as_const9 + 1))) &
   (return.V = 1)))
```

Figure 9.7: The enQueue ensures clause in a conjunctive form



(a) SpecGen

(b) Focus Graph

Figure 9.8: SPECGEN Interface and Output

Using the information provided by SPECGEN, several transformations of the specification in Figure 9.7 can be performed that simplify and introduce abstraction into the postcondition. First, based on the focus graph in Figure 9.8(b), conjuncts (a) and (c) are deleted due to the independence criteria. After the deletion of conjuncts (a) and (c), we can perform a textual substitution of all references to the `_param3` identifier with a more descriptive symbol, like `Q`. Finally, given that the term `as_const9` has the value `_pVal4` and in the precondition for the specification `_param3.tail.V == _pVal4`, we can replace `as_const9` with the term `Q.tail^`, which represents the pre-value for the tail component of the queue (i.e., the value of the tail component before execution of the

procedure). Given these transformations, the abstraction for the enQueue procedure can be derived as shown in Figure 9.9. Informally, the specification states that after execution of the procedure, the return value is set to 0 when the difference between the head and tail of the queue is the maximum array size. When the difference between the head and tail is not the maximum array size, then the element E is added to the array at the tail index, the tail index is incremented, and the return value is set to 1.

```
spec int enQueue(Queue *q, QDATA E)
requires
((q .> Q) &&
   (Q.tail == Q.tail^))
modifies
   Q.tail, Q.data
ensures
((((Q.tail - Q.head) = MAXSIZE) &&
   (return = 0)) ||
((((!((Q.tail^ - Q.head) = MAXSIZE)) &&
   (Q.data[Q.tail^ % MAXSIZE] = E)) &&
   (Q.tail' = (Q.tail^ + 1))) &&
   (return = 1)))
```

Figure 9.9: The enQueue abstraction

A process similar to the one used for enQueue can be applied to derive abstractions for the remainder of the queue as-built specifications. For the purposes of combining the reverse engineering suite with the reuse suite, the resulting specifications must be translated into the syntax for the ABRIE system. The reason for the differences between the syntax of the AUTOSPEC and ABRIE specification languages is historical. Our initial investigations for deriving specifications for programs focused on the analysis of the Dijkstra guarded command language [6]. As such, a general Larch Interface Language variant [42] was developed. The expansion of the AUTOSPEC tool to support the C programming language

has since prompted a need to modify the tools to generate Larch C (LCL) specifications, an activity that we are currently performing. In contrast, ABRIE was not developed for a specific programming language, but was intended to be tailorable to a given language. As such, Chen and Cheng used a generic procedure-oriented syntax for the Larch Interface Language. However, since the output formats for the AUTOSPEC and SPECGEN systems and the input format for the ABRIE system are all based on the Larch interface language (all contain header information and the **requires**, **modifies** and **ensures** clauses), the actual step for preparing the procedure specifications generated by AUTOSPEC and SPECGEN to the library format for the ABRIE system is straightforward and can be facilitated with automated tools. Appendix D.2 contains the module specification in the ABRIE syntax that was constructed for the example described in this section.

9.3.2 Specifying an Application

In the following discussion, we describe how a solution to the Josephus problem [59] can be specified and assembled from reusable components in ABRIE. In particular, we show how the formal specifications generated by the reverse engineering process can be used to semantically determine the reusability.

The Josephus game can be described as follows: N people, numbered 1 to N, are sitting in a circle. Starting at person 1, a hot potato is passed. After M passes, the person holding the hot potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining persons wins. Given N and M, the Josephus problem is to determine who will win.

Figure 9.10 shows the structure of a solution to the Josephus problem that uses a queue to represent people sitting in a circle. The solution is specified in ABRIE. Component

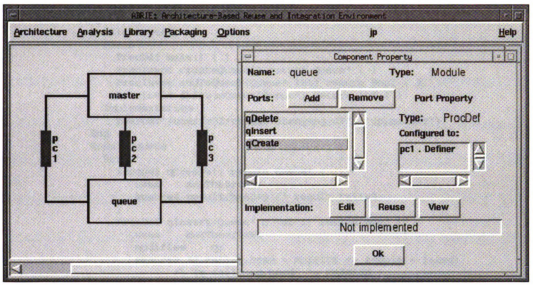


Figure 9.10: Architecture of a solution to Josephus problem

master simulates the game, and calls queue operations provided by component *queue*. The two components are connected through three connectors of procedure calls. As shown in the "Component Property" window of Figure 9.10, component *queue* has three ports, each of which defines and provides a queue operation. Figure 9.11 shows the textual specification of the architecture. As shown in Figure 9.11, component *master* is implemented using a C source file *jp_main.c*. Component *queue* needs to be implemented and will be the focus of the reuse activities. The required behaviors of its ports have been specified. In the next subsection, we discuss how a library component can be selected based on these behavioral specifications to implement the *queue* interface.

```
Architecture jp
   Components
     Module master
       Ports
         ProcDef main() { }
         ProcInvoc createQueue() return Queue* { }
         ProcInvoc addToQueue(Queue*,int) return Bool { }
         ProcInvoc delFromQueue(Queue*) return int { }
       Implementation
         source("/user/r02/chengb/chenyong/JP", "jp_main.c")
     End
     Module queue
       Ports
         ProcDef qCreate() return Queue* {
            uses    auxTheories;
            ensures result.head=0 /\ result.tail=0;
         }
         ProcDef qInsert(Queue* q,int i) return Bool {
            uses    auxTheories;
            modifies    q;
            ensures (q.tail-q.head = MAXSIZE => result = false)
                  /\ (q.tail^ -q.head^ ~= MAXSIZE
                        => (   result = true
                             /\ q.tail' = q.tail^ + 1
                             /\ q.data[mod(q.tail^, MAXSIZE)] = i));
         }
         ProcDef qDelete(Queue* q) return int {
            uses    auxTheories;
            requires    q.head^ ~= q.tail^;
            modifies    q;
            ensures result=q.data[mod(q.head^, MAXSIZE)]
                  /\ q.head'=q.head^+1;
         }
     End
   Connections
     CallProc pc1
       Roles
         Caller -> master . createQueue
         Definer -> queue . qCreate
       End
     CallProc pc2
       Roles
         Caller -> master . addToQueue
         Definer -> queue . qInsert
       End
     CallProc pc3
       Roles
         Caller -> master . delFromQueue
         Definer -> queue . qDelete
       End
End
```

Figure 9.11: Architecture specification

9.3.3 Component Reuse

ABRIE incorporates a library manager for organizing and managing existing components. Components are classified and retrieved based on their interfaces (i.e., types and ports). When implementing an abstract component (interface) in an architecture, a single click on the *reuse* button in the "Component Property" window (see Figure 9.10) triggers ABRIE to search for the current library (which is loaded through the library manager). All components of the same type as the query interface will be presented to the user. Based on their specifications, the user selects one candidate for further evaluation. Figure 9.12 shows the scenario of matching the library component *circqueue* for satisfying interface *queue.*

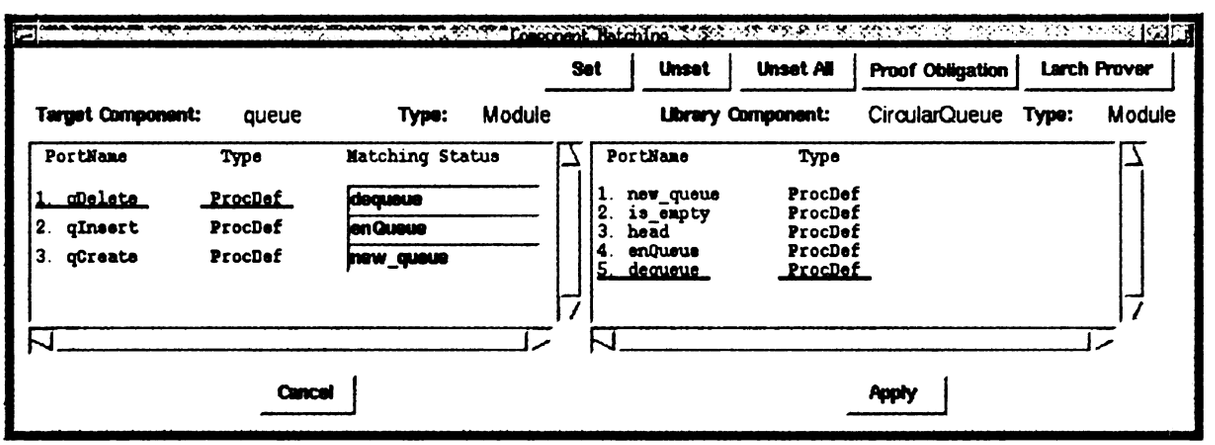


Figure 9.12: Component Matching

In order to determine the reusability of *circqueue*, we need to establish a mapping from the ports of the target component *queue* to those of *circqueue* so that each operation specified in *queue* can be implemented by a corresponding operation in *circqueue*. As shown in Figure 9.12, we conjecture that *qDelete* can be matched (implemented) by

dequeue, *qInsert* by *enQueue*, and *qCreate* by *new_queue*. Given a match, specification-based proof obligations may be generated to validate the matching.

As exhibited in the mapping between *circqueue* and *queue*, naming conflicts, such as *qDelete* of *queue* and *dequeue* of *circqueue*, may exist between a query specification and the reused component. Resolving these mismatches is one of the tasks of the packaging process. Figure 9.13 shows the wrappers generated by ABRIE for resolving the naming conflicts between *circqueue* and *queue*, where wrappers are generated based on the established port mappings. The packaging process also checks connectors and generates

```
// _circqueue_wrapper.cc
// Generated by ABRIE for wrapping component circqueue

#include "auxTypes.h"

extern int dequeue(Queue *);
int qDelete(Queue *q) {
   return dequeue(q);
}

extern Bool enQueue(Queue *, int);
Bool qInsert(Queue *q, int i) {
   return enQueue(q,i);
}

extern Queue* new_queue();
Queue *qCreate() {
  return new_queue();
}
```

Figure 9.13: Wrappers generated by ABRIE for resolving naming conflicts

their implementation, as well as a system construction file (a *makefile*) that describes how an executable system is produced.

Chapter 10

Related Work

10.1 Introduction

Software maintenance has long been recognized as one of the most costly phases in software projects [9]. A software system is termed a *legacy system* if that system has a long maintenance history. Many techniques have been suggested for the maintenance of legacy software as is clearly indicated by the number of surveys that have been used to catalog these techniques [60, 61, 62]. Due to the increasing visibility of the *Year 2000 Problem* (e.g., Y2K) [1] many more tools have been suggested and subsequently catalogued [63].

Given the large number of tools, identifying one appropriate for the goals of an individual organization can be difficult. Currently, the information gathered on software maintenance tools focuses on surface characteristics for the given tools. That is, the gathered information typically lists the languages that are supported and the type of by-products (i.e., artifacts) generated from analyzing the input software with the particular

[1] The Y2K problem refers to the potential failure of systems due to the use of a two-digit encoding for the year field in software systems.

tool. For instance, Bellay and Gall [62] describe capabilities related to usability, parsing speed, type of by-product, editing facilities, and report generation. Based on feedback and interaction with industry, it is our claim that in addition to these surveys, it is also useful to have an analysis of the actual by-products (i.e., function reports, call graphs, data flow diagrams) in order to gain an understanding of the value of the by-products.

In this chapter, we describe a framework for analyzing software reverse engineering and design recovery tools and techniques. Within this framework we provide a context by which software reverse engineering and design recovery tools can be classified according to the underlying approach used to analyze software, and we define several criteria for comparing and contrasting tools according to the semantic quality of their by-products.

10.2 Background

Chapter 2 described background information about the area of software maintenance and reverse engineering. In the context of software maintenance, we define a *structural* abstraction to be a description of a software system that is based on the syntactic properties of a programming language. For example, encapsulation of a sequence of programming statements into a module is a structural abstraction. We contrast structural abstraction with the term *functional* abstraction. A functional abstraction is a description of a software system that is based on the semantics of a program. That is, a functional abstraction describes program behavior. For instance, if a sequence of statements is grouped into a module, the high-level description of the function of that module is a functional abstraction. Recent work in the area of reverse engineering has focused on both the derivation of structural and functional abstractions from program code.

10.2.1 Evaluation of Software Technology

Brown and Wallnau [64] describe a framework for evaluating software technology that is based on two primary goals: (1) understanding how the evaluated technology differs from other technologies, and (2) understanding how these differences address the needs of specific usage contexts. In order to achieve these goals, Brown and Wallnau suggest a three phase process for technology evaluation. These phases are:

1. Descriptive modeling
2. Experiment design, and
3. Experiment evaluation

The *descriptive modeling* phase is used to create a context for candidate technologies. A descriptive model is a description of the assumptions concerning features and their relationship to usage contexts [64]. Two types of descriptive models are the *technology genealogy* and the *problem domain habitat*. The technology genealogy describes the historical context for a given technology, and a problem habitat describes how the features of a given technology can be used as well as what the benefits of their use will be.

The *experiment design* phase involves three primary activities: (1) comparitive feature analysis, (2) hypothesis formulation, and (3) experiment design. In this phase, the goals are to develop a set of hypotheses about the added value of a technology that can be established by experiments, and to identify the experiments that are used to substantiate or refute the hypotheses [64].

The final phase, *experiment evaluation*, involves performing experiments to confirm or refute the hypotheses. Brown and Wallnau identify a few different classes of experiments that can be useful in evaluating hypotheses [64]. These experiment categories include:

- *Model problems*: narrowly defined problems that are easily addressed by the candidate technologies. Model problems allow alternative technologies to be directly compared.
- *Compatibility studies*: experiments that study how well candidate technologies operate when combined
- *Demonstrator studies*: full scale trial applications of a technology
- *Synthetic benchmarks*: standard contrived problems that can be used to evaluate the differences between candidate technologies

In this chapter, we analyze several reverse engineering support tools using an assessment technique that is similar to the Brown and Wallnau "Technology Delta Framework". Specifically, we present the results of the descriptive modeling phase, where we describe a hierarchical genealogy of reverse engineering techniques. In addition, we define several semantic dimensions that are used to qualitatively evaluate some representative reverse engineering support tools, an activity that corresponds to constructing a reference model in the experiment design phase in the technology delta framework. Next, we describe the informal and formal techniques for reverse engineering, respectively. Finally, we provides a comparative analysis of all of the techniques.

10.2.2 Previous Surveys

The Air Force Software Technology Support Center (STSC) published a two volume report that compiles information about hundreds of tools that are available for reengineering purposes [61]. While the report lists many tools, the descriptions of the tools are often limited to high-level properties, such as supported languages and vendor contact information. Similar surveys by Zvegintzov [60, 63] also collect descriptions of Reengineering and Y2K tools with the same shortcomings of the STSC report. However,

the Y2K survey [63] does classify the tools based on their intended capabilities. For instance, some of the categories used to group tools are based on whether the tools support activities such as *inventory analysis* (e.g., identification of the executable software inventory), *recovering source from object* (e.g., analysis of binaries in the case that source is not available) , and *conversion* (e.g., identification of code and data structures that require modification).

A recent survey by Bellay and Gall [62] compares four reverse engineering tools using several criteria that are used to analyze the effectiveness of the input parsers, the by-product representations, the editing and browsing capabilities, and the general usability of the tools. While the survey in this chapter does provide a more in-depth view of tools when compared to the previous surveys, it focuses primarily on tool properties as opposed to the characteristics and qualities of the tool by-products.

Our approach to surveying and analyzing software reverse engineering and design recovery tools and techniques is meant to provide a framework for assessing the quality and usability of the by-products. As such, this survey provides a complementary approach to the assessment and comparison of tools such as those contained in the surveys described above.

10.3 Taxonomy

In order to classify automated and semi-automated reverse engineering techniques, we have developed the hierarchical taxonomy shown in Figure 10.1. At the highest level, the techniques can be subdivided into two classes: *informal* and *formal*. *Informal* approaches are those methods that rely on pattern matching and user-driven clustering based on

the syntactic structure of code. The pattern matching and clustering approaches are considered informal because either the representations that are constructed are informal, or the consistency between the design specification and the source code cannot be rigorously verified. Nonetheless, the informal techniques do provide a means for deriving abstractions about the general function of a program. The *formal* approaches are those techniques that are based on using some type of formal analytical method for deriving a specification from source code. The basis for the formal techniques are grounded in mathematical logic so that each step can be formally verified. The primary difference between the informal techniques and the formal techniques is the use of formal specification languages that have well-defined syntax and semantics. In addition, the formal techniques have associated inference rules that can be used to construct proofs in order to rigorously verify the correctness of each step of the reverse engineering process. The remainder of the section describes each of the categories shown in Figure 10.1.

Various reverse engineering and program understanding techniques can be evaluated and classified using the taxonomy in Figure 10.1. The utility of classifying tools using this taxonomy is that it provides a means for determining the current trends in supporting reverse engineering and design recovery, and aids in identifying the areas that require further investigation. In Sections 10.5.1 and 10.5.2 we describe several representative techniques and classify them accordingly. As a notational convention, a numerical tag follows the name of each approach to indicate the classification of the technique within the taxonomy. For instance, a tool "**foo**" might fall in class "**2**" to indicate that the technique is an informal, plan-based, commercial tool. The annotations at the leaves of the classification hierarchy in Figure 10.1 associate each tag to a location in the classification.

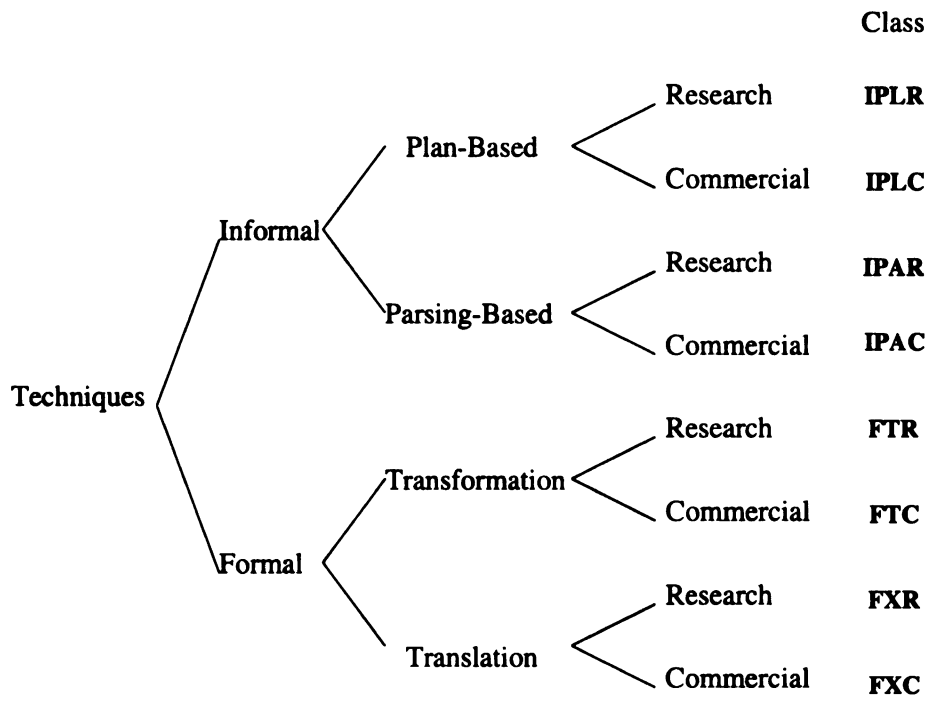


Figure 10.1: A Taxonomy of Reverse Engineering Techniques

10.3.1 Informal Techniques

In the context of reverse engineering and program understanding, a technique is classified as *informal* if the methods used to recover designs from source code is based on pattern matching or analysis of syntactic structures as opposed to semantic structures. The informal techniques can be decomposed into two additional subcategories: *plan-based* and *parsing-based.* The *plan-based* techniques rely primarily on using pattern matching to identify clichés or *plans* within source code and have been a major focus in both research and commercial organizations. A program plan is a description of a computational unit contained within a program where a computational unit performs some abstract function [44]. A program plan can be *localized* or *de-localized* in the sense that the code recognized as satisfying the plan can be located in contiguous (localized) or non-contiguous

(de-localized) sequences of code [65]. To date, most plan-based approaches have been developed by research organizations [36, 66, 67], although some industrial adoption of this approach is occurring [68].

A *parsing-based* approach is one in which a program is analyzed using the properties of the syntactic structure of a language. In general, the parsing-based approach is used to construct a high-level structural abstraction of the source code. These abstractions typically come in the form of data flow diagrams or some other graphical representation of the design. A significant number of commercial tools use a parsing-based technique for supporting reverse engineering [69, 70], and research organizations continue to investigate the use of advanced parsing-based approaches [37, 71].

10.3.2 Formal Techniques

Formal methods for software development are analytical techniques for assuring, by construction, that a derived specification is correct with respect to some other specification. A reverse engineering technique is formal if the steps of the method have a formal mathematical basis. When applied to reverse engineering, a formal method takes as input a source program (a low-level specification) and derives a formal specification. In the formal context, reverse engineering techniques can be subdivided into two categories: techniques that use a knowledge-base or *transformation* library to derive formal specifications from code, and techniques that use derivation or *translation* to derive formal specifications from code.

A *transformation* is a means for changing a specification from one form to another while preserving the semantics of the specification. In the context of programs, a *program*

transformation is a means for changing a program from one form to another while preserving the semantics of the program. Each program transformation is typically used to change a group of programming statements at a time, where the group is determined by the author of the particular transformation.

Transformation is contrasted with *translation*, where a translation is also a means for changing a program from one form to another while preserving semantics but at an atomic level of granularity. The primary difference between transformation and translation is the degree to which high-level knowledge about a problem domain or programming language is incorporated into the transformation or translation rules. In the case of transformation, the rules typically involve transforming aggregations of programming statements into simpler, equivalent sequences of statements (as is the case in restructuring transformations) or concise formal specifications. In many cases, a large library of transformations is required to capture the many different possible code constructions. Translation, in contrast, involves much simpler rules that are based on single atomic statements such as assignments, conditionals, and iteratives, thus requiring fewer rules. A program compiler can be considered a translator since each program statement is translated into an equivalent binary form. In the context of program reverse engineering, a translation technique is one that translates a program into an equivalent formal specification.

Research into the use of formal methods for reverse engineering has addressed both the use of transformation [72, 73] and translation [6]. Industrial adoption of such techniques has begun but is limited [39, 74].

10.4 Semantic Dimensions

A *by-product* is an artifact that is constructed by a reverse engineering tool as a result of analyzing program code. One way to evaluate the by-products of a tool or technique is to simply list the formats and representations that are produced by a particular tool. For instance, one tool might produce reports about the data structure formats, as well as visual representations such as callgraphs and data flow diagrams. While this knowledge about a tool is extremely helpful, it is of equal importance to understand the nature of these by-products and to evaluate a tool based on this information. In order to analyze the value of the by-products of the various tools, we define four semantic dimensions: *distance*, *accuracy*, *precision*, and *traceability*. These measures enable a software maintainer to evaluate a tool based on the level of importance placed on the consistency between an abstract representation as compared to a given implementation.

10.4.1 Semantic Distance

The *semantic distance* describes the number of levels of abstraction that separate an input and an output of a particular technique. The semantic distance is a relative distance, since no absolute measure of abstractness can reasonably be developed. Instead, a subjective measure based on the level of algorithmic detail must be considered.

As a rule of thumb, the greater the semantic distance, the more abstract the by-product. Suppose, for instance, we translated source code from FORTRAN to C. Since there is no difference in the level of abstraction between the two representations, the semantic distance is low or non-existent. On the other hand, if we reverse engineer source code from C into a data-flow diagram representation, the semantic distance is higher. At the extreme, we

might reverse engineer source code from C into a description of the concept of the program; a transformation that would result in the highest degree of semantic distance.

A concept related to the semantic distance is the *inter-step distance* that measures the semantic distance between each intermediate step of a technique. For example, if a reverse engineering technique is comprised of three steps, where each step produces a representation that is more abstract than the previous step, the semantic distance that separates each step in the technique is the inter-step distance.

10.4.2 Semantic Accuracy

The *semantic accuracy* describes the level of confidence that a specification is correct with respect to the input (i.e., source code). Many of the by-products derived from an analysis of syntactic information rarely have a low semantic accuracy. That is, the information that is recovered from the source code is accurate with a high degree of confidence. In contrast, the techniques that derive by-products based on semantic information may not be as accurate. For instance, those techniques that are based on the plan abstraction approach may rely on the assumption that plans are not interleaved [65], and, as such, may ignore the effect of *cancellation* or composition in their description of a particular sequence of software. That is, two or more program plans may be identified in the same sequence of code, but their combined effects may not be well-understood and thus, the accuracy of the design abstraction may be reduced.

One of the factors that impacts the semantic accuracy of a given technique is the number of analysis stages and the inter-step distances between the stages. This is due to the fact that

an abstraction omits certain information that is embedded in a lower-level representation. The composition of these stages results in an increased potential for a loss of accuracy.

10.4.3 Semantic Precision

Semantic precision describes the level of detail of a specification and the degree that the specification is formal. Figure 10.2 depicts a precision hierarchy for a set of tool by-products. A formal specification is the most precise given the well-defined syntax and semantics associated with this form of description. The least precise by-product is natural language due to its potential for ambiguity. A more precise specification is apt to be more amenable to automated analytical processing while a less precise specification better suited for discussions between programmers.

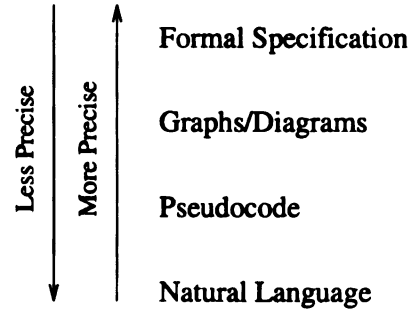


Figure 10.2: Precision Hierarchy

10.4.4 Semantic Traceability

Semantic traceability describes the degree that a specification can be used to reconstruct an equivalent program. Semantic traceability highly depends on the semantic accuracy and semantic precision of the end by-product since the accuracy will contribute to the

degree to which the original program and the new program correspond semantically, and the precision will contribute to the degree that the representation is free of ambiguity. Furthermore, the semantic precision will impact the amount of semantic information that can be used to construct the new program. For instance, a formal specification might have a high degree of semantic traceability while a graphical design has a low degree of semantic traceability. The ability of a programmer to reproduce a working system varies greatly between a formal specification and a graphical design since semantic information is contained in the formal specification while, in general, only syntactic information is contained in a graphical design.

10.4.5 Discussion

Ideally, a design derived from program code has a balance between all of the semantic dimensions. A large semantic distance may produce a more abstract specification but if that specification lacks accuracy and precision, there is a low degree of confidence that the specification captures the actual functionality of the source code. On the other hand, a specification with a high degree of precision and traceability that lacks a reasonably large semantic distance may be difficult to understand. In the end, it is the software maintenance programmer that must weigh the goals of a project against the relative advantages and disadvantages offered by the by-products of the various techniques in order to make the appropriate decision for a particular project or organization.

10.5 A Representative Tools Survey

In this section, we survey a number of tools that can be used to support reverse engineering and design recovery. Several of the tools are commercially available systems while a number of other tools are systems that are currently being developed as part of research activities. There are far too many reverse engineering and design recovery systems available to enumerate them all in this context. Instead, we have selected a number of representative tools that exhibit several of the properties discussed earlier. The survey is decomposed into two broad categories that correspond to the taxonomy in Section 10.3: informal and formal-based techniques. Within each category, additional criteria are used to describe the various techniques and tools.

10.5.1 Informal Techniques

This section describes different informal approaches that have been applied to reverse engineering and design recovery. As a convention, the name of each approach is followed by a numerical tag corresponding to the classification hierarchy given in Figure 10.1.

Plan-Based Approaches (IPLR and IPLC Classes)

A program cliché is a commonly used sequence of code that performs some specific function. The term *plan* is used to refer to the knowledge representation for describing clichés [44]. Typically a plan contains an *event* section for describing the conditions that must exist in order for an instance of the cliché to be present.

An example plan is shown in Figure 10.3 [36]. The plan has two sections: an event section (*consists of*), and a constraints section (*such that*). The event section is as described

above. The constraints section provides additional conditions for evaluating the events within a specific context of the program. Essentially, the plan in Figure 10.3 states that if the events of `reader`, `eof-test`, and `repeater` are recognized with respect to the constraints, then the concept `READ-PROCESS-ALL-VALUES` has been recognized.

