AN EMPIRICAL COMPARISON OF STATIC CONCURRENCY ANALYSIS TECHNIQUES

A Dissertation Presented

by

ALBERT T. CHAMILLARD

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1996

Department of Computer Science

© Copyright by Albert T. Chamillard 1996

All Rights Reserved

AN EMPIRICAL COMPARISON OF STATIC CONCURRENCY ANALYSIS TECHNIQUES

A Dissertation Presented

by

ALBERT T. CHAMILLARD

Approved as to style and content by:

Lori A. Clarke, Chair

W. Richards Adrion, Member

George S. Avrunin, Member

Leon J. Osterweil, Member

David W. Stemple, Department Head
Computer Science

DEDICATION

To my parents, for loving and supporting me, and for always showing me I could do anything I set my mind to.

To Chris, Timothy, Nicholas, and Emily, for loving me, believing in me, giving me perspective, helping me find my keys, and making more sacrifices than I did so I could complete this dissertation.

ACKNOWLEDGMENTS

I would like to thank Lori Clarke, my advisor, for all her help during the course of this work. Without her many contributions of both time and lessons about being a good researcher, this dissertation would not have been possible. She also made sure I always remembered her most important lesson - to always be proud of my work.

I would like to thank Lee Osterweil for our many interesting discussions about my work, research, teaching, and life in general.

I would like to thank George Avrunin for his careful review of this dissertation, for the time he took to help me, and for our sometimes loud but always productive discussions about my work.

I would like to thank Rick Adrion for our discussions about my work and teaching.

ABSTRACT

AN EMPIRICAL COMPARISON OF STATIC CONCURRENCY ANALYSIS TECHNIQUES

SEPTEMBER 1996

ALBERT T. CHAMILLARD

B.E.E., GEORGIA INSTITUTE OF TECHNOLOGY

M.Sc., UNIVERSITY OF SOUTHERN CALIFORNIA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lori A. Clarke

Developers of concurrent software need cost-effective analysis techniques to acquire confidence in the reliability of that software. Analysis of concurrent programs is difficult because, in many cases, the patterns of communication among the various parts of the program are complicated and the number of possible communications is large.

One class of techniques that can be used for analysis of concurrent programs is static analysis, which uses compile-time information to prove properties about a program. Given the variety of concurrency analysis tools available, analysts need assistance when selecting tools to use to check a specific program and property. Despite exponential worst-case bounds for most of the techniques, average case analysis times may help differentiate between the techniques in practice. The techniques provide a range of analysis accuracies, but these accuracies have not been formally or empirically quantified. Empirical tool comparisons can therefore provide useful insight into which tool would be most suitable for a given program and property.

The main contribution of the work presented here is the development of a sound methodology for comparing concurrency analysis tools, with a thorough description of the experimental design and constraints, discussion of the issues and tradeoffs involved in developing such a methodology, and valid application of statistical analysis. We apply

this methodology to conduct an experiment to compare a number of concurrency analysis tools. Comparisons are accomplished for analysis time, analysis failures, and analysis accuracy of the tools.

Secondary contributions of the work presented here include development of predictive models and preliminary examination of several "real" programs. We develop, with varying degrees of success, predictive models that may help an analyst estimate the analysis time, analysis failure, and analysis accuracy of each tool given a program and a property to be checked. We also provide a preliminary examination of several "real" programs, including a discussion of the program constructs used in the programs and observations about program characteristics that are likely to affect the applicability of static concurrency analysis tools to these programs.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT vi

LIST OF TABLES xiv

LIST OF FIGURES xvi

Chapter

1. INTRODUCTION 1

2. RELATED WORK 4

2.1 Static Concurrency Analysis Techniques 4

2.1.1 Reachability Analysis 4

2.1.1.1 Reachability Analysis Approaches 5
2.1.1.2 State Space Reduction Approaches 6

2.1.2 Symbolic Model Checking 11
2.1.3 Inequality Necessary Condition Analysis 11
2.1.4 Dataflow Analysis 12
2.1.5 Compositional Analysis 17
2.1.6 Combinations of Techniques 19

2.2 Empirical Work in Software Engineering 21

2.2.1 Flowcharting Experiments 23
2.2.2 Metrics Experiments 23
2.2.3 Reliability Experiments 24
2.2.4 Inspection Experiments 25
2.2.5 Test Data Selection Experiments 26
2.2.6 Static Concurrency Analysis Experiments 27

3. EXPERIMENTAL METHODOLOGY 30

3.1 Concurrent Programs and Program Representations 30

3.1.1 Example Program 31
3.1.2 Program Representations 33

3.1.2.1 Control Flow Graphs 34
3.1.2.2 Finite State Automata 36

3.2 Concurrency Analysis Tools 39

3.2.1 Reachability Analysis 40

3.2.1.1 SPIN..40
3.2.1.2 SPIN + Partial Orders 48
3.2.1.3 TRACC..48

3.2.2 Symbolic Model Checking 50
3.2.3 Inequality Necessary Condition Analysis..54
3.2.4 Data Flow Analysis 56

3.3 Comparison Methodology..58

3.3.1 Program Representations..59
3.3.2 Property Representations..61
3.3.3 Checking for Bias..62
3.3.4 Input Domain..63
3.3.5 Data Comparison..64

4. PROGRAMS AND PROPERTIES FOR THE EXPERIMENT..68

4.1 Cyclic 69
4.2 Divide and Conquer (DAC) 76
4.3 Dining Philosophers 82

4.3.1 Standard Problem (dp)..82
4.3.2 Dining Philosophers with Dictionary (dpd) 85
4.3.3 Dining Philosophers with Fork Manager (dpfm) 89
4.3.4 Dining Philosophers with Host (dph)..89

4.4 Elevator 90
4.5 Gas Station 96
4.6 Hartstone 103
4.7 Memory Management 106
4.8 Ring..113

5. METRICS AND MEASUREMENTS 117

5.1 Metrics..117

5.1.1 Program Metrics 118
5.1.2 Internal Representation Metrics 120
5.1.3 Property Metrics 122

5.2 Measurements 123

6. STATISTICAL ANALYSIS TECHNIQUES 124

6.1 Data Collection Strategy 124
6.2 Checking for Bias Statistically 125
6.3 Preprocessing the Data 127
6.4 Building the Models 128

6.4.1 Linear Regression 129
6.4.2 Logistic Regression 130

6.5 Analyzing the Models 131

6.5.1 Goodness of Fit 131
6.5.2 Residual Analysis 132
6.5.3 Identifying Outliers 133

6.6 Summary of Statistical Analysis 134

7. EMPIRICAL RESULTS 135

7.1 Experimental Environment 135
7.2 Checking for Bias Statistically 136
7.3 Experimental Data 140
7.4 Analysis Time Comparisons 142

7.4.1 Native Input Analysis Times 142
7.4.2 Total Analysis Times 144

7.5 Failure Comparisons 146
7.6 Spurious Result Comparisons 147
7.7 Successful Analysis Case Comparisons 148
7.8 Preprocessing the Data 149
7.9 Predictive Models for Analysis Time 151

7.9.1 SPIN, Never Claims 151

7.9.1.1 Predictive Model for Deadlock 152
7.9.1.2 Predictive Model for Other Properties 155

7.9.2 SPIN, Assertions 156
7.9.3 SPIN+PO 157

7.9.3.1 Predictive Model for Deadlock 157
7.9.3.2 Predictive Model for Other Properties 158

7.9.4 TRACC 159

7.9.4.1 Predictive Model for Deadlock 159
7.9.4.2 Predictive Model for Other Properties 159

7.9.5 SMV 160

7.9.5.1 Predictive Model for Deadlock....160
7.9.5.2 Predictive Model for Other Properties....161

7.9.6 INCA....162

7.9.6.1 Predictive Model for Deadlock....162
7.9.6.2 Predictive Model for Other Properties....163

7.9.7 FLAVERS....164

7.10 Predictive Models for Failures....164

7.10.1 SPIN, Never Claims....166

7.10.1.1 Predictive Model for Deadlock....166
7.10.1.2 Predictive Model for Other Properties....168

7.10.2 SPIN, Assertions....169
7.10.3 SPIN+PO....170

7.10.3.1 Predictive Model for Deadlock....170
7.10.3.2 Predictive Model for Other Properties....171

7.10.4 TRACC....172

7.10.4.1 Predictive Model for Deadlock....172
7.10.4.2 Predictive Model for Other Properties....173

7.10.5 SMV....173

7.10.5.1 Predictive Model for Deadlock....174
7.10.5.2 Predictive Model for Other Properties....175

7.10.6 INCA....176

7.10.6.1 Predictive Model for Deadlock....176
7.10.6.2 Predictive Model for Other Properties....177

7.10.7 FLAVERS....177

7.11 Predictive Models for Spurious Results....178

7.11.1 SPIN, Never Claims....178

7.11.1.1 Predictive Model for Deadlock....178
7.11.1.2 Predictive Model for Other Properties....180

7.11.2 SPIN, Assertions....181
7.11.3 SPIN+PO....182

7.11.3.1 Predictive Model for Deadlock....182

7.11.3.2 Predictive Model for Other Properties....183

7.11.4 TRACC....184

7.11.4.1 Predictive Model for Deadlock....184
7.11.4.2 Predictive Model for Other Properties....185

7.11.5 SMV....186

7.11.5.1 Predictive Model for Deadlock....186
7.11.5.2 Predictive Model for Other Properties....186

7.11.6 INCA....187

7.11.6.1 Predictive Model for Deadlock....188
7.11.6.2 Predictive Model for Other Properties....188

7.11.7 FLAVERS....189

7.12 Validating the Models....190
7.13 Summary....195

8. CASE STUDIES....197

8.1 Programs Considered....197

8.1.1 Border Defense System (BDS)....198
8.1.2 Train Control Program....198
8.1.3 ALSP Common Module (ACM)....199

8.2 Conversion to Control Flow Graphs....199

8.2.1 Task Interactions in Called Procedures....199
8.2.2 Separate Packages....201
8.2.3 Generics....202
8.2.4 Use of Attributes....202
8.2.5 Use of Pragmas....202
8.2.6 Use of Compiler-Dependent Packages....202
8.2.7 Use of Discriminated Types....203
8.2.8 Use of Exception Handlers for Control Flow....203

8.3 Characteristics Affecting Analysis....204

8.3.1 Dynamic Task Allocation....204
8.3.2 Task Interactions Within Exception Handlers....205
8.3.3 Complexity of Individual Tasks....205
8.3.4 Task Types in Complicated Data Structures....206

8.4 Discussion....206

9. IMPROVING PETRI NET-BASED STATIC ANALYSIS ACCURACY 208

9.1 Program Representations 210

9.1.1 Petri Nets 211
9.1.2 Reachability Graphs 213

9.2 Improving Accuracy 214

9.2.1 Enforcing Impossible Pairs 215
9.2.2 Representing Variable Values 220
9.2.3 Choosing Between the Two Techniques 225

9.3 Empirical Results 226
9.4 Conclusions 231

10. CONCLUSION 235

APPENDIX: PREDICTIVE MODELS 243

A.1 Analysis Time Predictive Models 243
A.2 Failure Predictive Models 245
A.3 Spurious Result Predictive Models 246

BIBLIOGRAPHY 248

LIST OF TABLES

Table | Page

7.1 Program Metric Data for Experiment 141

7.2 Property Metric Data for Experiment 141

7.3 Mean Native Input Analysis Times....................142

7.4 Fastest Case Counts, Native Input Analysis Time....................143

7.5 Average Rankings, Native Input Analysis Time....................144

7.6 Mean Total Analysis Times 145

7.7 Average Rankings, Total Analysis Time 146

7.8 Counts for Failures....................147

7.9 Counts for Spurious Results 148

7.10 Successful Analysis Percentages....................149

7.11 Collinear Sets of Metrics 150

7.12 R2 Values for Analysis Time Models....................152

7.13 Deviances and Percents Correct for Failure Models....................165

7.14 SPIN Failure Classification Table for Deadlock....................167

7.15 SPIN, Never Claims, Failure Classification Table 168

7.16 SPIN, Assertions, Failure Classification Table....................169

7.17 SPIN+PO Failure Classification Table for Deadlock 171

7.18 SPIN+PO Failure Classification Table for Other Properties 172

7.19 TRACC Failure Classification Table for Deadlock....................173

7.20 SMV Failure Classification Table for Deadlock....................174

7.21 SMV Failure Classification Table for Other Properties....................175

7.22 INCA Failure Classification Table for Deadlock....................176

7.23 Deviances and Percents Correct for Spurious Result Models....................179

7.24 SPIN Spurious Results Classification Table for Deadlock....................180

7.25 SPIN, Never Claims, Spurious Results Classification Table....................181

7.26 SPIN, Assertions, Spurious Results Classification Table......................................182
7.27 SPIN+PO Spurious Results Classification Table for Deadlock.............................183
7.28 SPIN+PO Spurious Results Classification Table for Other Properties...................184
7.29 TRACC Spurious Results Classification Table for Deadlock 185
7.30 SMV Spurious Results Classification Table for Deadlock....................................186
7.31 SMV Spurious Results Classification Table for Other Properties..........................187
7.32 INCA Spurious Results Classification Table for Deadlock...................................188
7.33 INCA Spurious Results Classification Table for Other Properties.........................189
7.34 FLAVERS Spurious Results Classification Table for Other Properties.................190
7.35 Counts of Fastest Analysis Times..192
7.36 Specific Case Predictions...193
7.37 Effect of Using Predictive Models..194
9.1 Effects of Approach on Petri Nets and Reachability Graphs...................................228
9.2 Program Properties...231

LIST OF FIGURES

Figure Page

3.1 Ada Program for 1 Reader/1 Writer 32

3.2 Example Control Flow Graph 36

3.3 FSA for Control Task, No Variables Modeled 37

3.4 FSA for Control Task, Writer Variable Modeled 39

3.5 PROMELA Program for Readers/Writers Example 42

3.6 Never Claims for no_r1w and no_w1w2 45

3.7 Assertions for no_r1w 46

3.8 Assertions for no_w1w2 47

3.9 Example SMV Input 51

3.10 SMV Specification for no_r1w 53

3.11 Alternate SMV Specification for no_r1w 53

3.12 SMV Specification for no_w1w2 54

3.13 INCA Query for Deadlock 55

3.14 INCA Queries for no_r1w and no_w1w2 56

3.15 QREs for no_r1w and no_w1w2..58

4.1 Never Claim for no_c3c2 70

4.2 Assertions for no_c3c2 71

4.3 SMV Specification for no_c3c2...71

4.4 Alternate SMV Specification for no_c3c2 71

4.5 INCA Query for no_c3c2 72

4.6 Alternate INCA Query for no_c3c2 72

4.7 QRE for no_c3c2 73

4.8 Never Claim for no_c2ss 73

4.9 Assertions for no_c2ss 74

4.10 SMV Specification for no_c2ss 74

4.11 Alternate SMV Specification for no_c2ss 74
4.12 INCA Query for no_c2ss 75
4.13 Alternate INCA Query for no_c2ss...75
4.14 QRE for no_c2ss 76
4.15 Never Claim for no_s1js3j 77
4.16 Assertions for no_s1js3j..78
4.17 SMV Specification for no_s1js3j 78
4.18 Alternate SMV Specification for no_s1js3j 79
4.19 INCA Query for no_s1js3j 79
4.20 QRE for no_s1js3j...79
4.21 Never Claim for no_s3f 80
4.22 Assertions for no_s3f 80
4.23 SMV Specification for no_s3f 81
4.24 Alternate SMV Specification for no_s3f 81
4.25 INCA Query for no_s3f 81
4.26 QRE for no_s3f 81
4.27 Never Claim for no_p1p2 83
4.28 Assertions for no_p1p2 83
4.29 SMV Specification for no_p1p2 84
4.30 INCA Query for no_p1p2 84
4.31 QRE for no_p1p2 84
4.32 Never Claim for no_p2d 86
4.33 Assertions for no_p2d 86
4.34 SMV Specification for no_p2d 87
4.35 Alternate SMV Specification for no_p2d 87
4.36 INCA Query for no_p2d 88
4.37 Alternate INCA Query for no_p2d..88

4.38 QRE for no_p2d..89
4.39 Never Claim for no_omc..91
4.40 Assertions for no_omc..92
4.41 SMV Specification for no_omc..92
4.42 INCA Query for no_omc..93
4.43 Alternate INCA Query for no_omc..93
4.44 QRE for no_omc..93
4.45 Never Claim for no_sdni..94
4.46 Assertions for no_sdni..95
4.47 SMV Specification for no_sdni..95
4.48 INCA Query for no_sdni..95
4.49 QRE for no_sdni..96
4.50 Never Claim for no_c1c2..97
4.51 Assertions for no_c1c2..97
4.52 SMV Specification for no_c1c2..98
4.53 INCA Query for no_c1c2..98
4.54 QRE for no_c1c2..99
4.55 Never Claim for no_c1p2..100
4.56 Assertions for no_c1p2..100
4.57 SMV Specification for no_c1p2..101
4.58 Alternate SMV Specification for no_c1p2..101
4.59 INCA Query for no_c1p2..102
4.60 Alternate INCA Query for no_c1p2..102
4.61 QRE for no_c1p2..102
4.62 Never Claim for no_t3t2..104
4.63 Assertions for no_t3t2..104
4.64 SMV Specification for no_t3t2..105

4.65 Alternate SMV Specification for no_t3t2 105
4.66 INCA Query for no_t3t2 105
4.67 Alternate INCA Query for no_t3t2 106
4.68 QRE for no_t3t2 106
4.69 Never Claim for no_u1u2 108
4.70 Assertions for no_u1u2 108
4.71 SMV Specification for no_u1u2 109
4.72 INCA Query for no_u1u2 109
4.73 QRE for no_u1u2 110
4.74 Never Claim for no_sdu1a 110
4.75 Assertions for no_sdu1a 111
4.76 SMV Specification for no_sdu1a 111
4.77 Alternate SMV Specification for no_sdu1a 111
4.78 INCA Query for no_sdu1a 112
4.79 Alternate INCA Query for no_sdu1a 112
4.80 QRE for no_sdu1a 113
4.81 Never Claim for no_m1m2 114
4.82 Assertions for no_m1m2 115
4.83 SMV Specification for no_m1m2 115
4.84 INCA Query for no_m1m2 115
4.85 QRE for no_m1m2 116
7.1 Plot of Standardized Cnd' Residuals vs Predicted Time 153
7.2 Plot of Standardized MaxTRANS Residuals vs Predicted Time 154
7.3 Plot of Standardized Residuals vs Failures 167
9.1 Example Program 208
9.2 Petri Net 212
9.3 Reachability Graph 214

9.4 Petri Net With Impossible Pairs Represented 217
9.5 Reachability Graph With Impossible Pairs Represented 220
9.6 Boolean Variable Subnet 222
9.7 Petri Net With Variable Subnet Added...224
9.8 Reachability Graph Using Variable Subnet 225

CHAPTER 1

INTRODUCTION

Developers of concurrent software need cost-effective analysis methods to acquire confidence in the reliability of that software. Analysis of concurrent programs is difficult because, in many cases, the patterns of communication among the various parts of the program are complicated and the number of possible communications is large. One class of methods that can be used for analysis of concurrent programs is static analysis, which uses compile-time information to prove properties about a program.

A number of techniques have been proposed for *static concurrency analysis* (i.e., static analysis of concurrent programs). These techniques include: reachability analysis, which generates the state space of the concurrent program and checks the property of interest on that state space; symbolic model checking, which checks the property on a symbolic representation of the state space; inequality necessary condition analysis, which specifies the program and property as a system of integer inequalities and looks for a solution to that system; and dataflow analysis, which checks the property by solving a dataflow problem on a graphical representation of the program. For each of the techniques, one or more tools have been developed to implement the technique.

Unfortunately, there is little information available to help analysts choose between the analysis tools. Most of the static concurrency analysis techniques are NP-complete, leading to exponential analysis times in the worst case. Despite these exponential worst-case bounds, average case analysis times may help differentiate between the techniques in practice. All static analysis tools may produce *spurious results* -- that is, report that a property fails when in fact the cases in which it fails do not correspond to actual program behaviors. Usually, a tool produces a spurious result as a consequence of considering paths that can never be executed in the program (commonly called *infeasible paths*) or of considering aliasing that can never occur in the program. The tools provide a range of

analysis accuracies, but these accuracies have not been formally or empirically quantified. Empirical tool comparisons can therefore provide useful insight into which tool would be most suitable for a given program and property.

The main contribution of the work presented here is the development of a sound methodology for comparing concurrency analysis tools, with a thorough description of the experimental design and constraints, discussion of the issues and tradeoffs involved in developing such a methodology, and valid application of statistical analysis. Fair concurrency analysis tool comparisons are difficult to accomplish given the diverse program semantics and property specification formalisms of the tools. Our methodology includes a process to try to ensure each tool examines the same programs and properties. We also need to guard against introducing bias against one or more of the tools. Our methodology includes recognition of a number of biases our methodology could introduce and statistical testing for these biases. We apply our methodology to conduct an experiment to compare a number of concurrency analysis tools. Comparisons are accomplished for analysis time, analysis failures, and analysis accuracy of the tools.

Secondary contributions of the work presented here include development of predictive models and preliminary examination of several "real" programs. We hypothesize that the behavior of the tools, both in terms of performance and accuracy, is affected by characteristics of the program being analyzed and the property being checked for that program. These characteristics are measured using existing and newly-developed metrics. We develop, with varying degrees of success, predictive models that may help an analyst estimate the analysis time, analysis failure, and analysis accuracy of each tool given a program and a property to be checked. These predictive models are in the form of mathematical equations. We conjecture a scenario in which an analyst calculates the program and property metrics, solves the predictive equations, and selects the tool whose predicted behavior meets their accuracy and time requirements.

To be most useful, the analysis tools need to be applicable to programs of realistic size, containing realistic communication structures. In almost all cases, static concurrency analysis tools have been demonstrated using programs from the concurrency analysis literature. It is unlikely that these academic programs are representative of concurrent programs in general. Most tasks in these programs are relatively small, for instance, and the program constructs used in these programs are relatively simple. We provide a preliminary examination of several "real" programs, including a discussion of the program constructs used in the programs and observations about program characteristics that are likely to affect the applicability of static concurrency analysis tools to these programs.

The remainder of the thesis is organized as follows. Chapter 2 contains a review of the related work in static concurrency analysis and experimental software engineering. Chapter 3 presents the experimental methodology we use for the experiment, illustrating the methodology with analysis of a concurrent program and several properties of interest for that program. Chapter 4 describes the other concurrent programs and properties included in the experiment. Chapter 5 introduces the program and property metrics we use as the variables in the predictive models and the measurements we predict with those models. Chapter 6 describes the statistical analysis techniques we use to check for bias in the experiment and to build the predictive models. Chapter 7 presents our empirical results. Chapter 8 provides the results of our examination of several "real" programs. A method for improving the accuracy of certain kinds of static concurrency analysis is provided in chapter 9. Chapter 10 provides our conclusions and directions for future research.

CHAPTER 2

RELATED WORK

Work related to this dissertation can be separated into two categories. Because we conduct experiments using a number of static concurrency analysis techniques, we survey the static concurrency analysis literature in the first section. Because our work is empirical, the second section contains a discussion of the difficulties that must be addressed in software engineering experiments and a review of prior work in experimental software engineering.

2.1 Static Concurrency Analysis Techniques

Static analysis can be used to check whether a selected property, often called the *property of interest*, holds for a specific program. Numerous techniques for static analysis of concurrent programs have been proposed. The major approaches include reachability analysis, symbolic model checking, integer programming, dataflow analysis, compositional analysis, and combinations of these. In this section we survey these major approaches.

2.1.1 Reachability Analysis

Reachability analysis checks whether a property of interest holds on all executions (or no executions) of a concurrent program by considering all reachable states of the program being analyzed. Theoretical results [Tay83b] have shown that using reachability analysis to answer various analysis questions is NP-complete. Because the best known solutions to NP-complete problems are exponential, Taylor's results imply that, in general, the time and space requirements for this technique are exponential.

Taylor presents complexity results for various analysis questions about synchronization events in concurrent programs [Tay83b]. These analysis questions include determining points of possible synchronization, determining actions that can occur in parallel, and determining errors inherent in the synchronization structure

(deadlock, for instance). Taylor shows that most analysis tasks are NP-complete, even under severe restrictions on program structure. The restrictions include prohibiting branches, loops, and select statements in all tasks, or prohibiting branches and loops in the tasks and only allowing one task to have entry calls on a given entry. It is clear that a variety of important questions in static analysis of concurrent programs are intractable; indeed, Taylor points out that, even when feasibility of program paths is ignored, the problems only become tractable when enough restrictions are applied to make a system fully deterministic.

2.1.1.1 Reachability Analysis Approaches

The set of reachable program states used in reachability analysis can be generated using a variety of program representations, including flow graphs [Tay83a, YTF+89] and Petri nets [Pet77, SC88, DCN95].

Taylor's algorithm [Tay83a] implements a graph-based approach for reachability analysis. The algorithm provides a means for checking properties of interest using a flow graph model of the program to generate the set of reachable states. Using a program call graph to mark units (tasks) that can directly or indirectly perform a tasking activity, Taylor defines a *concurrency state* as an ordered tuple of task state nodes. To generate the set of reachable states, a successor function is used to generate the successor states from each concurrency state. The resulting graph is called the Concurrency History Graph (CHG). A complete concurrency history of a program is defined as all non-loop paths through the concurrency states of the program. Properties of interest are checked on the CHG.

Because the size of the CHG often grows exponentially in the number of tasks in the program (commonly called *state-space explosion*), Taylor suggests parceling the analysis by connected components. Analysis can be performed on each connected component, with the results of these analyses combined in the global reachability analysis. Taylor also discusses several problems associated with static concurrency analysis techniques;

these problems include imprecision caused by aliasing and delay statements, as well as difficulties analyzing programs containing dynamic task creation.

Shatz and Cheng [SC88] implement a Petri net-based approach for reachability analysis of Ada programs. An Ada program is converted to a Petri net using a translation table of Ada constructs to Petri net building blocks. A reachability graph is generated from the Petri net, where each node in the graph represents a reachable marking of the net and each arc in the graph represents the firing of a single transition. Shatz and Cheng check properties of interest about states of the program using the generated reachability graph.

A variety of methods can be used to check properties of a reachability graph. For some properties, examination of each state is sufficient to check the property. For others, information about the path to each state is required; these properties can be checked using dataflow analysis or model checking. Clarke et al [CES86] present a model checking technique for checking properties on a reachability graph. Each state is assigned the set of atomic propositions true in that state. The property of interest is expressed in Computation Tree Logic (CTL), a propositional, branching-time temporal logic. The technique works through the reachability graph in stages, processing all subformulas of length 1, then length 2, and so on up to the length of the property formula. In each stage, each state in the reachability graph is marked with the subformulas that are true at that state. After all stages have been completed, the property holds if and only if for each state the property formula is true in that state. Proving properties using this technique requires O(length of formula * (# states + # state transitions)) time.

2.1.1.2 State Space Reduction Approaches

To combat the state-space explosion problem in reachability analysis, various approaches have been suggested to reduce the size of the reachable state space. The approaches discussed below attempt these reduction in two different ways - by reducing the program model from which the reachability graph is generated or by reducing the

reachable state space as it is generated. We note that, for all the state space reduction techniques, in the worst case the size of the reachable state space remains exponential in the number of tasks.

Long and Clarke [LC89] suggest using Task Interaction Graphs (TIGs) as a reduced program representation that retains task interaction information. The TIG consists of a finite set of nodes, $N = \{n_i\}$, and a finite set of directed edges, $E = \{e_i\}$. Each node n_i represents a maximal region of sequential code, and each directed edge represents a task interaction (either the start or end of an entry call or an accept). The set of nodes includes a single start node and a set of terminal nodes for the TIG. There is an edge from n_i to n_j if and only if the task can potentially participate in the task interaction represented by the edge, causing the task to exit the sequential region represented by n_i and enter the sequential region represented by n_j. Use of TIGs as the program model results in reduced representations of the reachable state space, thereby increasing the size of the programs that can be analyzed.