```
READ-PROCESS-ALL-VALUES( value: ?value, PROCESS: ?body)

consists of
   reader:      FETCH-INPUT-VALUE(RESULT: ?inp-res, VALUES: ?value)
   eof-test:    NOT-EQUAL(OP1: ?inp-res, OP2: EOF)
   repeater:    LOOP(TEST: ?test, BODY: ?body)

such that
   contained-in(reader, ?test)
   contained-in(eof-test, ?test)
   data-dep(eof-test, reader, ?inp-res)
```

Figure 10.3: Example Plan

Many of the plan-based techniques use a three step process that involves parsing the program, identifying the events, and matching the events with the plans contained in a plan library. The variations, to be described in the next few sections, are often related to the methods used to construct the plans to either make the techniques faster, more efficient, or convey some other view of the design of a system.

Cobol/SRE (IPLR, IPLC). The Cobol System Renovation Environment, or Cobol/SRE, is a toolset developed by the Andersen Consulting Center for Strategic Technology Research [66, 75]. The approach used in Cobol/SRE is based on the use of program plans with the intent of identifying abstract concepts in code. These abstract concepts can be classified as *programming concepts*, *architectural concepts*, and *domain concepts* [75]. While programming concepts can be automatically determined by parsing, architectural

and domain concepts require knowledge about architecture and domains to be encoded into a plan library.

In the Cobol/SRE approach, concepts in programs are recognized by decomposing programs into their equivalent abstract syntax tree and performing syntactic pattern matching against the cliché library. Higher level concepts are recognized via a method of substitution whereby constraints (or sub-concepts) for a high-level concept are instantiated with previously recognized lower level concepts. In addition, Cobol/SRE has features that support flow analysis, slicing (a decomposition technique that extracts program statements that are relevant to the scope of a particular computation from a program [76]), complexity analysis, and anomaly detection.

Cobol/SRE allows users to determine which program segments to analyze and which rules to use. Criteria for selection of segments include selection using condition-based slicing [76], forward slicing [76], and ripple-effect analysis [76]. Upon completion of the recognition process, a window lists which concepts were recognized. Other information, including which rules were used in recognizing the concepts, is available for user analysis purposes. The toolset has been applied to a commercial production control system consisting of approximately 8000 COBOL modules.

COBOL/SRE Summary.

Name	COBOL/SRE
Class	Informal-Plan-Research/Commercial
By-Products	Code decomposed into commented segments
Language	COBOL
Operating System	Unix

DECODE (IPLR). DECODE uses a plan-based approach to provide an environment for supporting the cooperative understanding of programs via the construction of an object-oriented design from COBOL code [36]. The program understanding system is used to recognize as much of the program as possible with the programmer filling in where DECODE fails. Based on the COBOL/SRE approach, DECODE uses three major components to support program understanding: an automated program recognizer (APU), a knowledge base for storing information about a given program, and a design notebook for allowing a user to edit retrieved designs as well as construct queries for answering questions about the designs.

The DECODE technique has three primary steps: an automated understanding step, a user-driven, machine-aided understanding step, and a query step. In the automated understanding step, the APU is used to identify the existence of both low-level (*incremental*) and design level or *design-oriented* concepts in a system. To support this activity, plans are extended to have links to high-level conceptual design elements. In addition, special associations such as specializations and implications are allowed.

In cases where the APU is only able to understand parts of a system, DECODE aids the programmer in understanding the remainder by use of a structured notebook. This activity works by allowing a programmer to browse through the code and the initial design. Once the user recognizes new design concepts, those concepts can be added to the design and the code can be linked to the new design element. Once an appropriate design has been constructed, queries about the design and program can be made. DECODE supports queries about the function of certain sections of code, the location of code corresponding to the design, and the status of the design (e.g., has the design been completed?).

The graphical user interface of DECODE consists of many elements including a code browser and a design-editor, which provides a graphical depiction of the extracted design.

DECODE Summary.

Name	DECODE
Class	Informal-Plan-Research
By-Products	Graphical Design Representation
Language	C, COBOL
Operating System	Unix

LANTeRN (IPLR). The "Loop ANalysis Tool for Recognizing Natural concepts" or LANTeRN is an approach that uses a multi-step process to construct predicate logic annotations for loops [67]. The analysis process translates and normalizes loop programs into forms that are amenable to matching various components of loops. A knowledge base or plan library is used to identify stereotypical loop *events*, where events are in the form of *basic events* and *augmentation events*.

A basic event (BE) is a fragment of a loop that forms the control aspect of a loop. These are typically made up of *conditions*, *enumerations*, and *initializations*, where a condition is a clause of the loop guarding condition, the enumeration is the segment of code that ensures that data flows into the condition, and the initializations are the statements responsible for the initializations of the variables into the loop condition. Augmented events (AE) make up the remaining components of the loop body. The AE's are subdivided into two subcategories: the *body* and the *initialization*. The initialization is the set of statements used to initialize the variables contained in the loop body, while the body is all other statements in the loop not associated to data flow into the loop conditions.

Upon analysis of the events contained within a loop, a pattern matching library is used to characterize the loop. When a match of a rule antecedent occurs, the corresponding formal specification is constructed by using the consequent of the fired rule. These consequents have information such as *preconditions*, *postconditions*, and *invariants*.

The structure of the plan library is based on a classification of stereotypical loops. This classification is based on the characterization of the structural forms of loop conditions (i.e., single condition vs. multiple condition), the loop bodies, and the variables used to determine the loop conditions. LANTeRN uses these characterizations in order to identify the appropriate rules to apply a loop program. In order to facilitate efficiency, many rules are abstracted and generalized into a hierarchy of plans.

The approach taken by the LANTeRN system moves in the direction of making plan-based approaches more formal in that the final product of the loop analysis activity is the construction of a formal specification. However, while the activity produces a formal specification, there is no formal basis for the verification that the specification of the plan matches the true semantics of a loop that is being analyzed.

LANTeRN Summary.

Name	LANTeRN
Class	Informal-Plan-Research
By-Products	Formal specification (axiomatic)
Language	Pascal
Operating System	NA

Xinotech (IPLC). Xinotech is an interactive environment that is based on the use of a *meta-language* called the *Xinotech Meta-Language*, or *XML* [68]. The Xinotech approach

is based on the plan, or cliche approach described earlier. The process used for analyzing software is comprised of steps that are used to translate source code into the intermediate XML representation, apply concept or plan recognition techniques, and then represent the design of the system using multiple textual and graphical views. In addition, the Xinotech system provides support for several different methodologies for analysis and code transformations.

The Xinotech approach is characterized primarily by the extensive use of meta-languages. These meta-languages are general purpose languages that make the Xinotech tool applicable to many different source languages through translation into the XML language. Plan-based transformations in Xinotech are specified using the *Xinotech Plan Abstraction Meta-Language* (XPAL) and allow Xinotech to make transformations of code into higher level abstractions. Features of Xinotech include the ability to support many views or models of a particular system.

Xinotech Summary.

Name	Xinotech
Class	Informal-Plan-Commercial
By-Products	Textual and graphical designs
Language	Several
Operating System	Unix, Windows

Parsing-Based Approaches (IPAR and IPAC Classes)

Many commercial tools have been developed to address software maintenance issues. In general, these tools are typically parser-based. The by-products of these techniques generally consist of call graphs and flow diagrams although many other representations

exist and are produced by these systems. This section describes three commercially available systems and two systems developed by research organizations.

Refine (IPAC). The *Software Refinery* and *Refine Language Tools* by Reasoning Systems have been the basis for many reverse and re-engineering tools [77]. By supporting such features as user extensions, *Refine*-based tools have grown in popularity.

The Refine Language tools support reverse and re-engineering efforts for programs written in various programming languages including Ada, C, COBOL, and FORTRAN. Features of the Refine tools include interactive source code browsing, generation of various reports such as structure charts and identifier (i.e., variable) definitions. A major feature of the Refine tools is the open architecture that allows users to tailor Refine to specific language dialects.

Refine Language tools, when combined with the Software Refinery, provide an environment for producing reverse and re-engineering applications through the use of a three part process of loading code into an object database, selecting code and operations to be applied to the code, and executing the operations. The Software Refinery is divided into three tools that support this process and allow users to construct custom re-engineering tools: DIALECT, REFINE, and INTERVISTA. These tools are used to support parsing, symbolic computation, and user interface construction, respectively.

The Refine-based tools have been used in a number of well-documented instances for building reverse and re-engineering applications [78]. In addition, the Refine-based tools have been used to support many research-oriented activities [75].

Refine Summary.

Name	Refine
Class	Informal-Parsing-Commercial
By-Products	Graphical views
Language	Several
Operating System	Unix

McCabe Visual Reengineering Toolset (IPAC). The McCabe Visual Reengineering Toolset (VRT) provides a graphical environment for supporting code analysis [70]. The VRT tools combine metric, complexity, and static information to aid in many reengineering tasks.

The key feature of the McCabe VRT is the production of graphical views of a program including structures charts that are combined with information about various complexity measures of a system (e.g., cyclomatic). In addition, McCabe VRT supports the identification and elimination of redundant and dead code. Other features include testing aids for determining logic and data complexity tests.

One of the strongest characteristics of the McCabe VRT is the number of languages (over 15) that are supported, including Ada, COBOL, C, C++, and ASM370. These tools operate on numerous platforms and operating systems.

McCabe VRT Summary.

Name	McCabe VRT
Class	Informal-Parsing-Commercial
By-Products	Text and Graphical views
Language	Several
Operating System	Unix, Windows

Imagix 4D (IPAC) Imagix 4D is a graphical tool for supporting program understanding through the use of multiple views of a system [69]. The main features include a 3 dimensional view of the software structure on an *xyz*-axis and supports hypertext browsing of source code.

Imagix 4D uses a static syntactic analysis technique of various software sources including code and makefiles to build a database of information about a subject system. Structure charts, control flow, data usage, and inheritance information is used to aid the user in the analysis process, and support for multiple views enables a user to analyze the system based on data types, file dependencies, and function calls. Imagix 4D allows a user to automatically construct documents from information gathered during analysis. Imagix 4D supports C and C++ source code and runs on Sun workstations.

Imagix 4D Summary.

Name	Imagix 4D
Class	Informal-Parsing-Commercial
By-Products	Text and Graphical views
Language	C, C++
Operating System	Unix

Rigi (IPAR). Rigi is a parsing-based tool that focuses on constructing structural abstractions by facilitating the management of the complexity of a graph derived from source code [37]. Rigi uses a three step process to support program understanding. The first step, *parsing*, constructs a representation suitable for proceeding to the second step, *graph construction and visualization.* The initial graph (e.g., a call graph) can be passed through filters that allow a user to select the subsystems of interest. The final step, an interactive and

iterative one, allows a user to reduce the complexity of the graphs by collapsing vertices in the graph into functional groups, and supports the hierarchical browsing of the graph.

The underlying approach in Rigi for automatically constructing subsystems and functional groups is a bottom-up technique. The strength of Rigi is the use of composition operations based on well-established software engineering concepts such as coupling and cohesion that aid in the construction of graphical specifications that depict either the calling hierarchy of a system or some other view of that hierarchy, such as subsystems and abstract data types. Rigi has been applied to a number of real projects including applications from IBM, NASA, and a commercially available system called Doctor's Practice Management System [79].

Rigi Summary.

Name	Rigi
Class	Informal-Parsing-Research
By-Products	Graphically-oriented design
Language	C, C++, COBOL
Operating System	Unix, Windows, Linux

Reflexion Models (IPAR). A *Software Reflexion Model* is a model that is used to represent differences between an engineer's high-level model and a corresponding low-level model of the original source code [71]. This approach consists of three major steps:

1. High-level model definition
2. Source (low-level) model extraction from the source
3. Definition of a declarative mapping between the high-level model and low-level model
4. Computation of a reflexion model

Steps 1 and 3 are performed by the software maintenance programmer while automated tools can be used to perform steps 2 and 4. The by-products of this approach consist of high-level, low-level, and reflexion models. The representation used in these models depends entirely upon the software maintenance programmer as well as the tools used to construct the low-level source model. A reflexion model is represented by a graph that closely resembles a high-level model provided by a user. The primary modules of the high-level model are retained and arcs between the modules (typically represented by rectangles or circles) indicate whether or not flows between the modules are consistent (or inconsistent) with the user-defined mapping.

The primary value of the reflexion models is the capability to communicate to the software maintenance programmer the differences between the perceived structure of the system (i.e., the high-level model) and the actual structure of the system (i.e., the source model). A tool to support this approach, called RMTool, has been used to analyze C and C++ source code, but the tool (and approach) is not limited to these languages. The size of the programs analyzed range in size with the largest being an industrial system with over a million lines of code [80].

RMTool Summary.

Name	RMTool
Class	Informal-Parsing-Research
By-Products	Graphically-oriented design, Reflexion model
Language	Primarily C. Easily retargetted.
Operating System	Unix, Windows NT

10.5.2 Formal Techniques

This section describes the different formal approaches that have been applied to reverse engineering and design recovery.

Transformation (FTR and FTC Classes)

Program transformations have been used primarily for forward engineering and the development of programs [81, 82, 83, 40]. A program transformation is a semantic preserving operation where a part of a program is replaced with a semantically equivalent construct. When applied in forward engineering, a transformation may replace a high-level specification with an implementation of the specification. In reverse engineering, transformations are generally aimed at replacing sequences of code with semantically equivalent formal specifications.

In general, the theoretical foundations of transformations are based on proving the equivalence of the components of a transformation. For instance, if a construct α is to be replaced with some other construct β, the black box behavior of α and β must be proven to be equivalent with respect to the initial and final states. Transformations can be at the same level (as is typical with a restructuring transformation), refinements (commonly found in program synthesis), or abstractions (which are appropriate for reverse engineering).