A few analysis techniques have used TIGs as the underlying program model. The Concurrency Analysis Tool Suite (CATS) [YTF+89] provides an analysis toolset for concurrent programs. CATS uses a graph-based model of the program with tasks modeled as TIGs. A Task Interaction Concurrency Graph (reachability graph) is generated from the set of TIGs of the program. The toolset can be used to evaluate assertions about sequences of task interactions by performing temporal logic assertion checking on the reachability graph. The toolset separately checks for deadlock in the reachability graph.

TIGs can also be used in a Petri net-based approach to reachability analysis. Dwyer et al [DCN95] generate a Petri net model of the program (called a TIG-based Petri Net, or TPN) using TIGs for each task. Property predicates for properties of interest can be defined and checked at each state in the reachability graph generated from the TPN. Using TPNs reduces the size of the enumerated state space, sometimes at the expense of

increased analysis costs at each state. For the programs examined in [DCN95], the compaction in reachability graph size is two orders of magnitude. Essentially, using a TPN trades space for time; the reachability graph is smaller than for a control flow-based technique, but checking the property predicate at each reachable state can be more expensive than for a control flow-based technique. For the examples examined in [DCN95], the total cost of analysis was less using TPNs.

As an alternative to reducing the representation of the program model, the set of reachable states can potentially be reduced during reachability graph generation. A variety of techniques have been proposed for these types of reductions; the major approaches are discussed below. We note that most of these techniques apply independently of the choice of program model.

One major contributor to the state space explosion is the consideration of all possible interleavings of potentially concurrent activities in the program. For certain kinds of properties, Valmari introduces an approach called *stubborn sets* [Val90], in which the effects of interleavings are reduced through consideration of a subset of each set of possible interleavings. A stubborn set is defined as a set of state transitions that can affect each other. More precisely, any disabled transition in the set can only be enabled by a transition in the set, the transitions in the set are independent of transitions outside the set, and at least one of the transitions in the set is enabled. A linear algorithm exists for finding "almost" optimum stubborn sets for a given state, and a quadratic algorithm can be used to find optimum stubborn sets for that state.

Using this technique to generate the next states from the current state, only the enabled transitions in the stubborn sets are used; in ordinary reachability analysis, all enabled transitions in the system are used to generate the next states. Using stubborn sets, if the number of enabled transitions for a particular state is smaller than the number of enabled transitions in the system, the state will have fewer next states. This can, in turn, lead to a reduction in the size of the reachable state space. Valmari proves that the

stubborn set method preserves Linear Temporal Logic (LTL) properties of the state space, as long as the LTL operators "next state" and "previous state" are not used. The LTL formulas specifying the property of interest must be known before the state space is generated, since they are used during state space generation. After the reduced state space is generated using stubborn sets, the LTL formulas can be checked on the reduced state space.

The partial orders approach of Godefroid and Wolper attempts to reduce the effects of interleavings on the size of the reachable state space through the use of *sleep sets* [GW91]. During the generation of the reachable state space, only one instance of equivalent interleavings are considered at each state, where equivalence depends on the property of interest. To accomplish this, the technique considers traces through an automaton representing the concurrent program. A dependency relation on transitions in the system is developed and this relation is used in conjunction with the set of transitions to explore only one interleaving for each possible trace of the system. This restriction to one interleaving tends to reduce the size of the generated reachable state space significantly.

The partial orders technique can be combined with existing reachability analysis techniques to check properties on a reduced reachable state space. Any property to be checked must be specifiable as a finite state automaton, since the automaton for the property is combined with the program automaton to perform the analysis. Because interleavings that could affect the property being checked are not removed by the technique, the reductions are property-preserving.

Rather than explicitly trying to eliminate the effects of interleavings, McDowell tries to reduce the size of the reachable state space by combining sets of related states into single states [McD89]. If two tasks are executing the same sequence of statements, it may not be necessary to distinguish between them. Similarly, if several tasks are executing the same sequence of statements, it may not be necessary to know how many

tasks are at a particular statement; it may be sufficient to know that at least one task is at that statement. McDowell uses a CHG [Tay83a] to represent the reachable state space, noting that several equivalent CHGs are possible for a given program when his mapping for identical tasks is used. Two CHGs are defined to be equivalent if they contain the same synchronization and parallel access anomalies. Taylor's reachability analysis [Tay83a] generates a CHG containing all reachable states; McDowell's technique attempts to generate an equivalent CHG containing fewer states. This technique is only useful for programs with sets of identical tasks; it is not clear how often this program structure occurs in practice.

Murata et al detect deadlock in Petri net models of Ada programs using structural properties (invariants) of the Petri nets [MSS89]. In some cases deadlock can be detected without generating the reachable state space; these are called *inconsistency deadlocks*. In other cases the reachable state space must be generated to detect the deadlock; these are called *circular deadlocks*. T-invariants are employed to support the deadlock checking, where a T-invariant represents the number of times each transition in the Petri net fires to move a Petri net from a given marking back to that marking. This technique specifically excludes deadlock caused by loop statements, and these deadlocks will go undetected.

To detect inconsistency deadlocks, a set of linearly independent T-invariants, called the Ada T-invariant, is calculated. If this set does not exist, or if some transition is not in any of the T-invariants composing the set, then the transition is not on any executable path, and the program has at least one inconsistency deadlock. To detect circular deadlocks, circular directed paths are identified where task segments on the paths start and end with communication transitions. Existence of at least one such path is a necessary (but not sufficient) condition for actual deadlock in the program. To identify these paths, T-invariants are used for comparison of transition firing counts to guide reachability graph generation. If such a path is identified in the resulting reachability graph, a circular deadlock is reported. Because a program can have multiple Ada T-

invariants, a reachability graph is generated for each Ada T-invariant. While the total number of states in the set of resulting reachability graphs is less than the total number of reachable states for the single example given in the paper, it is not clear whether this will commonly be the case in practice.

2.1.2 Symbolic Model Checking

Symbolic model checking techniques [BCM+90] represent the program state space symbolically rather than explicitly. With this technique, the state transition relation for the program to be analyzed is modeled using Ordered Binary Decision Diagrams (OBDDs). An OBDD is a directed acyclic graph with a strict total order on the occurrence of variables on any path from the root to any leaf in the OBDD. OBDDs can be used to represent arbitrary boolean functions. The program to be analyzed is encoded as a set of variables and operations on those variables, and this encoding is then used to generate an OBDD model of the program. The property of interest is specified in the temporal logic Computation Tree Logic (CTL), and a least fixed point algorithm is used to build an OBDD that symbolically represents the set of states in which the property holds and to check whether all reachable states in the program satisfy the property of interest.

Symbolic CTL model checking is known to be PSPACE-complete [McM93]. In the worst case, the number of iterations required to reach a fixed point can be exponential in the number of variables in the OBDDs. Burch et al [BCM+90] note that the size of the OBDD is extremely sensitive to the variable ordering, so a poor choice for the variable ordering can degrade the performance of the technique significantly.

2.1.3 Inequality Necessary Condition Analysis

The Inequality Necessary Condition Analysis technique [ABC+91, CA95] avoids representing the state space of the program altogether. The system is represented as a set of communicating finite state automata. Transitions in a given automaton represent internal actions of that automaton, initiation of a communication with another automaton

(an entry call or accept), or a communication error (such as hanging waiting for a communication that never occurs). Flow equations are created for each state in the set of automata to specify that the number of times a given state is entered is equal to the number of times that state is exited. Communication equations are generated for each communication channel (entry) in the system to specify constraints on well-formed behavior. For example, these equations enforce the constraint that the number of times the accepting automaton transitions on accepts of this entry is equal to the total number of times the callers transition on entry calls for this entry. Restriction inequalities are produced to disallow certain impossible behaviors. For example, a calling and accepting automaton can not both hang waiting for a communication on the same entry, so restriction inequalities are produced to prohibit this. Property inequalities are derived from a specification of the negation of the property of interest. Integer linear programming techniques are then used to check for an integer solution to the set of flow equations, communication equations, restriction inequalities and property inequalities for the system. If there is no integer solution, the necessary conditions for the negation of the property are not met, and the property must hold.

Integer linear programming is known to be NP-complete, and thus in the worst case this technique can require exponential time to find a solution or determine that none exists.

The technique described in [ABC+91] can verify some interesting properties, such as freedom from deadlock, but can not be used to check liveness properties or properties involving the relative order of events in a system trace. This technique was subsequently extended to handle both infinite traces and properties involving relative event orders [Cor92]. Including information about certain types of infeasible synchronization events and certain program variable values [Cor93] in the set of inequalities has been proposed as one way to reduce the size of the set of inequalities.

2.1.4 Dataflow Analysis

Dataflow analysis has been used extensively in compiler construction to recognize opportunities for optimizations and has also been used for anomaly detection in sequential and concurrent programs.

Taylor and Osterweil [TO80] use dataflow analysis to determine the presence or absence of errors specified as anomalous or illegal sequences of events in a concurrent program. The analysis is performed on a Process Augmented Flowgraph (PAF), which is constructed by connecting the flowgraphs of each process with special edges indicating all synchronization constraints. Taylor and Osterweil specify algorithms for detecting a variety of data flow and synchronization anomalies. Each algorithm is specified with a definition of the gen and kill functions, with the algorithms using standard (or in some cases slightly modified) AVAIL and LIVE procedures. A technique is also described for parceling the analysis by creating summary information for each task, then substituting this information at task schedule and wait nodes.

Long and Clarke [LC91] refine the anomaly detection techniques of [TO80], presenting a technique for dataflow analysis of rendezvous model concurrent programs to detect anomalies specified as patterns of events. Tasks are broken into task fragments and summary information is calculated on each fragment. The order of calculation is given by a rendezvous graph, which is analogous to a call graph with task fragments treated as procedures. Each entry call and accept is interpreted as a procedure call. For each fragment, a pessimistic (minimal) gen and an optimistic (maximal) kill are calculated for the input/output of the fragment, and the gen/kill information is used to solve AVAIL and LIVE (or, more generally, forward and backward flow) problems. Calculating the minimal gen and the maximal kill gives the "coarsest" gen/kill information possible about the fragment, which is required because the calling context is unknown at the time of gen/kill calculation. Called fragments are analyzed before their callers so that summary information can be used during calculation of the summary information for those fragments that invoke the given fragment; to improve accuracy at

this point, the technique accounts for formal parameters in entry calls (explicit procedure calls) and local variables in the scope of accepts (implicit procedure calls). For the technique described, two assumptions are made about the structure of the program being analyzed; each entry has only one accept, and the rendezvous graph is acyclic. Entries with multiple accept statements would require multiple versions of the summary information, one for each accept statement. The calls each accept services could be determined to refine the analysis, though this refinement is NP-complete. Alternatively, as in the approach suggested by Long and Clarke, worst case analysis of the summary information can be used, with potentially less accurate results. Assuming an acyclic rendezvous graph seems reasonable, since a cycle in the graph (assuming no recursive procedures) generally, though not always, indicates the potential for deadlock.

Reif and Smolka [RS90] consider an asynchronous message passing (not rendezvous) model of communication in concurrent programs they analyze. Initially, static communication patterns are assumed; in other words, channel arguments to message primitives are constants. The system state is described as the state of each process in the program, the value of each variable, and the contents of each communication channel. Communication channels can either be First In-First Out (FIFO) or unordered. The technique is subsequently extended to dynamic communication, in which channel arguments to message primitives are expressions.

With their technique, each process in the program is modeled by a process flow graph, which is a control flow graph in which only assignment, transmit, receive, and no-op nodes are included. The program is modeled with an Event Spanning Graph (ESG). An ESG is composed of the spanning tree for each process flow graph and a set of message links, which are ordered pairs of transmit/receive statements specifying the same communication channel. Restrictions on the ESG are that each node must be reachable in its process flow graph and that at least one transmit node matches each receive node. The existence of an ESG is a necessary condition for all nodes to be reachable; if an ESG does

not exist, at least one node in the set of process flow graph nodes is unreachable. Reif and Smolka provide a linear algorithm for creating the ESG or recognizing that it does not exist.

Reif and Smolka use dataflow analysis to determine the possible values of program expressions at each node in the ESG. To solve the dataflow problem on the ESG, input and output predicates are computed for each node, and a message predicate for each channel is computed to record all messages sent over that channel. Nodes in the ESG are visited in topological order, because very often convergence is obtained quickly when topological order is used. At convergence, an estimate of the input and output variable values for each node in the ESG is available.

The above dataflow analysis techniques can be applied to check a variety of properties. In contrast, some dataflow analyses have focused on a single property of interest, usually deadlock. Masticola and Ryder [MR91] present a polynomial time algorithm for deadlock detection in Ada programs. The program is modeled as a sync hypergraph, with nodes for rendezvous statements, control edges for control flow between statements in each task, and synchronization edges for possible rendezvous between tasks. A sync hyperedge, connecting an entry call node to the begin and end of the accept body, is used to force the entry caller to wait until the accept body is executed. A Can't Happen Together relation (CHT) is calculated on the sync hypergraph, and this relation is used with the sync hypergraph to detect cycles in the graph corresponding to potential deadlocks.

The CHT relationship [MR93] identifies pairs of statements that cannot execute concurrently. CHT can be calculated in polynomial time through iterative application of a set of predefined refinements. Refinements are applied until a fixed point is reached, meaning that no refinement can add a new node to the set of CHT nodes. The CHT set generated by the technique is not guaranteed to be perfect (to contain all nodes that Can't

Happen Together). In an experiment on 127 programs, at least 95% of the CHT pairs were found in 90 of the 115 programs that had CHT pairs.

Rather than applying dataflow analysis for a single property, Dwyer and Clarke [DC94] present a more general technique that uses polynomial time algorithms to check whether or not a user-specified sequence of program events occurs on all paths or any path in the concurrent program. The program is modeled as a Trace Flow Graph (TFG), a conservative representation of program event traces. Nodes in a TFG represent control states of individual tasks. There are three kinds of edges in a TFG: control flow edges, which represent program events local to a task; communication edges, which are used to capture the communication predecessor in the task with which a given task is engaging in a communication; and May Immediately Precede (MIP) edges, which are used to explicitly capture potential interleaving of asynchronously executing program events. The property of interest is specified as a Quantified Regular Expression (QRE), which is converted to a deterministic finite automaton called the Property Automaton (PA). To solve the dataflow problem, states of the PA are propagated through the TFG using an iterative worklist algorithm. The state propagation requires $O(|PA|*|E|)$ time, where $|PA|$ is the number of states in the property automaton and $|E|$ is the number of edges in the TFG. $|E|$ is $O(|N|^2)$, where $|N|$ is the number of nodes in the TFG. To check whether the property holds, the PA states that are possible at program termination are compared to the accepting states of the PA. For an all-paths property, the possible PA states at program termination must be a subset of the accepting states of the PA for the property to hold. For an any-path problem, the intersection of the possible PA states at program termination and the accepting state of the PA must be non-empty for the property to hold.

A major strength of the approach described in [DC94] is the flexibility an analyst has when applying accuracy-improving techniques to control the tradeoff between efficiency of the analysis and accuracy of the analysis results. The TFG can be refined prior to analysis, using program- and property-specific information, to improve analysis efficiency

with a potential gain in accuracy. Dwyer and Clarke describe two such refinements; one refines the TFG by eliminating representation of events not contained in the PA and the other removes certain MIP edges based on communication events in the TFG. In addition to TFG refinements, *feasibility constraints*, based on the program and programming language, can be used during the analysis to improve analysis accuracy. Feasibility constraints encode necessary conditions for paths in the TFG to correspond to executable paths in the program. The feasibility constraint described by Dwyer and Clarke enforces a local event ordering constraint by including information about control flow orderings in single tasks. Feasibility constraints are included in the analysis by forming the product automaton of the PA and all feasibility constraints; the resulting automaton is used for the state propagation described above.

Empirical results are provided for three programs, where combinations of the refinements and feasibility constraints described above are used to check a variety of properties on the programs. For the programs and properties examined, the actual performance is quadratic in the number of TFG nodes, rather than the cubic theoretical upper bound.

Naumovich et al [NCO96] conduct a case study using FLAVERS to verify protocol behavioral requirement specifications for two communication protocols. A variety of feasibility constraints are used to verify the specifications. *Variable automata* are used to model selected variables, *task automata* are used to enforce the control flow in selected processes, and customized feasibility constraints are also used. Properties are verified for the three-way handshake connection establishment protocol and the alternating bit transfer protocol. The case study also shows how assumptions about the operating environment of the software can be incorporated into the analysis, using message losses to illustrate the technique.

2.1.5 Compositional Analysis

To control the exponential cost of most of the techniques described above, it may be possible to analyze portions of the system being analyzed, then combine the results for the global analysis [Tay83a, YY91, CA94, CK95]. Several approaches for performing this compositional analysis are described below.

In the compositional reachability analysis of Yeh and Young [YY91], reachability graph representations for individual components are derived, then the representations are hierarchically composed to generate a global reachability graph. Individual components are described as process algebra expressions. The process algebra expressions can be transformed into process graphs, which are essentially reachability graphs with additional algebraic structure. The process graph for multiple components is generated using the algebraic product operation on component process graphs. To reduce process graph sizes, it may be possible to verify that the implementation satisfies a simpler specification (by finding a bisimulation between them); the process graph for the implementation can then be replaced with the process graph for the simulation. Reducing and composing process graphs is repeated iteratively until the system process graph has been generated, at which point the property of interest can be checked. Yeh and Young note that applicability of these techniques depends on clean modular decomposition of the system and the ability to describe complicated implementations with simpler specifications of their behavior.

Noting that proving equivalence between two processes (implementation and specification, for instance) is required for compositional analysis and may require comparison of potentially large reachability graphs, Corbett and Avrunin [CA94] present a method for equivalence checking of two processes without enumeration of the states of the processes. The component processes of each process are used to generate a set of necessary conditions for the existence of a system trace showing that the equivalence does not hold. The necessary conditions are expressed in the form of a set of integer linear equations. Integer linear programming techniques are then used to search for a solution to the set of equations. If no solution exists, the necessary conditions can not be satisfied,

and the processes are equivalent. The analysis is conservative, so it may be unable to prove equivalence of two equivalent processes, but will never prove equivalence of two inequivalent processes. The technique is only applicable to deterministic, divergence-free processes. A process is deterministic if the set of actions in which a process has engaged completely determines the set of actions in which it can engage in the future; a process is divergence free if it can not engage in an unbounded number of internal actions, thereby ignoring external requests indefinitely. The technique has been successfully applied to several large problems, though in the worst case solving the set of integer linear equations can require exponential time.

2.1.6 Combinations of Techniques

Since each of the techniques described above exhibits both strengths and weaknesses, a natural step is to consider how multiple techniques can be combined to take advantage of the strengths and avoid the weaknesses of each.

Young and Taylor [YT88] propose combining reachability analysis and symbolic execution to improve the accuracy of reachability analysis for less cost than full symbolic execution. Conceptually, the reachability graph provides path selection for the symbolic execution, while the symbolic execution provides pruning of the reachability graph through elimination of infeasible paths. When the techniques are used in isolation, every path in a symbolic execution of the program corresponds to a path in the reachability graph. The reverse is not true, since the reachability graph can contain infeasible paths, which are not included in symbolic execution paths.

The techniques can be combined in both a serial and an interleaved manner. In a serial application, reachability analysis is performed to mark which reachable states are "interesting". Symbolic execution is then performed, where any "interesting" states encountered are also marked "feasible". The analysis results only include "interesting, feasible" states. We note that the entire reachable state space is always generated in a serial application. In an interleaved application, reachability analysis is performed to

mark "promising" states until some criteria is met; for instance, until a state of interest is discovered or a certain number of new states have been generated. Symbolic execution is then run through the "promising" states from the reachability analysis. The two techniques are applied in an alternating fashion until the analysis is complete. In the interleaved application, the symbolic execution provides pruning of infeasible paths as the reachable state space is generated; the reachable state space generated is more accurate than in general reachability analysis, and is also potentially smaller if the program contains one or more infeasible paths.

Several methods are proposed to allow scaling of this combined technique. To help control the combinatorial explosion in the size of the reachable state space, biconnected components can be analyzed separately with the results then combined into a global result, weak monitors can be used to parcel components of the system into modules to be analyzed separately, and heuristic search can be used to guide partial exploration of the state space. Use of heuristic search invalidates the guarantee that the combined technique will detect all possible errors (i.e., the technique is no longer conservative).

Cheung and Kramer [CK94] suggest combining reachability analysis with dataflow analysis. These techniques are considered to be complimentary because reachability analysis provides an exhaustive analysis of the program states but carries an exponential complexity, while dataflow analysis provides a tractable, but more approximate, analysis of the program. Dataflow analysis is applied in the early stages of development, when the design is unstable and an approximate technique is sufficient. Reachability analysis is applied in later stages, when stronger assurances of correct program behavior are required. We note that the combination of techniques as described is not as tightly coupled as the combined technique described in [YT88]. There is no information sharing between the two techniques, so neither technique is used to improve the accuracy or reduce the cost of the other. More correctly, the combined technique proposed in [CK94]

consists of selecting the appropriate analysis technique based on the development phase, rather than a synergistic combination of the two techniques.

2.2 Empirical Work in Software Engineering

While some empirical work has been and is being performed in software engineering, the volume, and often the quality, of such work is lacking compared to other scientific disciplines. To demonstrate the lack of empirical work in computer science, Tichy et al [TLP+95] classify 400 research articles based on the amount of empirical work contained in each. The computer science articles are extracted from refereed journals, the 1993 SIGPLAN Conference on Programming Language Design and Implementation (refereed conference), and a random sample of 50 articles drawn using the INSPEC data base. The journals of Neural Computation (NC) and Optical Engineering (OE) are used for comparison. Of the papers in software engineering presenting new methods that would require experimental validation, over 50% contained no experimental validation whatsoever. In contrast, of the similar papers in NC and OE, only 15% and 12%, respectively, lacked experimental validation. Conversely, the fraction of these papers in NC and OE that devoted 20% or more space to experimental validation was almost 70%, while only 20% of the corresponding software engineering papers devoted as much space to validation. These results seem to demonstrate a lack of empirical work in software engineering, though this has been disputed. One factor that may affect these results is that many software engineering techniques deal with human behavior (i.e., code understanding, effectiveness of design methodologies, etc.), while the experiments presented in NC and OE probably did not use human subjects. Osterweil and Clarke call for more empirical work in software engineering, both in the form of small, repeatable experiments and larger case studies on complex systems [OC92].

Fenton et al [FPG94] note that many research findings present new methods with a theoretical analysis of the benefits, but no empirical evaluation to quantify the benefit. They also point out that a large number of the experiments that are performed are poorly designed. In addition, most experiments are conducted on "toy" programs -- programs that are so small they can not be considered to be a representative sample. For example,

Vessey and Weber consider 9 experiments on structured programming [VW84]; four of the experiments consider programs of 10 to 25 lines of code, three consider programs of 26 to 57 lines of code, one considers programs of 46 to 85 lines of code, and one considers programs of 25 to 225 lines of code. Fenton et al also indicate that many experiments use statistical methods incorrectly.

The problems discussed above are often caused by the difficulties facing an experimenter in software engineering. Basili et al [BSH86] indicate the range of these problems. There are wide variations in the environments in which software engineering techniques are applied; desired costs, quality goals, personnel experience, the problem domain, and other constraints can all affect the applicability of a certain technique. Designing an experiment to account for these many variations is difficult, but is necessary if the experimental results are to be generalized. Individual performance can also vary widely, so the actual individuals used in an experiment are a critical factor in the generalizability of the experimental results. Precisely stating the goals of an experiment is a non-trivial task, particularly when addressing areas that do not have commonly accepted definitions, like software quality. Experimental results must be carefully quantified, based on the sample used and how well it represents the set of environments to which the results are to be generalized.

Basili and Weiss [BW84] point out additional difficulties with conducting software engineering experiments. These problems include the fact that there is often a large number of potentially confounding factors that can affect the results of the experiment and the prohibitive expense of attempting controlled studies in an industrial environment with medium or large scale systems. They also note that timely data collection and validation is important. Unmeasured data cannot be accurately recaptured, and without validation, as much as 50% of the data that is collected may be erroneous.

Pfleeger 94 [Pfl94] also points out that exerting control over the independent variables (i.e., those that can affect the truth of the hypothesis) to do a formal experiment

can be impossible or prohibitively expensive. In addition, experimenters often must measure the factor of interest (quality, for example) indirectly; selecting the appropriate indirect measures (number of defects, for instance) can be particularly difficult.

Despite the pitfalls facing an experimenter in software engineering, a number of software engineering experiments have been conducted. We briefly survey a sample of experiments from a number of areas of software engineering and discuss the experiments related to testing and analysis in greater detail.

2.2.1 Flowcharting Experiments

A flowchart can be used to express a high-level definition of a solution to some problem. A flowchart consists of boxes corresponding to operations and alternatives in the program, with edges connecting the boxes to reflect potential flow of control from one box to the next. Flowcharts have often been used as graphical representations of computer programs.

Shneiderman et al [SMM+77] conducted a series of five experiments to determine the utility of detailed flowcharts in program composition, comprehension, debugging, and modification. Shneiderman et al conclude from the results of these five experiments that flowcharts do not contribute to program composition, comprehension, debugging, and modification. Scanlan [Sca89] investigated a related set of hypotheses, namely that structured flowcharts take less time than pseudocode to comprehend, produce fewer errors in understanding, give students more confidence in their understanding, reduce time spent answering questions, and reduce the number of times students look at an algorithm. On the basis of these experiments, Scanlan concludes that flowcharts do have a positive, statistically significant effect.

2.2.2 Metrics Experiments

Program metrics have been proposed as a means of measuring various characteristics of programs, such as program quality. Example metrics include Halstead's software science metrics [Hal77] and McCabe's cylomatic complexity [McC76]. The experiments

surveyed in this section examine the value of a variety of metrics as predictors of certain program characteristics or examine the relationships among metrics.

Li and Cheung [LC87] classified 31 different metrics and examined the relationships among them. While some metric pairs have low correlations, the more conventional metrics are highly correlated, with Lines Of Code being as useful as other, more complicated, metrics for measuring program complexity. Compton and Withrow [CW90] explored how well the presence of predelivery defects predict postdelivery defects and how well program complexity measures predict defect density. The empirical data revealed that packages with predelivery defects detected had a postdelivery defect density (defects/SLOC) six times as large as those with no predelivery defects detected. Porter and Selby [PS90] conducted an experiment using metrics to classify programs in a classification tree according to some user-specified property (fault-prone, change-prone, etc.). The classification tree can be used to identify components (in other systems) that share the same property. This latter work is noteworthy because of the realistic programs used, the thorough description of experimental design and results, and the careful use of statistical analysis on the experimental data.

2.2.3 Reliability Experiments

Software reliability models typically use data about the past performance of a program to estimate the future reliability of the program [Lit91]. For example, Shooman [Sho75] developed software reliability models using data from three operating systems and calculated the model constants using data from 17 additional programs. Iannino et al [IMO+84] propose a set of criteria for comparing the various software reliability models that have been developed. Musa and Okumoto [MO84] used regression analysis on 15 sets of failure data to perform a model-independent comparison of the use of execution time and calendar time in reliability models. They discovered that models using execution time will almost always be superior to those using calendar time. The

experiments surveyed in this section examine n-version programming (a reliability improvement technique) and analyze reliability data from large operational systems.

Avizienis and Kelly [AK84] conducted an experiment using n-version programming, in which multiple versions of code meeting a given specification are independently developed to improve the reliability of the system. Because the effectiveness of n-version programming is based on the independence of the multiple versions, Knight and Leveson [KL86] conducted an experiment to test this hypothesis. They conclude that dependent errors do exist, and these must be considered when calculating the effects of n-version programming. Data from another experiment [Dun86] also indicated that the independence assumption requires further investigation. To help determine the cause of operating system failures, Iyer and Rossetti [IR84] analyzed reliability data from a large operational system. They discovered that the level of interactive processing on the system had a larger effect on operating system failures than CPU execution rate.

2.2.4 Inspection Experiments

The use of software inspections has been proposed as a cost-effective technique for discovering errors in specifications and code. The experiments surveyed here examine the feasibility and effectiveness of inspections.

Porter and Votta [PV94] conducted an experiment using different defect detection methods for inspections of software requirements. They show that a Scenario-based method has a higher defect detection rate than other methods. Schneider et al [SMT92] also examined defect detection methods for requirements and found that replicating the inspection process (N-fold inspections) yielded increased fault detection. Russell [Rus91] relates experiences conducting inspections in a large-scale, industrial setting, and Porter et al [PST+95] provide a status report of an ongoing experiment using inspections in a large scale software development.

2.2.5 Test Data Selection Experiments

Numerous techniques have been proposed for selection of test data and various criteria have been introduced for measuring how well the generated test data "covers" the program. A number of experiments have been conducted to determine how the data selection techniques and coverage criteria perform and how they compare in practice.

Duran and Ntafos [DN84] experimentally examined the effectiveness of random test data generation. The results of the random testing were compared to those based on a form of partition testing called path testing. They concluded that random testing is slightly weaker than path testing. Duran and Ntafos also determined how well randomly generated test data covered each of five programs, using a variety of coverage criteria. On average, random testing yielded a high level of segment and branch coverage, but less coverage for the other criteria. Unfortunately, neither experiment quantified the statistical significance of the results.

Basili and Selby [BS87] examined the effectiveness of code reading, functional testing (equivalence partitioning and boundary value analysis) and 100% statement coverage in terms of fault detection effectiveness, fault detection cost, and classes of faults detected. Basili and Selby found that the number of faults observed depends on the program type, but make no statements about which techniques seem better suited for which program types. Their data analysis uses statistically valid techniques, and Basili and Selby provide a thorough summary of the results.

DeMillo and Offutt [DO88] examined the effectiveness of automatic test data generation to support mutation testing. Adequacy of automatically generated test cases was compared to adequacy of test cases selected using a number of coverage criteria, including statement coverage, branch coverage, and others. The adequacies were compared for a single, 27 SLOC program. DeMillo and Offutt found that the automatically generated test cases yielded high adequacy values, but lower precision

values. Overall, the experiment is not very satisfying, given the small sample size and a number of flaws in the experimental design.

Frankl and Weiss [FW93] compared the fault exposing capabilities of all-edges (branch testing) and all-uses (an instance of dataflow testing) coverage criteria, using test data selected randomly for comparison. The experiment provides an overall comparison of the criteria, a comparison for a fixed test set size, and the relationship between coverage and effectiveness for each criteria. Frankl and Weiss found that the all-uses criteria was more effective than the all-edges criteria in 5 of the 9 subjects at the 0.01 level. Finally, Frankl and Weiss point out that effectiveness had a clear dependence on percent coverage for only 3 of the 9 subjects. Their experiment design and data analysis is noteworthy because it includes avoidance of ceiling effects, effective statistical hypothesis testing, proper use of logistic regression, and recognition of potential bias in the experiment.

Hutchins et al [HFG+94] experimentally compared the effectiveness of all-edges, all-DUs and random criteria. The data implies that there are no discernible syntactic or semantic characteristics of the faults that correlate with high fault detection by any of the methods. It was also determined that high coverage (even 100%) is not a good indicator of testing adequacy (i.e., fault detection). The experiment was designed to avoid floor and ceiling effects. Hutchins et al censored a large part of their data without justification, however, and the effects of the censoring are not quantified or discussed. Also, a second order curve was fitted to several plots, though Hutchins et al do not justify why a second order curve is the appropriate choice.

2.2.6 Static Concurrency Analysis Experiments

Several experiments have been conducted using a subset of the static concurrency analysis techniques described in Section 2.1. Because most of the techniques are exponential in the worst case, experimentation is needed to distinguish average costs from worst case cost for the techniques. In addition, empirical work will support

performance quantification of the techniques, both in terms of optimizations within a given technique and through comparisons among the techniques.

Duri et al [DBD+93] experimented with various optimization techniques for Petri net-based reachability analysis. Net reduction was used to reduce the Petri net model of the program, while stubborn sets, partial orders (sleep sets), and net symmetry were used to guide the reachability graph construction. Experiments were conducted on programs of 3 to 100 dining philosophers, 3 to 10 customers for the 1-pump gas station problem, 3 to 5 customers for the 2-pump gas station problem, and readers/writers programs from 2 readers/1 writer to 10 readers/10 writers. The Border Defense System (BDS) program, an 11,000 line, 15 task program was included as well. In all, 32 programs of various sizes, with and without deadlock, were included in the experiment. For each program, Duri et al checked for deadlock without using any optimizations, using each optimization separately, using net reduction with each of the remaining three optimizations, and using net reduction with stubborn sets and net symmetry.

Data analysis consisted of comparisons of reachability graph sizes and generation times for the various optimization combinations. This sort of comparison can give informal evidence of certain relationships between the optimization techniques, but no statistical analysis is provided to quantify the significance of the results. In addition, there was no apparent attempt to formally characterize the growth rate for each of the optimization combinations. Because the experiment is conducted on academic programs (with the exception of BDS), the results of the experiment may not be generalizable to "typical" concurrent programs. This experiment provides insight into applicability of the combinations of optimization techniques, but the experimental design and informal data analysis prevent Duri et al from making general comments about the performance of these techniques.

Corbett [Cor94] provides an experimental evaluation of three static concurrency analysis techniques: reachability analysis is performed using SPIN (general reachability)

and SPIN+PO (general reachability + partial orders); symbolic model checking is performed using SMV; and INCA is used for inequality necessary condition analysis. The property checked by all of the tools is freedom from deadlock. The experiment is conducted on 7 scalable programs and one "real" program. Each scalable program was analyzed with 4 different sizes in an arithmetic progression; the communication skeleton of BDS (the "real" program) was analyzed as is. Corbett notes that conducting a fair evaluation of these methods is extremely difficult. To help guarantee fairness, Finite State Automata were built for the tasks in each program (using the INCA front end) and these FSAs were automatically converted to the input language of each tool. The FSAs provide a canonical model of the concurrent programs, ensuring all tools are solving the same problem. Corbett also points out a potential bias against SMV because the FSAs generated may present variables in an arbitrary order, and BDD size is sensitive to the variable ordering. Time to check for deadlock was measured for each of the tools on the programs in the sample.

Analysis of the data provides insight into the applicability of each tool. Using SPIN+PO generally allowed analysis of larger programs than SPIN, but the state space continued to grow quickly. SPIN and SPIN+PO performed best on programs with a small number of tasks and performed better than the other methods on the most data-intensive program. SMV exhibited subexponential growth in most programs, worked significantly faster than reachability on programs with many tasks, and provided comparable performance to the other methods on the gas station examples, despite potential biases from variable orderings. INCA excelled on programs containing many small tasks, though adding a single large task seriously degraded performance.

CHAPTER 3

EXPERIMENTAL METHODOLOGY

This chapter describes the experimental methodology we have developed to provide a basis for valid comparisons of the performance, in terms of both analysis time and accuracy, of various static concurrency analysis tools. We begin with a description of concurrent programs and some useful representations of those programs, then describe the tools used in the experiment. We close with a presentation of our comparison methodology.

3.1 Concurrent Programs and Program Representations

Because Ada is one of the few commonly used languages supporting concurrency, we use Ada programs as the canonical model of concurrent programs to be analyzed. We briefly describe here the principal concurrency constructs in Ada and several sources of nondeterminism in concurrent Ada programs. The inputs to the tools included in the experiment are based on program representations derived from the Ada programs. We describe these program representations, discuss their relationship to the canonical Ada program model, and describe how these representations can be converted to the input for each tool.

In Ada programs, potentially concurrent activities occur in *tasks*[1]. Ada tasks typically communicate with each other using a *rendezvous*. In a rendezvous, the calling task makes an *entry call* on a specific *entry* in the called task; the calling task then suspends execution until the called task terminates the rendezvous. The called task executes any statements contained in the *accept* for the entry, then terminates the rendezvous and, like the calling task, continues execution. Data can also be passed

[1]Ada also supports concurrent procedures, but for simplicity we only consider the tasking mechanism in our discussion.

between the two tasks at the start and termination of the rendezvous through parameters. The rendezvous thus acts as a synchronization and communication point between two tasks.

Nondeterminism is introduced into an Ada program's execution in several ways. When a calling task makes an entry call on a given entry, the calling task is placed on a task queue. When the called task reaches the corresponding entry, the run-time system selects the calling task for the rendezvous from the front of the queue. Since we cannot in general know the order of this task queue, this is essentially equivalent to the run-time system nondeterministically selecting a calling task for the rendezvous. Another source of nondeterminism is the *select* statement, which consists of one or more alternatives, each potentially including a guard that controls selection of that alternative. When a select statement is executed, the guard of each alternative is evaluated, with unguarded alternatives treated as though their guards are true. If more than one guard is true, one of the alternatives with a true guard and a waiting entry call is nondeterministically selected for execution. If there are no waiting entry calls on the alternatives with true guards, the task stalls until an entry call is made on one of these alternatives. If none of the guards are true, the task containing the select statement is terminated with a program error.

3.1.1 Example Program

To solidify our description of the program representations and the various concurrency analysis tools, we consider the readers/writers problem, an example that is commonly studied in the concurrency analysis literature. The readers/writers problem includes a set of readers and a set of writers that may be simultaneously accessing the same document, with the restriction that when a writer is accessing the document no readers or other writers can be accessing the document at that time. Our solution for the readers/writers problem uses a task for each reader, a task for each writer, and a single task to control access to the document. An example program showing one reader and one

writer can be found in Figure 3.1. To increase the size of the example program, we add additional readers and writers with the same structure as reader_1 and writer_1 below.

```
task body reader_1 is
begin
  loop
    control.start_read;
    control.stop_read;
  end loop;
end reader_1;
```

```
task body control is
  Readers : Natural range 1 .. 1 := 0;
  Writer  : Boolean := false;
begin
  loop
    select
      when (not Writer) =>
        accept start_read;
        Readers := Readers + 1;
      or
        accept stop_read;
        Readers := Readers - 1;
      or when (not Writer) and
              (Readers = 0) =>
        accept start_write;
        Writer := true;
      or
        accept stop_write;
        Writer := false;
    end select;
  end loop;
end control;
```

```
task body writer_1 is
begin
  loop
    control.start_write;
    control.stop_write;
  end loop;
end writer_1;
```

Figure 3.1. Ada Program for 1 Reader/1 Writer

We have selected three properties to check for the readers/writers program. The first of these is deadlock, which occurs when the program reaches a non-terminal state in which none of the tasks can continue executing. The second property can be phrased as "Can a reader ever read before some writer has written?" Our rationale for selecting this property is to ensure no reader can read an empty document. Because of symmetry, we do not need to check if each reader can read before some writer writes. All readers behave in the same way as far as the control task is concerned, so checking a single reader is sufficient; if the property is not possible for a specific reader, it is not possible for any of them. In our experiment, we check this property for reader_1. For notational convenience, we call this property no_r1w. The third property can be phrased as "Can two writers ever be writing at the same time?" We check this property to ensure that writers have mutually exclusive access to the document. Again by symmetry, checking two specific writers is sufficient; if these two writers can not write concurrently, no two writers can. In our experiment, we check this property for writer_1 and writer_2. For notational convenience, we call this property no_w1w2. Another property one would expect to check for this program is whether a reader and a writer can be accessing the

document at the same time. This property is similar to the third property above, so it is not described further.

3.1.2 Program Representations

All the tools in our experiment analyze the same Ada program. None of the tools accept an Ada program directly as input, however, so we convert the canonical Ada program to each tool's input representation. We build a set of Control Flow Graphs (CFGs) from the Ada source code, creating a CFG for each task in the program. Several of the tools use CFGs directly as the program description. Several other tools use program descriptions based on Finite State Automata (FSAs). For those tools, we convert each CFG to a corresponding FSA and then use the set of FSAs to generate a tool's input.

Since the tools use two different program representations for the program being analyzed, we try to ensure that each tool is analyzing the same program so that our comparison will be valid. The use of an Ada program as the canonical representation, with conversion to other representations as necessary, is intended to provide a common program for analysis. We use a straight-forward algorithmic translation from the Ada representation to each tool's required input representation. We examine the benefits and drawbacks of this approach further in Section 3.3.

In general, we would like any static analysis method to be *conservative*; for a given property, the analysis must not overlook cases where the property fails to hold. To ensure conservativeness, most methods use program representations that overestimate the behavior of the program being analyzed. This overestimate can lead to inaccuracy in the analysis results. If a tool reports that a property does not hold, when in fact the cases when it does not hold do not correspond to actual program behaviors, then this is called a *spurious result*. For example, if the program representation contains paths that can never be executed in the program (commonly called *infeasible paths*), the tool may report that the property fails to hold when it only fails on infeasible paths. The CFGs generated from our canonical Ada program can contain infeasible paths because some information, such

as each variable's values, is not included in the CFG. Since the inputs to SPIN, SPIN+PO, TRACC, SMV, and FLAVERS are based on these CFGs, the possibility exists that each of these tools will yield spurious results. Similarly, in INCA an integer solution to the set of inequalities could correspond to an infeasible trace (path) in the program. It is important, therefore, that we consider the effects of our program representations on the accuracy of the analysis.

As part of our experiment, we will improve the accuracy of the analysis results by improving the accuracy of the program representations. One way to do this is by *modeling* the values of user-selected variables in the representations. To be conservative, our representations initially include all possible values of the variables in the program. It may be possible, however, to statically determine the actual values of the variables and to include this information in the representations. When we include the actual values of a variable in a representation, we say we have *modeled* that variable.

For example, the **Writer** variable in the control task of the readers/writers program ensures that only a single writer can be writing at a time. The **Readers** variable ensures that there is never a situation in which the reader is reading at the same time the writer is writing. By modeling the values of one or both of these variables, we can generate representations that more accurately represents the control task behavior.

3.1.2.1 Control Flow Graphs

One way to represent the behavior of a program is with a control flow graph [Hec77]. A control flow graph (CFG) is similar to a flow chart, in that it represents all paths through a procedure or task. A control flow graph consists of a finite set of nodes, $N = \{n_i \mid i = 1, ..., j\}$, where j is the total number of nodes in the CFG, and a finite set of directed edges, $E = \{e_i \mid i = 1, ..., k\}$, where k is the total number of edges in the CFG. In our representation, the set of nodes includes a single start node and a single end node for the CFG. In addition, there is a single node in the CFG for each of the following: the declaration of the task and any local variables in the task (this node is called a

Decl_Region node), the begin statement, the end statement, and each executable statement. The start node in a CFG is always the Decl_Region node. There is an edge from n_i to n_j if the statement corresponding to n_j is potentially executable immediately after execution of the statement corresponding to n_i. There is also an edge from the start node to the node generated for the begin statement, an edge from the node generated for the begin statement to the node generated for the first executable statement in the task, and an edge from each of the exit nodes in the task to the node generated for the end statement. Each CFG node is annotated with the statement associated with that node.

Each entry call in the task is represented by a single node. Each accept statement in the task is represented by an Accept node followed by zero or more nodes representing the executable statements in the accept body, followed by an Accept_End node. Accept statements with no executable statements in the body are the only instance in which we add two CFG nodes for a single statement; therefore, the number of nodes in a CFG is never greater than twice the number of statements in the corresponding task. For the CFG for the control task in Figure 3.1, see Figure 3.2. In the figure, for the convenience of the reader we annotate each node in the CFG with the kind of statement (i.e., loop, assign, etc.) associated with it and each guard edge with the predicate for that guard.

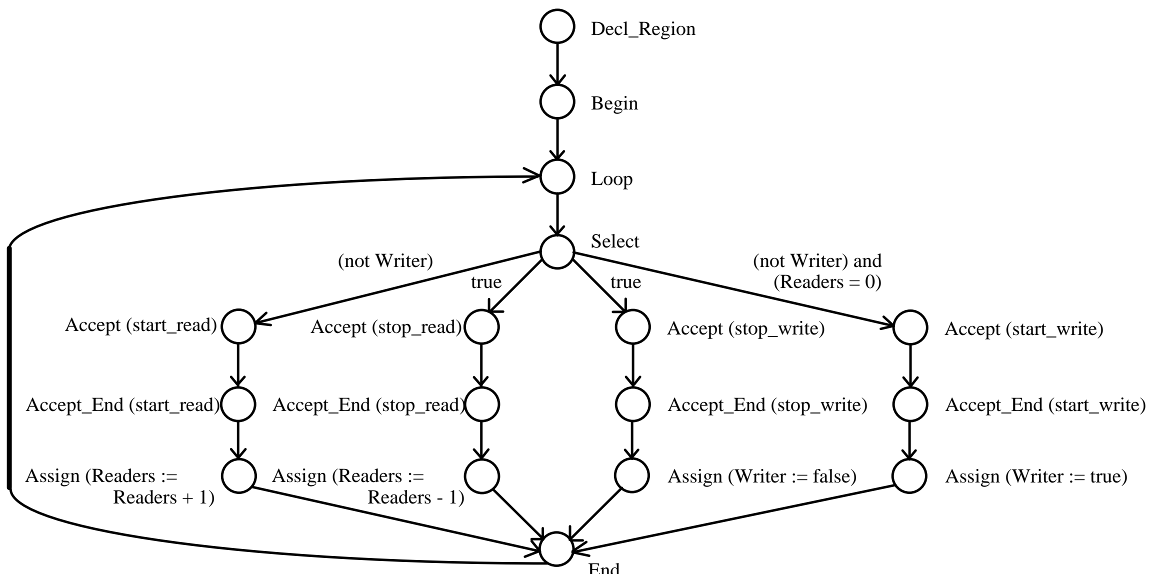


Figure 3.2. Example Control Flow Graph

The current form of the CFGs we use do not provide the capability to include variable values within the CFG representation. While it would be possible to revise the CFG representation to include this information, it is not necessary, since the two tools that use CFGs as their inputs provide methods for modeling variables. We therefore do not model variables in the CFGs; instead, we use the methods provided by these two tools to model the variables.

3.1.2.2 Finite State Automata

As an alternative to the CFG representation, Finite State Automata (FSAs) can be used to represent the behavior of the program. For each task in the program, we convert the CFG for the task into an FSA for the task. An FSA consists of a finite set of states, $S = \{s_i \mid i = 1, ..., m\}$, where m is the total number of states in the FSA, and a finite set of state transitions, $T = \{t_i \mid i = 1, ..., n\}$, where n is the total number of transitions in the FSA. The set of states includes a single start state and one or more final states. Each state in the FSA corresponds to one or more statements in a sequential region of code in the task and may also encode the values of variables that affect the synchronization behavior of the task. Each state transition in the FSA corresponds to a rendezvous point in the task or to an internal action of the task. Thus, there is a state transition t_i from s_j (the source) to s_k (the target) if the communication event (entry call or accept) represented by t_i causes the task represented by the automaton to transition from the region represented by s_j to the region represented by s_k, or if the internal action represented by t_i occurs in the task. We note that, if multiple tasks can rendezvous with the task at a rendezvous point, the FSA for the task contains a transition for each of those rendezvous.

The conversion from a set of CFGs to a set of FSAs starts with a translation of each CFG to the S-Expression Design Language (SEDL), one of several input languages accepted by the INCA toolset (discussed further in Section 3.3). The SEDL for a task is

similar to the original Ada for the task converted to Lisp syntax. We then provide the SEDL for the set of tasks comprising the program to INCA, which generates the FSAs described above.

A textual form of the FSA for the control task in Figure 3.1 is given in Figure 3.3. Because both the **Writer** and **Readers** variables are included in the guards of the select statement in the control task in this example, they can both affect the synchronization behavior of the task. The states in the FSA therefore include encodings of all possible values of those variables. State 2 encodes (**Readers** = 1, **Writer** = false), State 3 encodes (**Readers** = 0, **Writer** = false), State 4 encodes (**Readers** = 0, **Writer** = true), and State 5 encodes (**Readers** = 1, **Writer** = true).

```
State 1:
        T1 : internal ---> State 2
        T2 : internal ---> State 3
        T3 : internal ---> State 4
        T4 : internal ---> State 5
State 2:
        T5 : accept (writer_1, stop_write) ---> State 2
        T6 : accept (writer_1, stop_write) ---> State 5
        T7 : accept (reader_1, stop_read) ---> State 2
        T8 : accept (reader_1, stop_read) ---> State 3
        T9 : accept (reader_1, start_read) ---> State 2
        T10 : accept (reader_1, start_read) ---> State 3
State 3:
        T11 : accept (writer_1, stop_write) ---> State 3
        T12 : accept (writer_1, stop_write) ---> State 4
        T13 : accept (writer_1, start_write) ---> State 3
        T14 : accept (writer_1, start_write) ---> State 4
        T15 : accept (reader_1, stop_read) ---> State 3
        T16 : accept (reader_1, stop_read) ---> State 2
        T17 : accept (reader_1, start_read) ---> State 3
        T18 : accept (reader_1, start_read) ---> State 2
State 4:
        T19 : accept (writer_1, stop_write) ---> State 4
        T20 : accept (writer_1, stop_write) ---> State 3
        T21 : accept (reader_1, stop_read) ---> State 4
        T22 : accept (reader_1, stop_read) ---> State 5
State 5:
        T23 : accept (writer_1, stop_write) ---> State 5
        T24 : accept (writer_1, stop_write) ---> State 2
        T25 : accept (reader_1, stop_read) ---> State 5
        T26 : accept (reader_1, stop_read) ---> State 4
```

Figure 3.3. FSA for Control Task, No Variables Modeled

The transitions from State 1 represent a nondeterministic choice of the initial values of the **Writer** and **Readers** variables. Since we initially do not model these variables, the FSA must consider all possible combinations of their values. Transitions that result in changes to a variable lead to states encoding both possible values of that variable[2]. For example, transitions from State 2 (where **Writer** = false) on the stop_write entry lead to State 2 (**Writer** = false) and State 5 (**Writer** = true). This is because the **Writer** variable is changed as a result of the stop_write interaction, but without modeling the variable the FSA does not reflect the actual new variable value.

Because CFGs typically overestimate task behavior, using the FSAs generated from those CFGs may lead to spurious results. We can improve the accuracy of the analysis results by modeling variables in the FSAs. We do this by considering the values of user-selected variables during the conversion from the CFG to the SEDL. Information about variable values can be extracted from the abstract syntax tree annotation of each CFG node. The FSA that includes modeling of the **Writer** variable is shown in Figure 3.4.

By modeling the value of the **Writer** variable, we have pruned transitions 3, 4, 6, 12, 13, 19, and 23 from the original FSA. As an example of this pruning, consider transitions 3 and 4 in the original FSA. These transitions assume the initial value of the **Writer** variable can be true; when we model the **Writer** variable (and its initial value of false), these transitions are no longer possible. The other transitions are pruned in a similar manner.

We note that, when one or more variables are considered in the conversion from a CFG to an FSA, the two representations are no longer equivalent. The FSA contains additional information about task behavior and therefore represents a more accurate representation of the task. To make a fair comparison between the analysis results of a

---

[2]In general, variables can have more than two values. In these cases, the transitions that result in changes to the variable lead to states encoding all possible values of that variable.

tool using the more accurate FSA representation and those from a tool using CFGs, we must ensure that the tool using the CFGs also accounts for the same variables included in the FSA.

```
State 1:
        T1 : internal ---> State 2
        T2 : internal ---> State 3
State 2:
        T5 : accept (writer_1, stop_write) ---> State 2
        T7 : accept (reader_1, stop_read) ---> State 2
        T8 : accept (reader_1, stop_read) ---> State 3
        T9 : accept (reader_1, start_read) ---> State 2
        T10 : accept (reader_1, start_read) ---> State 3
State 3:
        T11 : accept (writer_1, stop_write) ---> State 3
        T14 : accept (writer_1, start_write) ---> State 4
        T15 : accept (reader_1, stop_read) ---> State 3
        T16 : accept (reader_1, stop_read) ---> State 2
        T17 : accept (reader_1, start_read) ---> State 3
        T18 : accept (reader_1, start_read) ---> State 2
State 4:
        T20 : accept (writer_1, stop_write) ---> State 3
        T21 : accept (reader_1, stop_read) ---> State 4
        T22 : accept (reader_1, stop_read) ---> State 5
State 5:
        T24 : accept (writer_1, stop_write) ---> State 2
        T25 : accept (reader_1, stop_read) ---> State 5
        T26 : accept (reader_1, stop_read) ---> State 4
```

Figure 3.4. FSA for Control Task, Writer Variable Modeled

In the examples that follow, our descriptions assume use of the controller FSA shown in Figure 3.4. However, to help quantify the effect of modeling variables, in our experiment all analyses were performed with three different versions of the controller FSA - the version in Figure 3.3, the version in Figure 3.4, and a version that models both the Writer and Readers variables.

3.2 Concurrency Analysis Tools

In our experiment, we consider several concurrency analysis methods and the tools implementing those methods. Specifically, we consider the reachability analysis tools SPIN, SPIN plus Partial Orders (SPIN+PO), and TRACC, the symbolic model checking

tool SMV, the integer programming tool INCA, and the data flow analysis tool FLAVERS.

3.2.1 Reachability Analysis

Reachability analysis enumerates the reachable states of the program being analyzed and checks the property of interest on the reachable state space. *State properties* can be checked by considering each state in isolation. Freedom from deadlock and writers 1 and 2 writing concurrently are examples of state properties. *Path properties* require consideration of paths through the reachable state space. Reader 1 reading before some writer writes is an example of a path property.

3.2.1.1 SPIN

The Simple Promela INterpreter (SPIN) [Hol91] performs reachability analysis on a program represented as a set of finite state automata. The program is described in the PROMELA language [Hol91], a language that was developed for specification of network protocols. SPIN automatically checks for deadlock. Other properties to be checked must be specified using *never claims* or *assertions*. In a never claim, the property is represented as an FSA that should never reach an accept state. An assertion is an expression that evaluates to true or false and is specified at user-selected points in a PROMELA program.

Given the program and property specifications, SPIN builds a transition matrix with an entry for each statement in the program. Each matrix entry consists of a specification of the conditions under which the statement can be executed and a specification of the effect of executing the statement. Starting from the initial state of the program, the tool generates the reachable state space with a depth-first traversal algorithm, using the transition matrix to generate next states from any given state. If at any time during the analysis a potential deadlock state is found, the FSA for a never claim reaches an accept state, or an assertion evaluates to false, the tool reports the error and terminates.

To analyze the readers/writers problem with SPIN, we need to translate the Ada program to a PROMELA program. A specification of a program in PROMELA consists of a declaration of the communication channels and global variables, a specification of a process type for each task in the program, and an initialization function that specifies the initial state of the program. In PROMELA it is possible to simulate a simple Ada rendezvous by declaring a communication channel with 0 message capacity. Such a channel forces a synchronization between two processes participating in a rendezvous, reflecting the semantics of the Ada rendezvous. For the readers/writers problem, we specify a single channel for each entry in the corresponding Ada program. Multiple processes can send to each channel but only a single process (in this case, the control process) can receive from each channel. This is consistent with the Ada rules for task entries, where multiple tasks can make entry calls on a given entry but only one task can accept the entry call. In PROMELA, the syntax <channel-name>!<var-name> specifies a process trying to send variable var-name on the channel channel-name and <channel-name>?<var-name> specifies a process trying to receive variable var-name from the channel channel-name.

The PROMELA specification of a process is based on a finite state automaton, with transitions between the states of the automaton specified as gotos. An if statement in PROMELA consists of one or more alternatives with guards and an optional unguarded else clause, and closely follows the semantics of the Ada select statement. When an if statement is executed, the guard of each alternative is evaluated. If more than one guard is true, one of the alternatives with a true guard is nondeterministically selected for execution. If none of the guards are true and an else clause exists, the else clause is executed. Unlike Ada, if none of the guards are true and there is no else clause, the process containing the if statement hangs until one or more of the alternative guards becomes true.