The main difference between a transformation and a program plan is that the transformations are semantically preserving, meaning that a part being replaced during a transformation is provably equivalent to the part it is being replaced by. A program plan, on the other hand, is a knowledge representation and recognition rule from which claims about correctness cannot be formally verified.

Maintainer's Assistant (FTR). The Maintainer's Assistant [84] is a reverse engineering environment used for reverse engineering program code into formal specifications using semantics-preserving transformations. The primary feature of the Maintainer's Assistant is the use of a formally defined *wide spectrum language*. A wide spectrum language is a multi-purpose language that combines low-level programming constructs such as assignment, alternation, and iteration with high-level formal specification constructs. In the context of the wide spectrum language **wsl**, several transformations have been developed for supporting software maintenance activities [84].

An example transformation, called a *loop inversion* [84], is as follows. Assuming that the statements S_1 and S_2 have no **exits**, a code sequence "**do** S_1**;** S_2 **od**" can be inverted to "S_1**; do** S_2**;** S_1 **od**". The library of semantic preserving transformations used by the Maintainer's Assistant plays a major role in the reverse engineering process, where the system keeps track of information about the applicability of a particular transformation. The program transformation process is a user-driven activity where a programmer browses through code with a graphical interface and chooses when to apply transformations. The final by-product of the Maintainer's Assistant is a formal specification written in the **wsl** language [85]. As such, the specification can consist of several statements expressed at different levels of abstraction. As such, the specification may retain the sequential style of the original program.

The Maintainer's Assistant toolset was initially developed to support the IBM370 Assembler language and a subset of BASIC and has been applied to portions of the IBM CICS product. In addition, the system has been expanded to support the reverse engineering of concurrent programs [86].

Maintainer's Assistant Summary.

Name	Maintainer's Assistant
Class	Formal-Transformational-Research
By-Products	First-order logic (WSL) specification
Language	WSL, IBM370 Assembler
Operating System	Unix

Design Maintenance System (FTC). Baxter and Mehlich [39] describe an approach to reverse engineering that is based on the idea that reverse engineering consists of the "backwards" application of program transformations. In order to construct a library of transformations, the approach records the transformations that are used to instantiate program plans in a forward transformation system. The approach advocates *design maintenance* as the primary means for maintaining a system, thus avoiding the need to continually reverse engineer a system over its lifetime.

While the primary emphasis in this approach is the use of a transformational engine, plan recognition technology is used extensively as a means for retrieving "clues" about various aspects of the input code [39]. That is, program plans are used to guide the transformation process. The approach, supported by a domain based transformation system called the *Design Maintenance System* or (DMS) has been used to analyze source code written in Motorola 6809 assembler code.

DMS Summary.

Name	Design Maintenance System
Class	Formal-Transformational-Commercial
By-Products	NA
Language	NA
Operating System	NA

REDO (FTR). The REDO Project [73] produced tools for reverse engineering COBOL program code into Z and Z^{++} specifications. The technique involves a three step process where:

- COBOL programs are translated into an intermediate language called UNIFORM,
- Functional abstractions are derived from UNIFORM code, and
- Simplifying transformations are applied to the functional abstractions and objects are derived by combining functions.

Therefore, the REDO approach can be considered to be a hybrid between translation and transformation techniques although the primary reverse engineering activity is transformational.

In order to derive high-level abstractions from the UNIFORM code, a data-flow analysis is performed in order to identify data variables and functions associated to various data structures. The technique also attempts to find logically connected pieces of single entry and single exit code as a means for identifying abstract functional units. Transformations are then applied to these abstractions in order to derive object-oriented specifications using Z^{++} [73].

REDO Summary.

Name	REDO
Class	Formal-Transformation/Translation-Research
By-Products	Z^{++} Specification
Language	COBOL
Operating System	Unix

Translation (FXR and FXC Classes)

Formal *translation* is the process of deriving semantically equivalent representations of atomic programming constructs using the formal semantics of a language. The primary difference between a translation and a transformation is the level of granularity of the translation. A translation occurs at the atomic level and is associated directly to programming constructs, whereas a transformation may involve longer sequences. As a result, translation is more accurate and traceable (i.e., reproducible). However, translation often produces by-products with a smaller semantic distance between code and specification, resulting in a representation that contains an implementation bias.

Peritus Software Services (FXC). Peritus Software Services is an organization that specializes in the support of software evolution activities. Many of their techniques focus on the application of *weakest precondition* for logical code analysis. The *weakest precondition* predicate transformer $wp(S, R)$ is defined as the set of all states in which the statement S can begin execution and terminate with postcondition R true, meaning that given S and R, if the computation of S begins in state $wp(S, R)$, then the program S *will* halt with condition R true.

The Pertitus Code Analyzer (PCA) is a tool that has been developed to support the Peritus approach to logical code analysis [74]. The approach used by the PCA system is a three step process. First, the input source code is translated into the Peritus Intermediate Language (PIL). Second, the PIL program is analyzed using several static analysis techniques including slicing. In addition functions are highlighted and identified for further processing. Finally, the code is analyzed using logical analysis techniques based

on the use of *wp*. The logical analysis step is the primary reverse engineering and design recovery activity, where the analysis of the source is decomposed into four parts including the analysis of (1) terminating, non-iterating code, (2) terminating, iterating code, (3) non-terminating execution of independently terminating programs, and (4) multi-tasking code.

PCA Summary.

Name	Peritus Code Analyzer
Class	Formal-Translation-Commercial
By-Products	First-order logic specifications
Language	COBOL, C, RPG, PL/I
Operating System	NA

AUTOSPEC (FXC). The AUTOSPEC suite of tools use a formal translation-based approach to derive formal specifications. For a complete description of the AUTOSPEC tools, please refer to Chapter 8.

AUTOSPEC Summary.

Name	AUTOSPEC
Class	Formal-Translation-Research
By-Products	Formal Specification, Graphically-oriented diagram
Language	Dijkstra, C
Operating System	Unix, Linux

10.6 Comparison

In this section we evaluate the different approaches by comparing them based on surface or *informational* criteria as well as the semantic dimensions of the tool by-products.

10.6.1 Comparison Criteria

The criteria to be used in comparing the different approaches are subdivided into two groups: *informational* and *evaluational*. The informational criterion are a high-level list of surface characteristics, such as source language, platform, and technique. These criterion serve to provide a quick glance index to the reader and a means for quickly finding more information about a system if so desired. The evaluational criteria are a list of detailed characteristics that allow a user to evaluate the differences between the respective approaches. These characteristics include extensibility, high-level abstractions, formal specifications, metrics, standard diagrams, precision level, and traceability level.

10.6.2 Informational Criteria

Informational criteria provide a quantitative means for measuring each of the tools described in this chapter. That is, each of the criteria can be used as a feature "checkbox" for a tool. In this paper we use a small set of informational criteria consisting of *Languages*, *Platforms*, and *Techniques*. Bellay and Gall [62] list several other criterion of this type. The language criteria indicate the languages supported by a particular tool. The languages that the various tools support range from *C*, *C++*, *COBOL*, *ADA*, and *FORTRAN*. Platform criteria are used to indicate on which hardware platforms the tools can execute. The platforms include support for *PC*, *Sun*, *IBM RS6000*, *HP*, and *Macintosh*. The approach (e.g., informal/formal, plan/parsing, etc.) is also used to further classify each technique. The survey by Bellay and Gall covers many more characteristics that are informational in nature [62].

10.6.3 Evaluational Criteria

The evaluational criteria provide a more in-depth means for categorizing different tools. These criteria provide a means for differentiating tools according to the by-products. The by-products include structure charts, flow diagrams, data dictionaries, metrics, complexity measures, and formal specifications. Another type of evaluational criteria is the *Open Interface* characteristic which indicates whether a tool has an application programming interface (API) to allow users to build applications. In addition, we compare the characteristics of the by-products by indicating whether the tool produces structural or functional abstractions. In addition, the evaluational criterion describe the by-products using the four semantic dimensions described in Section 10.4 (i.e., distance, accuracy, precision, and traceability).

10.6.4 A note about by-products

Tool *by-products* are the artifacts generated by tools as a result of program analysis. Using the informational and evaluational criteria, different inferences can be made about the value of a tool with respect to the by-products. For instance, a formal specification is a form of by-product that has the properties of being precise, and in general, traceable. However, formal specifications are not generally perceived to be user-friendly (that is, they may require some specific background education). Additionally, structure charts are user-friendly, precise, and traceable but lack high-level abstraction. In the remainder of this section we evaluate the primary by-products of each tool. We also provide an evaluation of the by-products along each of the semantic dimensions described in Section 10.4. Inferences about usability, productivity, etc. are all dependent on the final end users.

10.6.5 Informational Comparison

In this section we compare a number of tools based on the informational criteria listed above. In addition to evaluating the tools described earlier, we also include the Logiscope [87], Ensemble [88], and PAT [89] toolsets in the comparison. An index of tools is provided in Table 10.1. Tables 10.2 and 10.3 summarize the tools based on the informational criteria.

Commercial Tools			Research Tools		
SR	=	Software Refinery	PA	=	PAT
VR	=	McCabe VRT	CS	=	COBOL/SRE
4D	=	Imagix 4D	DE	=	DECODE
XI	=	Xinotech Research	LT	=	LANTRN
LS	=	Logiscope	MA	=	Maintainer's Assistant
EN	=	Ensemble Software	RE	=	REDO Toolset
			RI	=	Rigi
			AS	=	AutoSpec

Table 10.1: Tool Index

Table 10.2 compares commercially available tools using the informational criteria. This table shows that C and COBOL are the most widely supported languages among commercial tools and that the McCabe VRT tool supports the largest number of languages. Among platforms, Sun is the most widely supported, although in this comparison we make no distinction between the Solaris and SunOS Operating Systems. Again, the McCabe VRT tool supports the largest number of platforms. Among the techniques used, parsing-based is the most popular. Of note is the fact that the Software Refinery supports the use of transformations although the built-in tools do not use formal transformation as an analysis

technique. Finally, of all the commercial tools, only the Xinotech tool uses a plan-based approach.

		SR	VR	4D	XI	LS	EN
Languages	C	●	●	●	●	●	
	C++		●	●		●	
	COBOL	●	●		●	●	●
	ADA	●	●		●	●	
	FORTRAN	●	●		●	●	
	Other		●		●	●	
Platform	PC		●				●
	Sun	●	●	●	●	●	
	IBM RS6000		●		●		
	HP	●	●				
	Macintosh						
	Other		●		●		
Technique	Plan-Based				●		
	Parsing-Based	●	●	●		●	●
	Transformation	●*					
	Translation						

Table 10.2: Comparison of Commercial Tools by informational criterion

Table 10.3 compares research tools using the informational criterion. This table shows that, like the commercial tools, C and COBOL are the most widely supported languages. Of the research tools, Rigi supports the largest number of languages (COBOL, C, C++). "Other" languages are also more widely supported than FORTRAN and ADA due to the fact that most research tools use source languages that resemble production languages with the caveat that translation to and from production languages from the research languages is

theoretically possible. Among platforms, Sun is supported the most, with the Rigi system supporting the largest number of platforms (Unix, Windows). The approaches used by the research tools are divided mainly into two groups: those approaches that use plan-based techniques, and those approaches that use some formal technique. Only the Rigi system uses a parsing-based technique.

		PA	CS	DE	LT	MA	RE	RI	AS
Languages	C			●				●	●
	C++							●	
	COBOL		●	●			●	●	
	ADA								
	FORTRAN								
	Other	●			●	●			
Platform	PC							●	
	Sun		●	●		●	●	●	●
	IBM RS6000							●	
	HP								
	Macintosh				●				
	Other								
Technique	Plan-Based	●	●	●	●				
	Parsing-Based							●	
	Transformation					●			
	Translation								●

Table 10.3: Comparison of Research Tools by informational criterion

Parsing-based techniques are the most widely used technique among the commercial tools, which reflects the fact that the parsing techniques are more mature. The research tools focus on the use of plans or formal methods, although the plan-based technique has

been adopted by the commercial tool offered by Xinotech. A possible conjecture is that the plan-based approach is becoming more mature and is beginning to be adopted by industry.

10.6.6 Evaluational Comparison

Evaluational criteria provide a more qualitative means for comparing the various approaches. Tables 10.4 and 10.5 summarize the by-products produced by each tool, grouped by commercial tools and research tools, respectively. Tables 10.6 and 10.7 summarize the characteristics of the by-products using the criterion described in Section 10.6.3. Again, these tables are grouped by commercial and research tools, respectively.

	SR	VR	4D	XI	LS	EN
Structure Charts	●	●	●		●	●
Flow Diagrams		●			●	●
Data Dictionaries	●	●	●			●
Metrics		●			●	●
Complexity Measures		●			●	●
Formal Specifications						
Other		●	●		●	
Open Interface	●			●		

Table 10.4: Comparison of Commercial Tools by By-products

Table 10.4 shows the by-products of the various commercial tools. Among commercial tools, creation of structure charts is the most widely supported activity and the McCabe VRT and the Ensemble tools create the largest number of by-products. The Software

Refinery and Xinotech tools provide support for user-defined applications via their programmer interfaces. Of all the commercial tools, none support the construction of formal specifications, and only the Xinotech tool creates functional abstractions in the form of recognized program plans.

	PA	CS	DE	LT	MA	RE	RI	AS
Structure Charts							●	●
Flow Diagrams							●	
Data Dictionaries								
Metrics					●		●	
Complexity Measures								
Formal Specifications				●	●	●		●
Other	●	●	●					
Open Interface							●	

Table 10.5: Comparison of Research Tools by By-products

Table 10.5 shows the by-products of the various research tools. Most research approaches focus on the creation of either formal specifications or some other functional abstraction with only the Rigi tool supporting the creation of structural by-products and abstractions.

Overall, the main difference between the commercial and the research tools is the nature of the by-products. That is, the research by-products focus on creating functional abstractions whereas the commercial by-products focus on generating structural abstractions, as shown in Tables 10.6 and 10.7. Specifically, Table 10.6 shows that only the Xinotech tool produces functional abstractions while Table 10.7 shows that