To generate the PROMELA program for our readers/writers problem, we convert each of the tasks in our canonical Ada representation into a CFG and then into an FSA as described in Section 3.1. We then translate the set of FSAs into a PROMELA program, where the FSA for the control task is as shown in Figure 3.4. The resulting PROMELA program can be found in Figure 3.5.

```
mtype = { synch };

chan control_start__read = [0] of { byte };
chan control_stop__read = [0] of { byte };
chan control_start__write = [0] of { byte };
chan control_stop__write = [0] of { byte };

proctype writer__1()
{
state_1:
  if
  :: control_start__write!synch -> goto state_2
  fi;
state_2:
  if
  :: control_stop__write!synch -> goto state_1
  fi
}

proctype reader__1()
{
state_1:
  if
  :: control_start__read!synch -> goto state_2
  fi;
state_2:
  if
  :: control_stop__read!synch -> goto state_1
  fi
}

proctype control()
{
state_1:
  if
  :: skip -> goto state_3
  :: skip -> goto state_2
  fi;
state_2:
  if
  :: control_stop__write?synch -> goto state_2
  :: control_stop__read?synch -> goto state_3
  :: control_stop__read?synch -> goto state_2
```

```
:: control_start__read?synch -> goto state_3
:: control_start__read?synch -> goto state_2
fi;
```

Figure 3.5. PROMELA Program for Readers/Writers Example

Continued, next page

Figure 3.5, continued

```
state_3:
  if
  :: control_stop__write?synch -> goto state_3
  :: control_start__write?synch -> goto state_4
  :: control_stop__read?synch -> goto state_3
  :: control_stop__read?synch -> goto state_2
  :: control_start__read?synch -> goto state_3
  :: control_start__read?synch -> goto state_2
  fi;
state_4:
  if
  :: control_stop__write?synch -> goto state_3
  :: control_stop__read?synch -> goto state_4
  :: control_stop__read?synch -> goto state_5
  fi;
state_5:
  if
  :: control_stop__write?synch -> goto state_2
  :: control_stop__read?synch -> goto state_4
  :: control_stop__read?synch -> goto state_5
  fi
}

init {
  atomic { run writer__1();
    run reader__1();
    run control()
  }
}
```

We note that this approach for generating a PROMELA input closely follows that described by Corbett [Cor94], with the difference that our technique uses an Ada program rather than a set of FSAs as the canonical model of the program. In addition, we use a separate message channel for each entry in the program, while Corbett uses a separate message channel for each pair of communicating processes for each entry. Our approach is conceptually closer to the semantics of the underlying Ada program. Despite these differences, our PROMELA input is very similar to that of Corbett, and both approaches generate essentially the same state space.

We note that specifying each process as a distinct FSA is probably not standard PROMELA "programming style". We recognize that this may introduce some bias, since SPIN could potentially take advantage of multiple instantiations of process types to

provide more efficient state space generation. Our approach would preclude the use of such specialized techniques. Our approach, however, greatly facilitates the process of translating from the canonical Ada representation to PROMELA, providing the benefits discussed in Section 3.3. In addition, Corbett [Cor94] discovered that converting Ada programs to the standard PROMELA programming style was difficult to do correctly, even for experienced PROMELA users, and did not result in a noticeable difference in performance compared to specifying each process as a distinct FSA.

After we have the specification of the program in PROMELA, we need to represent the three properties we are interested in checking. The first property, freedom from deadlock, is automatically checked by SPIN, so no further specification of this property is required. The second and third properties can be checked using either never claims or assertions.

Example never claims for no_r1w and no_w1w2 are found in Figure 3.6. Never claims are typically formulated in terms of the states of one or more processes, so PROMELA provides syntax to check the state of a given process. For example, the string reader__1[reader_1_pid]@state_2 checks whether the reader_1 process is in state 2. PROMELA also provides a skip statement, which is simply a null statement.

The FSA for the never claim for no_r1w stays in the initial state until either some writer writes or reader_1 reads (and goes to s2). We keep track of whether or not a writer has written with an additional variable, called **wrote**, in the PROMELA input. The **wrote** variable is initialized to false, and set to true whenever a writer writes. If some writer writes, the FSA can never exit the second do loop, and the FSA for the never claim can never reach the accept state. If reader_1 reads, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim. The FSA for the never claim for no_w1w2 stays in the initial state until both writer_1 and writer_2 are at s2; in other words, both writers are writing. If this occurs, the FSA for the

never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim.

no_r1w
```
never {
  do
    :: (wrote == true)  -> break                           -- if any writer writes, exit loop
    :: reader__1[reader_1_pid]@state_2 -> goto accept-- if reader_1 reads, go to accept state
    :: else -> skip                                        -- if neither of above, loop back
  od;
  do
    :: skip                                 -- infinite loop; property is not possible
  od;
accept:                                                    -- accept state of FSA
  do
    :: skip                                                -- infinite loop; reader_1 reading before
  od                                                       -- some writer writes has been found
}
```

no_w1w2
```
never {
  do
     :: writer__1[writer_1_pid]@state_2 &                  -- if writer_1 and writer_2 are both writing,
        writer__2[writer_2_pid]@state_2 -> goto accept     -- go to accept state
    :: else -> skip                                        -- otherwise, loop back
  od;
accept:                                                    -- accept state of FSA
  do
    :: skip                                 -- infinite loop; writer_1 and writer_2 both
  od                                                       -- writing has been found
}
```

Figure 3.6. Never Claims for no_r1w and no_w1w2

We have discovered several occasions on which we have found it necessary to add additional variables to the PROMELA input to check properties of interest. In many cases, the properties are specified in terms of events (i.e., rendezvous) rather than states of the processes. While it is sometimes possible to infer the event occurrences from the sequence of states one or more of the processes pass through, this can be difficult for non-trivial programs. We have found it effective to use variables (such as the **wrote** variable above) to keep track of the occurrence of events of interest. For instance, when a writer writes, we set the **wrote** variable to true as the writer transitions from one state to the next. We have also found it necessary to add additional variables when one task needs to

know the state of another task in the system, which often occurs when we use assertions. Because tasks are prohibited from querying the status of other tasks in the system, we have found it to be effective to add additional variables to keep track of this status information.

To use assertions to check if reader_1 reads before some writer writes, we modify the PROMELA input as shown in Figure 3.7. Basically, we set the **wrote** variable when any writer writes and assert that the variable has been set when reader_1 reads.

```
reader_1                                          control
    . . .                                              . . .
  :: control_start_read!synch ->                    :: control_start__write?synch -> atomic { wrote =
true;
       atomic { assert (wrote == true);                                          goto state_9 }
                goto state_2 }                         . . .
    . . .
```

Figure 3.7. Assertions for no_r1w

The assertions to check for writer_1 and writer_2 concurrently writing are somewhat more complicated. When writer_1 starts to write, the assertion that writer_2 is not writing is checked and the flag indicating that writer_1 is writing is set. Before writer_1 stops writing, the flag indicating that writer_1 is writing is cleared. We note that the flag can not be cleared after writer_1 stops writing, because SPIN then finds a violation of the assertion by having writer_2 start to write before the flag is cleared. Similar assertions are embedded in the writer_2 process. The required changes are shown in Figure 3.8.

We specify properties using both never claims and assertions to ensure that our choice of property specification technique does not bias our results against SPIN. Since SPIN+PO (discussed below) requires the use of assertions, we use assertions in SPIN to allow comparison of the results. It may be the case that using never claims yields better performance by SPIN, however, so we specify properties using never claims as well.

Modeling of the Writer variable is controlled by the FSAs generated for the program. When the FSAs are built without considering the Writer variable, the generated PROMELA code does not incorporate information about the Writer variable in the

analysis. When the FSAs are built taking the Writer variable into account, the generated PROMELA code incorporates this information as well.

writer_1

```
proctype writer_1()

{
state_1:
  if
        :: control_start_write!synch ->
             atomic { writer_1_writing = true;
                         assert (writer_2_writing == false);
                         goto state_2 }
  fi;
state_2:
  writer_1_writing = false;
  if
    :: control_stop_write!synch -> goto state_1
state_1
  fi
}
```

writer_2

```
proctype writer_2()

{
state_1:
  if
    :: control_start_write!synch ->
         atomic { writer_2_writing = true;
              assert (writer_1_writing == false);
              goto state_2 }
fi;
state_2:
 writer_2_writing = false;
  if
    :: control_stop_write!synch -> goto

  fi
}
```

Figure 3.8. Assertions for the no_w1w2

Converting the canonical Ada representation of the readers/writers problem into a PROMELA program was straightforward, and most of this process is automated. To provide a fair comparison, we specified the second and third properties using both never claims and assertions. Specifying the properties as never claims was relatively straightforward, but required an understanding of the internal operation of the tool to achieve the proper behavior. Specifically, we needed to realize that "execution" of the program and never claim are interleaved during the analysis and that evaluation of a guard is distinct from the execution of the guarded statement. This problem might be avoided by specifying the property in Linear Temporal Logic (LTL). Specifying the property in LTL is provided as a SPIN option, but we have yet to investigate it. Using assertions was straightforward for the second property, but the third property required some non-intuitive manipulations to specify the property correctly. Again, specifying this properly required the recognition that evaluation of a guard is distinct from the execution of the guarded statement.

3.2.1.2 SPIN + Partial Orders

The partial orders approach of Godefroid and Wolper (described in Section 2.1.1.2) has been implemented as an addition to SPIN; we refer to the resulting tool as SPIN+PO. The SPIN+PO tool takes input in the form of PROMELA, so the SPIN+PO input for the readers/writers program is as shown in Figure 3.5. Like SPIN, the SPIN+PO tool checks for deadlock automatically. The current version of SPIN+PO does not support the use of never claims for the specification of the property of interest, so the other two properties are specified as assertions embedded in the PROMELA input as shown in Figures 3.7 and 3.8. SPIN+PO checks those assertions, just as SPIN does during state space generation, and reports a violation and terminates if an assertion evaluates to false.

We note that other Partial Order additions to SPIN have been implemented, and using these additions could yield different empirical results. At the time we conducted this experiment, these additions did not support the use of rendezvous channels, while the SPIN+PO tool allows using these channels.

3.2.1.3 TRACC

To combat the state space explosion, Godefroid and Wolper try to reduce the size of the reachable state space as it is generated. An alternative approach is to reduce the size of the model from which the reachable state space is generated. This is the approach taken in the TPN-based Reachability Analysis for Concurrent Code (TRACC) tool.

The TRACC tool accepts the set of CFGs generated from the canonical Ada program as the program specification. A property of interest is not explicitly specified; rather, a specialized program must be written to check the property. For state properties (freedom from deadlock, no writer 1 and writer 2 writing concurrently), the property checking program examines each state in the reachability graph. For path properties (no reader 1 before any writer), the property checking program solves a dataflow problem on the reachability graph to check the property.

The TRACC tool uses a variety of representations of a concurrent program to capture information about the program. The CFG for each task is first converted into a Task Interaction Graph (TIG) [LC89], where each node represents a sequential region of control flow and each edge represents a possible interaction (entry calls/accepts) between that task and other tasks in the program. This is basically an optimization that greatly reduces the number of nodes in the flow graph. These TIGs are then combined into a single Petri net, which is used to generate a reachability graph. Preliminary experimental results [DCN94] show that using TIGs rather than CFGs as the basis for the Petri net can greatly reduce the size of the resulting reachability graph. The algorithm to check for deadlock on the reachability graph generated by the TRACC tool is given in [DCN94]. To check for writer 1 and writer 2 writing concurrently, each node in the reachability graph is examined. If a node is found where both writer 1 and writer 2 are writing, the property checking program reports the violation and terminates. To check for reader 1 reading before some writer writes, the property checking program solves a dataflow problem on the reachability graph. Each time some writer writes, a write flag is set to true. After the dataflow problem reaches a fixed point, each state in the reachability graph is examined to see if reader 1 is reading when the write flag is false. If so, the property checking program reports the violation and terminates.

Because the TRACC tool uses a set of CFGs as input, the accuracy improvements we make to FSAs have no affect on TRACC analysis accuracy. The TRACC tool includes several ways to improve analysis accuracy [CC96], including the capability to model some types of variables as variable subnets. We use a variable subnet to model the Writer variable in our experiment.

While the generation of the reachability graph is fully automated in the TRACC tool, the requirement to write a special program to check each property is inconvenient. We would not expect a typical analyst to undertake this effort. In addition, the TRACC tool can only be applied to very small versions of the readers/writers program.

3.2.2 Symbolic Model Checking

Symbolic Model Checking was described in Section 2.1.2. In our experiment, we used an implementation of a symbolic model checker called the Symbolic Model Verifier (SMV) [McM93]. Although SMV was originally designed as a hardware verification tool, it can also be used for analysis of concurrent software. A program specification is provided to the tool, which encodes the possible variable values for each variable and generates an OBDD for the program from those variables. The property of interest is specified in CTL and a least fixed point algorithm is used to check the property as described above. If the property is ever false, SMV reports the violation and terminates.

The usual method for specifying a program for SMV involves specifying a set of processes and a next state function for each process, but a capability for explicitly specifying the system transitions is also provided by SMV. We would have liked to specify the SMV input with the usual specification style, but were unable to impose rendezvous semantics with this style. This style changes a single state variable at a time for a state transition, but we need to change two state variables concurrently to represent a rendezvous. Using the style that explicitly specifies the transition relation also facilitates our translation from the canonical Ada program into the SMV input. For these reasons we use the latter specification style. Using this style, the input to SMV consists of four parts. The VAR declaration defines a variable to represent each process, with the number of variable values given by the number of states for the corresponding process. The INIT declaration sets the initial values (states) for the process variables. The TRANS declaration fully specifies the transition relation for the system, which determines which variable values change on each state transition of the program. For example, rendezvous semantics are explicitly incorporated in the transition relation by changing the two variables associated with the calling and accepting tasks on each transition. The SPEC declaration is a specification of a property in the temporal logic CTL.

To generate the SMV input for our readers/writers problem, we convert each of the tasks in our canonical Ada representation into a CFG and then convert each of the CFGs into an FSA as described in Section 2. We then automatically translate the set of FSAs into the SMV input, where the FSAs for the reader_1 and writer_1 tasks are as shown in Figure 3.3 and the FSA for the control task is as shown in Figure 3.4. A variable for each task is defined in the VAR declaration and initialized in the INIT declaration as described above. The TRANS declaration is generated by matching entry calls and accepts on the transitions in the set of FSAs. For each matching entry call and accept, a transition that changes the values of the variables representing the calling and accepting tasks is added to the transition relation. The resulting SMV input is shown in Figure 3.9. Our specification of the SMV input closely follows that of Corbett [Cor94]. We note that this may bias our results against SMV somewhat, since techniques that organize the OBDDs to efficiently represent the multiple, duplicate, processes can not be used.

```
MODULE main
VAR
  writer__1 : { s1, s2 };
  reader__1 : { s1, s2 };
  control : { s1, s2, s3, s4, s5 };
INIT
  ( ( writer__1 = s1 ) & ( reader__1 = s1 ) & ( control = s1 ) )
TRANS
  ( ( ( control = s1 ) & ( next(control) = s3 ) & ( next(writer__1) = writer__1 ) &
      ( next(reader__1) = reader__1 ) )
    |
    ( ( control = s1 ) & ( next(control) = s2 ) & ( next(writer__1) = writer__1 ) &
      ( next(reader__1) = reader__1 ) )
    |
    ( ( control = s2 ) & ( next(control) = s2 ) & ( writer__1 = s2 ) & ( next(writer__1) = s1 ) &
      ( next(reader__1) = reader__1 ) )
    |
    ( ( control = s2 ) & ( next(control) = s3 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
      ( next(writer__1) = writer__1 ) )
    |
    ( ( control = s2 ) & ( next(control) = s2 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
      ( next(writer__1) = writer__1 ) )
    |
```

Figure 3.9. Example SMV Input

Continued, next page

Figure 3.9, continued

```
( ( control = s2 ) & ( next(control) = s3 ) & ( reader__1 = s1 ) & ( next(reader__1) = s2 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s2 ) & ( next(control) = s2 )  &  ( reader__1 = s1 )  &  ( next(reader__1) = s2 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s3 ) & ( next(control) = s3 ) & ( writer__1 = s2 ) & ( next(writer__1) = s1 ) &
  ( next(reader__1) = reader__1 ) )
|
( ( control = s3 ) & ( next(control) = s4 ) & ( writer__1 = s1 ) & ( next(writer__1) = s2 ) &
  ( next(reader__1) = reader__1 ) )
|
( ( control = s3 ) & ( next(control) = s3 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s3 ) & ( next(control) = s2 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s3 ) & ( next(control) = s3 ) & ( reader__1 = s1 ) & ( next(reader__1) = s2 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s3 ) & ( next(control) = s2 ) & ( reader__1 = s1 ) & ( next(reader__1) = s2 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s4 ) & ( next(control) = s3 ) & ( writer__1 = s2 ) & ( next(writer__1) = s1 ) &
  ( next(reader__1) = reader__1 ) )
|
( ( control = s4 ) & ( next(control) = s4 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s4 ) & ( next(control) = s5 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
  ( next(writer__1) = writer__1 ) )
|
( ( control = s5 ) & ( next(control) = s2 ) & ( writer__1 = s2 ) & ( next(writer__1) = s1 ) &
  ( next(reader__1) = reader__1 ) )
|
( ( control = s5 ) & ( next(control) = s4 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
  ( next(writer__1) = writer__1 ) )
 |
( ( control = s5 ) & ( next(control) = s5 ) & ( reader__1 = s2 ) & ( next(reader__1) = s1 ) &
  ( next(writer__1) = writer__1 ) ) )
SPEC
 AG ( EX 1 )
```

The SPEC declaration in Figure 3.9 specifies a check for deadlock. The specification states Always, Globally, there exists an enabled state transition; in other words, the system does not deadlock.

To check no_r1w, we need to include additional variables in the system to keep track of when reader_1 has read and when any writer has written. Each transition in the transition relation must also be revised to modify these variables as appropriate. Transitions in which reader_1 moves from s1 to s2 change the **reader_1_read** variable to true, and transitions in which any writer moves from s1 to s2 change the **any_writer_wrote** variable to true. The SMV specification for no_r1w is shown in Figure 3.10. The specification states Always, Globally, if reader_1 read then **any_writer_wrote**. If the system can reach an execution state in which reader_1 has read but no writers have written yet, the implication is false and SMV reports the violation and terminates.

While we found it intuitive to check no_r1w by including variables that keep track of the occurrence of events of interest, it is also possible in SMV to avoid including these additional variables by specifying the property as a more complicated CTL formula. Because adding variables could increase the size of the state space, thereby adversely affecting SMV's analysis times, we have also checked no_r1w using the specification shown in Figure 3.11. The specification states that reader_1 will not enter state 2 (will not read) until writer_1 has gone to state 2 (has written).

```
SPEC
 AG ( reader_1_read -> any_writer_wrote )
```

Figure 3.10. SMV Specification for no_r1w

```
SPEC
 A [ !( reader__1 = s2 ) U ( writer__1 = s2 ) ]
```

Figure 3.11. Alternate SMV Specification for no_r1w

To check no_w1w2, we use the VAR, INIT, and TRANS declarations from Figure 3.9 to describe the program. No additional variables are needed, since the property can be checked by examining the values of the writer_1 and writer_2 variables. The SMV specification can be found in Figure 3.12. The specification states Always, Globally, if writer_1 is in state 2 (i.e., is writing) then writer_2 is not in state 2 (i.e., is not writing)

and that if writer_2 is in state 2 then writer_1 is not in state 2. If the system can reach an execution state in which both writer_1 and writer_2 are writing, both implications are false and SMV reports the violation and terminates.

As for the PROMELA input, modeling of the Writer variable is controlled by the FSAs used to generate the SMV input.

```
SPEC
  AG ( ( ( writer_1 = s2 ) -> !( writer_2 = s2 ) ) &
       ( ( writer_2 = s2 ) -> !( writer_1 = s2 ) ) )
```

Figure 3.12. SMV Specification for no_w1w2

Specifying the program in the VAR, INIT, and TRANS declarations was straightforward and mostly automated. Generating the SPEC declarations for our three properties of interest was also not difficult, but modifying the entire transition relation for the second property was tedious, though we quickly developed a tool to automate this as well.

3.2.3 Inequality Necessary Condition Analysis

The Inequality Necessary Condition Analysis technique (described in Section 2.1.3) has been implemented in a tool called INCA. The INCA tool accepts a specification of the program in an Ada-like language or in the SEDL discussed in Section 3.1.2.2. Properties of interest are formulated as *INCA queries*, which specify the properties as sequences of event symbols.

INCA converts the program specification into a set of communicating finite state automata for the tasks in the program. Communication equations and restriction inequalities are generated based on Ada rendezvous semantics. The INCA query is converted into a set of property inequalities. A commercial integer linear programming package (CPLEX) is then used to search for a solution to the set of flow equations, communication equations, restriction inequalities and property inequalities for the system.

A program specification is provided to the INCA tool in an Ada-like language or in SEDL. We use a set of (mostly) automated tools to convert the canonical Ada program

into SEDL. An INCA query consists of a definition of the query name, whether or not fairness constraints should be applied, and the query itself, which is written in terms of a set of sequences of intervals. None of our properties require fairness constraints, so we simply specify "nofair" in the queries below. The query is expressed as an ω-star-less expression [CA95], which is similar to a regular expression. The query can contain several sequences of intervals, though for our properties a single sequence of intervals is sufficient.

The query to check for deadlock can be found in Figure 3.13. For this query, we consider a single interval starting at the beginning of the program (":initial t") and enforce the constraint that progress in the program is always possible (":progress t").

```
(defquery "deadlock" "nofair"
  (omega-star-less (sequence
      (interval :initial t
                :progress t
                :costs "connect-arc-unit"))))
```

Figure 3.13. INCA Query for Deadlock

For more complicated properties, such as no_r1w and no_w1w2, we can specify more complicated constraints on intervals in the program execution. We specify a set of start (with ":initial") and end (with ":ends-with") points for each interval in the execution, as well as the events that are forbidden within each interval (with ":forbid"). We use "rend <caller>;<acceptor>.<entry>" to check for the occurrence of a specific rendezvous in an interval. When a rendezvous is included in the ":ends-with" portion of the query, the rendezvous is allowed but neither of the tasks participating in the rendezvous is allowed to progress further. We use "call(<caller>;<acceptor>.<entry>)" if we want to check the occurrence of a rendezvous but also need to let the accepting task progress past the rendezvous point. Queries for no_r1w and no_w1w2 are shown in Figure 3.14.

Recall that an INCA query specifies the negation of the property we would like to prove. For no_r1w, to show that reader_1 can not read before some writer has written, we specify the necessary conditions for reader_1 to read before any writer has written. To

do so, we specify an interval that begins at the initial state of the program and ends when reader_1 reads. Within this interval, none of the writers are allowed to write. If such an interval exists, it is possible for reader_1 to read before some writer has written. For no_w1w2, we specify an interval starting at the initial state of the program, ending after both writer_1 and writer_2 have started to write an arbitrary number of times (without the ":open t" flag, the property would be checked for the first time both writers start to write). Because neither writer_1 nor writer_2 is allowed to progress past their last calls on start_write, if such an interval exists, it is possible for writer_1 and writer_2 to be writing concurrently.

no_r1w
```
(defquery "no_reader_1_before_some_write" "nofair"
   (omega-star-less (sequence
      (interval :initial t :ends-with '((rend "reader_1;control.start_read"))
                :forbid '((rend "writer_1;control.start_write")
                          (rend "writer_2;control.start_write"))))))
```

no_w1w2
```
(defquery "no_w1w2" "nofair"
   (omega-star-less (sequence
      (interval :initial t
         :open t
         :ends-with '("call(writer_1;control.start_write)"
                      "call(writer_2;control.start_write)")))))
```

Figure 3.14. INCA Queries for no_r1w and no_w1w2

As discussed above, control over the variables that are modeled in the analysis is provided in the conversion from CFGs to SEDL. The conversion from the canonical Ada program to SEDL is almost fully automated and thus was straightforward, but proper specification of the queries required several discussions with the INCA developers.

3.2.4 Data Flow Analysis

The Dwyer and Clarke technique for dataflow analysis (discussed in Section 2.1.4) has been implemented in a tool called FLow Analysis VERifier for Software (FLAVERS). The FLAVERS tool accepts a set of CFGs as the specification of the program to be analyzed. The property of interest is specified as a Quantified Regular

Expression (QRE), which contains three parts. The first part of a QRE is the alphabet of events that are included in the property. The second part of a QRE is a quantifier, which specifies whether the tool should check if the property holds on all paths or whether the tool should look for the existence of a path on which the property holds. The final part of a QRE is a specification of a sequence of events as a regular expression.

The concurrent program is modeled as a Trace Flow Graph (TFG), which is a set of CFGs with additional edges to capture program events that may immediately precede the program event at each node. The QRE is then converted to a deterministic finite automaton called the Property Automaton (PA). To solve the dataflow problem, states of the PA are propagated through the TFG using an iterative worklist algorithm. To check whether the property holds, the PA states that are possible at program termination are compared to the accepting states of the PA.

Checking for deadlock using FLAVERS is not currently supported. The QREs for no_r1w and no_w1w2 are shown in Figure 3.15. For no_r1w, the events of interest are when reader_1 reads and when any writer writes. The tool should check for the existence of a path in which reader_1 reads before some writer writes, so the quantifier is "exist". The sequence of events in the property is specified (informally) as: "Any event except the events of interest occurs 0 or more times, then reader 1 reads, then any event, including the events of interest, occurs 0 or more times". For no_w1w2 the events of interest are when writer_1 and writer_2 start and stop writing. The tool should check that the specified sequence occurs on all paths. The sequence is (informally) specified as "Any event but writer_1 or writer_2 starting to write occurs 0 or more times, then either writer_1 starts to write and stops writing without an intervening writer_2 start write, or writer_2 starts to write and stops writing without an intervening writer_1 start write, then anything but writer_1 or writer_2 starting to write occurs 0 or more times".

Because the FLAVERS tool uses a set of CFGs as input, the accuracy improvements we make to FSAs are not incorporated into the FLAVERS analysis. The FLAVERS tool

includes a variety of refinement techniques that can be used to improve analysis accuracy, including the capability to model some types of variables as variable automata. When we use the variable automata technique to model the Writer variable, FLAVERS uses a representation that incorporates the same information as the FSAs used for the PROMELA, SMV, and INCA inputs and the variable subnet for TRACC.

```
no_r1w
{reader_1_read, any_writer_wrote} exist

[-any_writer_wrote, reader_1_read]*;
reader_1_read;
[any_writer_wrote, reader_1_read]*

no_w1w2
{writer_1_start_write, writer_1_stop_write,
  writer_2_start_write, writer_2_stop_write} all

 [-writer_1_start_write, writer_2_start_write]*;
(((writer_1_start_write;
   [-writer_2_start_write,writer_1_stop_write]*;
   writer_1_stop_write)
  |
  (writer_2_start_write;
   [-writer_1_start_write,writer_2_stop_write]*;
   writer_2_stop_write));
 [-writer_1_start_write,writer_2_start_write]*)*
```

Figure 3.15. QREs for no_r1w and no_w1w2

The conversion from the canonical Ada program to FLAVERS input is fully automated. Specifying properties as QREs seemed natural, but it was sometimes difficult to write the QRE correctly.

3.3 Comparison Methodology

To ensure our comparisons are as unbiased as possible, we must consider many separate issues. We must try to make sure each tool is analyzing the same program and property of interest. We must follow a methodology that tries not to bias our results against one or more of the tools. We must carefully select which programs and properties to include in the experiment. Finally, we must decide what measurements to compare after we have collected our experimental data.

3.3.1 Program Representations

To try to ensure the analysis tools are evaluating the same program, we used an Ada program as the canonical model of the program and translated that program into each tool's input language. For all the tools, the Ada program was first converted to a set of CFGs, which were then converted, when necessary, to the input language of each of the tools. The FLAVERS tool uses CFGs as input directly, so no further conversion was required for FLAVERS. For SPIN, SPIN+PO, and SMV, we converted the set of CFGs to a set of Finite State Automata, which we then used to generate the PROMELA and SMV inputs. For INCA, the SEDL generated when converting from CFGs to FSAs served as the input language.

There is some question, however, about whether it is even possible to have each of the tools analyze the exact same program. For example, PROMELA semantics are not exactly the same as Ada semantics, so specifying a PROMELA program to behave exactly as the corresponding Ada program would behave may not be possible. Similarly, standard SMV input specifications cannot represent rendezvous semantics, so we have used an alternate SMV specification form that also does not exactly match Ada semantics. We have applied extensive effort to try to ensure the tools are analyzing the same program, but because of differing tool semantics we may not have been completely successful. While we believe our approach to this problem is reasonable, there may be other approaches that are more successful at providing equivalent programs to each of the tools.