only the Rigi tool produces structural abstractions. Tables 10.6 and 10.7 also show the difference between commercial and research tools with respect to the semantic dimensions (i.e., distance, accuracy, precision, and traceability) of the by-products. In the table, "H" indicates high, "M" indicates medium, and "L" indicates low so that an H in the distance row for a tool means that the by-products have a high semantic distance. The commercial by-products tend to have a low semantic distance but are very accurate in their representations. On the other hand, the research tools have a high degree of semantic distance but the accuracy tends to suffer. A few of the research tools also focus on higher precision but few do well in terms of traceability and accuracy.

		SR	VR	4D	XI	LS	EN
Structural	As-built	●	●	●		●	●
	Abstraction						
Functional	As-built						
	Abstraction				●		
Semantic Dimensions	Distance	L	L	L	H	L	L
	Accuracy	H	H	H	M	H	H
	Precision	L	L	L	L	L	L
	Traceability	L	L	L	L	L	L

Table 10.6: Comparison of Commercial Tools by evaluational criterion

		PA	CS	DE	LT	MA	RE	RI	AS
Structural	As-built							●	●
	Abstraction							●	
Functional	As-built								●
	Abstraction	●	●	●	●	●	●		
Semantic Dimensions	Distance	H	H	H	H	M	M	H	L
	Accuracy	M	M	M	M	M	M	M	H
	Precision	L	L	L	H	H	H	L	H
	Traceability	L	L	L	L	M	M	L	H

Table 10.7: Comparison of Research Tools by evaluational criterion

Chapter 11

Case Study

Several of the examples that we have presented throughout this dissertation have been self-contained entities that demonstrated a particular aspect of formal analysis. In this chapter we present a case study that applies all of the methods for reverse engineering described in this dissertation to a software system used by the NASA Jet Propulsion Laboratory.

11.1 Overview

In this section we provide an overview of the case study system and outline the objectives of the analysis.

11.1.1 System Overview

The Command Subsystem provides access and facilitates the command and control of spacecraft via a user interface. The system supports the control of multiple spacecraft and provides real-time feedback about the status of radiating commands at each operational point during the uploading of commands to the spacecraft [90]. In addition, the Command subsystem supports command file reformatting and direct access to project databases.

The overall Command process is a six-step sequence as follows [91]:

1. A user accesses the Command Subsystem and prepares a mnemonic command file.
2. The system translates the mnemonic file to binary format.
3. The binary file is converted into a format required by the Deep Space Network (DSN) for radiation (i.e., transmission).
4. The Command file is transferred to the DSN for radiation to the spacecraft.
5. The user can then control and monitor the command file from the workstation.
6. Exit.

The *command translation* module of the command subsystem is responsible for two of the items in the sequence, namely items 2 and 3. Figure 11.1, taken from the Multimission Ground Data System User's Guide for Workstation End Users [91] provides a flowchart of the Command process. The overall size of the command translation subsystem is approximately five thousand lines of code while the overall command system is approximately fifty thousand lines of code. The command translation system has an interesting history that motivates the analysis of the software. The system was originally developed to support the control of a specific set of spacecraft. Every time a new mission is developed (for example, the 1997 Cassini mission to Saturn), the software is updated to handle the translation of spacecraft specific mnemonics. Given the constant change associated with the system, the analysis of the command translation software justifies its study using reverse engineering techniques.

11.1.2 Analysis Objectives

In this chapter, we analyze the command translation subsystem in order to demonstrate the use of a combined informal and formal technique for reverse engineering. The primary

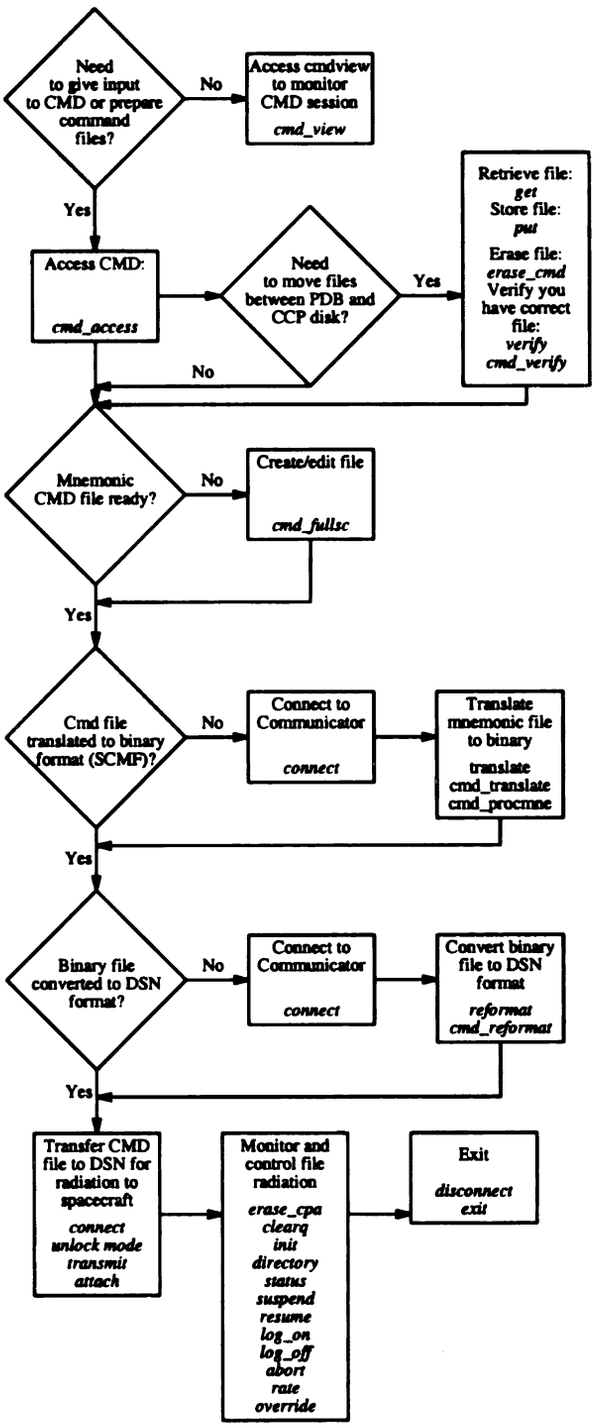


Figure 11.1: Steps for Preparing, Transferring, and Radiating a Command file

objective of the case study is to illustrate how a formal method can be used along with informal methods to derive information about the functionality of a software system.

The application of formal methods to large systems has yet to be effectively demonstrated. However, formal methods have been shown to yield the highest payoff when applied to systems that are critical in nature. In the area of reverse engineering, the highest payoff for formal methods occurs not when applied to an entire system, but rather when applied to a *critical* part of a system. In this Chapter, we apply our technique to a portion of the command translation system that is responsible for processing user mnemonics (messages). The failure of the translation system can have several impacts including erroneous messages being transmitted to spacecraft. Our objective is to investigate various properties of the system such as termination and translation failure.

11.2 Project-Specific Process

Reengineering projects often have project-specific process models that are used to direct the re-analysis and re-development of software [92]. For this case study we used a process that involved the following steps:

1. High-level informal analysis
2. Low-level informal analysis
3. Formal analysis

This process is identical in every respect to the framework described in Section 7. At the macroscopic level this process is not project-specific. However, at the microscopic level this process has many elements that are specific to the project. For instance, in this case study, the high-level informal analysis was facilitated by the existence of documentation that was written by the original developers of the software. In many other projects, the existence of documents as a resource for high-level analysis can not be assumed.

One of the assumptions that was made concerning the existing documents was that constant modifications to the software were not reflected in the documentation. Due to this lack of document maintenance, it was assumed that only a certain level of detail from the documents could be determined to be reliable.

The low-level informal analysis was based on the construction of source models (i.e., call graphs) in order to recover structural information about the system. Using some standard visualization tools, the source models were used to determine potential *points of failure*. In this context, we use the phrase *point of failure* to mean those parts of the source model where there is a large difference between the in-degree and out-degree of a vertex in the graph. The reason that these vertices of the graph are interesting is that the high out-degree means that a procedure invokes many other procedures and thus is potentially a critical procedure. High in-degree vertices in a graph indicate that a procedure is called often and thus is also a potentially critical procedure.

The formal analysis follows a top-down, bottom-up approach as described in Chapter 7. In the analysis we focused primarily on issues of mnemonic translation and spacecraft message construction. Our intent was to examine properties of *process termination* and *process failure*. With process termination, we were interested in determining what conditions were required for ensuring that the translation process terminates and for process failure, we were interested in determining what conditions force the translation process to fail.

11.3 High-Level Analysis

The first step in the process was to construct a high-level model that described the overall functionality of the command translation software system. Our primary source of information for constructing the high-level models were the *User's Guide* [91], the *Software Specifications Document* [93], and the *Detailed Capabilities and Adaptation Guide* [90]. The purpose of the *User's Guide* and *Software Specifications Document* are self-evident. The *Detailed Capabilities and Adaptation Guide* provided an *executive overview* of the functionality of various parts of the command system.

One of our main assumptions in deriving high-level models from the documents listed above was that the *User's Guide* and *Detailed Capabilities and Adaptation Guide* were sources of high-level information and, hence, could be viewed as reliable since conceptual information rarely changes over the lifetime of a product. For the *Software Specifications Document* we assumed that, contrary to the view held towards the *User's Guide* and *Detailed Capabilities and Adaptation Guide*, the documentation would progressively become less accurate as more detailed implementation information was encountered. This assumption is based on the fact that the software document had few revisions from the initial writing and so the correspondence between the document and the source code as the models moved closer to the implementation would decrease.

11.3.1 Context Overview

Before any translation operations can occur during on-line commanding, a communicator must be allocated and connected to be a user at a command workstation. Allocation to a communicator is performed by a member of the Data System Operations Team (DSOT)

and is restricted to users at specific workstations who are authorized to command specific spacecraft. A *Communicator* is an abstract entity that relates a spacecraft to specific radiation facilities. Figure 11.2 depicts an object model of the relationship between a Communicator and various entities and concepts. In particular, a Communicator Table is an aggregation of many Communicators. When a Communicator is allocated by a DSOT member, the Command Control processor places that Communicator in the Communicator table.

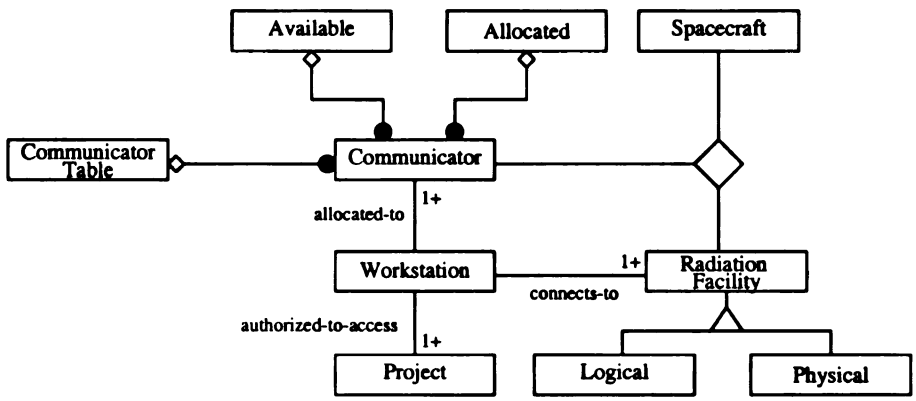


Figure 11.2: Communicator and Related Data Structures

Two abstract data objects depicted in Figure 11.2 are *Available* and *Allocated.* These are used to model the fact that resources (i.e., communicators) are either available or allocated. In the case of allocation, it is possible that an allocated communicator may not be present in the Communicator table if there is not enough room in memory. Finally, the ternary relationship between the Communicator, Spacecraft, and Radiation Facility shows that these entities have some dependent relationship, namely that the Communicator is used to allocate the resources for communicating to a Spacecraft via some Radiation Facility.

Another entity depicted in Figure 11.2 is the *Workstation* data object. The relationship between Workstation and *Radiation Facility* models the fact that a Workstation *connects-to* a Radiation Facility in order to send a command file for radiation to a spacecraft. This connection can be either *Logical* or *Physical* in the case where a Command Translation (defined later) is either performed on-line or off-line, respectively. A Workstation also has a relation to a *Project* where a Workstation can be designated as being project-specific or multimission.

11.3.2 Command Translation

Figure 11.3 contains the context diagram for the command translation software subsystem. This diagram contains one process (bubble) labeled "command translate", an external entity (rectangle) labeled "command control", and three data stores (parallel lines) labeled "MasterFile Table", "Communicator Table", and "Directive Table". The collector (circle) is used to abstract the inputs to command translate into one flow (arc).

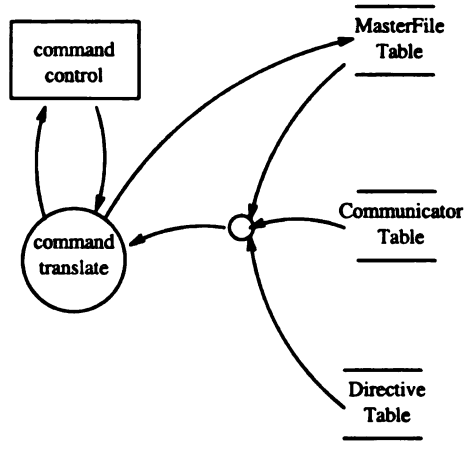


Figure 11.3: Command Translation Context Diagram

The Command Control process invokes and passes a communicator index to the Command Translate process. The Command Translate process uses this index to determine what operation is to be performed by indexing the Directive Table. The communicator index is also used to access project files via indirect access through the Communicator Table and the MasterFile Table. Once the appropriate project files are determined, Command Translate will perform the desired operation as indicated by the Directive Table. If final output is written to new output files, the MasterFile Table is updated to reflect the creation of the new files. Otherwise, no changes are made. Upon completion, Command Translate writes a return code that is accessed by the Command Control process.

Figure 11.4 contains the data flow diagram for the command translation software subsystem. This diagram is a refinement of Figure 11.3, where the dashed rectangle represents the command translate process bubble of Figure 11.3.

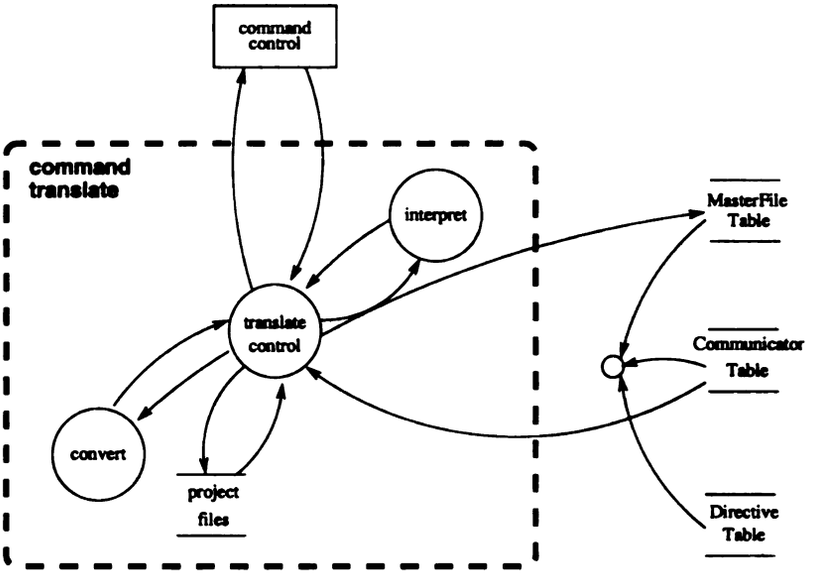


Figure 11.4: Command Translation Data Flow Diagram

The *Translate Control* process uses the communicator index in two ways: first, *Translate Control* uses the communicator index to reference the *Directive Table* in order to determine the operation to be performed, and second, *Translate Control* uses the communicator index to resolve the file names of input files and configuration files.

Once the project files have been located, translate control invokes the appropriate process (either the interpret process or the convert process) for performing the desired operation. The interpretation process, and similarly conversion, reads mnemonic input and translates that input into appropriate binary commands. The translation is based on formats specific to each project. Interpretation (conversion) of mnemonics (binary inputs) proceeds until either all of the items within an input file have been processed, an error is encountered, or the user issues a cancel.

Once the mnemonic or binary input has been processed either an output file has been created and stored in the project directory, the MasterFile table is updated to reflect the change. If the processing resulted in an error or a cancel, no updates are made. In either case, a return code is written and accessed by the Command Control process.

Figure 11.5 contains the object model for the command translation data structures. The Communicator Table is an aggregation of many Communicator Entries. The qualified relation *Communicator Index* between *User* and *Communicator Table* indicates that the *Communicator Table* is accessed using the *Communicator Index.* This allows for access to a *Communicator Entry*, which is used to access the *MasterFile Table.* A *MasterFile Table* is an aggregation of many *Project MasterFile* entries and are indexed via the qualified association *Project Id.* The *Project MasterFile* entries are then used to access *Project Command Files* through the qualified association *Project Dir.*

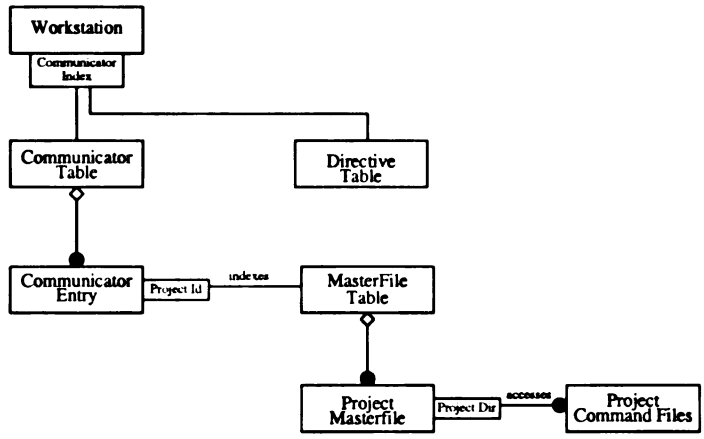


Figure 11.5: Major Data Structures for Command Translation

Figure 11.6 contains the object model of Project Command Files. This model is used to describe the different kinds of files that may be located in a Project Directory. Included in this model are the *Translation Files* that come in the form of input and output files. The *Command Files* entry shows a multiple inheritance from input and output, thus indicating that Command Files can be the output of mnemonic *translation* and the input to binary file *reformatting*. The Project Translation entry is a file that is used during translation.

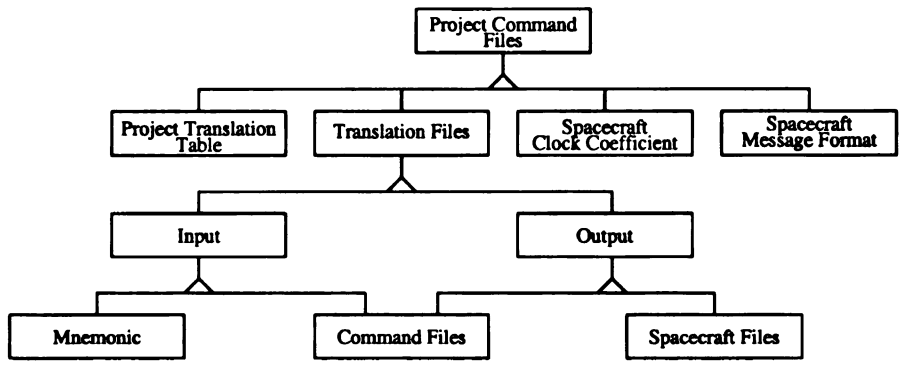


Figure 11.6: Project Files Model for Command Translation

At this point in the analysis of the command translation system we had determined that the primary function of the command translation system is to perform interpretation and conversion of mnemonic and command input files. In addition, we had determined that the command translation system reads a directive file in order to determine the mode of operation for the system. The latter fact provided an important clue regarding how to proceed with the low-level analysis of the source code in that we could use this information as a means for focusing our analysis effort to specific operating modes..

11.4 Low-Level Analysis

The software for the command system was organized into several directories that were partitioned by subsystem. Accordingly, the command translation system resided in a single directory. We began our analysis by first using a combination of tools ranging from call graph browsers to source file editors. In addition we used the unix command "grep" to perform keyword searches.

The first step involved the construction of the call graph for the *main* procedure for the command translation system. Figure 11.7 shows the call graph for the top level of the command translation system. One of the cues that was used for identifying procedures to be analyzed was procedure names. In the case of the command translation system, we were interested in analyzing *translation*. As such, we focused our investigations on the *translate* procedure, shown in the middle of the column to the right of *main* in Figure 11.7.

Figure 11.8 shows the source code for the `translate` routine of the command translation subsystem. The corresponding call graph is given in Figure 11.9. During the informal analysis, the source code and call graph were used in tandem in order to help

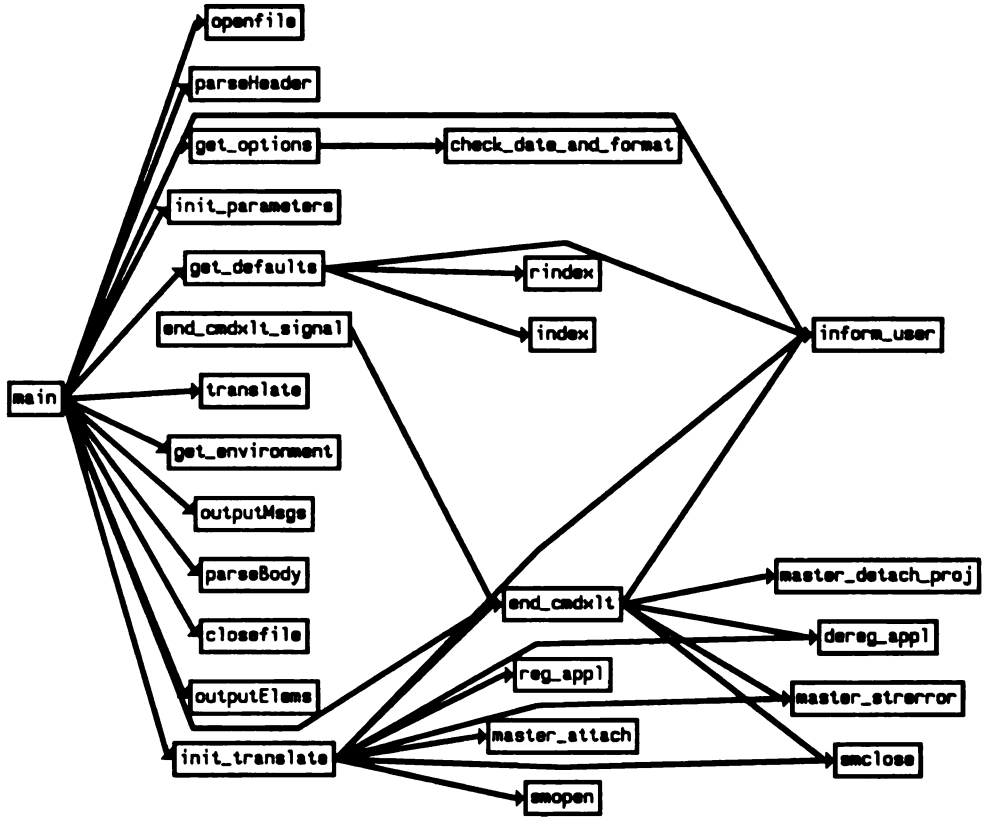


Figure 11.7: Command Translation: *Main* source model

identify procedures that required further study. For instance, the source code in Figure 11.8 consists of a switch statement with three cases: (1) `INIT`, (2) `XLT`, and (3) `CARG`. These cases correspond to different operating modes for the software for initialization, translation, and command file copying, respectively. In our analysis we were interested in the `XLT` or *translation* mode and so two functions, `process_mnemonic_input` and `process_binary_output` were tagged as requiring further study.

The next step in the process was to generate and analyze the call graph for the `process_mnemonic_input` procedure. The call graph, shown in Figure 11.10, led to the observation that the `process_msg` procedure controls a majority of the mnemonic