We note that the CFGs that are automatically generated from the canonical Ada program are a general abstraction of program control flow and were not explicitly developed to support one or more of the analysis tools evaluated. Similarly, the FSAs that are created to represent the program are a general abstraction and were not tuned to one or more of the analysis tools. While we know that the CFGs and FSAs are valid

representations of the program, using these representations could introduce bias in the experiment in some unknown manner.

As discussed in Section 3.1.2.2, when we convert a CFG to an FSA we decide which variables to consider during the conversion. To quantify the effect of modeling variables, we generated the inputs for all the tools with three different variable combinations - not modeling any variables, modeling the Writer variable only, and modeling both the Writer and Readers variables. For SPIN, SPIN+PO, SMV, and INCA, we controlled which variables were modeled during the conversion from CFGs to SEDL, and for FLAVERS we included a variable automaton in the analysis.

By modeling different combinations of the variables rather than simply modeling both variables, we may be introducing bias against some of the tools. For example, SPIN, SPIN+PO, SMV, and INCA all run faster when both variables are modeled. However, in general we believe an analyst will add accuracy to the program representations incrementally, rather than adding all variable information initially. For a program that contains a large number of variables, trying to model all those variables might make building the program representations or performing the analysis on those representations intractable. We therefore believe an analyst would start without modeling variables, and would incrementally model additional variables until the analysis results meet their accuracy requirements. The first two properties do not require any variable modeling for the tools to produce accurate results and, for the third property, the tools can produce accurate results when we model only the **Writer** variable. Thus for these properties, we believe an analyst would be unlikely to run the analysis modeling the **Readers** variable if they were using the incremental approach described above. Of course, checking some properties accurately, such as whether or not a reader and a writer can be accessing the document concurrently, requires modeling both variables, in which case we would expect an analyst to add both variables to the analysis.

Other empirical work [Cor94] has assumed that all or almost all variables in the program will be modeled. This approach seems to reflect a different analysis process, in which an analyst would start modeling information about all variables in the program, or at least all variables that affect inter-task communications. If the analyst discovered that this variable modeling led to intractable analyses, they could then incrementally remove variable modeling until the analysis was tractable. While this approach is often feasible for the example programs from the concurrency analysis literature, it is not clear to us that this assumption will hold for real programs that contain hundreds or thousands of variables, particularly since determining which variables affect inter-task communication is a non-trivial task. Since we plan to use the methodology presented here to build predictive models for the performance of the analysis tools on larger, more realistic programs, we work from the assumption that we believe is more likely to scale up. Of course, only extensive practical experience applying the analysis tools to realistic programs will indicate which assumption holds in general.

3.3.2 Property Representations

Guaranteeing that the tools are evaluating the same property on a given program is also difficult. Because each tool uses a different specification technique, and often a different logic, automatic translation between the property representations is not straightforward. FLAVERS can, however, generate an FSA from the QRE for a property of interest. When the QRE specifies an exists property this FSA can then be used to create a never claim for SPIN or an INCA query, and when the QRE specifies an all paths property this FSA can be used to generate the SMV SPEC declaration. These translations must be carefully performed, since QREs are in terms of events while SPIN never claims and SMV SPECs are in terms of process states. The SPIN assertions and TRACC property checking program were hand crafted for each of the three properties. In these cases, the only assurance of comparable properties of interest is a careful, manual

translation from the property to the appropriate SPIN assertions and TRACC property checker.

3.3.3 Checking for Bias

We must also try to ensure that our methodology avoids bias against one or more of the tools as much as possible.

For example, the sizes of the OBDDs used by SMV are sensitive to the order of the variables in the SMV input. To account for this, we checked the tool's performance using the variable ordering that results from our automatic translation and also ran the tool with the REORDER option, which applies a heuristic reordering algorithm before generating the OBDDs for the system. Similarly, SMV tends to be more efficient when processes are used rather than an explicit specification of the global transition relation. Modeling the semantics of the Ada rendezvous using the semantics of the SMV processes is not possible, however, and would have precluded our using the FSAs generated from our canonical CFGs for the SMV input. We set a higher priority on the requirement that the programs be the same for each tool, at the risk of a potential degradation in tool efficiency. We also noted in Section 3.2.2 that we found adding additional variables to the SMV input and embedding operations on those variables in the system transitions to be an intuitive approach to checking properties. Since this could degrade SMV's performance by growing the state space, we also specified the second property using an alternate CTL specification that did not require additional variables. We note that, in general, these alternate CTL specifications seem more complicated (i.e., contain more terms and temporal logic operators) than those using additional variables, but this is not always the case.

We noted in Section 3.2.1.1 that SPIN can use either never claims or assertions to check properties. We chose to use assertions to allow comparison with SPIN+PO, but this could introduce a bias against SPIN if the use of never claims is more efficient. We therefore ran SPIN with both the never claims and the assertions and compared the execution times.

In our implementation of the readers/writers problem, none of the accept statements have bodies. This does not affect the tools using inputs based on FSAs because the accept bodies are collapsed into single FSA states. Similarly, it does not affect FLAVERS, since this tool optimizes the accept bodies away (for the readers/writers problem). It is not clear, however, whether this affects the performance of INCA, since the INCA input used in [Cor94] for the readers/writers problem used accept bodies. Therefore, for INCA we ran the analysis cases on our version of the program with no accept bodies and on a version of the program containing accept bodies. We also note that most examples of INCA input that we have seen represent sets of identical tasks as arrays of task types, while the Ada program we use as a canonical model contains each reader and writer task specified uniquely. Since it is unclear how this affects INCA performance, we ran the analysis cases on INCA with a conversion from the canonical Ada program and also with arrays of reader and writer tasks. Finally, for some of the properties described in Chapter 4, we found it intuitive to specify the INCA query using two intervals, which can cause a significant growth in the size of the inequality system. In these cases, we also specified the queries using single intervals and adding additional constraints to the system. We ran the analysis cases using both types of queries.

3.3.4 Input Domain

From an empirical point of view, we would have liked to randomly select the programs for our experiment from the population of all concurrent Ada programs. This is not feasible, however, since the population of concurrent Ada programs available for public access is fairly limited in size and is certainly not complete. Unfortunately, there is no evidence that the programs we have selected are representative of the population of "real" Ada programs. In addition, the properties we tend to specify are relatively straightforward and may not be representative of the properties analysts specify in practice. Our uncertainty about the representativeness of both the programs and properties we are likely to include in our dataset means that our ability to make general

inferences from our empirical results is limited. On the other hand, we can use the relationships discovered in our experiment as a point of comparison when we do gain access to other, more realistic, concurrent programs.

A comparison of analysis times for a specific program and property can be useful. Since the size of the programs included in the experiment can be increased by including more tasks into the system, it is also interesting to consider how the analysis times for the tools grow as the problem size is increased. Toward this end, we collected experimental data for a range of program sizes.

We determined this range by finding the maximum size the LEAST effective tool on that program could accomplish in less than five hours and without exhausting memory. We then used an arithmetic progression of six sizes, with the maximum size mentioned above as the fifth or sixth size in the progression. While this may introduce some bias against tools that can scale to much larger sizes for this program, our rationale is that the comparison between the analysis tools should be made on the same input domain of programs, properties, and sizes.

3.3.5 Data Comparison

There are a number of measurements we can use for comparison. For example, Corbett [Cor94] uses a calculated growth rate to compare the performance of concurrency analysis tools. We suggest using analysis times and information about tool failures and analysis accuracy for comparison purposes.

In an effort to ensure a fair comparison of the tools, we propose using the time each tool takes to generate the analysis results from its native input as the analysis time. This time does not include the cost of translating the Ada programs into each tool's input language as part of the analysis time for that tool. Because this translation is a cost of our methodology rather than of each of the tools, we do not believe it would be fair to "charge" each tool for the cost of using our translation tools. On the other hand, the

translation times are often much larger than the actual analysis times, so using this measure of analysis time may not give a clear picture of the true cost of using each tool.

In a practical sense our real interest is in how long each of the tools takes to analyze Ada programs. To gain more insight into this practical issue, we propose an alternative definition of analysis time that uses the total analysis time for comparison, including timing information for all the translation steps in the analysis process and for the compilation of the C programs generated by SPIN and SPIN+PO. This comparison probably gives better insight into the true cost of analysis, at least for Ada programs, but the times also include potential inefficiencies in our conversion tools.

Once we have selected which analysis time to use and collected our data, we need to compare the resulting analysis times. One way to do the comparison would be to compare the mean analysis times for each tool; the tools with the lower mean times would fare best the comparison. Unfortunately, outliers can have a significant effect on the mean. For example, a tool with consistently small analysis times but one (or a few) very large analysis times could easily have a larger mean analysis time than a tool that has consistently larger analysis times but no outliers. The median can be used to give a rough idea about the effects of the outliers, but we still do not believe the mean analysis times are the best choice for comparison.

Another way to do the comparison would be to count the number of cases for which each tool has the fastest analysis time; tools with the largest numbers of "fastest cases" would fare best in the comparison. This measure also has problems, however. Specifically, a tool that consistently had the second or third fastest analysis times, but seldom had the fastest, would do worse in the comparison than a tool that had the fastest analysis times more often than the first tool, but generally had the slowest analysis times. We would like a measure that not only captures how well a tool compares to the others for each case, but also includes some (indirect) measure of consistency.

We believe a reasonable summary statistic to use for comparison of analysis times is the average ranking for each tool. For each case, we rank the tools (1 = fastest, 2 = second fastest, etc.) based on analysis time. For each tool, we then average these rankings across all cases and use this average for comparison; tools with the smallest average ranking would fare best in the comparison. This average can still be affected by outliers, but because the worst ranking a tool can have on a given case is given by the number of tools in the experiment, the effect of outliers is not of much concern. Because it is an average, this measure also (indirectly) includes consistency. The choice of what to compare for analysis times is a difficult one, but we believe the average ranking is a reasonable summary statistic for analysis time comparisons.

Another useful measurement for comparison is the failure rate for each tool. In our methodology, any analysis that takes over 5 hours is classified as a failed analysis. The selection of 5 hours is somewhat arbitrary, but in an experimental environment we need to choose a limit to ensure the experiments run in a reasonable period of time. The analysis can also fail because the tool exhausts available memory, terminates with some internal error, or can not be compiled (for SPIN and SPIN+PO).. Whether or not each analysis fails (takes more than 5 hours or terminates because of exhausted memory or an internal error) is therefore measured and used for comparison. One way to compare failures would be to compare the counts of failure cases for each tool; the tools with the lowest number of failed cases would fare best in the comparison. For this comparison to be meaningful, all the tools would need to be run on the same number of analysis case. While this is typically the case in an experiment using our methodology, it is not required. We therefore propose a comparison of percentage failures. For each tool, we calculate the percentage of analysis cases (for that tool) on which the tool failed. We can then compare these percentages across the tools without being concerned about the number of cases run for each tool; the tools with the lowest failure percentages would fare best in the comparison.

The utility of the tools is also determined by the accuracy of their analysis results. Given the relative simplicity of the programs included in most experiments, we can determine the correct answer for each of the analyses, and can therefore recognize spurious results reported by an analysis tool. Whether or not each analysis yields spurious results is considered to be a good indicator of accuracy, so we measure spurious results and use them for comparison. As for failures, we could use counts of the spurious results for comparisons. Because a spurious result would only be counted for a non-failure case, however, and because the tools are unlikely to fail on the exact same number of cases, comparing spurious result counts is problematic. We instead use percentages of spurious results for comparison. For each tool, we calculate the percentage of analysis cases (for that tool) on which the tool yielded spurious results. We can then compare these percentages across the tools without being concerned about the number of cases on which each tool failed; the tools with the lowest failure percentages would fare best in the comparison.

CHAPTER 4

PROGRAMS AND PROPERTIES FOR THE EXPERIMENT

This chapter describes the programs and properties that were included in the experiment. The programs that were included provide a diverse range of program structures and functionalities and were all readily available. Some of the programs, such as readers/writers and dining philosophers, had already been developed by the Arcadia consortium. For the other programs, we acquired the INCA inputs used by Corbett [Cor94] and converted them to Ada programs. For a given program, properties were selected to check key aspects of the functional behavior of the program. The program and property specifications included in the experiment can be obtained from ftp://laser.cs.umass.edu/pub/.

In all the programs in the experiment, we checked each property without including any variable modeling information in the FSAs. In some cases, we needed to include some variable modeling information to accurately check certain properties. Those cases are explicitly indicated below.

With the small, academic programs in the experiment, we knew which properties should be violated for each program, property, and modeled variables. If we specified a property that we knew should not be violated and the analysis reported that the property was violated, we iteratively modified our property specification until we achieved the "correct" analysis result given the program, property, and modeled variables (or could no longer see reasonable ways to modify the property specification). In some cases, specifying the property was very difficult and reaching a correct property specification required many iterations. We used this iterative process to try to factor out our inexperience using the tools, since the original spurious results were caused by our incorrect property specifications rather than by weaknesses in the tools. We believe that the spurious results measured in the experiment therefore more accurately represent the

strengths and weaknesses of the tools rather than our skill (or lack of it) specifying properties. It can be argued, of course, that our original property specifications should be used, since they may better reflect how a "typical" user would specify the properties and also informally include ease of use for each specification formalism in the results. Additionally, an analyst analyzing a real concurrent program probably does not know the "correct" analysis result, so the analyst would not know when to iteratively modify the property specification. It would therefore also be interesting to design a different experiment that used the original property specifications and tried to measure how easy or difficult it was to properly specify the properties on the first attempt with each formalism.

4.1 Cyclic

The cyclic program provides a loosely synchronized ring of processes, where the processes start in order as the ring is traversed, but each process can complete its task at any time [Mil80]. The program thus enforces the start order for each process, but not the stop order. Our implementation of a size N cyclic program consists of N customer tasks and N scheduler tasks. Each customer task executes a simple loop, first accepting a start from its scheduler then signaling the scheduler that it is finished. Each scheduler loops through the following actions - signaling its customer to start, signaling the next scheduler to begin, then waiting until both its customer has finished and the previous scheduler has signaled it to begin.

We have selected three properties to check for the cyclic program. The first of these is deadlock. The second property can be phrased as "On any iteration, can customer_3 start before customer_2 starts?" This checks to see if the start ordering is enforced as required. For ease of reference, we call this property no_c3c2. The third property can be phrased as "On any iteration, can customer_2 accept start twice without an intervening call to finish?" This property can be checked by considering only the control flow in customer_2, but it is interesting because it ensures that the customer task completes its current processing before starting again. If we can prove this for an arbitrarily selected

customer task, we have shown it for all customer tasks. For ease of reference, we call this property no_c2ss. The never claim for no_c3c2 is shown in Figure 4.1. The FSA for the never claim stays in the initial state until scheduler_3 has started customer_3 and customer_2 was not started on this iteration. If this occurs, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim.

```
never {
 do
   :: cyclic_sched_3[sched_3_pid]@s3 &            -- if scheduler 3 has just started customer 3 and
      cust_2_started == false   -> goto accept    -- customer 1 was not started on this iteration, accept
   :: else -> skip                                -- otherwise, loop back
 od;
accept:                                           -- accept state of FSA
 do
   :: skip                               -- invalid sequence of customer_3 starting without
 od                                               -- customer_1 starting was found
}
```

Figure 4.1. Never Claim for no_c3c2

To check this property, we needed to add an additional variable to the PROMELA program to keep track of whether or not customer_2 had been started on the current iteration. This was necessary because we needed to recognize certain events that occur during program execution, specifically customer_2 being started. Since customer_2 can start and finish before we reach the point at which customer_3 is started, it is not possible to check the state of the customer_2 task to determine if it was started on the current iteration. The **cust_2_started** variable is set to true when customer_2 is started and set to false when customer_3 is started (to reset it for the next iteration).

The assertions to check no_c3c2 are shown in Figure 4.2. Whenever customer_2 is started, the **cust_2_started** flag is set. Whenever customer_3 is started, the assertion that customer_2 was started is checked and the flag is cleared. If the assertion is false, customer_3 has started before customer_2, and SPIN reports the violation and terminates.

The SMV specification for no_c3c2 is shown in Figure 4.3. Essentially, the specification states that Always, Globally, if customer_3 has been started then

customer_2 has also been started. The transition that starts customer_2 was modified to set the **cust_2_started** flag to true and the transition that starts customer_3 was modified to set the **cust_3_started** flag to true. The transition on which scheduler_3 signals scheduler_4 was modified to clear both flags. A violation of the property will thus only be found if customer_3 is started on some iteration before customer_2 is started.

```
scheduler_2
        . . .
  :: cust__2_start!synch -> atomic { cust_2_started = true;
                                     goto state_3 }
        . . .
scheduler_3
        . . .
  :: cust__3_start!synch -> atomic {
                                    assert (cust_2_started == true);
                                    cust_2_started = false;
                                    goto state_3 }
        . . .
```

Figure 4.2. Assertions for no_c3c2

```
SPEC
   AG ( cust_3_started -> cust_2_started )
```

Figure 4.3. SMV Specification for no_c3c2

It is also possible to check no_c3c2 with SMV without modifying the transitions in the transition relation to model the **cust_2_started** and **cust_3_started** variables. Alternatively, we can specify the property using the alternate CTL formula shown in Figure 4.4. The formula states that Always, Globally, if scheduler_2 is in state 2 (just prior to starting customer_2), then on All execution paths from this point, scheduler_3 is not in state 3 (has not started customer_3) until scheduler_2 is in state 3 (has started customer_2).

```
SPEC
  AG ( ( sched__2 = s2 ) -> A [ !( sched__3 = s3 ) U ( sched__2 = s3 ) ] )
```

Figure 4.4. Alternate SMV Specification for no_c3c2

The INCA query for no_c3c2 is shown in Figure 4.5. We specify an interval, starting at the initial state of the program and ending with some occurrence of the rendezvous

between scheduler_1 and scheduler_2 on the next entry and some occurrence of the rendezvous in which scheduler_3 starts customer_3. The rendezvous between scheduler_1 and scheduler_2 represents the start of a cycle around the ring, and because the interval ends with this rendezvous, scheduler_2 is not allowed to progress further (i.e., cannot start customer_2). If such an interval exists, it is possible for customer_3 to start before customer_2 on some cycle around the ring of schedulers.

```
(defquery "no_c3c2" "nofair"
  (omega-star-less (sequence
    (interval :initial t
          :open t
          :ends-with '((rend "sched_1;sched_2.next")
                       (rend "sched_3;cust_3.start"))))))
```

Figure 4.5. INCA Query for no_c3c2

Because the query above was somewhat difficult to formulate properly, we also formulated the property by adding an additional constraint to the system of inequalities; the resulting query is shown in Figure 4.6. The query specifies an interval, starting at the initial state of the program, in which the number of times scheduler_3 has started customer_3 is greater than the number of times scheduler_2 has started customer_2. If such an interval exists, it is possible for customer_3 to start before customer_2 on some cycle around the ring of schedulers.

```
(defquery "no_c3c2_con" "nofair"
  (omega-star-less (sequence
    (interval :initial t
      :constraints '((>= (- "call(sched_3;cust_3.start)"
                            "call(sched_2;cust_2.start)")
                     1))))))
```

Figure 4.6. Alternate INCA Query for no_c3c2

The FLAVERS QRE for no_c3c2 is shown in Figure 4.7. The events of interest are when customer_2 and customer_3 are started. The tool should check if the specified sequence occurs on some path in the program. The sequence is informally specified as "0 or more valid sequences of customer_2 then customer_3 starting are followed by customer_3 starting before customer_2 starts."

```
{cust_2_start,cust_3_start} none

[-cust_2_start,cust_3_start]*;
(cust_2_start;
 [-cust_3_start]*;
 cust_3_start;
 [-cust_2_start,cust_3_start]*)*;
cust_3_start;
[cust_2_start,cust_3_start]*
```

Figure 4.7. QRE for no_c3c2

The third property we check on the cyclic program is no_c2ss. The never claim for no_c2ss is shown in Figure 8. The FSA for the never claim stays in the initial state until scheduler_2 has started customer_2 before customer_2 was finished. To check this property, we added a variable to the PROMELA program to keep track of whether or not customer_2 has finished the previous processing. We set this variable to false when customer_2 is started and true when customer_2 is finished.

```
never
{
 do
   :: sched__2[sched_2_pid]@state_3  &              -- if scheduler_2 has started customer_2 and
      cust_2_finished == false -> goto accept       -- customer_2 has not finished, accept
   :: else -> skip                                  -- otherwise, loop back
 od;
accept:                                             -- accept state of FSA
 do
   :: skip                                  -- customer_2 started before finishing previous
 od                                                 -- processing
}
```

Figure 4.8. Never Claim for no_c2ss

The assertions to check no_c2ss are shown in Figure 4.9. Whenever customer_2 is started, the assertion that customer_2 finished is checked and the flag is set to false. When customer_2 finishes, the flag is cleared. If the assertion is violated, customer_2 can be started without finishing the previous processing.

The SMV specification for no_c2ss is shown in Figure 4.10. The specification states that Always, Globally, the error flag is never set to true. We use a flag for customer_2 finishing as described above, and set the error flag to true if customer_2 is started when

cust_2_finished is false. The specification is thus only false if customer_2 is started before it has finished the previous processing.

```
scheduler_2
        . . .
 :: cust__2_start!synch -> atomic {
                                   assert (cust_2_finished == true);
                                   cust_2_finished = false;
                                   goto state_3 }
        . . .
```

Figure 4.9. Assertions for no_c2ss

```
SPEC
   AG ( !error )
```

Figure 4.10. SMV Specification for no_c2ss

Alternatively, we can avoid modeling the additional variables in SMV by using the alternate CTL specification shown in Figure 4.11. The specification states that Always, Globally, if scheduler_2 is in state 3 (has just started customer_2), then scheduler_2 does not reach state_2 again (ready to start customer_2 again) until scheduler_2 has reached either state 5 or 8 (i.e., has received finished notification from customer_2). The specification is thus only false if customer_2 is started before it has finished the previous processing.

```
SPEC
  AG ( ( sched__2 = s3 ) -> A [ !( sched__2 = s2 ) U ( ( sched__2 = s5 ) |
                                                        ( sched__2 = s8 ) ) ] )
```

Figure 411. Alternate SMV Specification for no_c2ss

The INCA query for no_c2ss is shown in Figure 4.12. We specify one interval starting at the initial state of the program and ending when the customer_2 task is started. The first interval thus skips an arbitrary number of cycles around the ring of schedulers before checking the second interval. The second interval specifies that customer_2 is started without customer_2 finishing. Since the first interval ends with customer_2 starting, the combination of the two intervals specifies the no_c2sf property.

```
(defquery "no_c2ss" "nofair"
   (omega-star-less (sequence
      (interval :initial t
         :open t
         :ends-with '((rend "sched_2;cust_2.start")))
      (interval
         :ends-with '((rend "sched_2;cust_2.start"))
         :forbid    '((rend "cust_2;sched_2.finished"))))))
```

Figure 4.12. INCA Query for no_c2ss

Alternatively, because using multiple intervals can increase INCA analysis times, rather than using two intervals to specify no_c2ss we can add an additional constraint as shown in Figure 4.13. The query specifies an interval, starting at the initial program state, in which the number of times that customer_2 has been started is two greater than the number of times customer_2 has finished. Note that it is valid for the number of times customer_2 is started to be one greater than the number of times it has finished; this occurs on each cycle, after customer_2 has been started but before it has finished.

```
(defquery "no_c2ss_con" "nofair"
  (omega-star-less (sequence
    (interval :initial t
      :constraints '((>= (- "call(sched_2;cust_2.start)"
                             "call(cust_2;sched_2.finished)")
                     2))))))
```

Figure 4.13. Alternate INCA Query for no_c2ss

The FLAVERS QRE for no_c2ss is shown in Figure 4.14. The events of interest are when customer_2 is started and finished. Customer_2 finishing is specified with the scheduler_2_finished event, since FLAVERS annotates rendezvous with the name of the accepting task and entry name. The tool should check if the specified sequence occurs on all paths in the program. The sequence is informally specified as "Any time customer_2 is started, customer_2 finishes before it is started again."

The inputs to the tools contained sufficient information to check the deadlock and no_c2ss properties without modeling any variables in the program. However, our analysis of the no_c3c2 property indicated that customer_3 could be started before customer_2 on some iterations around the ring of processes, violating the start order

requirement. To prove that the no_c3c2 property holds, we needed to model two variables in each scheduler task - a variable that indicates when the corresponding customer has finished and a variable that indicates when the scheduler has been signaled by the preceding scheduler in the ring.

```
{customer_2_start,scheduler_2_finished} all

[-customer_2_start]*;
(customer_2_start;
 [-customer_2_start,scheduler_2_finished]*;
 scheduler_2_finished;
 [-customer_2_start]*)*
```

Figure 4.14. QRE for no_c2ss

4.2 Divide and Conquer (DAC)

The divide and conquer program [ACD+94] provides a set of solvers that can cooperatively solve a problem. Each $solver_i$ can be activated by a fork from $solver_{i-1}$ or simply terminate if it is not activated by $solver_{i-1}$. If $solver_i$ is forked, it uses an internal condition to either (conceptually) solve the problem and join $solver_{i-1}$ (indicating that it is done) or fork $solver_{i+1}$ and wait for $solver_{i-1}$ to join it before joining $solver_{i-1}$. Our implementation of a size N divide and conquer program consists of N solver tasks and a single main task that activates (forks) solver_1.

We have selected three properties to check for the cyclic program. The first of these is deadlock. The second property can be phrased as "If solver_3 is forked, is it possible for solver_1 to join the main task before solver_3 has joined solver_2?" This checks to see if the join ordering is enforced as required. For ease of reference, we call this property no_s1js3j. Note that we could have also checked the (more intuitive) property no_s1js2j, but no_s1js2j could be checked by examining the control flow in a single task (solver_1), and we preferred the more challenging no_s1js3j. The third property can be phrased as "Is solver_3 forked on every execution of the program?" This property is interesting because it checks to see if there are instances in which solver_1 or solver_2 decide NOT to fork additional solvers. It is "legal behavior" for the program to execute

without forking solver_3, but an analyst may still want to know if this is possible. For ease of reference, we call this property no_s3f.

The never claim for no_s1js3j is shown in Figure 4.15. The FSA for the never claim stays in the initial state until either solver_3 is forked (when solver_2 moves to s4) or solver_1 joins the master task. If solver_3 is forked, it must join solver_2 before solver_1 joins the master task for no_s1js3j to hold. If solver_3 joins solver_2, the FSA moves to the ok state, and the property holds. If solver_1 joins the main task before solver_3 joins solver_2, the FSA moves to the accept state and SPIN reports the violation of the never claim.

```
never {
 do
   :: solver__2[solver_2_pid]@state_4 -> goto solver_3_forked           -- solver_2 forked solver_3
   :: solver__1[solver_1_pid]@endstate_2 -> goto ok                     -- solver_1 joined main task
   :: else -> skip                                                      -- otherwise, loop back
 od;
ok:                                                                     -- ok state, property not violated
 do
   :: skip                                                  -- infinite loop
 od;
solver_3_forked:                                                        -- solver_3 was forked
 do
   :: solver__2[solver_2_pid]@state_5 -> goto ok                        -- solver_3 joined solver_2
   :: solver__1[solver_1_pid]@endstate_2 -> goto accept                 -- solver_1 joined main task
   :: else -> skip                                                      -- otherwise, loop back
 od;
accept:                                                     -- accept state of FSA
 do
   :: skip                                        -- infinite loop; solver_3 was forked but
 od                                                         -- solver_1 joined main task before solver_3
}                                                           -- joined solver_2
```

Figure 4.15. Never Claim for no_s1js3j

The assertions to check no_s1js3j are shown in Figure 4.16. To check this property using assertions, we needed to add two additional variables, called **solver_3_forked** and **solver_3_joined**, to keep track of whether or not solver_3 had been forked and joined. When solver_3 is forked, **solver_3_forked** is set to true and when solver_3 joins, **solver_3_joined** is set to true. When solver_1 joins the main task, the assertion that either solver_3 both forked and joined or solver_3 was not forked at all is checked. If the

assertion is false, solver_3 was forked but solver_1 joined the main task before solver_3 joined solver_2, violating no_s1js3j.

```
solver_1
        . . .
state_5:
  if
  :: solver__1_join?synch ->  atomic {
        assert ( ((solver_3_forked == true) & (solver_3_joined == true))
                | (solver_3_forked == false) );
        goto endstate_2 }
  fi
        . . .
solver_2
        . . .
state_3:
  if
  :: solver__3_fork!synch -> atomic { solver_3_forked = true;
                                     goto state_4 }
  fi;
state_4:
  if
  :: solver__3_join!synch -> atomic { solver_3_joined = true;
                                     goto state_5 }

  fi;
        . . .
```

Figure 4.16. Assertions for no_s1js3j

The SMV specification for no_s1js3j is shown in Figure 4.17. We use **solver_3_forked** and **solver_3_joined** variables as described above to keep track of when solver_3 has been forked and joined. The specification states that Always, Globally, when solver_1 has joined the main task, either solver_3 both forked and joined or solver_3 was not forked.

```
SPEC
  AG ( ( solver_1 = s3 ) -> ( ( solver_3_forked & solver_3_joined )
                        | !solver_3_forked ) )
```

Figure 4.17. SMV Specification for no_s1js3j

Alternatively, we can avoid adding the **solver_3_forked** and **solver_3_joined** variables by using the alternate CTL specification shown in Figure 4.18. The specification checks to see if there Exists an execution path on which, in the Future,

solver_1 has joined the master task and solver_3 is in state 3 or 5 (has been forked but has not joined). If this specification is true, no_s1js3j can be violated.

```
SPEC
  EF ( ( solver__1 = s2 ) & ( ( solver__3 = s3 ) |
                              ( solver__3 = s5 ) ) )
```

Figure 4.18. Alternate SMV Specification for no_s1js3j

The INCA query for no_s1js3j is shown in Figure 4.19. We specify an interval starting at the initial state of the program in which solver_1 joins the main task and solver_3 is forked but does not join solver_2. If such an interval exists, it is possible for solver_3 to be forked and solver_1 to join the main task before solver_3 joins solver_2.

```
(defquery "no_s1js3j" "nofair"
  (omega-star-less (sequence
    (interval :initial t
          :ends-with '((rend "main;solver_1.join"))
          :require   '((rend "solver_2;solver_3.fork"))
          :forbid    '((rend "solver_2;solver_3.join"))))))
```

Figure 4.19. INCA Query for no_s1js3j

The FLAVERS QRE for no_s1js3j is shown in Figure 4.20. The events of interest are when solver_1 joins and when solver_3 is forked and joins. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "Either solver_3 is forked and joins before solver_1 joins or solver_1_joins (i.e., solver_3 is not forked)."

```
{solver_1_join, solver_3_fork, solver_3_join} all

[-solver_1_join, solver_3_fork, solver_3_join]*;
((solver_3_fork;
 [-solver_1_join, solver_3_join]*;
 solver_3_join;
 [-solver_1_join]*;
 solver_1_join)
 |
 solver_1_join);
[solver_1_join, solver_3_fork, solver_3_join]*
```

Figure 4.20. QRE for no_s1js3j

The third property we check for the divide and conquer program is no_s3f. The never claim for no_s3f is shown in Figure 4.21. The FSA for the never claim stays in the initial state until either solver_3 is forked (when solver_2 moves to s4) or solver_2 joins solver_1. If solver_3 is forked, the FSA enters an infinite loop. If solver_2 joins solver_1 (without forking solver_3), the FSA moves to the accept state and SPIN reports the violation of the never claim.

```
never {
  do
    :: solver__2[solver_2_pid]@state_4 -> break              -- solver_3 was forked
    :: solver__1[solver_1_pid]@state_5 -> goto accept        -- solver_3 was not forked
    :: else -> skip                                          -- otherwise, loop back
  od;
  do
    :: skip                                          -- infinite loop
  od;
accept:                                                      -- accept state of FSA
  do
    :: skip                                          -- infinite loop; solver_3 was not forked
  od
}
```

Figure 4.21. Never Claim for no_s3f

The assertions to check no_s3f are shown in Figure 4.22. When solver_3 is forked, **solver_3_forked** is set to true. After solver_1 joins, the assertion that solver_3 was forked is checked. If the assertion is false, solver_3 was never forked, violating no_s3f.

```
main
      . . .
  :: solver__1_join!synch -> atomic { assert (solver_3_forked == true );
                              goto endstate_3 }
      . . .
solver_2
      . . .
  :: solver__3_fork!synch -> atomic { solver_3_forked = true;
                                  goto state_4 }
      . . .
```

Figure 4.22. Assertions for no_s3f

The SMV specification for no_s3f is shown in Figure 4.23. We use **solver_3_forked** as described above to keep track of when solver_3 has been forked.

The specification states that Always, Globally, when solver_1 has joined the main task, solver_3 was forked.

```
SPEC
  AG ( ( solver_1 = s2 ) -> solver_3_forked )
```

Figure 4.23. SMV Specification for no_s3f

Alternatively, we can avoid adding the **solver_3_forked** variable by using the alternate CTL specification shown in Figure 4.24. The specification states that Always, the main task is not in state 3 (has not terminated) until solver_3 is in state 3 or 5 (has been forked). If this specification is false, it is possible for the program to execute without forking solver_3, violating no_s3f.

```
SPEC
  A [ !( main = s3 ) U ( ( solver__3 = s3 ) |
                        ( solver__3 = s5 ) ) ]
```

Figure 4.24. Alternate SMV Specification for no_s3f

The INCA query for no_s3f is shown in Figure 4.25. We specify an interval starting at the initial state of the program in which solver_1 joins the main task but solver_2 is not allowed to fork solver_3. If such an interval exists, it is possible for solver_1 to join the master task without solver_3 being forked, violating no_s3f.

```
(defquery "no_s3f" "nofair"
  (omega-star-less (sequence
    (interval :initial t
          :ends-with '((rend "main;solver_1.join"))
          :forbid    '((rend "solver_2;solver_3.fork"))))))
```

Figure 4.25. INCA Query for no_s3f

The FLAVERS QRE for no_s3f is shown in Figure 4.26. The events of interest are when solver_1 joins and when solver_3 is forked. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "Solver_3 is forked before solver_1 joins."

```
{solver_1_join, solver_3_fork} all

[-solver_1_join, solver_3_fork]*;
solver_3_fork;
```

[solver_1_join, solver_3_fork]*

Figure 4.26. QRE for no_s3f

4.3 Dining Philosophers

The dining philosophers problem has been analyzed extensively in the literature. We included the standard problem and three variations of it in our experiment.

4.3.1 Standard Problem (dp)

In the standard dining philosophers problem, a certain number of philosophers sit around a table, with a single fork between a philosopher and the neighbor to its left. Each philosopher thinks for a while, then picks up both forks (one at a time, left fork first) to eat, then puts the forks back down and thinks some more. Because the forks between the philosophers are shared, it is not possible for all the philosophers to eat at the same time. Our solution for the dining philosophers problem uses a task for each fork and a task for each philosopher. Because all the philosophers can pick up their left forks and wait to pick up their right forks, deadlock is possible in this program.

We have selected two properties to check for the standard dining philosophers program. The first of these is deadlock. The second property can be phrased as "Can two adjacent philosophers ever be eating at the same time?" If we can prove that an arbitrarily selected pair of adjacent philosophers can not be eating concurrently, we can show that it is not possible for all the philosophers to be eating concurrently. By symmetry, checking two specific adjacent philosophers is sufficient; if these two philosophers can not be eating concurrently, no two adjacent philosophers can. In our experiment, we check this property for philosopher 1 and philosopher 2. For ease of reference, we call this property no_p1p2. We note that the property specifications for no_p1p2 are essentially the same as the property specifications to check for two writers writing concurrently in the readers/writers problem.

The never claim for no_p1p2 is shown in Figure 4.27. The FSA for the never claim stays in the initial state until both philosopher_1 and philosopher_2 are at s3; in other

words, both philosophers are eating. If this occurs, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim.

```
never {
 do
  :: phil__1[phil_1_pid]@state_3 &                    -- if philosophers 1 and 2 are both eating,
     phil__2[phil_2_pid]@state_3 -> goto accept       -- go to accept state
  :: else -> skip                                     -- otherwise, loop back
 od;
accept:                                               -- accept state of FSA
 do
  :: skip                                   -- infinite loop; philosophers 1 and 2 both
 od                                                   -- eating has been found
}
```

Figure 4.27. Never Claim for no_p1p2

The assertions to check no_p1p2 are shown in Figure 4.28. When philosopher_1 starts to eat, the flag indicating that philosopher_1 is eating is set and the assertion that philosopher_2 is not eating is checked. Before philosopher_1 stops eating, the flag indicating that philosopher_1 is eating is cleared. We note that the flag can not be cleared after philosopher_1 stops eating, because SPIN then finds a violation of the assertion by having philosopher_2 start to eat before the flag is cleared. Similar assertions are embedded in the philosopher_2 process.

philosopher_1

```
    . . .
s2: if
    :: fork__1_up!synch ->
         atomic { phil_1_eating = true;
                  assert (phil_2_eating == false);
                  goto s3 }
    fi;
s3: phil_1_eating = false;
    if
    :: fork_2_down!synch -> goto s4
    fi
    . . .
```

philosopher_2

```
    . . .
s2: if
          :: fork__2_up!synch ->
       atomic { phil_2_eating = true;
          assert (phil_1_eating == false);
          goto s2 }
    fi;
    s3:   phil_2_eating = false;
    if
    :: fork_3_down!synch -> goto s4
    fi
    . . .
```

Figure 4.28. Assertions for no_p1p2

The SMV specification for no_p1p2 is shown in Figure 4.29. Essentially, the specification states that Always, Globally, if philosopher_1 is eating philosopher_2 is not eating and if philosopher_2 is eating philosopher_1 is not eating.

```
SPEC
  AG   ( ( ( phil__1 = s3 ) -> !( phil__2 = s3 ) ) &
         ( ( phil__2 = s3 ) -> !( phil__1 = s3 ) ) )
```

Figure 4.29. SMV Specification for no_p1p2

The INCA query for no_p1p2 is shown in Figure 4.30. We specify an interval, starting at the initial state of the program, that ends after philosopher_1 and philosopher_2 have both started eating an arbitrary number of times. If such an interval exists, it is possible for philosopher_1 and philosopher_2 to be eating concurrently.

```
(defquery "no_p1p2" "nofair"
   (omega-star-less (sequence
      (interval :initial t
            :open t
            :ends-with '((rend "phil_1;fork_1.up")
                        (rend "phil_2;fork_2.up"))))))
```

Figure 4.30. INCA Query for no_p1p2

The FLAVERS QRE for no_p1p2 is shown in Figure 4.31. The events of interest are when philosopher_1 and philosopher_2 start and stop eating. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "Any event but philosopher_1 or philosopher_2 starting to eat occurs 0 or more times, then either philosopher_1 starts eating and stops eating without an intervening philosopher_2 starting to eat, or philosopher_2 starts eating and stops eating without an intervening philosopher_1 starting to eat, then any events but philosopher_1 or philosopher_2 starting to eat occurs 0 or more times".

```
{phil_1_start_eating, phil_1_stop_eating,
 phil_2_start_eating, phil_2_stop_eating} all

 [-phil_1_start_eating, phil_2_start_eating]*;
(((phil_1_start_eating;
   [-phil_2_start_eating, phil_1_stop_eating]*;
   phil_1_stop_eating)
  |
  (phil_2_start_eating;
   [-phil_1_start_eating, phil_2_stop_eating]*;
   phil_2_stop_eating));
 [-phil_1_start_eating, phil_2_start_eating]*)*
```

Figure 4.31. QRE for no_p1p2

4.3.2 Dining Philosophers with Dictionary (dpd)

In this variation of the standard dining philosophers problem, the philosophers eat and think as described above, but also pass a dictionary around the table. The philosopher currently holding the dictionary can not be eating, since it can not pick up any forks until it passes the dictionary to the next philosopher. This removes the possibility of deadlock in the system.

As with the standard version, we check for deadlock and for philosopher_1 and philosopher_2 eating concurrently. We also check a third property, which can be stated "Can philosopher i ever start eating while holding the dictionary?" By symmetry we can check this for a single philosopher and generalize the results to most of the philosophers in the system (all philosophers but philosopher_1), so we check this property for philosopher 2. Because philosopher_1 starts out holding the dictionary, and all other philosophers start out not holding the dictionary, our symmetry argument only applies to philosophers 2 through N for a size N version of this program For notational convenience we call this property no_p2d.

The property specifications for deadlock and no_p1p2 are as described for dp, with the minor change that philosopher_1 is eating in state 5 and philosopher_2 is eating in state 4 in this variation of the problem. The property specifications for no_p2d are provided below.

The never claim for no_p2d is shown in Figure 4.32. To check this property, we needed to add an additional variable to keep track of whether or not philosopher_2 was holding the dictionary. The **holding_dictionary** variable is set to false when philosopher_2 hands off the dictionary and set to true when philosopher_2 accepts the dictionary. The FSA for the never claim stays in the initial state until philosopher_2 is in state 4 (eating) and philosopher_2 is holding the dictionary (**holding_dictionary** == true). If this occurs, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim.

```
never {
  do
    :: phil__2[phil_2_pid]@state_4 &                  -- if philosopher 2 is eating and holding
       (holding_dictionary == true) -> goto accept   -- the dictionary, go to accept state
    :: else -> skip                                   -- otherwise, loop back
  od;
accept:                                               -- accept state of FSA
  do
    :: skip
  od
}
```

Figure 4.32. Never Claim for no_p2d

The assertions to check for philosopher_2 eating while holding the dictionary are shown in Figure 4.33. When philosopher_2 starts to eat, the assertion that philosopher_2 is not holding the dictionary is checked. As for the never claim, we use the **holding_dictionary** variable to recognize whether or not philosopher_2 is holding the dictionary.

```
philosopher_2
. . .
  :: phil__2_dictionary?synch -> atomic { holding_dictionary = true;
                                          goto state_6 }
  fi;
state_3:
  if
  :: fork__2_up!synch -> atomic { assert (holding_dictionary == false);
                                  goto state_4 }
  fi;
      . . .
  :: phil__3_dictionary!synch -> atomic { holding_dictionary = false;
                                          goto state_1 }
      . . .                                               . . .
```

Figure 4.33. Assertions for no_p2d

The SMV specification for no_p2d is shown in Figure 4.34. Essentially, the specification states that Always, Globally, if philosopher_2 is eating then philosopher_2 is not holding the dictionary. As for the PROMELA programs, the **holding_dictionary** variable is set to false when philosopher_2 hands off the dictionary and set to true when philosopher_2 accepts the dictionary.

```
SPEC
  AG  ( ( phil_2 = s3 ) -> !holding_dictionary )
```

Figure 4.34. SMV Specification for no_p2d

Alternatively, we can avoid using the **holding_dictionary** variable by using the alternate CTL specification shown in Figure 4.35. The specification states that Always, Globally, if philosopher_2 is in state 6 (holding the dictionary), philosopher_2 can not go to state 4 (eating) until it has gone to state 1 (handed off the dictionary). An interesting side effect of using this specification is that we had to add a fairness constraint to check the property. The semantics of the Until operator require that (phil__2 = s1) be true at some time in the future, otherwise the formula evaluates to false. Since there are executions in which philosopher_2 accepts the dictionary but never passes it off again (essentially, philosopher_2 "starves" holding the dictionary), the specification evaluates to false without the fairness constraint. The fairness constraint specifies that philosopher_2 enters state 1 (passes off the dictionary) infinitely often, at which point we can successfully check the property.

```
FAIRNESS
  (phil__2 = s1)
SPEC
  AG ( ( phil__2 = s6 ) -> A [ !( phil__2 = s4 ) U ( phil__2 = s1 ) ] )
```

Figure 4.35. Alternate SMV Specification for no_p2d

It is important to note that, by adding the fairness constraint, we have changed the property, at least in some sense. The specification still checks whether or not philosopher_2 can eat while holding the dictionary, but including the constraint may have eliminated a large number of program executions that SMV would have had to consider without the constraint. Although we were unable to formulate the property without the fairness constraint (and without additional variables), it may be possible to do so, with the resulting property equivalent to no_p2d.

The INCA query for no_p2d is shown in Figure 4.36. We specify an interval starting at the initial state of the program that ends after philosopher_1 has handed the dictionary

to philosopher_2 an arbitrary number of times. We then specify a second interval in which philosopher_2 starts to eat, but does not hand off the dictionary in the interval. If such a pair of intervals exists, it is possible for philosopher_2 to be eating while holding the dictionary.

```
(defquery "no_p2d" "nofair"
   (omega-star-less (sequence
        (interval :initial t
                 :open t
                 :ends-with '((rend "phil_1;phil_2.dictionary")))
        (interval :ends-with '((rend "phil_2;fork_2.up"))
                 :forbid  '((rend "phil_2;phil_3.dictionary"))))))
```

Figure 4.36. INCA Query for no_p2d

Alternatively, we can avoid using multiple intervals in the query by adding an additional constraint to the query as shown in Figure 4.37. The query specifies an interval, starting at the initial program state, ending after philosopher_2 has started eating an arbitrary number of times, in which philosopher_2 has accepted the dictionary more times than it has passed off the dictionary. If such an interval exists, philosopher_2 can eat while holding the dictionary.

```
(defquery "no_p2d_con" "nofair"
  (omega-star-less (sequence
    (interval :initial t
         :open t
         :ends-with '((rend "phil_2;fork_2.up"))
         :constraints '((<= (- "call(phil_2;phil_3.dictionary)"
                               "call(phil_1;phil_2.dictionary)")
                            -1))))))
```

Figure 4.37. Alternate INCA Query for no_p2d

The FLAVERS QRE for no_p2d is shown in Figure 4.38. The events of interest are when philosopher_2 and philosopher_3 accept the dictionary and when philosopher_2 starts eating. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "The first event of interest to occur is philosopher_2 accepting the dictionary (philosopher_1 hands off the dictionary to philosopher_2), then

philosopher_2 hands off the dictionary without eating in the interim, then events other than philosopher_2 accepting the dictionary occur 0 or more times".

```
{phil_2_dictionary, phil_3_dictionary, phil_2_eating} all

[-phil_2_dictionary]*;
(phil_2_dictionary;
 [-phil_2_eating,phil_3_dictionary]*;
 phil_3_dictionary;
 [-phil_2_dictionary]*)*
```

Figure 4.38. QRE for no_p2d

4.3.3 Dining Philosophers with Fork Manager (dpfm)

In this variation of the dining philosophers problem we replace the fork tasks with a single fork manager that keeps track of the status of all the forks in the system. To start eating, a philosopher calls a single entry in the fork manager task, and to stop eating the philosopher calls a different entry in the fork manger task. The possibility of deadlock is removed, and the fork manager task enforces the constraint that no two adjacent philosophers can be eating concurrently.

The property specifications for deadlock and no_p1p2 are as described above, with the minor change that the philosophers are now eating in state 2 rather than in state 3 for the standard problem.

To accurately check no_p1p2 we need information about the fork shared between philosopher_1 and philosopher_2. To include this information in the analysis, we model the status of the fork (fork 2) between philosopher_1 and philosopher_2.

4.3.4 Dining Philosophers with Host (dph)

In this variation of the standard dining philosophers problem, the philosophers eat and think as described for the standard problem, but also must get permission from a host task to "enter the dining room" before starting to eat and to "exit the room" after finishing eating. The host task admits no more than one philosopher fewer than the number of forks in the system into the dining room, thereby avoiding deadlock.

For this variation we check for deadlock and philosopher_1 and philosopher_2 eating concurrently (no_p1p2). The property specifications are as described above, with the minor change that the philosophers are now eating in state 4 rather than in state 3 for the standard problem.

We can accurately check no_p1p2 using simply the structure of the program, but to check for deadlock we need information about the number of philosophers currently in the dining room. To include this information in the analysis, we model the philosopher count variable maintained in the host task to keep track of the number of philosophers in the dining room.

4.4 Elevator

The elevator program provides a simulation of a set of elevators; the version we use was developed by the Arcadia consortium. The elevators can be called to certain floors, sent to certain floors from within the elevator, set to idle if no requests are pending, or shut down. Our implementation of a size N elevator program consists of a simulation driver, a controller task to handle requests for the elevator to go to certain floors, an elevator task to provide an interface to the elevators, N elevator simulation tasks to simulate the N elevators, and N doorman tasks to simulate doormen for the N elevators. Because of certain limitations in the current version of the CFG to SEDL translation tool, the elevator program was not included in our experiment. We include the property specifications below for future reference.

We have selected three properties to check for the elevator program. The first of these is deadlock. The second property can be phrased as "Can an elevator ever be moved while its doors are open?" The significance of this property should be clear, since a violation could lead to severe injury. By symmetry, we can check this property on an arbitrary elevator - for our experiment, we check the property for elevator_1. For ease of reference, we call this property no_omc (for no open doors, move elevator, close doors). The third property can be phrased as "Can an elevator ever be shut down while it has

pending requests?" If this can occur, either someone will be left waiting for the elevator or, even worse, someone will be trapped within the elevator. Again by symmetry we can check this property on a single elevator, so we check it for elevator_1. When an elevator has no pending requests, its direction of motion is set to idle. For ease of reference, we call this property no_sdni (for no shut down elevator when it is not idle).

The never claim for no_omc is shown in Figure 4.39. The FSA for the never claim stays in the initial state until the error flag is set to true (which occurs if elevator_1 moves while its doors are open). If this occurs, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim.

We added two variables to help us check this property. One variable, called **door_open**, was set to true when the doors of elevator_1 were opened and false when the doors of elevator_1 were closed. The other variable, called **error**, was initialized to false and set to **door_open** when elevator_1 moved. Thus, **error** was only set to true if elevator_1 moved while its doors were open, violating no_omc.

```
never {
  do
    :: error == true -> goto accept        -- if elevator_1 moved while door open, go to accept
    :: else -> skip                        -- otherwise, loop back
  od;
accept:                                    -- accept state
  do
    :: skip                         -- infinite loop; elevator_1 moved with the doors open
  od
}
```

Figure 4.39. Never Claim for no_omc

The assertions to check no_omc are shown in Figure 4.40. When the doors of elevator_1 are opened, **door_open** is set to true. **Door_open** is then set to false in the next state, just before the doors are closed again (for the same reason as for no_w1w2 for the readers/writers problem). When elevator_1 moves, at which point it notifies the controller that it is at a certain floor, the assertion that the doors are closed (i.e., not open) is checked. If the assertion is ever false, elevator_1 can move while its doors are open.

```
doorman_1
       . . .
s3:
  if
  :: elevator_open_door_1!synch -> atomic {
                                        door_open = true;
                                        goto s4 }
  fi;
s4:
  door_open = false;
  if
  :: elevator_close_door_1!synch -> goto s5
  fi;
       . . .
elevator_1
       . . .
  :: controller_at_floor!synch -> atomic {
                                       assert(door_open == false);
                                       goto s4 }
       . . .
```

Figure 4.40. Assertions for no_omc

The SMV specification for no_omc is shown in Figure 4.41. We use a **door_open** variable and an **error** variable as described for the never claim, and specify that Always, Globally, the error does not occur.

```
SPEC
  AG ( !error )
```

Figure 4.41. SMV Specification for no_omc

The INCA query for no_omc is shown in Figure 4.42. We specify an interval, starting at the initial state of the program, that ends after the door of elevator_1 has been opened an arbitrary number of times. We specify a second interval in which elevator_1 moves but does not close its doors before moving. We can infer that elevator_1 has moved by the call on the controller.at_floor entry, since this call only occurs if the elevator moves. If the interval exists, it is possible for elevator_1 to move while its doors are open, violating no_omc.

Alternatively, we can avoid using two intervals by adding a constraint to the query as shown in Figure 4.43. The query specifies an interval, starting at the initial program state,

that ends with elevator_1 moving, and the doors of elevator_1 have been opened more times than they have been closed.

```
(defquery "no_omc" "nofair"
    (omega-star-less (sequence
        (interval :initial t
                  :open t
                  :ends-with '((rend "doorman_1;elevator.open_door_1")))
        (interval
                  :ends-with '((rend "elevator_1;controller.at_floor"))
                  :forbid   '((rend "doorman_1;elevator.close_door_1"))))))
```

Figure 4.42. INCA Query for no_omc

```
(defquery "no_omc_con" "nofair"
  (omega-star-less (sequence
    (interval : initial t
                  :ends-with '((rend "elevator_1;controller.at_floor"))
                  :constraints '((>= (- "call(doorman_1;elevator.open_door_1)"
                                        "call(doorman_1;elevator.close_door_1)")
                                 1))))))
```

Figure 4.43. Alternate INCA Query for no_omc

The FLAVERS QRE for no_omc is shown in Figure 4.44. The events of interest are when the doors of elevator_1 are opened and closed and when elevator_1 moves. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "Any event but the doors of elevator_1 being opened occurs 0 or more times, followed by the doors of elevator_1 being opened and closed without an intervening movement of elevator_1, followed by any event but the doors of elevator_1 being opened occurs 0 or more times."

```
{elevator_open_door_1, elevator_close_door_1, elevator_1_moved} all

[-elevator_open_door_1]*;
(elevator_open_door_1;
 [-elevator_close_door_1, elevator_1_moved]*;
 elevator_close_door_1;
 [-elevator_open_door_1]*)*;
```

Figure 4.44. QRE for no_omc

We note that we could not use a rendezvous on controller.at_floor to check for elevator_1 moving, because any of the elevators can call that entry. We therefore

embedded an internal event in the elevator_1 task to reflect when it was making a call on that entry.

The third property we check on the elevator program is no_sdni. The never claim for no_sdni is shown in Figure 4.45. The FSA for the never claim stays in the initial state until elevator_1 is shut down (goes to state 3) while it is not idle. We use the **elev_1_idle** flag to keep track of when elevator_1 is idle. This flag is initialized to false, set to true when elevator_1 is initialized (to idle), and set to false when the direction of movement for elevator_1 is set.

```
never {
 do
   :: elevator__1[elev_1_pid]@endstate_3 & -- if elevator_1 is shut down while
      elev_1_idle == false -> goto accept          -- it is not idle, go to accept state
   :: else -> skip                                 -- otherwise, loop back
 od;
accept:                                            -- accept state
 do
   :: skip                                -- infinite loop;  elevator_1 shut down
 od                                                -- while not idle
}
```

Figure 4.45. Never Claim for no_sdni

The assertions to check no_sdni are shown in Figure 4.46. When the elevator_1 is initialize, **elev_1_idle** is set to true, and when the direction for elevator_1 is set, **elev_1_idle** is set to false. When elevator_1 is shut down, the assertion that it is idle is checked; if this assertion is false, elevator_1 can be shut down while it still has pending requests.

The SMV specification for no_sdni is shown in Figure 4.47. We use an **elev_1_idle** variable as described above, and specify that Always, Globally, if elevator_1 is shut down then it is idle.

The INCA query for no_sdni is shown in Figure 4.48. We specify an interval starting at the initial state of the program and ending after the direction for elevator_1 has been set (i.e., elevator_1 is not idle) an arbitrary number of times. We specify a second interval in

which the system is shut down and elevator_1 is not set to idle. If such an interval exists, it is possible for the system to shut down while elevator_1 is not idle, violating no_sdni.

```
elevator_1
s1:
 if
 :: elevator_1_init?synch -> atomic { elev_1_idle = true;
                                      goto s2 }
 fi;
s2:
 if
 :: elevator_1_set_direction?synch -> atomic {
                                        elev_1_idle = false;
                                        goto s2 }
 :: elevator_1_shut_down?synch -> atomic {
                                   assert (elev_1_idle == true);
                                   goto end_s3 }
 :: controller_at_floor!synch -> goto s2
 fi;
     . . .
```

Figure 4.46. Assertions for no_sdni

```
SPEC
  AG ( ( elevator_1 = s3 ) -> elev_1_idle )
```

Figure 4.47. SMV Specification for no_sdni

```
(defquery "no_sdni" "nofair"
    (omega-star-less (sequence
        (interval :initial t
                  :ends-with '((rend "elevator;elevator_1.set_direction"))
        (interval
                  :ends-with '((rend "driver;controller.shut_down"))
                  :forbid   '((rend "elevator-task;elevator_1-task.set_idle"))))))
```

Figure 4.48. INCA Query for no_sdni

The FLAVERS QRE for no_sdni is shown in Figure 4.49. The events of interest are when elevator_1 is made idle, when elevator_1 is given a direction to move, and when the system is shut down. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "Any event but elevator_1 given a direction to move or system shut down, followed by the sequence elevator_1_set_direction; elevator_1_set_idle occurring 0 or more times, followed by system shut down."

```
{elevator_1_set_idle, elevator_1_set_direction, controller_shut_down} all

[-elevator_1_set_direction, controller_shut_down]*;
(elevator_1_set_direction;
 [-elevator_1_set_idle, controller_shut_down]*;
 elevator_1_set_idle;
 [-elevator_1_set_direction, controller_shut_down]*)*;
controller_shut_down
```

Figure 4.49. QRE for no_sdni

4.5 Gas Station

The gas station program, originally described in [HL85], provides a simulation of a self-service gas station. Customers prepay the operator for a specific pump, at which point the operator queues the customer and activates the pump. The customer then starts and stops pumping on the selected pump. The pump reports the charge to the operator, who then provides change to the customer. Our implementation of a size N gas station program consists of an operator task, 2 pump tasks, and N customer tasks.