```
struct msg *translate(op, args)
    int op;
    char *args;
{
    extern int dontoutput;
    static struct project_parameters *pp;
    struct msg *mp = NULL;

    switch (op)
    {
       case INIT:      /* initialize the interpreter */
          pp = initialize_interpreter();
          break;

       case XLT:       /* interpret a message */
          while (args[0] != '\0')
          {
             if (process_mnemonic_input(&args, pp))
             {
                if (mp == NULL)
                   mp = process_binary_output(pp);
                else
                {
                   mp->next = process_binary_output(pp);
                   mp = mp->next;
                }
             }
             else
                dontoutput = 1;
          }
          break;

       case CARG:      /* set a value for a control argument */
          process_carg(&args, pp);
          break;

       default:
          inform_user("internal error: bad op in translate");
          end_cmdxlt(CMD_ERROR);
    }
    /* only translation returns a value;
       return NULL on error or no value */
    return(mp);
}
```

Figure 11.8: Translate source code

input processing. Specifically, given that the `process_msg` has a large difference between the out-degree and in-degree, with the out-degree dominating, it led us to identify `process_msg` as a critical procedure.

To simplify the analysis, we used the *VCG* [46] tool to aid in the visualization and analysis of the call graphs. Specifically, the *VCG* tool allowed us to abstract various functions into entities that are contained within the same by source file, as shown in

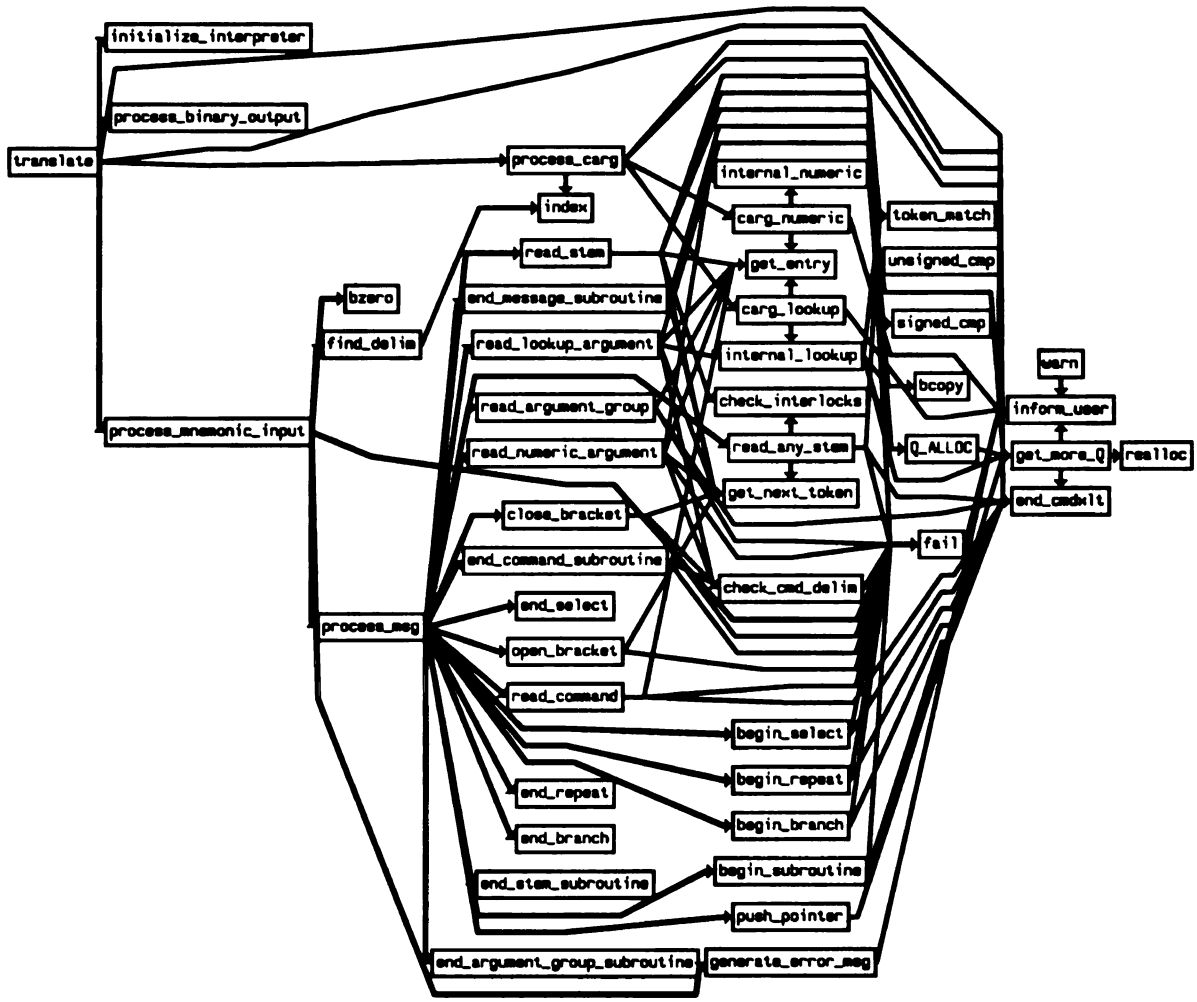


Figure 11.9: Translate source model

Figure 11.11. By *folding* the graph in this manner, much of the visual complexity was removed, thus providing a level of structural abstraction.

At this point in the study of the command translation system we were able to begin formulating questions to be answered by the formal specification phase of the analysis. For instance, a quick analysis of the `process_msg` procedure, shown in Figure 11.12, revealed that a loop is executed until the value of the variable `sp->msg_complete = 1`. Using this information, we were interested in determining when the value of `sp->msg_complete`

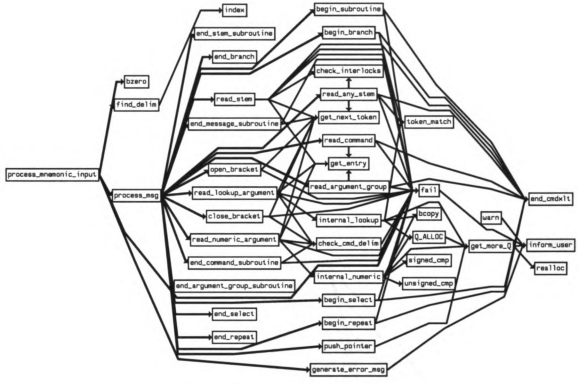


Figure 11.10: Process Mnemonic subgraph

changes from 0 to 1. In addition, given that the return value of the `process_msg` procedure is the negation of `sp->failed`, we were also interested in determining what conditions needed to be present in order for `sp->msg_complete = 1` and `sp-failed = 1` or `sp-failed = 1`. These cases would indicate that the message was syntactically correct and that the processing either failed or succeeded. Specifically, we were interested in the case where the message was constructed correctly but the processing still failed.

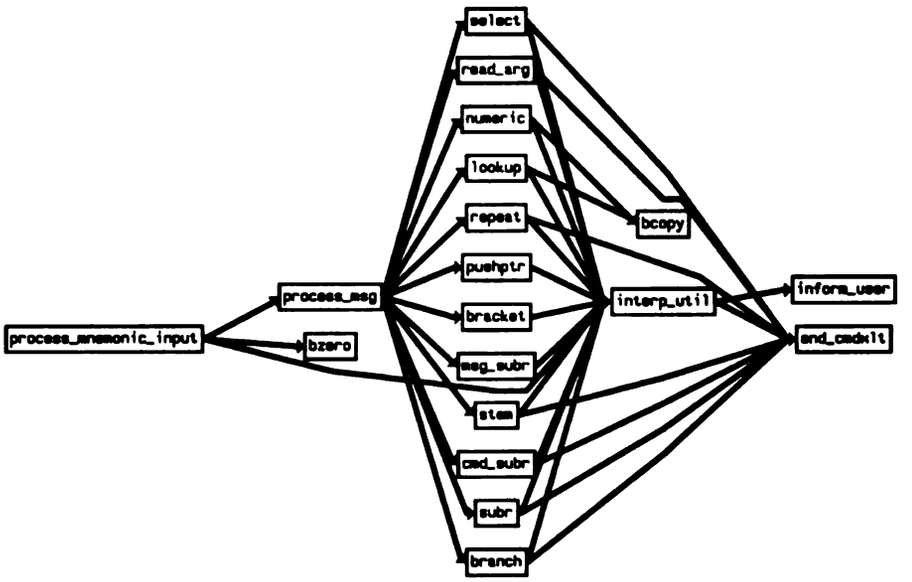


Figure 11.11: Alternative view of Process Mnemonic subgraph

```
 0.  static void (*rtn[])() =
 1.  { (void (*)())0, read_lookup_argument, read_numeric_argument,
 2.    read_argument_group, read_stem, read_any_stem, read_command,
 3.    begin_subroutine, end_argument_group_subroutine,
 4.    end_stem_subroutine, end_command_subroutine,
 5.    end_message_subroutine, begin_select, end_select,
 6.    begin_branch, end_branch, begin_repeat, end_repeat,
 7.    open_bracket, close_bracket, push_pointer
 8.  };
 9.
10.  int process_msg(ep, tp, sp, parms)
11.      U16 *ep;
12.      struct tokens *tp;
13.      struct interp_state *sp;
14.      struct project_parameters *parms;
15.  {
16.      U16 code;
17.
18.      /* check to see if this message is excluded at this site */
19.      if (ep[1]&SITE_BIT)
20.      {
21.          sp->failed = 1;
22.          fail(EXCLUDED_MSG, tp, sp);
23.          return(0);
24.      }
25.
26.      P = ep + 3 + ep[2];    /* move P to input processing instrs */
27.      sp->msg_level = 1;    /* we are at the message level */
28.      sp->msg_complete = 0; /* the message isn't complete */
29.
30.      /* interpret instructions until we have a message */
31.      while (!sp->msg_complete)
32.      {
33.          /* on failure, a new value for P
34.             will be on top of the stack */
35.          if (sp->failed)
36.              P = (U16 *)STACK(0);
37.          code = *P++;
38.          (*(rtn[code]))(tp,sp,parms);
39.      }
40.
41.      return(!sp->failed);
42.  }
```

Figure 11.12: process_msg source code

11.5 Formal Analysis

Prior to the formal analysis of the command translation system, the following details about the functionality had been determined via the informal analysis of the source code:

- The sequence of calls from originating from the `translate` procedure and proceeding to `process_mnemonic_input` and finally to `process_msg` constitutes a "critical path" of execution.
- The `process_msg` routine terminates only when the variable `sp->msg_complete` variable is set to the value 1.
- The routine `end_cmdxlt` is invoked by every one of the `begin_*` routines (among others) as shown in Figure 11.10. This led to the conjecture that `end_cmdxlt` is a critical procedure.

Using this information, we formulated the following questions to be answered by the formal analysis:

- What are the conditions for terminating the translation process.
- What are the routines that exhibit representative behavior for successful and unsuccessful translation? That is, given known termination condition for the routine `process_msg`, what routines establish `sp->msg_complete = 1`?
- Are there other terminating paths that bypass `process_msg`?

In an attempt to answer these questions, we analyzed several procedures that potentially had an impact on the issues outlined above. That is, we analyzed the `process_mnemonic_input`, `process_msg`, and `end_cmdxlt` procedures, as well as the procedure named `end_message_subroutine`. We identified `end_message_subroutine` as a routine of interest after using the *grep* command to locate the places in the code where the variable `sp->msg_complete` was assigned a value of 1.

11.5.1 Analysis of `process_mnemonic_input`

Appendix E contains the source code for the `process_mnemonic_input` procedure. The most important sequence of the procedure appears in Figure 11.13. In lines `2-11` the `do-while` loop contains several assignment statements and a call to the `process_msg` procedure. The call is used to guard a `break` statement that, in essence, provides another termination condition for the loop. As such, the terminating condition for this loop is the following:

$$(process\_msg(ep, tp, sp, params) = 1) \vee (ep = collection[249]). \qquad (11.1)$$

Expression 11.1 states that either the `process_msg` routines returns 1, or the `ep` pointer takes the value of the 250th element of the `collection` array. The significance of the number 249 (or 250, depending on the perspective point of view) is that the `ep` pointer is used as a cursor to refer to the current position in a message. When the entire input has been processed, the `ep` pointer is moved to the adjacent memory locations until, finally, it refers to the next element in the `collection` array. The more interesting aspect of the terminating condition in Expression 11.1 is the term $process\_msg(ep, tp, sp, params) = 1$. In this case, we need to analyze the `process_msg` procedure in order to determine when `process_msg` returns 1.

11.5.2 Analysis of `process_msg`

Consider again Figure 11.12. At line `41`, the statement `return(!sp->failed)` indicates that the program returns the negated value of the `sp->failed` variable. Since line `6` of the program in Figure 11.13 states that `sp->failed = 0`, it is reasonable for us to infer that $(coset(sp).failed = 0)$ is a precondition for the `process_msg` procedure.

```
 0.   ep = get_first_entry(248); /* 248 contains the message entries */
 1.
 2.   do {
 3.       tp->token_index = tp->t;
 4.       Q = control_list;
 5.       sp->num_of_commands = 0;
 6.       sp->failed = 0;
 7.       sp->cmd_delimiter_deferred = 0;
 8.
 9.       if (process_msg(ep, tp, sp, parms))
10.           break;
11.   } while ((ep = get_next_entry(ep)) != collection[249]);
12.
13.   if (sp->failed)
14.       generate_error_msg(sp, tp);
15.
16.   *strp = s;
17.   stem_entry = sp->stem_name;
18.
19.   return(!sp->failed);
```

Figure 11.13: Source code sequence for `process_mnemonic_input`

Given this precondition, consider lines `26 - 39`. Line `28` establishes the condition that `sp->msg_complete = 0`, so in the initial iteration of the loop, $(coset(sp).failed = 0) \wedge (coset(sp).msg\_complete = 0)$. Using strongest postcondition, then, to formally specify the loop, we obtain the following postcondition (as generated by AUTOSPEC):

```
/* AutoSpec:
   "(((coset(sp).msg_complete.V == 1) /\
    ((((((R_i-1 /\
    (coset(sp).failed.V != 0)) /\ (as_const8 = S[0])) \\/
    ((R_i-1 /\
    (!(coset(sp).failed.V != 0))) /\
    (suif_tmp0 .> coset(P))) /\
    (P.V = ((2 * 1) + suif_tmp0.V))) /\
    (code.V = coset(suif_tmp0).V))) /\
    sp(rtn[(int)code](tp, sp, parms), R_i))" */
```

where the term `R_i` is used to represent the ith iteration of the loop. The specification states that message processing is complete and that either the message processing failed or it was

successful. The term (last line) `sp(rtn[(int)code](tp, sp, parms), R_i)` is the specification of the various calls to the procedures listed in lines `1-7` in Figure 11.12. In order to determine if after the loop is executed that indeed (coset(sp).msg_complete.V == 1), we must analyze the various procedures.

11.5.3 Analysis of `end_message_subroutine`

Figure 11.14 contains the annotated source code for the `end_message_subroutine` procedure. After performing a *grep* search for the references to the variable `sp->msg_complete`, it was determined that in only one location throughout the code is the value of `sp->msg_complete` set to 1.

The original specification for the `end_message_subroutine` procedure as generated by AUTOSPEC is as follows:

```
/* AutoSpec:
   "(((((((((parms .> _param5) /\ (_param5.V == _pVal6)) /\
   (((sp .> _param4) /\ (_param4.V == _pVal5)) /\
   ((tp .> _param3) /\ (_param3.V == _pVal4)))) /\
   (S.V = ((4 * 1) + as_const4))) /\
   (coset(sp).msg_complete.V = 1)) /\
   (!(as_const6 != 0))) /\ (get_next_token(tp.V) != 0)) /\
    (coset(sp).failed.V = 1)) \/
    ((R_1 /\ (!(coset(sp).failed.V != 0))) /\
   (!(get_next_token(tp.V) != 0))))" */
```

Since we were interested in conditions related to message processing completion and failure, we were able to use the SPECGEN system to derive an abstraction based on deleting conjuncts. The resulting postcondition specification of the `end_message_subroutine` procedure is as follows:

```
extern void end_message_subroutine(tp, sp, parms)
    struct tokens *tp;
    struct interp_state *sp;
    struct project_parameters *parms;
  {
    S = (unsigned int *)((char *)S + 4 * 1);
    sp->msg_complete = 1;

/* AutoSpec:
   R_1: (((((parms .> _param5) /\ (_param5.V == _pVal6)) /\
   (((sp .> _param4) /\ (_param4.V == _pVal5)) /\
   ((tp .> _param3) /\ (_param3.V == _pVal4)))) /\
   (S.V = ((4 * 1) + as_const4))) /\ (coset(sp).msg_complete.V = 1)) */

    if (sp->failed != 0) {
        return;
    }

/* AutoSpec:
   "((R_1 /\ (coset(sp).failed.V != 0)) \/
    (R_1 /\ (!(coset(sp).failed.V != 0))))" */

    if (get_next_token(tp) != (void *)0) {
        sp->failed = 1;
        fail("End of message expected", tp, sp);
    }

/* AutoSpec:
   "((((R_1 /\ (!(as_const6 != 0))) /\ (get_next_token(tp.V) != 0)) /\
    (coset(sp).failed.V = 1)) \/
    ((R_1 /\ (!(coset(sp).failed.V != 0))) /\
   (!(get_next_token(tp.V) != 0))))" */

    return;

/* AutoSpec:
   "(R_1 /\ (((((!(as_const6 != 0)) /\
    (get_next_token( tp.V ) != 0)) /\ (coset(sp).failed.V = 1)) \/
    ((!(coset(sp).failed.V != 0)) /\ (!(get_next_token( tp.V ) != 0))))))" */

  }

/* AutoSpec:
   "(coset(sp).msg_complete.V = 1) /\
    (((((!(as_const6 != 0)) /\
    (get_next_token( tp.V ) != 0)) /\
    (coset(sp).failed.V = 1)) \/
    ((!(coset(sp).failed.V != 0)) /\
    (!(get_next_token( tp.V ) != 0)))) */
```

Figure 11.14: Annotated source code for `end_message_subroutine`

```
/* AutoSpec:
   "(coset(sp).msg_complete.V = 1) /\
    ((((!(as_const6 != 0)) /\
    (get_next_token( tp.V ) != 0)) /\
    (coset(sp).failed.V = 1)) \/
    ((coset(sp).failed.V = 0) /\
    (!(get_next_token( tp.V ) != 0)))) */
```

This specification states that after executing this procedure, *(coset(sp).msg_complete.V = 1)* and that either *(coset(sp).failed.V = 1)* or *(coset(sp).failed.V = 0)*. In the case that *(coset(sp).failed.V = 1)*, the *get_next_token* procedure returned a non-zero value, indicating the message stream buffer was not empty. Conversely, in the case that *(coset(sp).failed.V = 0)*, the message processing was successfully completed.

The completion of the above specification allowed us to answer the question concerning the conditions for the termination of the translation process. In doing so, it was determined that in the event the *end_message_subroutine* never appears on the message stack, the *process_msg* procedure can potentially run forever (or at least until there is a message stack overflow).

11.5.4 Analysis of `end_cmdxlt`

Given our earlier observation about the termination of *process_msg*, we proceeded to analyze whether or not other conditions can cause the *process_msg* to terminate.

In the command translation source code there are several macros that are used to access the message stack. One such macro is given in Figure 11.15. The code contained in this macro, upon accessing the stack, will generate a failure condition and terminate the entire program if the stack overflows. The importance of this macro is that several routines called

by process_msg utilize this stack macro. As such, if the failure conditions are met, then the procedure end_cmdxlt will be called.

```
#define POPM(m) S+=(int)(m); \
  if (((U16 *)S<W) || (S>min_S)) \
{ \
    fail("stack overflow", NULL, NULL); \
    end_cmdxlt(-1); \
}
```

Figure 11.15: The POPM Macro

The formal specification of the end_cmdxlt is shown in Figure 11.16. The most important aspect of this routine is that it terminates the entire program if invoked. As such, the final specification of the program is "false", indicating that the routine will never reach line X in the code. Given this fact, the command translation system, specifically the process_msg procedure and subsequently the, process_mnemonic_input procedure, will terminate either by a successful (or partially successful) completion of a message translation, or by an eventual termination via the end_cmdxlt procedure.

11.6 Discussion

In the process of performing the case study, several discoveries concerning the structure and functionality of the command translation system were gathered. In addition to revealing functional properties of the system software, the case study allowed us to discover several non-functional properties regarding the code. In this section, we summarize the case study analysis.