We have selected three properties to check for the gas station program. The first of these is deadlock. The second property can be phrased as "Can two customers ever be pumping on the same pump at the same time?" By symmetry, we can check this property on an arbitrary pair of customers and an arbitrary pump - for our experiment, we check the property for customer_1, customer_2, and pump_1. For ease of reference, we call this property no_c1c2. The third property can be phrased as "Can a customer ever prepay on one pump and get change based on the charge from the other pump?" Again by symmetry we can check this property on an arbitrary customer and pump, so we check it for customer_1, prepaid on pump_1, getting change based on the charge from pump_2. For ease of reference, we call this property no_c1p2.

The never claim for no_c1c2 is shown in Figure 4.50. The FSA for the never claim stays in the initial state until customer_1 and customer_2 are in state 5 (pumping on pump_1) at the same time. If the two customers are ever pumping on pump_1 at the same time, the FSA moves to the accept state and SPIN reports the violation of the never claim and terminates.

```
never {
 do
   :: customer__1[cust_1_pid]@state_5 &                          -- if customer_1 and customer_2 are both
      customer__2[cust_2_pid]@state_5 -> goto accept             -- pumping on pump_1, go to accept state
   :: else -> skip                                               -- otherwise, loop back
 od;
accept:                                                          -- accept state
 do
   :: skip                                                 -- infinite loop; customer_1 and customer_2
 od                                                              -- were both pumping on pump_1
}
```

Figure 4.50. Never Claim for no_c1c2

The assertions to check no_c1c2 are shown in Figure 4.51. When customer_1 starts pumping on pump_1, the flag indicating that customer_1 is pumping is set and the assertion that customer_2 is not pumping is checked. Before customer_1 stops pumping, the flag indicating that customer_1 is pumping is cleared. We clear the flag before customer_1 stops pumping for the same reason as no_w1w2 in the readers/writers problem. Similar assertions are embedded in the customer_2 process.

customer_1

```
      . . .
state_4:
 if
 :: pump__1_start__pumping!synch ->
       atomic { cust_1_pumping = true;
                assert (cust_2_pumping == false);
                goto state_5 }
 fi;
state_5:
 cust_1_pumping = false;
      . . .
```

customer_2

```
      . . .
state_4:
 if
   :: pump__1_start__pumping!synch ->
         atomic { cust_2_pumping = true;
            assert (cust_1_pumping == false);
            goto state_5 }
 fi;
state_5:
 cust_2_pumping = false;
      . . .
```

Figure 4.51. Assertions for no_c1c2

The SMV specification for no_c1c2 is shown in Figure 4.52. The specification states that Always, Globally, if customer_1 is pumping on pump_1 then customer_2 is not and if customer_2 is pumping on pump_1 then customer_1 is not.

The INCA query for no_c1c2 is shown in Figure 4.53. We specify an interval, starting at the initial state of the program, which ends when both customer_1 and customer_2 have started pumping on pump_1 an arbitrary number of times. If such an

interval exists, it is possible for customer_1 and customer_2 to be pumping on pump_1 concurrently.

```
SPEC
  AG ( ( ( customer_1 = s5 ) -> !( customer_2 = s5 ) )
        &
         ( ( customer_2 = s5 ) -> !( customer_1 = s5 ) ) )
```

Figure 4.52. SMV Specification for no_c1c2

```
(defquery "no_c1c2" "nofair"
    (omega-star-less (sequence
        (interval :initial t
                  :open t
                  :ends-with '("call(cust_1-task;pump_1-task.start_pumping)"
                               "call(cust_2-task;pump_1-task.start_pumping)")))))
```

Figure 4.53. INCA Query for no_c1c2

The FLAVERS QRE for no_c1c2 is shown in Figure 4.54. The events of interest are when customer_1 and customer_2 start and stop pumping. The tool should check if the specified sequence occurs on any path. The sequence is informally specified as "Customer_1 and customer_2 start and stop pumping (without the other customer starting to pump in the interim) an arbitrary number of times, followed by either customer_1 starting to pump then customer_2 starting to pump before customer_1 stops pumping or customer_2 starting to pump then customer_1 starting to pump before customer_2 stops pumping".

We initially formulated this query as an all paths property, but the structure of the graphical representation FLAVERS generates for this program and property precluded accurately checking the all paths property. Specifically, the graph has a path from the point at which customer_1 (or customer_2) starts pumping to the terminal node for the program. Since the original property specified that the customers had to start and stop pumping on all paths, FLAVERS responded that the property did not hold. Because there does not exist a path on which the property specified above is violated, the above property can be accurately checked by FLAVERS. While one could argue that the original

specification should have been used, we followed the process described at the beginning of this chapter instead.

```
{cust_1_start_pumping,cust_1_stop_pumping,
 cust_2_start_pumping,cust_2_stop_pumping} none

[-cust_1_start_pumping,cust_2_start_pumping]*;
(((cust_1_start_pumping;
   [-cust_2_start_pumping,cust_1_stop_pumping]*;
   cust_1_stop_pumping)
  |
  (cust_2_start_pumping;
   [-cust_1_start_pumping,cust_2_stop_pumping]*;
   cust_2_stop_pumping));
 [-cust_1_start_pumping,cust_2_start_pumping]*)*;
((cust_1_start_pumping;
  [-cust_2_start_pumping,cust_1_stop_pumping]*;
  cust_2_start_pumping)
 |
 (cust_2_start_pumping;
  [-cust_1_start_pumping,cust_2_stop_pumping]*;
  cust_1_start_pumping));
[cust_1_start_pumping,cust_1_stop_pumping,
 cust_2_start_pumping,cust_2_stop_pumping]*
```

Figure 4.54. QRE for no_c1c2

The third property we check on the gas station program is no_c1p2. The never claim for no_c1p2 is shown in Figure 4.55. The FSA for the never claim stays in the initial state until customer_1 has prepaid on pump_1 and received change based on the charge for pump_2. If this ever occurs, the FSA moves to the accept state and SPIN reports the violation of the never claim and terminates.

We use some additional variables in the PROMELA to keep track of when customer_1 has prepaid on pump_1 (**prepay_1_pump_1**) and received change based on the charge for pump_2 (**cust_1_pump_2_change**). When customer_1 prepays on pump_1, **prepay_1_pump_1** is set to true and **cust_1_pump_2_change** is set to false. When customer_1 receives change based on the charge for pump_1, the **prepay_1_pump_1** flag is set to false. When customer_2 receives change based on the charge for pump_2, **cust_1_pump_2_change** is set to true. Therefore, if customer_1

prepays on pump_1 and receives change based on the charge for pump_2, both variables are set to true and the never claim catches the property violation.

```
never
{
 do
   :: prepay_1_pump_1 == true &                        -- if customer_1 prepaid on pump_1 but
      cust_1_pump_2_change == true -> goto accept      -- got change for pump_2, go to accept
   :: else -> skip                                     -- otherwise, loop back
 od;
accept:                                                -- accept state
 do
    :: skip                                            -- infinite loop; customer_1 got the wrong
change
 od
}
```

Figure 4.55. Never Claim for no_c1p2

The assertions to check no_c1p2 are included at a number of states in the operator task; an example is shown in Figure 4.56. We use the **prepay_1_pump_1** variable as described above. When customer_1 receives change based on the charge from pump_2 (as in state_22), the assertion that customer_1 did not prepay on pump_1 is checked. If this assertion is ever false, customer_1 prepaid on pump_1 but received change based on the charge for pump_2.

operator

```
      . . .
state_22:
 if
 :: customer__1_task_change!synch -> atomic {
                                assert (prepay_1_pump_1 == false);
                                goto state_2 }
 fi;
      . . .
```

Figure 4.56. Assertions for no_c1p2

The SMV specification for no_c1p2 is shown in Figure 4.57. The variables **prepay_1_pump_1** and **cust_1_pump_2_change** are used as described for the never claim above. The specification states that Always, Globally, if customer_1 has just received change based on the charge for pump_2 then customer_1 did not prepay on pump_1.

```
SPEC
  AG ( cust_1_pump_2_change -> !prepay_1_pump_1 )
```

Figure 4.57. SMV Specification for no_c1p2

Alternatively, we can avoid modeling the additional variables in SMV by using the alternate CTL specification shown in Figure 4.58. The specification states that if customer_1 enters state 4 (has prepaid on pump 1), customer_1 can not enter state 1 (just received change) until the operator has entered states 14, 20, 24, or 29 (received charge from pump_1). We also had to add a fairness constraint to ensure the customer_1 task does not "starve" waiting for its change. We note that adding the fairness constraint changes the property somewhat.

```
FAIRNESS
  ( customer__1_task = s1 )
SPEC
  AG ( ( customer__1_task = s4 ) -> A [ !( customer__1_task = s1 ) U
                                                ( ( operator_task = s29 ) |
                                                  ( operator_task = s24 ) |
                                                  ( operator_task = s14 ) |
                                                  ( operator_task = s20 ) ) ] )
```

Figure 4.58. Alternate SMV Specification for no_c1p2

The INCA query for no_c1p2 is shown in Figure 4.59. We specify an interval, starting at the initial state of the program, that ends after customer_1 has prepaid on pump_1 an arbitrary number of times. We specify a second interval that ends with the operator giving customer_1 its change, contains pump_2 providing the charge to the operator, and forbids pump_1 providing a charge to the operator. If such a pair of intervals exist, it is possible for customer_1 to receive change based on the charge for pump_2 after prepaying on pump_1, violating no_c1p2.

Alternatively, we can avoid using multiple intervals by adding a constraint as shown in Figure 4.60. The query specifies an interval, starting at the initial state of the program, ending after the operator has given customer_1 change an arbitrary number of times, in which the number of times customer_1 has prepaid on pump_1 is at least 2 greater than the number of times pump_1 has provided a charge for customer_1 to the operator. Note

that it is valid for the number of prepays to be one greater, which occurs when customer_1 has prepaid on pump_1 but has not yet finished pumping.

```
(defquery "no_c1p2" "nofair"
    (omega-star-less (sequence
      (interval :initial t
                :open t
                :ends-with '((rend "operator-task;customer_1-task.operator-prepay_1_pump_1-end")))
      (interval
                :ends-with '((rend "operator-task;customer_1-task.change"))
                :require   '((rend "pump_2-task;operator-task.charge_1_pump_2"))
                :forbid    '((rend "pump_1-task;operator-task.charge_1_pump_1"))))))
```

Figure 4.59. INCA Query for no_c1p2

```
(defquery "no_c1p2_con" "nofair"
  (omega-star-less (sequence
    (interval :initial t
        :open t
        :ends-with '((rend "operator-task;customer_1-task.change"))
        :constraints '((>= (- "call(operator-task;customer_1-task.operator-prepay_1_pump_1-end)"
                              "call(pump_1-task;operator-task.charge_1_pump_1)")
                          2))))))
```

Figure 4.60. Alternate INCA Query for no_c1p2

The FLAVERS QRE for no_c1p2 is shown in Figure 4.61. The events of interest are when customer_1 prepays on pump_1, when customer_1 receives change based on the charge for pump_1, and when customer_1 receives change based on the charge for pump_2. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "Whenever customer_1 prepays on pump_1, customer_1 receives change based on the charge for pump_1 without an intervening event where customer_1 receives change based on the charge for pump_2.

```
{operator_prepay_1_pump_1,cust_1_pump_1_change,cust_1_pump_2_change} all

[-operator_prepay_1_pump_1]*;
(operator_prepay_1_pump_1;
 [-cust_1_pump_1_change,cust_1_pump_2_change]*;
 cust_1_pump_1_change;
 [-operator_prepay_1_pump_1]*)*
```

Figure 4.61. QRE for no_c1p2

We can check no_c1c2 and no_c1p2 accurately without modeling any of the variables in the program. Without modeling variables, however, we receive spurious results saying that deadlock is possible. To remove these spurious results, we need to model the variables that keep track of the numbers of active customers on pump_1 and pump_2.

4.6 Hartstone

The hartstone problem is based on the hartstone benchmark program, which iteratively starts and stops a series of tasks, collecting information about whether or not each of the tasks meets certain timing deadlines. The problem commonly analyzed in the literature (and here as well) abstracts away the timing information, retaining the iterative start/stop communication structure. Our implementation of a size N hartstone program consists of a set of N tasks, each of which iteratively accepts a start/stop sequence or terminates, and a main task that iteratively starts and then stops the N tasks using for loops.

We have selected two properties to check for the hartstone program. The first of these is deadlock. The second property can be phrased as "On any iteration in the main task, can task_3 be started before task_2?" This property checks to see if the start ordering is preserved in the main task. For ease of reference, we call this property no_t3t2.

The never claim for no_t3t2 is shown in Figure 4.62. The FSA for the never claim stays in the initial state until the error condition (task_3 starting before task_2 on some iteration) is true. If this ever occurs, the FSA moves to the accept state and SPIN reports the violation of the never claim and terminates.

We have used two additional PROMELA variables to keep track of the status of task_2 and the error condition. When task_2 is started, the variable **t_2_started** is set to true and when task_2 is stopped, **t_2_started** is set to false (clearing the flag for the next iteration). If task_3 is started and the **t_2_started** variable is false, the **error** variable is

set to true. This indicates that task_3 has started before task_2 on some iteration, and SPIN reports the violation of the never claim and terminates.

```
never {
 do
   :: error == true -> goto accept          -- if the error condition occurs, go to accept state
   :: else -> skip                          -- otherwise, loop back
 od;
accept:                                     -- accept state
 do
   :: skip                          -- infinite loop; task_3 started before task_2 on some iteration
 od
}
```

Figure 4.62. Never Claim for no_t3t2

The assertions to check no_t3t2 are shown in Figure 4.63. When task_2 is started, we set the **t_2_started** variable to true. When task_3 is started, we check the assertion that task_2 was started and set the **t_2_started** variable to false to clear it for the next iteration. If the assertion is ever false, task_3 can start before task_2 on some iteration, and SPIN reports the violation and terminates.

```
main
    . . .
state_2:
 if
 :: t__2_start!synch -> atomic { t_2_started = true;
                        goto state_3 }
 fi;
state_3:
 if
 :: t__3_start!synch -> atomic { assert (t_2_started == true);
                        t_2_started = false;
                        goto state_4 }
 fi;
    . . .
```

Figure 4.63. Assertions for no_t3t2

The SMV specification for no_t3t2 is shown in Figure 4.64. The specification states that Always, Globally, task_2 goes first (i.e., before task_3). We use two variables to keep track of the status of task_2 and the fact that task_2 went first. When task_2 is started, the **t_2_started** variable is set to true. When task_3 is started, the **t_2_first** variable is set to **t_2_started** and the **t_2_started** variable is set to false. If task_3 is ever

started when **t_2_started** is false (task_2 has not been started yet), **t_2_first** is set to false, and the SMV specification is then false.

```
SPEC
  AG ( t_2_first )
```

Figure 4.64. SMV Specification for no_t3t2

Alternatively, we can avoid modeling the additional variables in SMV by using the alternate CTL specification shown in Figure 4.65. The specification states that Always, Globally, if t__1 is in state 3 (just been started), indicating the start of an iteration starting and stopping the tasks, then t__3 is not in state 3 (started) until t__2 is in state 3 (started).

```
SPEC
  AG ( ( t__1 = s3 ) -> A [ !( t__3 = s3 ) U ( t__2 = s3 ) ] )
```

Figure 4.65. Alternate SMV Specification for no_t3t2

The INCA query for no_t3t1 is shown in Figure 4.66. We specify an initial interval starting at the initial state of the program and ending, after an arbitrary number of iterations, at the beginning of the loop in the main task. The second interval ends with task_3 starting, but task_2 is not allowed to start within the interval. If such a pair of intervals exists, it is possible for task_3 to start before task_2 on some iteration of the loop, violating no_t3t2.

```
(defquery "no_t3t2" "nofair"
    (omega-star-less (sequence
        (interval :initial t
            :open t
            :ends-with '((rend "main;t_1.start")))
        (interval :ends-with '((rend "main;t_3.start"))
            :forbid    '((rend "main;t_2.start"))))))
```

Figure 4.66. INCA Query for no_t3t2

Alternatively, we can avoid using a query with multiple intervals by adding a constraint as shown in Figure 4.67. We specify an interval, starting at the initial program state, in which the number of times t_3 has been started is greater than the number of times t_2 has been started.

```
(defquery "no_t3t2_con" "nofair"
    (omega-star-less (sequence
        (interval :initial t
            :constraints '((>= (- "call(main;t_3.start)"
                                  "call(main;t_2.start)")
                             1))))))
```

Figure 4.67. Alternate INCA Query for no_t3t2

The FLAVERS QRE for no_t3t1 is shown in Figure 4.68. The events of interest are when task_1 starts and when task_3 starts. The tool should check that the specified sequence could occur on some path. The sequence is informally specified as "The sequence task_1 starting; task_3 starting occurs 0 or more times, followed by task_3 starting before task_1." The set of tasks that are started and stopped in this program is specified as an array of task types. FLAVERS does not currently support using elements of arrays of task types as communicating tasks, so FLAVERS was not used to check no_t3t2 in the experiment.

```
{task_1_start, task_3_start} none

[-task_1_start,task_3_start]*;
(task_1_start;
 [-task_3_start]*;
 task_3_start;
 [-task_1_start,task_3_start]*)*;
task_3_start;
[task_1_start,task_3_start]*
```

Figure 4.68. QRE for no_t3t2

4.7 Memory Management

The memory management problem is based on the conservative release and allocate memory management algorithms in [For88]. The problem consists of a set of user tasks, an allocation procedure that allocates memory to the users from three memory sources, a release procedure that frees memory no longer required by the users, and a mechanism for enforcing critical sections and atomic actions. Our implementation of a size N memory management problem consists of N user tasks, an allocate procedure with three additional procedures to support allocation, a release procedure with two additional procedures to support releasing memory, a task to enforce critical sections, a task to enforce atomic

actions, a task to monitor when all users are done, a procedure to shut down the system, and a driver task to start and stop the system.

We have selected three properties to check for the memory management program. The first of these is deadlock. The second property can be phrased as "Can two users ever be in the critical section at the same time?" If we can prove that an arbitrarily selected pair of users can not be in the critical section concurrently, we can show that mutual exclusion is enforced for the critical section. By symmetry, checking two specific users is sufficient; if these two users can not be using the resource concurrently, no two users can. In our experiment, we check this property for user_1 and user_2. For ease of reference, we call this property no_u1u2. The third property can be phrased as "Can the system ever be shut down while a user is allocating memory?" If this property is possible, the system could shut down before all users were done. By symmetry, checking this property for an arbitrary user is sufficient. In our experiment, we check this property for user_1. For ease of reference, we call this property no_sdu1a (for no shut down while user 1 allocating).

The never claim for no_u1u2 is shown in Figure 4.69. The FSA for the never claim stays in the initial state until both user_1 and user_2 are in the critical section. If this occurs, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim.

The assertions to check no_u1u2 are shown in Figure 4.70. When user_1 enters the critical section, **user_1_in_crit_sect** is set to true and the assertion that user_2 is not in the critical section is checked. Before user_1 leaves the critical section, the **user_1_in_crit_sect** variable is set to false. Similar assertions are embedded in the user_2 process. There are actually several places in each user task where the user enters or leaves the critical section, so we have shown a representative example of the assertions.

```
never {
 do
  :: ( user__1[user_1_pid]@state_3 | user__1[user_1_pid]@state_5 | user__1[user_1_pid]@state_6 |
      user__1[user_1_pid]@state_7 | user__1[user_1_pid]@state_8 | user__1[user_1_pid]@state_9 |
      user__1[user_1_pid]@state_11 | user__1[user_1_pid]@state_13 | user__1[user_1_pid]@state_14 |
      user__1[user_1_pid]@state_15 | user__1[user_1_pid]@state_16 | user__1[user_1_pid]@state_17 |
      user__1[user_1_pid]@state_18 | user__1[user_1_pid]@state_19 | user__1[user_1_pid]@state_20 |
      user__1[user_1_pid]@state_22 | user__1[user_1_pid]@state_23 | user__1[user_1_pid]@state_24 |
      user__1[user_1_pid]@state_25 | user__1[user_1_pid]@state_26 | user__1[user_1_pid]@state_31 |
      user__1[user_1_pid]@state_32 | user__1[user_1_pid]@state_33 | user__1[user_1_pid]@state_34 |
      user__1[user_1_pid]@state_35 | user__1[user_1_pid]@state_36 | user__1[user_1_pid]@state_37 |
      user__1[user_1_pid]@state_38 ) &
     ( user__2[user_2_pid]@state_3 | user__2[user_2_pid]@state_5 | user__2[user_2_pid]@state_6 |
      user__2[user_2_pid]@state_7 | user__2[user_2_pid]@state_8 | user__2[user_2_pid]@state_9 |
      user__2[user_2_pid]@state_11 | user__2[user_2_pid]@state_13 | user__2[user_2_pid]@state_14 |
      user__2[user_2_pid]@state_15 | user__2[user_2_pid]@state_16 | user__2[user_2_pid]@state_17 |
      user__2[user_2_pid]@state_18 | user__2[user_2_pid]@state_19 | user__2[user_2_pid]@state_20 |
      user__2[user_2_pid]@state_22 | user__2[user_2_pid]@state_23 | user__2[user_2_pid]@state_24 |
      user__2[user_2_pid]@state_25 | user__2[user_2_pid]@state_26 | user__2[user_2_pid]@state_31 |
      user__2[user_2_pid]@state_32 | user__2[user_2_pid]@state_33 | user__2[user_2_pid]@state_34 |
      user__2[user_2_pid]@state_35 | user__2[user_2_pid]@state_36 | user__2[user_2_pid]@state_37 |
      user__2[user_2_pid]@state_38 ) -> goto accept          -- go to accept state
  :: else -> skip                                            -- otherwise, loop back
 od;
accept:                                                      -- accept state
 do
  :: skip;                                                   -- infinite loop
 od
}
```

Figure 4.69. Never Claim for no_u1u2

user_1

```
    . . .
 :: crit_sect_cs_start!synch -> atomic {
          user_1_in_crit_sect = true;
          assert(user_2_in_crit_sect == false);
          goto state_21 }
    . . .
state_25:
 user_1_in_crit_sect = false;
 if
 :: crit_sect_cs_end!synch -> goto state_48
 fi;
    . . .
```

user_2

```
    . . .
 :: crit_sect_cs_start!synch -> atomic {
          user_2_in_crit_sect = true;
          assert(user_1_in_crit_sect == false);
          goto state_21 }
    . . .
state_25:
 user_2_in_crit_sect = false;
 if
 :: crit_sect_cs_end_synch!synch -> goto state_48
fi;
    . . .
```

Figure 4.70. Assertions for no_u1u2

The SMV specification for no_u1u2 is shown in Figure 4.71. The specification states that Always, Globally, if user_1 is not out of the critical section (i.e., it is in the

critical section) then user_2 is out of the critical section and if user_2 is not out of the critical section then user_1 is out of the critical section.

```
SPEC
 AG ( ( ( !( user__1 = s1 ) &  !( user__1 = s2 ) & !( user__1 = s4 ) & !( user__1 = s12 ) &
          !( user__1 = s21 ) & !( user__1 = s27 ) & !( user__1 = s28 ) ) ->
                                                   ( ( user__2 = s1 ) | ( user__2 = s2 ) | ( user__2 = s4 ) |
                                                    ( user__2 = s12 ) | ( user__2 = s21 ) | ( user__2 = s27 ) |
                                                    ( user__2 = s28 ) ) )
    &
    ( ( !( user__2 = s1 ) & !( user__2 = s2 ) & !( user__2 = s4 ) & !( user__2 = s12 ) &
       !( user__2 = s21 ) & !( user__2 = s27 ) & !( user__2 = s28 ) ) ->
                                                   ( ( user__1 = s1 ) | ( user__1 = s2 ) | ( user__1 = s4 ) |
                                                    ( user__1 = s12 ) | ( user__1 = s21 ) | ( user__1 = s27 ) |
                                                    ( user__1 = s28 ) ) ) )
```

Figure 4.71. SMV Specification for no_u1u2

The INCA query for no_u1u2 is shown in Figure 4.72. We specify an interval starting at the initial state of the program in which both user_1 and user_2 enter the critical section, and neither one is allowed to leave the critical section. If such an interval exists, it is possible for user_1 and user_2 to be in the critical section concurrently.

```
(defquery "no_u1u2" "nofair"
   (omega-star-less (sequence
      (interval :initial t
                :open t
                :ends-with '("call(user_1-task;crit_sect-task.cs_start)"
                   "call(user_2-task;crit_sect-task.cs_start)")))))
```

Figure 4.72. INCA Query for no_u1u2

The FLAVERS QRE for no_u1u2 is shown in Figure 4.73. The events of interest are when user_1 and user_2 enter and leave the critical section. The tool should check if the specified sequence occurs on any path. The sequence is informally specified as "User_1 and user_2 enter and leave the critical section (without the other user entering the critical section in the interim) an arbitrary number of times, followed by either user_1 entering the critical section then user_2 entering the critical section before user_1 leaves the critical section or user_2 entering the critical section then user_1 entering the critical section before user_2 leaves the critical section".

```
{user_1_in_crit_sect,user_1_not_in_crit_sect,
 user_2_in_crit_sect,user_2_not_in_crit_sect} none

[-user_1_in_crit_sect,user_2_in_crit_sect]*;
(((user_1_in_crit_sect;
   [-user_2_in_crit_sect,user_1_not_in_crit_sect]*;
   user_1_not_in_crit_sect)
  |
  (user_2_in_crit_sect;
   [-user_1_in_crit_sect,user_2_not_in_crit_sect]*;
   user_2_not_in_crit_sect));
 [-user_1_in_crit_sect,user_2_in_crit_sect]*)*;
((user_1_in_crit_sect;
  [-user_2_in_crit_sect,user_1_not_in_crit_sect]*;
  user_2_in_crit_sect)
 |
 (user_2_in_crit_sect;
  [-user_1_in_crit_sect,user_2_not_in_crit_sect]*;
  user_1_in_crit_sect));
[user_1_in_crit_sect,user_1_not_in_crit_sect,
 user_2_in_crit_sect,user_2_not_in_crit_sect]*
```

Figure 4.73. QRE for no_u1u2

The third property we check on the memory management program is no_sdu1a. The never claim for no_sdu1a is shown in Figure 4.74. The FSA for the never claim stays in the initial state until the system has shut down and user_1 is still in the process of allocating memory. If this occurs, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim. We set the **user_1_allocating** variable to true when user_1 starts allocating and to false when user_1 stops allocating.

```
never {
 do
   :: final[final_pid]@endstate_4 &            -- if the system has shut down and
      user_1_allocating == true -> goto accept         -- user_1 is allocating, go to accept
   :: else -> skip                                     -- otherwise, loop back
 od;
accept:                                                -- accept state
 do
   :: skip                                     -- infinite loop; system shut down while
 od                                                    -- user_1 was allocating
}
```

Figure 4.74. Never Claim for no_sdu1a

The assertions to check no_sdu1a are shown in Figure 4.75. When user_1 starts allocating memory **user_1_allocating** is set to true and when user_1 stops allocating memory **user_1_allocating** is set to false. The system shuts down when the task named final goes to state end_s3. If the assertion is ever false, the system has shut down while user_1 was allocating, violating no_sdu1a.

```
final
      . . .
  :: mem__driver_final_go_end!synch ->
                  atomic { assert(user_1_allocating == false);
                        goto endstate_4 }
      . . .
```

Figure 4.75. Assertions for no_sdu1a

The SMV specification for no_sdu1a is shown in Figure 4.76. The specification states that Always, Globally, if the system has been shut down (final = s3) then user_1 is not allocating. The **user_1_allocating** variable is set and cleared as described above.

```
SPEC
  AG ( ( final = s3 ) -> !user_1_allocating )
```

Figure 4.76. SMV Specification for no_sdu1a

Alternatively, we can avoid using the **user_1_allocating** variable in SMV by using the alternate CTL specification shown in Figure 4.77. The specification states that Always, Globally, if user_1 is in state 2 (just started allocating), final can not go to state 4 (terminate) until user_1 goes to state 4 or 27 (stops allocating). Note that we had to also add a fairness constraint to ensure user_1 doesn't "starve" waiting to stop allocating, which changes the property somewhat.

```
FAIRNESS
  ( user__1 = s27 )
SPEC
 AG ( ( user__1 = s2 ) -> A [ !( final = s4 ) U ( ( user__1 = s4 ) |
                                                  ( user__1 = s27 ) ) ] )
```

Figure 4.77. Alternate SMV Specification for no_sdu1a

The INCA query for no_sdu1a is shown in Figure 4.78. We specify an interval, starting at the initial state of the program, that ends after user_1 has started allocating an

arbitrary number of times. We specify a second interval in which the system is shut down (at the end of the go entry) and user_1 is not allowed to stop allocating. If such an interval exists, it is possible for the system to shut down while user_1 is allocating, violating no_sd1ua.

```
(defquery "no_sdu1a" "nofair"
   (omega-star-less (sequence
      (interval :initial t
         :open t
         :ends-with '("internal(user-task_1;user_1_allocating)"))
      (interval
         :ends-with '((rend "final;driver.final-go-end"))
         :forbid '("internal(user-task_1;user_1_not_allocating)")))))
```

Figure 4.78. INCA Query for no_sdu1a

Alternatively, we can avoid using multiple intervals by adding a constraint as shown in Figure 4.79. We specify an interval, starting at the initial state of the program, that ends with system shut down, and include a constraint that user_1 has started allocating more times than it has stopped allocating (and is thus allocating at the end of the interval).

```
(defquery "no_sdu1a_con" "nofair"
  (omega-star-less (sequence
    (interval :initial t
          :ends-with '((rend "final;driver.final-go-end"))
          :constraints '((>= (- "internal(user-task_1;user_1_allocating)"
                                  "internal(user-task_1;user_1_not_allocating)")
                          1))))))
```

Figure 4.79. Alternate INCA Query for no_sdu1a

The FLAVERS QRE for no_sdu1a is shown in Figure 4.80. The events of interest are when user_1 starts and stops allocating memory and when the system shuts down. The tool should check that the specified sequence could occur on some path. The sequence is informally specified as "The sequence user_1 allocating; user_1 not allocating occurs 0 or more times, followed by user_1 allocating and the system shutting down before user_1 stops allocating."