```
0.  extern void end_cmdxlt(int n) {
1.      if (params->cmdcntl != 0) {
2.          inform_user(1);
3.          comm_tbl_ptr->alloc[comm_index].xlt_pid = 0;
4.          xltdir->new = 0;
5.          if (smclose("directive") == DTS_ERROR) {
6.              fprintf(stderr,
7.              "translate: closing directives shared memory failed\n");
8.          }
9.
10. /* AutoSpec:
11.    "((((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
12.     (coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\
13.     (coset(xltdir).new.V = 0)) */
14.
15.         if (dereg_appl("SFOC CMD", "com_ws", SHM_ALLOC) == RES_ERROR) {
16.             fprintf(stderr,
17.             "translate: deregistration with SMC failed: %s\n",
18.             smc_errlist[smc_errno]);
19.         }
20.
21. /* AutoSpec:
22.    "((((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
23.     (coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\
24.     (coset(xltdir).new.V = 0)) */
25.
26.         if (master_detach_proj(-1) == -1) {
27.             fprintf(stderr,
28.             "translate: cannot detach from masterfile: %s\n",
29.             master_strerror(master_errno));
30.         }
31.
32. /* AutoSpec:
33.    "((((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
34.     (coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\
35.     (coset(xltdir).new.V = 0)) */
36.
37.     }
38.
39. /* AutoSpec:
40.    "((((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
41.     (coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\
42.     (coset(xltdir).new.V = 0)) \/
43.     ((n.V = _param0) /\ (!(coset(params).cmdcntl.V != 0)))" */
44.
45.     exit(n);
46.
47. /* AutoSpec: false */
48.
49.     return;
50. }
```

Figure 11.16: Annotated source code for `end_cmdxlt`

Command translation. The command translation system provides two types of command file interpretation: user mnemonic translation and command file conversion. In addition, it was determined that the command translation system relies heavily upon

communicating with other Command subsystems via the use of system files. From a low-level perspective, the `process_mnemonic_input` and `process_msg` procedures are two of the most critical procedures in the system. These procedure either directly or indirectly control the command translation process and they constitute a critical path of execution.

Termination. The termination of the command translation process depends heavily upon the termination of the `process_msg` procedure. The `process_msg` procedure terminates in one of two ways; gracefully or by fault.

Global Variables and Macros. The command translation system relies heavily upon the use of global variables and macros. While the source code is visually compact, the functional complexity seemed to increase with each encounter of one of these constructs.

11.7 Lessons Learned

Several lessons about our reverse engineering approach were learned while performing the case study described in this chapter. This section summarizes these lessons.

11.7.1 Combined Analysis Technique

The utilization of a combined informal and formal process enhanced the usefulness of both the informal and formal techniques. The informal analysis provided a structured method for early discovery and organization of the functionality of the system. During the low-level analysis, the informal techniques provided valuable information and cues regarding where to focus the formal analysis. The formal analysis facilitated the functional understanding of the underlying logic embedded in many of the structural models derived during the low-level analysis. In addition, given many of the questions that arose after the informal

analysis, the formal technique provided a method for understanding certain properties of the code.

11.7.2 Tools

The availability of tools greatly facilitated the analysis process both during the informal and the formal phases of analysis. However, while the tools were invaluable, they need to mature in regards to the functionality that they provide, especially in regards to user interface concerns.

Chapter 12

Conclusions and Future Investigations

Consider the following scenario:

> *Programmer X developed some software 6-18 months ago to handle activity Y. In the process of developing the software, programmer X used some standard semi-formal design notation until he felt he understood problem Y. Then he wrote the software, adjusting the functionality of the various routines when new sub-cases for problem Y were discovered. Today, programmer X has learned that he needs to modify the system to incorporate new requirements. As he traverses the code, he realizes that he does not recall the functionality for some of the routines.*

Most programmers most likely can recall at least one such similar experience. The presenting of the above scenario clearly points out the widespread need for reverse engineering and design recovery. The techniques that are available range from ad-hoc to mathematically rigorous methods. In this chapter we summarize the results of our investigations and suggest future investigations.

12.1 Summary of Contributions

In this section we summarize our contributions to the field of software engineering and software maintenance.

12.1.1 Strongest Postcondition

To date, the primary use of the strongest postcondition predicate transformer has been for the study of issues related to the theories underlying the semantics of programming [16]. In this dissertation we demonstrated how the strongest postcondition can be applied to the problems of reverse engineering and design recovery. In doing so, we have introduced the use of a formal technique for reverse engineering that is based on a derivational approach for program analysis. The technique incorporates the use of the strongest postcondition to transform an operational specification (i.e., a program) into a behavioral specification in terms of predicate logic expressions. In addition, we have applied the use of strongest postcondition to the definition of the semantics of the C programming language in order to demonstrate the applicability of such an approach to real languages and systems.

12.1.2 Abstraction

The construction of abstract specifications, or generalizations, from as-built specifications has primarily been focused on the use of *transformation* [72]. Starting with as-built formal specifications that are constructed from programs using the strongest postcondition, our approach facilitates deriving abstractions based on translation and the preservation of various ordering criteria. The end result is a specification that is a logical abstraction of the as-built specification. As a result, our approach ensures consistency and retains traceability between high-level abstractions and low-level as-built specifications. In addition, the results of this research can be used to support program understanding.

12.1.3 Support for Reuse

Many formal software reuse approaches depend on the assumption that a library of reusable components is available for use. This assumption, however, may not be reasonable under many conditions since the techniques used to develop the components may not have been based on formal methods. In this dissertation, we have demonstrated how a formal reverse engineering technique can be used to generate specification-based indices for existing components in order to populate component libraries.

12.2 Future Investigations

Our future work will explore three major areas: Reverse Engineering, Software Reuse, Software Reengineering. and Software Testing.

12.2.1 Reverse Engineering

One of the objectives of the research described in this dissertation was to explore the feasibility of developing a rigorous approach to the problem of reverse engineering. Our philosophy was based on a breadth approach in that the intent was to develop techniques that could be used as part of an overall reverse engineering process. Along the way, several different issues were identified that merit further study and investigation.

Loops. Abd-El-Hafiz [67] describes a knowledge-based approach for constructing specifications of looping constructs. Several other reverse engineering approaches make no explicit mention of a formal or informal treatment of loops. Our approach to loops was to provide a series of guidelines that can be applied during the loop specification process. In order to provide a more rigorous, and perhaps more automated, method for handling

loops, we intend to investigate the use of techniques such as abstract interpretation [94], and approximation algorithms for loop analysis.

Pointers. Several approaches have been suggested for handling pointer variables [95]. Our approach assumes a single level of indirection, which is appropriate for a moderately-sized class of programs, but requires extension to several levels of indirection in order to be applicable to a wider class of programs.

Fully integrated informal and formal approaches. While our approach incorporates the use of both informal and formal methods, a fully integrated approach in the respect that formal specifications are hidden from users has not yet been realized. For example, it would be desirable to allow a user to construct a series of diagrams that describe the structure of the system and then have the system construct a formal specification based on those diagrams. Similar work has been developed for the area of software requirements engineering and design [96]. Our intention is to investigate the feasibility of such an approach in the area of software maintenance and reverse engineering.

Tool environments. One of the most valuable assets that a programmer can have is access to a set of tools that support software maintenance. In addition to the tools that we have described in this dissertation, we intend to investigate how the use of several classes of tools such as those described in Chapter 10 can be combined into a single software maintenance environment. The intent is to determine how the relative advantages of each complementary reverse engineering approach can be used to provide a programmer with as much information as possible during the software maintenance process.

12.2.2 Software Reuse

In Chapter 9, we described our initial investigations into the support of software reuse via reverse engineering. Our future investigations in this area will focus on further demonstration of the use of our formal reverse engineering technique as a means for populating component libraries. Specifically, we intend to investigate how non-functional architectural information (e.g., is the module a pipe, filter, client, server, etc.) can be extracted from code in order to enrich the module specification in such a way that enhances the abilities of software reuse search engines.

12.2.3 Reengineering

Reverse engineering is the first stage of the reengineering lifecycle. The existence of formal specifications that have been recovered from code can be used to facilitate several reengineering activities. For instance, in our previous investigations we presented an approach for identifying and formally specifying objects that may be embedded in imperative program code [97]. Other potential applications of reengineering that are can be facilitated by the results of a reverse engineering phase are system modification and system re-implementation, where modification refers to changing a system to add new functionality and re-implementation refers to preservation of functionality during activities like system retargetting. In the case of system modification, a formal specification can be used as a means for verifying that modifications to various parts of a system have no adverse impact on the functionality of other parts of a system.

Our future investigations in the area of software reengineering will focus on addressing several issues:

Object-Oriented Systems. The increasing popularity of programming languages such as C++ and Java continue to force software organizations to make decisions regarding future development. In order to support the transition of current systems that have been written using imperative languages such as C and Fortran, we intend to investigate how the results of the formal reverse engineering approach can be used to facilitate a paradigm shift to object-oriented languages.

Impact analysis. *Impact analysis* is the study of the effects of software change on systems [98]. One of the primary tenets of software reengineering is that some form of change is imposed on a system to produce a new system. The use of impact analysis has been used to determine how changes affect the remainder of the system. We intend to investigate how formal specifications can be used to facilitate the impact analysis process.

APPENDICES

Appendix A

Semantics of C Expressions

This section describes the expression semantics of the C programming language using the functions $\mathcal{A}$ and $\mathcal{V}$ defined in Section 5.1.

A.1 Assignment Operators

Let v be a variable or an assignable expression[1] and e be an expression.

Let γ be an assignable object with an n-bit integer vector such that it has the following bitwise evaluation:

$$\mathcal{A}(\gamma) = \langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n \rangle \tag{A.1}$$

where the components γ_i take the value of 0 or 1 and let m be some integer. The definition of the semantics of the bitwise assignment operators <<=, >>=, &=, ^=, and |=, rely on the use of the representation in Expression A.1. As was the case with the non-bitwise operative assignment expressions:

[1] In terms of the C grammar, an assignable expression is a *unary-expression*, *postfix-expression*, or *primary-expression*.

$$\mathcal{V}(v \cong e) = \begin{cases} T & \text{if } \mathcal{A}(v \cong e) \neq 0 \\ F & \text{if } \mathcal{A}(v \cong e) = 0 \end{cases},$$

where $\cong$ is one of <<=, >>=, &=, ^=, and |=. Table A.1 defines the evaluation semantics of the bitwise operative assignment expressions.

$\mathcal{A}(\gamma$ <<= $m)$	=	$\langle \gamma_m, \gamma_{m+1}, \ldots, \gamma_n, 0_1, 0_2, \ldots, 0_m \rangle$	if $0 < m \leq n$
		$\langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n \rangle$	if $m \leq 0$
		0	if $m > n$
$\mathcal{A}(\gamma$ >>= $m)$	=	$\langle 0_1, 0_2, \ldots, 0_m, \gamma_0, \gamma_1, \ldots, \gamma_{m-1} \rangle$	if $0 < m \leq n$
		$\langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n \rangle$	if $m \leq 0$
		0	if $m > n$
$\mathcal{A}(v$ &= $\gamma)$	=	$\mathcal{A}(v)$ & $\mathcal{A}(\gamma)$	
$\mathcal{A}(v$ ^= $\gamma)$	=	$\mathcal{A}(v)$ ^ $\mathcal{A}(\gamma)$	
$\mathcal{A}(v$ |= $\gamma)$	=	$\mathcal{A}(v)$ | $\mathcal{A}(\gamma)$	

Table A.1: Bitwise Operative Assignment Operators

A.2 Logical Operators

Let α and β be expressions. The logical operators || and && are used to form logical expressions that are commonly used within the guards of conditional statements. Table A.2 describes both the evaluational and logical semantics of the operators.

A.3 Bitwise Operators

Let γ and ψ be objects with integer values that have the following bitwise representations:

$$\mathcal{A}(\gamma) = \langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n \rangle$$

$$\mathcal{A}(\psi) = \langle \psi_0, \psi_1, \psi_2, \ldots, \psi_n \rangle$$

$\mathcal{V}(\alpha \mid\mid \beta)$	$=$	$\begin{cases} T & \text{if } \mathcal{V}(\alpha) = T \text{ or } \mathcal{V}(\beta) = T \\ F & \text{if } \mathcal{V}(\alpha) = F \text{ and } \mathcal{V}(\beta) = F \end{cases}$
$\mathcal{A}(\alpha \mid\mid \beta)$	$=$	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha \mid\mid \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha \mid\mid \beta) = F \end{cases}$
$\mathcal{V}(\alpha \,\&\&\, \beta)$	$=$	$\begin{cases} T & \text{if } \mathcal{V}(\alpha) = T \text{ and } \mathcal{V}(\beta) = T \\ F & \text{if } \mathcal{V}(\alpha) = F \text{ or } \mathcal{V}(\beta) = F \end{cases}$
$\mathcal{A}(\alpha \,\&\&\, \beta)$	$=$	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha \,\&\&\, \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha \,\&\&\, \beta) = F \end{cases}$

Table A.2: Logical Operators

where the components γ_i and ψ_j take the value of 0 or 1. Table A.3 summarizes the semantics of the bitwise operators.

$\mathcal{V}(\gamma \mid \psi)$	$=$	$\begin{cases} T & \text{if } \exists i : 0 \le i \le n : \gamma_i = 1 \vee \psi_i = 1 \\ F & \text{if } \forall i : 0 \le i \le n : \gamma_i = 0 \wedge \psi_i = 0 \end{cases}$
$\mathcal{A}(\gamma \mid \psi)$	$=$	$\mathcal{A}(\gamma) \mid \mathcal{A}(\psi)$
$\mathcal{V}(\gamma \hat{} \psi)$	$=$	$\begin{cases} T & \text{if } \exists i : 0 \le i \le n : \gamma_i \ne \psi_i \\ F & \text{if } \forall i : 0 \le i \le n : \gamma_i = \psi_i \end{cases}$
$\mathcal{A}(\gamma \hat{} \psi)$	$=$	$\mathcal{A}(\gamma) \hat{} \mathcal{A}(\psi)$
$\mathcal{V}(\gamma \,\&\, \psi)$	$=$	$\begin{cases} T & \text{if } \exists i : 0 \le i \le n : \gamma_i = \psi_i = 1 \\ F & \text{if } \forall i : 0 \le i \le n : (\gamma_i \ne \psi_i) \vee (\gamma_i = \psi_i = 0) \end{cases}$
$\mathcal{A}(\gamma \,\&\, \psi)$	$=$	$\mathcal{A}(\gamma) \,\&\, \mathcal{A}(\psi)$

Table A.3: Bitwise Operators

A.4 Equality and Relational Operators

Let α and β be expressions. The logical evaluation of the equality and relational operators have the following semantics:

$$\mathcal{V}(\alpha \,\Omega\, \beta) = \begin{cases} T & \text{if } \mathcal{A}(\alpha) \,\Omega\, \mathcal{A}(\beta) \ne 0 \\ F & \text{if } \mathcal{A}(\alpha) \,\Omega\, \mathcal{A}(\beta) = 0 \end{cases}.$$

where Ω is one of ==, != , >, <, >=, and <=. The equality and relational operators have the semantics shown in Table A.4.

$\mathcal{A}(\alpha == \beta)$	=	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha == \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha == \beta) = F \end{cases}$
$\mathcal{A}(\alpha\ != \beta)$	=	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha\ != \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha\ != \beta) = F \end{cases}$
$\mathcal{A}(\alpha < \beta)$	=	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha < \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha < \beta) = F \end{cases}$
$\mathcal{A}(\alpha > \beta)$	=	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha > \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha > \beta) = F \end{cases}$
$\mathcal{A}(\alpha <= \beta)$	=	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha <= \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha <= \beta) = F \end{cases}$
$\mathcal{A}(\alpha >= \beta)$	=	$\begin{cases} 1 & \text{if } \mathcal{V}(\alpha >= \beta) = T \\ 0 & \text{if } \mathcal{V}(\alpha >= \beta) = F \end{cases}$

Table A.4: Equality and Relational Operators

A.5 Shift Operators

Let γ be an object with an integer value such that it has the following bitwise representation:

$$\mathcal{A}(\gamma) = \langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n \rangle$$

where the components γ_i take the value of 0 or 1 and let m be some integer. The logical evaluation of the shift operators has the following semantics:

$$\mathcal{V}(\gamma\ \Omega\ m) = \begin{cases} T & \text{if } \mathcal{A}(\gamma\ \Omega\ m) \neq 0 \\ F & \text{if } \mathcal{A}(\gamma\ \Omega\ m) = 0 \end{cases} .$$

where Ω is one of << and >>. Table A.5 describes the semantics of the shift operators.

A.6 Additive and Multiplicative Operators

Let α and β be expressions. The logical evaluation of the additive and multiplicative operators have the following semantics:

$$\begin{array}{|lcl|}
\hline
\mathcal{A}(\gamma << m) & = & \begin{cases} \langle \gamma_m, \gamma_{m+1}, \ldots, \gamma_n, 0_1, 0_2, \ldots, 0_m \rangle & \text{if } 0 < m \leq n \\ \langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n \rangle & \text{if } m \leq 0 \\ 0 & \text{if } m > n \end{cases} \\
\hline
\mathcal{A}(\gamma >> m) & = & \begin{cases} \langle 0_1, 0_2, \ldots, 0_m, \gamma_0, \gamma_1, \ldots, \gamma_{m-1} \rangle & \text{if } 0 < m \leq n \\ \langle \gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n \rangle & \text{if } m \leq 0 \\ 0 & \text{if } m > n \end{cases} \\
\hline
\end{array}$$

Table A.5: Shift Operators

$$\mathcal{V}(\alpha\ \Omega\ \beta) = \begin{cases} T & \text{if } \mathcal{A}(\alpha\ \Omega\ \beta) \neq 0 \\ F & \text{if } \mathcal{A}(\alpha\ \Omega\ \beta) = 0 \end{cases},$$

where Ω is one of +, -, *, /, and %. Table A.6 gives the evaluation semantics of the additive and multiplicative operators.

$\mathcal{A}(\alpha + \beta)$	=	$\mathcal{A}(\alpha) + \mathcal{A}(\beta)$
$\mathcal{A}(\alpha - \beta)$	=	$\mathcal{A}(\alpha) - \mathcal{A}(\beta)$
$\mathcal{A}(\alpha * \beta)$	=	$\mathcal{A}(\alpha) \times \mathcal{A}(\beta)$
$\mathcal{A}(\alpha / \beta)$	=	$\frac{\mathcal{A}(\alpha)}{\mathcal{A}(\beta)}$
$\mathcal{A}(\alpha \% \beta)$	=	$\mathcal{A}(\alpha) \bmod \mathcal{A}(\beta)$

Table A.6: Additive and Multiplicative Operators

Appendix B

Partial Order Lemmas

This appendix states and proves a number of lemmas regarding partial-order and weak partial-order relations. These lemmas substantiate the notion that the abstraction match operator is a partial-order relation. As such, the abstraction technique described in Chapter 6 is well-founded.

B.1 Lemma 1

The exact pre/post match is reflexive, symmetric, and transitive (i.e., the exact pre/post match is an equivalence relation).

Proof. By definition, the exact pre/post relation with respect to two specifications A and B (denoted $A \preceq_{ex} B$) is $(A_{pre} \leftrightarrow B_{pre}) \wedge (A_{post} \leftrightarrow B_{post})$. The following shows that $\preceq_{ex}$ is reflexive:

$$
\begin{aligned}
& A \preceq_{ex} A \\
& \langle \text{definition of } \preceq_{ex} \rangle \\
& \equiv (A_{pre} \leftrightarrow A_{pre}) \wedge (A_{post} \leftrightarrow A_{post}) \\
& \langle (X \leftrightarrow X) \equiv \text{true} \rangle \\
& \equiv \text{true} \wedge \text{true} \\
& \langle (\text{true} \wedge X) \equiv X \rangle \\
& \equiv \text{true}
\end{aligned}
$$

It is also straightforward to show that $\preceq_{ex}$ is symmetric so that $(A \preceq_{ex} B) \rightarrow (B \preceq_{ex} A)$, as shown below:

$$
\begin{aligned}
& A \preceq_{ex} B \\
& \langle \text{definition of } \preceq_{ex} \rangle \\
& \equiv (A_{pre} \leftrightarrow B_{pre}) \wedge (A_{post} \leftrightarrow B_{post}) \\
& \langle \text{commutativity of } \leftrightarrow \rangle \\
& \equiv (B_{pre} \leftrightarrow A_{pre}) \wedge (B_{post} \leftrightarrow A_{post}) \\
& \langle \text{definition of } \preceq_{ex} \rangle \\
& \equiv B \preceq_{ex} A
\end{aligned}
$$

Finally, the proof for the transitivity of $\preceq_{ex}$ so that $((A \preceq_{ex} B) \wedge (B \preceq_{ex} C)) \rightarrow (A \preceq_{ex} C)$ is as follows:

$$
\begin{aligned}
& (A \preceq_{ex} B) \wedge (B \preceq_{ex} A) \\
& \langle \text{definition of } \preceq_{ex} \rangle \\
& \equiv (A_{pre} \leftrightarrow B_{pre}) \wedge (A_{post} \leftrightarrow B_{post}) \wedge (B_{pre} \leftrightarrow C_{pre}) \wedge \\
& (B_{post} \leftrightarrow C_{post}) \\
& \langle \text{definition of } \leftrightarrow, \text{ substitution of } B_{pre} \text{ with } A_{pre}, \text{ and } B_{post} \\
& \text{with } A_{post} \rangle \\
& \equiv (A_{pre} \leftrightarrow A_{pre}) \wedge (A_{post} \leftrightarrow A_{post}) \wedge (A_{pre} \leftrightarrow C_{pre}) \wedge \\
& (A_{post} \leftrightarrow C_{post}) \\
& \langle (X \leftrightarrow X) \equiv \text{true}, (\text{true} \wedge X) \equiv X \rangle \\
& \equiv (A_{pre} \leftrightarrow C_{pre}) \wedge (A_{post} \leftrightarrow C_{post}) \\
& \langle \text{definition of } \preceq_{ex} \rangle \\
& \equiv A \preceq_{ex} C
\end{aligned}
$$

Since $\preceq_{ex}$ is reflexive, symmetric, and transitive, $\preceq_{ex}$ is an equivalence relation. □

B.2 Lemma 2

The plug-in match is reflexive, anti-symmetric, and transitive (i.e., the plug-in match is a partial order relation).

Proof. By definition, the plug-in relation with respect to two specifications A and B (denoted $B \preceq_{pi} A$) is $(A_{pre} \rightarrow B_{pre}) \wedge (B_{post} \rightarrow A_{post})$. The following shows that

$\preceq_{pi}$ is reflexive:

$$
\begin{aligned}
& A \preceq_{pi} A \\
& \langle \text{definition of } \preceq_{pi} \rangle \\
& \equiv (A_{pre} \rightarrow A_{pre}) \wedge (A_{post} \rightarrow A_{post}) \\
& \langle (X \rightarrow X) \equiv \text{true} \rangle \\
& \equiv \text{true} \wedge \text{true} \\
& \langle (\text{true} \wedge X) \equiv X \rangle \\
& \equiv \text{true}
\end{aligned}
$$

The following proof shows that $\preceq_{pi}$ is antisymmetric so that $((B \preceq_{pi} A) \wedge (A \preceq_{pi} B)) \rightarrow A \preceq_{ex} B$:

$$
\begin{aligned}
& ((B \preceq_{pi} A) \wedge (A \preceq_{pi} B)) \\
& \langle \text{definition of } \preceq_{pi} \rangle \\
& \equiv (A_{pre} \rightarrow B_{pre}) \wedge (B_{post} \rightarrow A_{post}) \wedge (B_{pre} \rightarrow A_{pre}) \wedge \\
& (A_{post} \rightarrow B_{post}) \\
& \langle \text{associativity of } \wedge \rangle \\
& \equiv (A_{pre} \rightarrow B_{pre}) \wedge (B_{pre} \rightarrow A_{pre}) \wedge (A_{post} \rightarrow B_{post}) \wedge \\
& (B_{post} \rightarrow A_{post}) \\
& \langle ((X \rightarrow Y) \wedge (Y \rightarrow X)) \equiv (X \leftrightarrow Y) \rangle \\
& \equiv (A_{pre} \leftrightarrow B_{pre}) \wedge (A_{post} \leftrightarrow B_{post}) \\
& \langle \text{definition of } \preceq_{ex} \rangle \\
& \equiv A \preceq_{ex} B
\end{aligned}
$$

Finally, the following proof shows that $\preceq_{pi}$ is transitive so that $((B \preceq_{pi} A) \wedge (C \preceq_{pi} B)) \rightarrow C \preceq_{pi} A$:

$$
\begin{aligned}
& ((B \preceq_{pi} A) \wedge (C \preceq_{pi} B)) \\
& \langle \text{definition of } \preceq_{pi} \rangle \\
& \equiv (A_{pre} \rightarrow B_{pre}) \wedge (B_{post} \rightarrow A_{post}) \wedge (B_{pre} \rightarrow C_{pre}) \wedge \\
& (C_{post} \rightarrow B_{post}) \\
& \langle \text{associativity of } \wedge \rangle \\
& \equiv (A_{pre} \rightarrow B_{pre}) \wedge (B_{pre} \rightarrow C_{pre}) \wedge (C_{post} \rightarrow B_{post}) \wedge \\
& (B_{post} \rightarrow A_{post}) \\
& \langle ((X \rightarrow Y) \wedge (Y \rightarrow Z)) \rightarrow (X \leftrightarrow Z) \rangle \\
& \Rightarrow (A_{pre} \leftrightarrow C_{pre}) \wedge (C_{post} \leftrightarrow A_{post}) \\
& \langle \text{definition of } \preceq_{pi} \rangle
\end{aligned}
$$

$$\equiv C \preceq_{pi} A$$

Since $\preceq_{pi}$ is reflexive, antisymmetric, and transitive, $\preceq_{pi}$ is a partial order relation. □

B.3 Lemma 3

The exact pre/post match is reflexive, symmetric, and transitive (i.e., the exact pre/post match is an equivalence relation).

Proof. By definition, the plug-in post match with respect to specifications A and B (denoted $B \preceq_{pip} A$) is ($B_{post} \rightarrow A_{post}$). Since the logical operator $\rightarrow$ is reflexive, antisymmetric, and transitive, $\preceq_{pip}$ is a partial order relation.

Appendix C

Application Program

This section contains the source code for the example discussed in Section 6.4. The application is a C program that is part of a mission control ground-based system used by the NASA Jet Propulsion Laboratory. The system is responsible for the translation of user commands into appropriate spacecraft mnemonics, enabling users to modify spacecraft mission operations. This particular module takes a sequence of elements from a file and returns an index to a subsequence of elements specified by begin and end indices. In our previous investigations, we described the *sp* semantics for C [7] and pointers [43]. Those semantics were used to construct the `/*AS AS*/` annotations for the code contained in this section.

```
 0. /*
 1.  * Inputs: file (file to read from)
 2.  *    begin (first element to copy)
 3.  *     end (last element to copy)
 4.  * Outputs: none
 5.  * Externally read: body_lineno (for errors)
 6.  *     sc (to translate to mission ID)
 7.  * Externally modified: dontoutput (errors)
 8.  * Returns: the elements copied (NULL on error)
 9.  *
10.  * This routine does the actual work of opening and parsing the GCMD
11.  * file, finding the elements, and returning the appropriate ones.
12.  */
13. struct gcmd_elem *doGCMDCopy(char *file, int begin, int end)
```

```
14. {
15.     int fd;
16.     U16 L2;
17.     struct gcmd_hdr gcmd_hdr;
18.     int i;
19.     register j;
20.     struct gcmd_elem *orig_elem;
21.     struct gcmd_elem *elem;
22.     struct gcmd_elem *ep;
23.     extern int body_lineno;
24.
25.     /* open the file */
26.     /*AS (begin = B0 & end = E0 & file .> F0) AS*/
27.
28.     fd = open_copy_file(file, &L2, CMD_DSN);
29.
30.     /*AS (fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
31.
32.     if (fd < 0)
33.     {
34.       /*AS (fd < 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
35.
36.       dontoutput = 1;
37.
38.       /*AS (dontoutput = 1 &
39.             fd < 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
40.
41.       return(NULL);
42.
43.       /*AS false AS*/
44.     }
45.
46.     /*AS (fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
47.
48.     if (!skip_gcmd_sfdu(fd, L2))
49.     {
50.       /*AS (skip_gcmd_sfdu(fd, L2) = 0 &
51.             fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
52.
53.       inform_user("line %d: copy failed: bad SFDU header (%s)",
54.                 body_lineno, file);
55.
56.       /*AS (skip_gcmd_sfdu(fd, L2) = 0 &
57.             fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
58.
59.       dontoutput = 1;
60.
61.       /*AS (dontoutput = 1 & skip_gcmd_sfdu(fd, L2) = 0 &
62.             fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
63.
64.       close(fd);
65.
66.       /*AS (closed(fd) & dontoutput = 1 & skip_gcmd_sfdu(fd, L2) = 0 &
67.             fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
68.
69.       if (params->cmdcntl) master_unlock();
70.
71.       /*AS (params->cmdcntl != 0 & sp(master_unlock(),
72.                closed(fd) & dontoutput = 1
73.              & skip_gcmd_sfdu(fd, L2) = 0 & fd >= 0 & fd = FH0 &
74.                begin = B0 & end = E0 & file .> F0)) |
75.             (params->cmdcntl = 0 & closed(fd) & dontoutput = 1 &
76.              skip_gcmd_sfdu(fd, L2) = 0 &
77.              fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
78.
79.       return(NULL);
80.
81.       /*AS false AS*/
```

```
 82.     }
 83.
 84.     /*AS (skip_gcmd_sfdu(fd, L2) != 0 &
 85.           fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
 86.
 87.     if (!get_gcmd_hdr(fd, &gcmd_hdr))
 88.     {
 89.       dontoutput = 1;
 90.       close(fd);
 91.       if (params->cmdcntl) master_unlock();
 92.       return(NULL);
 93.     }
 94.
 95.     /*AS (get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
 96.           fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
 97.
 98.     if (params->sc != gcmd_hdr.SC)
 99.     {
100.       inform_user("line %d: copy: invalid spacecraft in GCMD file (%s)",
101.                   body_lineno, file);
102.       dontoutput = 1;
103.       close(fd);
104.       if (params->cmdcntl) master_unlock();
105.       return(NULL);
106.     }
107.
108.     /*AS (params->sc = gcmd_hdr.SC &
109.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
110.           fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
111.
112.     /* make sure the file has enough elements */
113.     if (end == -1)
114.       end = gcmd_hdr.elem_count;
115.     else if (end > gcmd_hdr.elem_count)
116.     {
117.       inform_user("line %d: copy: not enough elements in GCMD file (%s)",
118.                   body_lineno, file);
119.       dontoutput = 1;
120.       close(fd);
121.       if (params->cmdcntl) master_unlock();
122.       return(NULL);
123.     }
124.
125.     /*AS
126.          (E0 = -1 & end = gcmd_hdr.elem_count &
127.           params->sc = gcmd_hdr.SC &
128.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
129.           fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
130.          |
131.          (end <= gcmd_hdr.elem_count & end != -1 &
132.           params->sc = gcmd_hdr.SC &
133.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
134.           fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0) AS*/
135.
136.     /* read in the elements */
137.     orig_elem = NULL;
138.
139.     /*AS orig_elem .> NULL & ((E0 = -1 & end = gcmd_hdr.elem_count &
140.           params->sc = gcmd_hdr.SC &
141.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
142.           fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
143.          |
144.          (end <= gcmd_hdr.elem_count & end != -1 &
145.           params->sc = gcmd_hdr.SC &
146.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
147.           fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
148.
149.     for (i = 1; i <= gcmd_hdr.elem_count; i++)
```

```
150.    {
151.
152.      /*AS Qi AS*/
153.
154.      if (orig_elem == NULL)
155.      {
156.
157.          /*AS Qi & orig_elem .> NULL AS*/
158.
159.          orig_elem = get_elem(fd);
160.
161.          /*AS Qi & orig_elem .> Obj0 AS*/
162.
163.          elem = orig_elem;
164.
165.          /*AS Qi & orig_elem .> Obj0 & elem .> coset(orig_elem) AS*/
166.
167.      }
168.      else
169.      {
170.
171.          /*AS !(orig_elem .> NULL) & Qi AS*/
172.
173.          elem->next = get_elem(fd);
174.
175.          /*AS !(orig_elem .> NULL) & Qi & elem->next .> Obji AS*/
176.
177.          elem = elem->next;
178.
179.          /*AS !(orig_elem .> NULL) & Qi & elem->next .> Obji &
180.              elem .> coset(elem->next) AS*/
181.      }
182.
183.      /*AS (orig_elem .> Obj0 & Qi & elem .> coset(orig_elem)) |
184.           (!(orig_elem .> NULL) & Qi & elem->next .> Obji &
185.              elem .> coset(elem->next)) AS*/
186.
187.      if (elem == NULL)
188.      {
189.          dontoutput = 1;
190.          close(fd);
191.          if (params->cmdcntl) master_unlock();
192.          for (elem = orig_elem; elem != NULL; elem = ep)
193.          {
194.            ep = elem->next;
195.            free(elem);
196.          }
197.          return(NULL);
198.      }
199.
200.      /*AS !(elem .> NULL) &
201.           (orig_elem .> Obj0 & Qi & elem .> coset(orig_elem)) |
202.           (!(orig_elem .> NULL) & Qi & elem->next .> Obji &
203.              elem .> coset(elem->next)) AS*/
204.
205.      /* make sure the data isn't corrupted */
206.      if (elem_chksum(elem) != elem->chksum)
207.      {
208.          inform_user("line %d: copy: checksum failed for element %d (%s)",
209.                  body_lineno, i, file);
210.          dontoutput = 1;
211.          close(fd);
212.          if (params->cmdcntl) master_unlock();
213.          for (elem = orig_elem; elem != NULL; elem = ep)
214.          {
215.            ep = elem->next;
216.            free(elem);
217.          }
```

```
218.            return(NULL);
219.        }
220.
221.        /*AS (elem_chksum(elem) = elem->chksum) & !(elem .> NULL) &
222.             (orig_elem .> Obj0 & Qi & elem .> coset(orig_elem)) |
223.             (!(orig_elem .> NULL) & Qi & elem->next .> Obji &
224.                 elem .> coset(elem->next)) AS*/
225.
226.        /*
227.         * 0 fields not to be copied;
228.         * note: proj, SC, chksum, id, file, and elem_num are filled in
229.         * in collapse_elem_chain();
230.         */
231.        elem->remaining_rad_time = 0;
232.        elem->gsoc = 0;
233.        elem->chksum = 0;
234.        elem->elem_num = 0;
235.        for (j = 0; j < (sizeof(elem->mccc)/sizeof(elem->mccc[0])); j++)
236.            elem->mccc[j] = 0;
237.
238.        /*AS zeroed(elem) &
239.             (elem_chksum(elem) = elem->chksum) & !(elem .> NULL) &
240.             (orig_elem .> Obj0 & Qi & elem .> coset(orig_elem)) |
241.             (!(orig_elem .> NULL) & Qi & elem->next .> Obji &
242.                 elem .> coset(elem->next)) AS*/
243.
244.    }
245.
246.    /*AS (forall k : 1 <= k < gcmd_hdr.elem_count :
247.          elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
248.          orig_elem .> Obj0 & elem_1 .> coset(orig_elem) &
249.         ((E0 = -1 & end = gcmd_hdr.elem_count &
250.          params->sc = gcmd_hdr.SC &
251.          get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
252.          fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
253.         |
254.         (end <= gcmd_hdr.elem_count & end != -1 &
255.          params->sc = gcmd_hdr.SC &
256.          get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
257.          fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
258.
259.    elem->next = NULL;
260.
261.    /*AS elem_gcmd_hdr.elem_count .> NULL &
262.         (forall k : 1 <= k < gcmd_hdr.elem_count :
263.          elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
264.          orig_elem .> Obj0 & elem_1 .> coset(orig_elem) &
265.         ((E0 = -1 & end = gcmd_hdr.elem_count &
266.          params->sc = gcmd_hdr.SC &
267.          get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
268.          fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
269.         |
270.         (end <= gcmd_hdr.elem_count & end != -1 &
271.          params->sc = gcmd_hdr.SC &
272.          get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
273.          fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
274.
275.    /* check the file checksum */
276.    if (!checksum_gcmd_chain(&gcmd_hdr, orig_elem))
277.    {
278.        inform_user("line %d: copy: checksum failed on GCMD file (%s)",
279.                    body_lineno, file);
280.        dontoutput = 1;
281.        close(fd);
282.        if (params->cmdcntl) master_unlock();
283.        for (elem = orig_elem; elem != NULL; elem = ep)
284.        {
285.            ep = elem->next;
```

```
286.          free(elem);
287.        }
288.        return(NULL);
289.      }
290.
291.      /*AS checksum_gcmd_chain(gcmd_hdr, orig_elem.V) = 0 &
292.           elem_gcmd_hdr.elem_count .> NULL &
293.           (forall k : 1 <= k < gcmd_hdr.elem_count :
294.            elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
295.            orig_elem .> Obj0 & elem_1 .> coset(orig_elem) &
296.           ((E0 = -1 & end = gcmd_hdr.elem_count &
297.            params->sc = gcmd_hdr.SC &
298.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
299.            fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
300.           |
301.           (end <= gcmd_hdr.elem_count & end != -1 &
302.            params->sc = gcmd_hdr.SC &
303.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
304.            fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
305.
306.      /* free any initial unneeded elements */
307.      for (i = 1, elem = orig_elem; i < begin; i++)
308.      {
309.        ep = elem->next;
310.        free(elem);
311.        elem = ep;
312.      }
313.
314.      /*AS checksum_gcmd_chain(gcmd_hdr, orig_elem.V) = 0 &
315.           elem_gcmd_hdr.elem_count .> NULL &
316.           (forall k : 1 <= k < begin : freed(elem_k)) &
317.           (forall k : begin <= k < gcmd_hdr.elem_count :
318.            elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
319.            orig_elem .> NULL & elem_1 .> coset(orig_elem) &
320.           ((E0 = -1 & end = gcmd_hdr.elem_count &
321.            params->sc = gcmd_hdr.SC &
322.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
323.            fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
324.           |
325.           (end <= gcmd_hdr.elem_count & end != -1 &
326.            params->sc = gcmd_hdr.SC &
327.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
328.            fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
329.
330.
331.      orig_elem = elem;
332.
333.      /*AS orig_elem .> coset(elem_begin) &
334.           checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
335.           elem_gcmd_hdr.elem_count .> NULL &
336.           (forall k : 1 <= k < begin : freed(elem_k)) &
337.           (forall k : begin <= k < gcmd_hdr.elem_count :
338.            elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
339.           ((E0 = -1 & end = gcmd_hdr.elem_count &
340.            params->sc = gcmd_hdr.SC &
341.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
342.            fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
343.           |
344.           (end <= gcmd_hdr.elem_count & end != -1 &
345.            params->sc = gcmd_hdr.SC &
346.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
347.            fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
348.
349.      /* zero out the first element copied, only */
350.      elem->delay = 0;
351.
352.      while (i++ < end)
353.        elem = elem->next;
```

```
354.
355.      ep = elem;
356.      elem = elem->next;
357.      ep->next = NULL;
358.
359.      /*AS ep .> coset(elem_end) & elem .> coset(elem_end->next) &
360.           elem_end->next .> NULL & orig_elem .> coset(elem_begin) &
361.           checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
362.           elem_gcmd_hdr.elem_count .> NULL &
363.           (forall k : 1 <= k < begin : freed(elem_k)) &
364.           (forall k : begin <= k < gcmd_hdr.elem_count :
365.            elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
366.           ((E0 = -1 & end = gcmd_hdr.elem_count &
367.            params->sc = gcmd_hdr.SC &
368.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
369.            fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
370.           |
371.           (end <= gcmd_hdr.elem_count & end != -1 &
372.            params->sc = gcmd_hdr.SC &
373.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
374.            fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
375.
376.      /* free any terminal unneeded elements */
377.      while (i++ <= gcmd_hdr.elem_count)
378.      {
379.        ep = elem->next;
380.        free(elem);
381.        elem = ep;
382.      }
383.
384.      /*AS (forall k : end < k < gcmd_hdr.elem_count : freed(elem_k)) &
385.           ep .> coset(elem_gcmd_hdr.elem_count) & elem .> coset(ep) &
386.           elem_end->next .> NULL & orig_elem .> coset(elem_begin) &
387.           checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
388.           elem_gcmd_hdr.elem_count .> NULL &
389.           (forall k : 1 <= k < begin : freed(elem_k)) &
390.           (forall k : begin <= k < end :
391.            elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
392.           ((E0 = -1 & end = gcmd_hdr.elem_count &
393.            params->sc = gcmd_hdr.SC &
394.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
395.            fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
396.           |
397.           (end <= gcmd_hdr.elem_count & end != -1 &
398.            params->sc = gcmd_hdr.SC &
399.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
400.            fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
401.
402.      close(fd);
403.
404.      /*AS closed(fd) &
405.           (forall k : end < k < gcmd_hdr.elem_count : freed(elem_k)) &
406.           ep .> coset(elem_gcmd_hdr.elem_count) & elem .> coset(ep) &
407.           elem_end->next .> NULL & orig_elem .> coset(elem_begin) &
408.           checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
409.           elem_gcmd_hdr.elem_count .> NULL &
410.           (forall k : 1 <= k < begin : freed(elem_k)) &
411.           (forall k : begin <= k < end :
412.            elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
413.           ((E0 = -1 & end = gcmd_hdr.elem_count & params->sc = gcmd_hdr.SC
414.            & get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
415.           fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
416.           |
417.           (end <= gcmd_hdr.elem_count & end != -1 &
418.            params->sc = gcmd_hdr.SC &
419.            get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
420.            fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)) AS*/
421.
```

```
422.     if (params->cmdcntl) master_unlock();
423.
424.     /*AS
425.       (params->cmdcntl != 0 & sp(master_unlock(),
426.          closed(fd) &
427.          (forall k : end < k < gcmd_hdr.elem_count : freed(elem_k)) &
428.          ep .> coset(elem_gcmd_hdr.elem_count) & elem .> coset(ep) &
429.          elem_end->next .> NULL & orig_elem .> coset(elem_begin) &
430.          checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
431.          elem_gcmd_hdr.elem_count .> NULL &
432.          (forall k : 1 <= k < begin : freed(elem_k)) &
433.          (forall k : begin <= k < end :
434.           elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
435.          ((E0 = -1 & end = gcmd_hdr.elem_count &
436.           params->sc = gcmd_hdr.SC &
437.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
438.           fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
439.          |
440.          (end <= gcmd_hdr.elem_count & end != -1 &
441.           params->sc = gcmd_hdr.SC &
442.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
443.           fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0))))
444.       |
445.          (closed(fd) &
446.          (forall k : end < k < gcmd_hdr.elem_count : freed(elem_k)) &
447.          ep .> coset(elem_end) & elem .> coset(elem_end->next) &
448.          elem_end->next .> NULL & orig_elem .> coset(elem_begin) &
449.          checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
450.          elem_gcmd_hdr.elem_count .> NULL &
451.          (forall k : 1 <= k < begin : freed(elem_k)) &
452.          (forall k : begin <= k < end :
453.           elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
454.          ((E0 = -1 & end = gcmd_hdr.elem_count &
455.           params->sc = gcmd_hdr.SC &
456.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
457.           fd >= 0 & fd = FH0 & begin = B0 & E0 = E0 & file .> F0)
458.          |
459.          (end <= gcmd_hdr.elem_count & end != -1 &
460.           params->sc = gcmd_hdr.SC &
461.           get_gcmd_hdr(fd, gcmd_hdr) != 0 & skip_gcmd_sfdu(fd, L2) != 0 &
462.           fd >= 0 & fd = FH0 & begin = B0 & end = E0 & file .> F0)))
463.
464.     return(orig_elem);
465. }
```

Appendix D

Software Reuse Specifications

This appendix contains the as-built specifications of the queue library and the corresponding library specification as presented in Section 9.3.

D.1 As-built specification for the `Queue` source code

Figure D.1 shows the as-built specifications for the queue source code as they were constructed by the AUTOSPEC system. The figure contains five specifications corresponding to the `dequeue`, `enQueue`, `new_queue`, `head`, and `is_empty` operations. The format for the specifications, based on the Larch interface language [42] is shown in Figure 6.1.

```
spec QDATA dequeue(Queue *q)
locals
   int temp
requires
   (q .> _param2) &&
   (_param2.tail.V = _pVal3.tail) &&
   (_param2.head.V = _pVal3.head)
modifies
   q (_param2)
ensures
   (q .> _param2) &&
   (_param2.tail.V = _pVal3.tail) &&
   ((as_const2 == _pVal3.head) &&
    (is_empty(_param2.V) != 1) &&
    (temp.V == (as_const2 % MAXSIZE)) &&
    (_param2.head.V = (as_const2 + 1)) &&
    (return.V = _param2.data[temp.V]))
   ||
   ((_param2.head.V = _pVal3.head) &&
    (!(is_empty(_param2.V) != 1)) &&
    (return.V = 0))

spec QDATA head(const Queue q)
requires
   (q.V = _param1) &&
   (q.tail.V == _pVal3.tail) &&
   (q.head.V == _pVal3.head)
ensures
   (q.V = _param1) &&
   (q.tail.V = _pVal3.tail) &&
   (q.head.V = _pVal3.head)
   (return.V = q.data[(q.head.V % MAXSIZE)])


spec Queue *new_queue()
requires
   true
ensures
   (newQ.V .> o) &&
   (o.head.V = 0) &&
   (o.tail.V = 0) &&
   (return.V = newQ.V)
```

```
spec int enQueue(Queue *q, QDATA *e)
requires
(((e .> _param4) &&
   (_param4.V == _pVal5)) &&
   ((q .> _param3) &&
   (_param3.tail.V == _pVal4)))
modifies
   q (_param3)
ensures
((((((e .> _param4) &&
   (_param4.V == _pVal5)) &&
   ((q .> _param3) &&
   (_param3.tail.V == _pVal4))) &&
   ((_param3.tail.V -
      _param3.head.V) == MAXSIZE)) &&
   (return.V = 0)) ||
(((((((e .> _param4) &&
   (_param4.V == _pVal5)) &&
   ((q .> _param3) &&
   (_param3.tail.V == _pVal4))) &&
   (!((_pVal4 - _param3.head.V) ==
      MAXSIZE))) &&
   (_param3.data[(_pVal4 % MAXSIZE)].V =
      _param4.V)) &&
   (_param3.tail.V = (_pVal4 + 1))) &&
   (return.V = 1)))

spec int is_empty(const Queue q)
requires
   (q.V = _param0)
ensures
   (q.V = _param0) &&
   (return.V = (q.head.V == q.tail.V))
```

Figure D.1: AUTOSPEC output of as-built specifications for queue source code

D.2 Circular Queue Library Specification

Figure D.2 shows the library specification for the queue source code. The format for the specifications is based on the Larch interface language [42] and is used by the ABRIE system as a means for storing specifications and references to supporting source code.

```
Module CircularQueue
  Ports
    ProcDef dequeue(Queue* q) return int {
       uses auxTheories;
       requires true;
       modifies q.head;
       ensures
            (q.head^ ~= q.tail^ /\ q.head' = q.head^ + 1 /\
             result = q.data[mod(q.head^,MAXSIZE)])
            \/
            (q.head' = q.head^ /\ q.head^ = q.tail^ /\ result = 0);
    }

    ProcDef enQueue(Queue* q, int e) return int {
       uses auxTheories;
       requires true
       modifies q.tail, q.data;
       ensures
           (q.tail - q.head = MAXSIZE) /\ result = 0)
            \/
           (q.tail^ - q.head^ ~= MAXSIZE /\
            q.data'[mod(q.tail^,MAXSIZE)] = e /\
            q.tail' = q.tail^ + 1 /\
            result = 1);
    }

    ProcDef head(Queue q) return int {
       uses auxTheories;
       requires true;
       ensures result = q.data[mod(q.head^,MAXSIZE)];
    }

    ProcDef is_empty(Queue q) return Bool {
       uses auxTheories;
       requires true
       ensures  result = (q.head == q.tail);
    }

    ProcDef new_queue() return Queue* {
       uses auxTheories;
       requires true;
       ensures o.head = 0 /\ o.tail = 0 /\ result = o;
    }

    Implementation
       source ("/user/r02/chengb/gannod/Research/CircQueue/queue/","queue.c")

End
```

Figure D.2: Circular Queue Library Specification

Appendix E

process_mnemonic_input Source Code

This appendix contains the source code for the process_mnemonic_input procedure. The purpose of this procedure is to parse an input stream and to invoke the process_msg translation routine.

```
int process_mnemonic_input(strp, parms)
    char **strp;
    struct project_parameters *parms;
{
    char *s = *strp;
    struct tokens tokens;
    struct tokens *tp = &tokens;
    struct interp_state state;
    struct interp_state *sp = &state;
    int len;
    U16 *ep;

    /* set up token list */
    bzero(tp, sizeof(*tp));
    tp->end_token = tp->t - 1;

    /* copy special character list into token list */
    strcpy("", tp->special_chars);
    if (parms->field_delimiter != '*') /* '*' indicates none specified */
sprintf(tp->special_chars, "%s%c",
tp->special_chars, parms->field_delimiter);
    if (parms->command_delimiter != '*')
sprintf(tp->special_chars, "%s%c",
tp->special_chars, parms->command_delimiter);
    if (parms->message_delimiter != '*')
sprintf(tp->special_chars, "%s%c",
tp->special_chars, parms->message_delimiter);
    if (parms->left_bracket != '*')
sprintf(tp->special_chars, "%s%c",
tp->special_chars, parms->left_bracket);
    if (parms->right_bracket != '*')
sprintf(tp->special_chars, "%s%c",
tp->special_chars, parms->right_bracket);

    /* initialize interpreter state */
```

```
    sp->fail_token = NULL;
    sp->fail_reason = NULL;
    sp->msg_entry = NULL;
    sp->stem_name = NULL;
    stem_entry = NULL;

    /* tokenize the input str */
    while (*s != '\0')
    {
char *cp = s;
char *delim;

/* skip initial blanks */
while (isspace(*cp))
    cp++;
s = cp;

/* find the end of the token */
delim = find_delim(s, tp->special_chars);
        cp = delim;

/* calculate length */
if (cp == s)
    len = 0;
else
{
    while (isspace(*--cp))
;

    cp++;
    len = cp - s;
}

save_tok(s, len);

if (*delim == '\0')
{
    s = delim;
    break;
}
else if (*delim == parms->message_delimiter)
{
    s = delim + 1;
    break; /* complete msg */
}
else if (*delim == parms->field_delimiter)
{
    s = delim + 1;
    if (*s == '\0') /* last (default) argument */
    {
save_tok(s, 0);
    }
}
else /* command delim or bracket */
{
    save_tok(delim, 1);
    s = delim + 1;
    if (*delim == parms->right_bracket)
    {
if (*s == parms->command_delimiter)
{
    save_tok(s, 1);
    s++;
}
    }
}
    }
```

```
    /* analyze the token stream */
    ep = get_first_entry(248); /* 248 contains the message entries */

    do
    {
/* set globals to initial values */
tp->token_index = tp->t;
Q = control_list;
sp->num_of_commands = 0;
sp->failed = 0;
sp->cmd_delimiter_deferred = 0;

if (process_msg(ep, tp, sp, parms))
    break;
    } while ((ep = get_next_entry(ep)) != collection[249]);

    /* if we didn't find any match in 248, generate error message */
    if (sp->failed)
generate_error_msg(sp, tp);
#ifdef DEBUG2
    else
if (strlen(*strp) > RESP_LN - 20)
    inform_user("parsed line: '%.*s...'", RESP_LN-20, *strp);
else
    inform_user("parsed line: '%s'", *strp);
#endif
    *strp = s;

    /* save the stem name for the comment in the message output */
    stem_entry = sp->stem_name;

    return(!sp->failed);
}
```

Bibliography

Bibliography

[1] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *IEEE Computer*, vol. 26, pp. 18–41, July 1993.

[2] Report by the Inquiry Board, "ARIANE 5 Flight 501 Failure," tech. rep., European Space Agency, 1996. J.L. Lions, Chairman of the Board.

[3] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, pp. 13–17, January 1990.

[4] R. Covington, ed., *Formal Methods Specification and Verification Guidebook for Software and Computer Systems; Volume 1: Planning and Technology Insertion*, vol. NASA-GB-002-95. National Aeronautics and Space Administration, July 1995.

[5] J. Crow, ed., *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems; Volume 2: A Practioner's Companion*, vol. NASA-GB-001-97. National Aeronautics and Space Administration, May 1997.

[6] G. C. Gannod and B. H. C. Cheng, "Strongest Postcondition as the Formal Basis for Reverse Engineering," *Journal of Automated Software Engineering*, vol. 3, pp. 139–164, June 1996. A preliminary version appeared in the *Proceedings for the IEEE Second Working Conference on Reverse Engineering*, July 1995.

[7] G. C. Gannod and B. H. C. Cheng, "Using Informal and Formal Methods for the Reverse Engineering of C Programs," in *Proceedings of the 1996 International Conference on Software Maintenance*, pp. 265–274, IEEE, 1996. Also appears in the Proceedings for the Third IEEE Working Conference on Reverse Engineering.

[8] B. H. C. Cheng, "Applying formal methods in automated software engineering," *Journal of Computer and Software Engineering*, vol. 2, no. 2, pp. 137–164, 1994.

[9] R. S. Pressman, *Software Engineering A Practitioner's Approach*. McGraw-Hill, fourth ed., 1997.

[10] E. J. Byrne, "A Conceptual Foundation for Software Re-engineering," in *Proceedings for the Conference on Software Maintenance*, pp. 226–235, IEEE, 1992.

[11] E. J. Byrne and D. A. Gustafson, "A Software Re-engineering Process Model," in *COMPSAC*, ACM, 1992.

[12] E. Yourdon and L. Constantine, *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.

[13] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, pp. 8–24, September 1990.

[14] J. Rushby, "Formal Methods and the Certification of Critical Systems," Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[15] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576–580, October 1969.

[16] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[17] B. H. C. Cheng and G. C. Gannod, "Abstraction of Formal Specifications from Program Code," in *Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*, pp. 125–128, IEEE, 1991.

[18] J. Jeng and B. H. C. Cheng, "Using Automated Reasoning Techniques to Determine Software Reuse," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, pp. 523–546, December 1992.

[19] A. M. Zaremski and J. M. Wing, "Specification Matching of Software Components," in *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[20] J. Penix and P. Alexander, "Toward Automated Component Adaptation," in *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, June 1997.

[21] A. M. Zaremski and J. M. Wing, "Signature Matching: a Tool for Using Software Libraries," *ACM Transactions on Software Engineering and Methodology*, April 1995.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[23] R. H. Bourdeau and B. H. C. Cheng, "A formal semantics of object models," *IEEE Trans. on Software Engineering*, vol. 21, pp. 799–821, October 1995.

[24] Y. Wang and B. H. C. Cheng, "Formalizing and integrating the functional model within omt," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, June 1998.

[25] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[26] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[27] S. Katz and Z. Manna, "Logical Analysis of Programs," *Communications of the ACM*, vol. 19, pp. 188–206, April 1976.

[28] G. C. Gannod and B. H. C. Cheng, "A Formal Automated Approach for Reverse Engineering Programs with Pointers," Tech. Rep. MSU-CPS-97-19, Michigan State University, 1997.

[29] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

[30] W. Landi, "Undecidability of Static Analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, 1992.

[31] J.-J. Jeng and B. H. C. Cheng, "Specification Matching for Software Reuse: A Foundation," in *Proceedings of the ACM Symposium on Software Reuse*, pp. 97–105, 1995.

[32] B. Fischer, M. Kievernagel, and W. Struckmann, "VCR: A VDM-Based software component retrieval tool," in *Proceedings of the ACM Symposium on Formal Methods Application in Engineering Practice*, 1995.

[33] B. Fischer, M. Kievernagel, and W. Struckmann, "Deduction-Based Software Component Retrieval," in *Proceedings of IJCAI '95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, August 1995.

[34] A. Mili, R. Mili, and R. T. Mittermeir, "Storing and Retrieving Software Components: A Refinement Based System," *IEEE Transactions on Software Engineering*, vol. 23, July 1997.

[35] J.-J. Jeng and B. H. C. Cheng, "Reusing Analogous Components," *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[36] A. Quilici, "A Memory-Based Approach to Recognizing Program Plans," *Communications of the ACM*, vol. 37, pp. 84–93, May 1994.

[37] S. R. Tilley, K. Wong, M.-A. Storey, and H. A. Müller, "Programmable Reverse Engineering," *The International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 4, pp. 501–520, 1994.

[38] M. P. Ward and K. H. Bennett, "A Practical Solution to Reverse Engineering Legacy Systems using Formal Methods," in *Proceedings of the Working Conference on Reverse Engineering*, 1993.

[39] I. D. Baxter and M. Mehlich, "Reverse Engineering is Reverse Forward Engineering," in *Proceedings of the Fourth IEEE Working Conference on Reverse Engineering*, IEEE, October 1997.

[40] D. Smith, "KIDS: A Semi-automatic Program Development System," *Transactions on Software Engineering*, vol. 16, pp. 1024–1043, September 1990.

[41] L. L. Jilani, J. Desharnais, M. Frappier, R. Mili, and A. Mili, "Retrieving Software Components That Minimize Adaptation Effort," in *Proceedings of the 12th Automated Software Engineering Conference*, pp. 255–262, Nov 1997.

[42] J. Guttag and J. Horning, *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[43] G. C. Gannod and B. H. C. Cheng, "A Formal Automated Approach for Reverse Engineering Programs with Pointers," in *Proceedings of the Twelfth IEEE International Automated Software Engineering Conference*, pp. 219–226, IEEE, 1997.

[44] C. Rich and R. C. Waters, *The Programmer's Apprentice.* ACM-Press, 1990.

[45] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *IEEE Computer*, December 1996.

[46] G. Sander, "Graph layout through the vcg tool," in *Proceedings of Graph Drawing, DIMACS International Workshop GD'94, Lecture Notes in Computer Science*, vol. 894, pp. 194–205, Springer-Verlag, 1995.

[47] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[48] J. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[49] A. Nerode and R. A. Shore, *Logic For Applications.* Springer-Verlag, 1993.

[50] J. Levine, T. Mason, and D. Brown, *lex & yacc.* O'Reilly & Associates, 1992.

[51] J. Wielemaker, *SWI-Prolog 3.0 Reference Manual.* University of Amsterdam, July 1998.

[52] Y. Chen and B. H. C. Cheng, "Formalizing and automating component reuse," in *Proc. of 9th IEEE Intl. Conference on Tools with Artificial Intelligence*, November 1997.

[53] Y. Chen and B. H. C. Cheng, "Facilitating an automated approach to architecture-based software reuse," in *Proceedings of the 12th International Conference on Automated Software Engineering*, 1997.

[54] Microsoft Corporation, *Microsoft Visual C++ MFC Library Reference, Part 1 & 2*, 1997.

[55] D. Garlan and D. Perry, "Introduction to the special issue on software architecture," *IEEE Transaction on Software Engineering*, vol. 21, April 1995.

[56] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, October 1992.

[57] M. Shaw and D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[58] G. C. Gannod and B. H. C. Cheng, "A formal approach to reverse engineering c programs," Tech. Rep. MSUCPS-TR98-12, Michigan State University, April 1998.

[59] M. A. Weiss, *Algorithms, Data Structures, and Problem Solving with C++*. Addison-Wesley Publishing Company, Inc., 1996.

[60] N. Zvegintzov, ed., *Software Management Technology Reference Guide*. Software Management News Inc., 1994.

[61] M. R. Olsem and C. Sittenauer, "Reengineering technology report (vol. 1 and 2)," tech. rep., Software Technology Support Center, Hill AFB, 1995.

[62] B. Bellay and H. Gall, "A Comparison of four Reverse Engineering Tools," in *Proceedings for the Fourth Working Conference on Reverse Engineering*, pp. 2–11, IEEE, 1997.

[63] N. Zvegintzov, "A Resource Guide to Year 2000 Tools," *Computer*, vol. 30, pp. 58–63, March 1997.

[64] A. W. Brown and K. C. Wallnau, "A Framework for Evaluating Software Technology," *Software*, vol. 13, pp. 39–49, September 1996.

[65] S. Rugaber, K. Stirewalt, and L. Wills, "The Interleaving Problem in Program Understanding," in *Proceedings for the Second Working Conference on Reverse Engineering*, IEEE, 1995.

[66] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Automated Support for Legacy Code Understanding," *Communications of the ACM*, vol. 37, pp. 50–57, May 1994.

[67] S. K. Abd-El-Hafiz and V. R. Basili, "A Knowledge-Based Approach to the Analysis of Loops," *Transactions on Software Engineering*, vol. 22, pp. 339–360, May 1996.

[68] "Xinotech." [Online] Available http://www.xinotech.com/tech-overview.html.

[69] "Imagix 4D." [Online] Available http://www.teleport.com/~imagix.

[70] "The McCabe Visual Reengineering Toolset." [Online] Available http://gate.mccabe.com/visual/reeng.html.

[71] G. C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," in *Proceedings of the third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[72] M. Ward, F. Calliss, and M. Munro, "The Maintainer's Assistant," in *Proceedings for the Conference on Software Maintenance*, IEEE, 1989.

[73] J. Bowen, P. Breuer, and K. Lano, "The REDO Project: Final Report," Tech. Rep. PRG-TR-23-91, Oxford University, 1991.

[74] "Peritus Software Services." [Online] Available http://www.peritus.com/.

[75] W. Kozaczynski and J. Q. Ning, "Automated Program Understanding by Concept Recognition," *Automated Software Engineering*, vol. 1, no. 1, pp. 61–78, 1994.

[76] D. W. Binkley and K. B. Gallagher, "Program Slicing," in *Advances in Computers* (M. Zelkowitz, ed.), vol. 43, Academic Press, 1996.

[77] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller, "Using an Enabling Technology to Reengineer Legacy Systems," *Communications of the ACM*, vol. 37, pp. 58–70, May 1994.

[78] P. Newcomb, "Reengineering Procedural Into Data Flow Programs," in *Proceedings for the Second Working Conference on Reverse Engineering*, pp. 32–38, IEEE, 1995.

[79] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey, "Structural redocumentation: A case study," *IEEE Software*, pp. 46–54, January 1995.

[80] G. C. Murphy and D. Notkin, "Reengineering with Reflexion Models: A Case Study," *Computer*, vol. 30, pp. 29–36, August 1997.

[81] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper, "Formal Program Consdtruction by Transformations– Computer-Aided, Intuition-Guided Programming," *IEEE Transactions on Software Engineering*, May 1991.

[82] P. E. London and M. S. Feather, "Implementing specification freedoms," in *Readings in Artificial Intelligence and Software Engineering* (C. Rich and R. C. Waters, eds.), pp. 285–305, Los Altos, CA: Morgan Kaufman, 1986.

[83] D. S. Wile, "Local formalisms: Widening the spectrum of wide-spectrum languages," in *Program Specification and Transformation*, pp. 459–481, 1987.

[84] M. Ward, "Abstracting a Specification from Code," *Journal of Software Maintenance: Research and Practice*, vol. 5, pp. 101–122, 1993.

[85] T. Bull, "An Introduction to the WSL Program Transformer," in *Proceedings for the Conference on Software Maintenance*, pp. 242–250, IEEE, 1990.

[86] E. Younger, Z. Luo, K. Bennett, and T. Bull, "Reverse Engineering Concurrent Programs using Formal Modelling and Analysis," in *Proceedings of the 1996 International Conference on Software Maintenance*, pp. 255–264, IEEE, 1996.

[87] "Verilog logiscope." [Online] Available http://www.verilogusa.com/log/logiscop.htm.

[88] "Cayenne ensemble." [Online] Available http://www.cayennesoft.com/products/datasheets/ensemsoft.html.

[89] M. T. Harandi and J. Q. Ning, "Knowledge-Based Program Analysis," *IEEE Software*, vol. 7, pp. 74–81, January 1990.

[90] R. A. Slusser, "Advanced Multimission Operations System (AMMOS) Detailed Capabilities Catalog and Adaptation Guide," Tech. Rep. JJ MOSO0020-00-06(JPL D-5140), Jet Propulsion Laboratory - California Institute of Technology, June 1995. Internal JPL Document.

[91] D. F. Miller and L. A. Palkovic, "Space Flight Operations Center User's Guide for Workstation End Users," Tech. Rep. U6 MOSO0088-00-11-04(JPL D-6060), Jet Propulsion Laboratory - California Institute of Technology, June 1994. Internal JPL Document.

[92] E. J. Byrne, "Generating Project-Specific Reengineering Process Models," in *Proceedings of the 6th Annual DoD Software Technology Conference*, Department of Defense, 1994.

[93] N. Dehghani, "Space Flight Operations Center Command Subsystem (CMD) Software Specifications Document," Tech. Rep. SCMD0007-00-02, Jet Propulsion Laboratory - California Institute of Technology, January 1990. Internal JPL Document.

[94] P. Cousot, "Abstract Interpretation," *ACM Computing Surveys*, vol. 28, pp. 324–328, June 1996.

[95] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *Proceedings of the 23rd ACM SIGPLAN Symposium on the Principles of Progamming Languages*, 1996.

[96] E. Y. Wang, *Integrating Informal and Formal Approaches to Object-Oriented Analysis and Design*. PhD thesis, Michigan State University, Department of Computer Science, May 1998.

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

[98] R. S. Arnold and S. A. Bohner, "Impact Analysis - Towards a Framework for Comparison," in *Proceedings of the Conference on Software Maintenance*, pp. 292–301, IEEE, 1993.