We can accurately check all the properties without modeling any variables in the program. We also analyzed the properties while modeling a variable in each user task

that keeps track of whether or not memory has been allocated to determine impact on analysis time.

```
{user_1_allocating,user_1_not_allocating,mmgt_crit_sect_whoa} none

[-user_1_allocating,user_1_not_allocating,mmgt_crit_sect_whoa]*;
(user_1_allocating;
 [-user_1_not_allocating,mmgt_crit_sect_whoa]*;
 user_1_not_allocating;
 [-user_1_allocating,user_1_not_allocating,mmgt_crit_sect_whoa]*)*;
user_1_allocating;
[-user_1_not_allocating,mmgt_crit_sect_whoa]*;
mmgt_crit_sect_whoa;
[user_1_allocating,user_1_not_allocating,mmgt_crit_sect_whoa]*
```

Figure 4.80. QRE for no_sdu1a

4.8 Ring

The ring problem [Cor94] is based on a simulation of token ring access to a resource. The problem contains a ring of servers, each of which has an associated master. Each master iteratively requests and then releases the resource. When a master requests the resource, the associated server checks if it is holding the token. If it is, the request is granted, otherwise the server tells the next server in the ring that it needs the token and waits to grant the request until it has received the token. When the master releases the resource, the server will pass the token along if the previous server in the ring needs it. If the previous server needs the token and the server is not holding the token, the server tells the next server in the ring that it needs the token, and passes it to the previous token when it has received it. Our implementation of a size N ring program consists of N server tasks and N master tasks. Each server task uses a token variable to indicate if it holding the token or not and a using variable to indicate whether the associated master is using the resource or not.

We have selected two properties to check for the ring program. The first of these is deadlock. The second property can be phrased as "Can two masters ever be using the resource at the same time?" Because there may be a difference between two adjacent masters, two masters with another master between them, and so on, we can not use a

symmetry argument to say that checking any two masters is sufficient. For our purposes, however, we check the property for master_1 and master_2, and view a proof that the property holds as one piece of evidence needed to show that the more general property holds. For ease of reference, we call this property no_m1m2.

The never claim for no_m1m2 is shown in Figure 4.81. The FSA for the never claim stays in the initial state until both master_1 and master_2 are using the resource. If this occurs, the FSA for the never claim goes to the accept state (and never leaves it), and SPIN reports the violation of the never claim.

```
never {
 do
   :: master__1[master_1_pid]@state_3 &                    -- if master_1 and master_2 are both using
      master__2[master_2_pid]@state_3 -> goto accept       -- the resource, go to accept state
   :: else -> skip                                         -- otherwise, loop back
 od;
accept:                                                    -- accept state
 do
   :: skip
 od
}
```

Figure 4.81. Never Claim for no_m1m2

While this was an intuitive property to specify, we had to use the "atomic" PROMELA construct to force the release of the resource and the transition of the releasing master to its new state to occur as an atomic action. Otherwise, when SPIN performs the analysis it would be possible for the resource to be released by master_1 (for instance), and then for master_2 to acquire the token and enter state 3 before master_1 executed its transition out of state 3.

The assertions to check no_m1m2 are shown in Figure 4.82. When master_1 starts using the resource, the **master_1_using** variable is set to true and the assertion that master_2 is not using the resource is checked. Before master_1 stops using the resource, the **master_1_using** variable is set to false. Similar assertions are embedded in the master_2 process. If either of the assertions is ever false, master_1 and master_2 are using the resource at the same time, and no_m1m2 is violated.

```
master_1
        . . .
state_2:
  if
  :: master_1_server_1_request_end?synch -> atomic {master_1_using = true;
                                                    assert(master_2_using == false);
                                                    goto state_3 }
  fi;
state_3:
  master_1_using = false;
        . . .
```

Figure 4.82. Assertions for no_m1m2

The SMV specification for no_m1m2 is shown in Figure 4.83. The specification states that Always, Globally, if master_1 is using the resource then master_2 is not and if master_2 is using the resource master_1 is not.

```
SPEC
   AG  ( ( ( master_1 = s3 ) -> !( master_2 = s3 ) ) &
         ( ( master_2 = s3 ) -> !( master_1 = s3 ) ) )
```

Figure 4.83. SMV Specification for no_m1m2

The INCA query for no_m1m2 is shown in Figure 4.84. We specify an interval starting at the initial state of the program ending after master_1 and master_2 have started using the resource an arbitrary number of times. If such an interval exists, it is possible for master_1 and master_2 to be using the resource concurrently.

```
(defquery "no_m1m2" "nofair"
    (omega-star-less (sequence
        (interval :initial t
           :ends-with '((rend "server_1;master_1.server_1-request-end")
                        (rend "server_2;master_2.server_2-request-end"))))))
```

Figure 4.84. INCA Query for no_m1m2

The FLAVERS QRE for no_m1m2 is shown in Figure 4.85. The events of interest are when master_1 and master_2 start using the resource and release the resource. The tool should check that the specified sequence occurs on all paths. The sequence is informally specified as "Any event but master_1 or master_2 starting to use the resource occurs 0 or more times, then either master_1 starts using the resource and releases it without an intervening master_2 starting to use it, or master_2 starts using the resource

and releases it without an intervening master_1 starting to use it, then any events but master_1 or master_2 starting to use the resource occurs 0 or more times".

```
{master_1_start_using, server_1_release,
 master_2_start_using, server_2_release} all

 [-master_1_start_using, master_2_start_using]*;
(((master_1_start_using;
   [-master_2_start_using, server_1_release]*;
   server_1_release)
  |
  (master_2_start_using;
   [-master_1_start_using, server_2_release]*;
   server_2_release));
 [-master_1_start_using, master_2_start_using]*)*
```

Figure 4.85. QRE for no_m1m2

We cannot accurately check no_m1m2 when we do not model any variables in the program, but we can check it accurately by modeling the token and using variable for each server task. We therefore include analysis runs modeling these variables in the experiment.

CHAPTER 5

METRICS AND MEASUREMENTS

This chapter describes the metrics used as predictor variables and the measurements used as response variables in our experiment. We hypothesize that there are certain characteristics of programs that affect the feasibility of analysis and the accuracy of the analysis results for those programs. In addition, we believe that certain characteristics of a property being checked may also affect feasibility and analysis accuracy for that property on a given program. Our goal is to use statistical regression techniques to determine how well each of the program and property characteristics predicts the values of the response variables. The resulting regression equations can then be used as predictive models to predict each tool's analysis performance given a specific program and property.

5.1 Metrics

For our purposes, a *metric* is defined as a measurement of some characteristic of the program or property of interest. We divide our metrics into three categories: program metrics, internal representation metrics, and property metrics. The program metrics are used to capture characteristics of the Ada programs being analyzed. The internal representation metrics are used to capture characteristics of the set of FSAs for that program, the set of TIGs for that program, and the state space and transition relation for SMV. The property metrics are used to capture characteristics of the SPIN never claim and assertions, INCA query, and FLAVERS Property Automaton for each property. We treat the program, internal representation, and property metrics as predictor variables in the experiment.

The metrics have been selected in a number of ways. Characteristics that affect analysis feasibility based on the theoretical bounds of the techniques are included, as are

other characteristics that we believe may have an effect on analysis performance. Metrics that have been proposed in the concurrency analysis literature are also included.

5.1.1 Program Metrics

The program metrics are used to capture certain characteristics of the Ada program being analyzed. These characteristics include several measures of the size of the program, various measures of nondeterminism in the program and other characteristics of the program structure, and a metric indicating how many variables are modeled in the representations.

The theoretical upper bound on the number of possible program states for a concurrent program is exponential in the number of tasks in that program. We therefore include the number of tasks in the program (T) as one of the program metrics.

We suspect that the number of possible communications in a program affects the number of reachable states for that program. To calculate the number of communications, C_i, for a task T_i, we add the number of accept statements in the task to the number of entry calls in the task. We use two measures of communication size as metrics - the average number of communications for the set of tasks in the program, given by $C = \frac{1}{T}\left(\sum_{i=1}^{T} C_i\right)$ and the maximum number of communications in the set of tasks for the program, given by $MaxC = \max(C_i)$.

One of the characteristics of concurrent Ada programs that makes them particularly difficult to analyze is nondeterminism. None of the metrics above try to account for nondeterminism in the program being analyzed. Damerla and Shatz [DS92] propose several metrics that we also include in our experiment; the metrics are intended to quantify the nondeterminism in Ada programs. A metric called Alpha is used to account for the nondeterminism in entries when several tasks can make entry calls on those entries (entry nondeterminism). Alpha is given by $\sum_{i=1}^{e} (Calls_i - 1)$, where e is the number of entries not contained in selects and $Calls_i$ is the number of calls on entry i. The one is subtracted

because an entry with only one caller is deterministic. A metric called Alpha' is similar to Alpha, but also takes into account the *clustering* and *spreading* of entry calls. Entry calls on a given accept are clustered when they occur in a single task; entry calls on a given accept are spread when they occur in multiple tasks. For example, if all the entry calls on a given accept are clustered in the same task, the entry nondeterminism for this accept should be 0. Alpha' for a particular accept a is given by $\text{Alpha}'_a = \Pi_i (xi) * (2^T - (T+1))$, where x_i is the sum of entry calls in task i on the accept a and T is the number of tasks making calls on the accept. Alpha' is given by $\sum_{a=1}^{e} \text{Alpha}'_a$. The metric Beta is used to account for the nondeterministic selection of rendezvous within select statements (select nondeterminism). Beta is given by $\sum_{i=1}^{s} (Calls_i - 1)$, where s is the number of selects and $Calls_i$ is the number of calls on entries within select i. The one is subtracted because a select with only one call on an entry within the select is deterministic. Similarly to Alpha' , a metric Beta' is defined to account for entry call spreading and clustering. Beta' for a particular select a is given by $\text{Beta}'_a = \Pi_i (xi) * (2^T - (T+1))$, where x_i is the sum of entry calls in task i on alternatives in the select and T is the number of tasks making calls on alternatives in the select. Beta' is given by $\sum_{a=1}^{s} \text{Beta}'_a$. The metrics Gamma (Alpha + Beta) and Gamma' (Alpha' + Beta') are used to account for total nondeterminism.

Levine and Taylor [LT93] propose a metric similar to Gamma called Cnd to account for nondeterminism. Cnd is given by $\sum_{i=1}^{e+s} (Calls_i - 1) + \sum_{i=1}^{s} (Called_i - 1)$, where $Calls_i$ is the number of entry calls on entry i and $Called_i$ is the number of select alternatives with one or more callers. The difference between Cnd and Gamma is that Cnd includes entry nondeterminism for all entries (as opposed to excluding those in selects) and counts the number of select alternatives with one or more callers when calculating select nondeterminism. To account for clustering and spreading, Cnd' is defined as

$$\sum_{i=1}^{e+s} \left(\prod_{j=1}^{T_i} x_{ij} * (2^{T_i} - (T_i + 1)) \right) + \sum_{i=1}^{s} \left(\prod_{j=1}^{R_i} z_{ij} * (2^{R_i} - (R_i + 1)) \right).$$

T_i is the number of task with calls on entry or select statement i, x_{ij} is the number of entry calls in task j on entry i, R_i is the smaller of T_i and the number of alternatives of select statement i with one or more callers, and z_{ij} is the number of entry calls in task j on entry alternatives in select statement i (if $R_i = T_i$) or the number of alternatives in select statement i with one or more calls in task j.

Levine and Taylor also propose a metric, Cif, for capturing the communication structure or information flow for the tasks comprising the program. Cif is given by

$$\left(\sum_{i=1}^{T}(in - edges_i)(out - edges_i)\right),$$

where T is the number of tasks in the program, in-edges$_i$ is the sum of task entries and shared variables read in task i, and out-edges$_i$ is the sum of entry calls and shared variables written by task i. Cnd, Cnd', and Cif are also included in our experiment.

As discussed earlier, we sometimes choose to model certain variables to try to improve the accuracy of the analysis. When we do so, both the accuracy and the time to complete the analysis are almost always affected. While we capture some of the effects of this modeling indirectly through the metrics described above, we also explicitly include a metric, Vars, that specifies the number of variables that are modeled in the program.

5.1.2 Internal Representation Metrics

The internal representation metrics are used to capture characteristics of the set of FSAs for a given program, the set of TIGs for that program, and the state space and transition relation for the SMV representation of the program. These characteristics include several measures of the sizes of the representations and a measure of the graph theoretic complexity of the program in terms of TIGs.

As noted above, the upper bound for the number of states in a concurrent program is exponential in the number of tasks, T. When the program is represented by a set of FSAs,

the upper bound is given by N^T, where N is given by $\frac{1}{T}\left(\sum_{i=1} n_i\right)$ and n_i is the number of states in task i. We therefore include N as a predictor variable. We also include the maximum number of states in an FSA, $MaxN = \max(n_i)$, as a predictor variable, because a large number of states in an FSA for one of the tasks could significantly affect N. Wampler has proposed the metric $N^{T/2}$ as a good predictor of reachability graph size, at least for some programs [Wam85] and we include the WFSA (for Wampler, FSAs) metric in our experiment as well.

Because we believe the communications between the FSAs will affect the analysis, we include two measures of communication size for the FSAs, noting that in general transitions in the FSAs represent accepts or entry calls in the original program. We include the average number of transitions in the set of FSAs for the program, given by $TRANS = \frac{1}{T}\left(\sum_{i=1} Trans_i\right)$ and the maximum number of transitions in the set of FSAs for the program, given by $MaxTRANS = \max(Trans_i)$.

The above metrics can also be calculated for the set of TIGs for a given program (rather than the set of FSAs). We call the average number of nodes in the set of TIGs TN, the maximum number of nodes MaxTN, the average number of edges TE, the maximum number of edges MaxTE, and the Wampler metric WTIG (for Wampler, TIGs). We calculate these metrics for TIGs as well because a TIG is a conceptually different representation of a task than an FSA. The key difference is that the FSAs include information about choices in the task based on variable values, while TIGs abstract that information away. We note that the elision of variable information tends to yield TIGs that are smaller, in some cases much smaller, than the FSAs for the same tasks.

Levine and Taylor [LT93] propose a metric, called Cgt, intended to capture the graph theoretic complexity of the program. Cgt is given by E - N + T + 1, where E is the number of entry calls and accepts in the program, N is the number of TIG nodes in the

program, and T is the number of tasks in the program. Cgt is included as a predictor variable.

In addition to the metrics above, we also include two characteristics of the SMV system as predictor variables. We include the total number of task states (SMV St) because this number is related to the total number of sequential regions in the program. We also include the number of transitions in the transition relation (SMV Tr) as a predictor variable, because each transition represents a possible communication in the program.

5.1.3 Property Metrics

We believe that characteristics of the property being analyzed might affect the feasibility and accuracy of analysis of that property on a given program. We therefore attempt to capture characteristics of these properties through certain metrics on the property specifications for the tools. The property metrics are used to capture characteristics of the SPIN never claim and assertions, INCA query, and FLAVERS Property Automaton for each property.

Since expressing a property as an FSA seems to be a general and intuitive technique, we include three metrics on FLAVERS Property Automata to capture the size of the property. We include the size of the event alphabets (i.e., number of events of interest) and the number of states in the automaton as predictor variables. We include the number of transitions in the automaton that do not directly lead to a violation of the property, which gives us another measure of the size of the property.

We can capture the number of events of interest in the property by considering the INCA query as well, so we include the number of distinct events in the INCA query as a predictor variable. We also include the number of intervals in the INCA query, since multiple intervals in the query can significantly increase the size of the system of inequalities.

While FLAVERS QREs and INCA queries tend to be in terms of events, checking a property in SPIN entails specifying the property in terms of states. The SPIN never claim is essentially an FSA for the property, so we include the number of states and transitions in the never claim as predictor variables. We also include the number of assertions and the number of assignments to variables used in the assertions as measures of the amount of information needed to check the property.

5.2 Measurements

We consider a variety of measurements as response variables in the experiment. These measurements have been chosen as indicators of the feasibility and utility of using a particular tool to analyze a given program and property. The measurements can be broken into two categories: feasibility measurements and accuracy measurements.

The feasibility measurements are used to indicate whether each tool could be used to analyze a given program and property. The total analysis time for the program and property is a good indicator of feasibility, so we include analysis time as a response variable. Whether or not each analysis fails (takes more than 5 hours or terminates because of exhausted memory, an internal error, or inability to compile) is considered to be a good indicator of feasibility, so we include a boolean measure of failed/not failed as a response variable.

While the feasibility of using an analysis tool on a given program and property is clearly an important consideration, the utility of the tool is also determined by the accuracy of the analysis results. Whether or not each analysis yields spurious results is considered to be a good indicator of accuracy, so we include a boolean measure of spurious/not spurious as a response variable.

CHAPTER 6

STATISTICAL ANALYSIS TECHNIQUES

This chapter describes the statistical analysis techniques we use to analyze our experimental data and to generate our predictive models. We present our data collection strategy, discuss the statistical tests we apply to check for bias, explain the preprocessing required before applying the regression techniques, discuss our techniques for generating the predictive models from the data, and describe our analysis of the resulting models and their associated parameters.

All statistical analysis is performed using SPSS from SPSS Inc. and CLASP from the University of Massachusetts, Amherst.

6.1 Data Collection Strategy

To collect the data for the experiment, we attempted five analysis runs for each analysis case; each analysis case represents a certain tool/configuration/size/property combination. With the exception of INCA, which must be run in a Lisp environment, the set of analysis cases for all the tools were run in a random order. The run order was randomized to reduce caching effects, which could cause runs 2 through 5 to run more quickly than the first run for a given analysis case. For INCA, we randomized the order of INCA analysis cases, though INCA runs are not interspersed with runs of other tools since INCA is the only tool that must be run in a Lisp environment.

For each analysis case, we took the mean of the five runs as the analysis time. We also calculated confidence intervals for each of these means to ensure that we are using an analysis time that is a reasonable estimate of the "true" analysis time. Large variations in the measurement times for a given analysis case lead to wide confidence intervals, showing that we are "less sure" of the accuracy of our time measurement. Since the tools are deterministic, large confidence intervals may indicate that other factors (such as

system loading) are impacting the analysis times. In extreme cases, large confidence intervals led us to rerun the set of analysis cases.

To calculate the confidence intervals, we took the mean, $\bar{x}$, of the five runs as an estimate of the analysis time. We used the standard deviation, s, of the set of five times to calculate the estimated standard error of the mean, given by $\hat{\sigma}_{\bar{x}} = \frac{s}{\sqrt{5}}$. We then calculated the confidence interval as $\bar{x} \pm 2.776 * \hat{\sigma}_{\bar{x}}$. The 2.776 value is from a t-distribution with 4 degrees of freedom for confidence at the 0.05 level for a *two-tailed test*. Because we do not know if our mean analysis time is higher or lower than the true population mean, a two-tailed test is appropriate.

6.2 Checking for Bias Statistically

Running each tool/configuration/size/property five times also provides data for statistically checking for bias in our experiment. To explain our technique, we discuss checking to see whether using assertions rather than never claims introduces bias against SPIN, but the methodology for the checking the other biases is identical.

To check whether using assertions adversely impacted SPIN analysis times, we perform a standard form of hypothesis testing. In hypothesis testing, a *null hypothesis* (H_0) and an *alternative hypothesis* (H_1) are formed, a set of data is collected, and the probability of collecting that set of data given the null hypothesis is calculated. Note that H_1 does not have to be the exact opposite of H_0. If this probability is very small (less than 0.05 is typically considered significant), we can reject the null hypothesis (and accept the alternative hypothesis) with a small probability of doing so incorrectly. If we do not reject the null hypothesis, we have not proved it - we have simply been unable to reject it given the data at hand.

The null hypothesis for our example is that analysis times using assertions are equal to analysis times using never claims. For our alternative hypothesis, we check whether the analysis times are different. This is called a *two-tailed test*, since our alternative

hypothesis considers both ends of the distribution of possible data samples given by the null hypothesis. Strictly speaking, our hypotheses are actually concerned with the means of sets of analysis times (sets of 5, given our data collection strategy), which gives us a standard test for our hypotheses - the two sample t-test.

To use the two sample t-test, we calculate $t = \frac{\overline{x_N} - \overline{x_A}}{\hat{\sigma}_{\overline{x}_N - \overline{x}_A}}$, where $\overline{x}_N$ is the mean of the analysis times using never claims, $\overline{x}_A$ is the mean of the analysis times using assertions, and $\hat{\sigma}_{\overline{x}_N - \overline{x}_A}$ is calculated from the standard deviations of the two samples and the sample size (5). Essentially, the t value quantifies the probability that both samples were drawn from populations with equal means.

Given a t value, we reference a table (or let our software reference a table) of t-values and probabilities given the sample size. We can determine the probability of the t value given the null hypothesis, and if that probability is less than 0.05, we reject the null hypothesis and accept the alternative hypothesis that the analysis times are actually different. If this occurs, we then conduct a *one-tailed test* to check the alternative hypothesis that assertion analysis times are actually faster than never claim analysis times. If we can reject the null hypothesis for this case, this will imply that we have not introduced bias against SPIN by using assertions.

As described above, we use a two sample t-test to check the possibility of bias for a given tool/configuration/size/property. We also would like to know whether bias has been introduced over all the programs, sizes, and properties. To do this, we can use a paired-sample t-test.

In a paired sample t-test, both the never claim and assertion analyses are run on the same set of programs, sizes, and properties. For each such program/size/property, the difference between the two analysis times is calculated. The mean of the resulting distribution of differences is called $\overline{x}_\delta$ and the standard deviation of the distribution is called s_δ. The t value is given by $\frac{\overline{x}_\delta - \mu_\delta}{s_\delta / \sqrt{N}}$, with $\mu_\delta = 0$ given our null hypothesis that the

means are equal (so the mean of the differences will be 0). The t value is checked as before, and we reject the null hypothesis at the 0.05 level when possible. If we reject the null hypothesis, we should then determine whether or not using assertions yields smaller analysis times by conducting a one-tailed test.

6.3 Preprocessing the Data

It has been noted in the literature that high linear correlations between several (or many) of our predictor variables can cause problems [Bla70, HL89] in both of the regression techniques that we use. It is therefore necessary for us to preprocess our experimental data, removing predictor variables that are highly correlated to other predictors.

One relatively straightforward way to detect multicollinearity is to consider the pairwise Pearson correlation coefficients for the predictor variables [NWK85]. Pearson's correlation coefficient provides an estimate of the linear relationship between two variables x and y. The coefficient ranges from -1.0 to 1.0, with a coefficient magnitude close to 1.0 indicating a strong relationship and a magnitude close to 0.0 indicating no linear relationship. We note that a low correlation only indicates that the variables are not linearly associated; they could still be related in some non-linear way. If we find a high correlation coefficient between two predictor variables, this provides strong evidence that the variables are collinear, implying that we should elide one of them from the model.

Before omitting certain variables from the model, we would like some assurance that the correlation coefficients represent a systematic linear relationship and did not simply occur by chance. Standard statistical tests are not applicable, since our concern is about distributions of the correlation coefficients rather than distributions of the mean. However, we can use randomization tests, in conjunction with correlation, to test the hypothesis that two samples are linearly dependent [Coh95].

To conduct the randomization test, we randomly pair up values of the first and second variables and calculate the correlation coefficient. This gives us a single point in

the distribution of correlation coefficients that are possible for our set of data, given the null hypothesis that the two variables are in fact linearly independent. We then repeat the random pairing and coefficient calculation many times (in our case, 1000) to build a distribution of possible correlation coefficients. We then take the correlation coefficient with the true pairing (i.e., matching variable values for the same analysis cases) and determine where this correlation coefficient falls on the generated distribution. If the coefficient falls below the 5th value in the distribution or above the 995th value (conceptually, $p < 0.05$), we can reject the null hypothesis with high confidence, i.e., we can state that there is a linear dependence between the two variables with only a small probability that we are wrong. We conduct the randomization test on all variable pairs that have a correlation coefficient magnitude greater than 0.75. We note that randomization tests do not provide results that are generalizable to populations, so the two variables could in fact be linearly independent over the set of all possible data. The tests do, however, provide sufficient power given our specific data set.

When we discover a set of variables that are collinear to each other, we remove all but one of those variables from the regression analysis. The decision about which collinear variables to elide is not critical from the standpoint of the fit of the model we create, since the reason we're omitting the variables is because they provide the same influence as the variables we include in the model. In an effort to make the predictive models more intuitive, however, our tendency is to prefer variables representing the program metrics over those representing the internal representation metrics, and to prefer simpler internal representation metrics over more complicated ones.

6.4 Building the Models

One of the goals of our experiment is to provide a set of data on which we can apply statistical analysis techniques to generate predictive models. These predictive models can then be used by an analyst to select an appropriate analysis tool given a specific program and property. We have selected our response variables to let us predict analysis time,

whether or not an analysis will fail, and whether or not an analysis is likely to yield spurious results. We note that standard linear regression techniques are appropriate for building the predictive models of analysis time, while logistic regression is a sounder choice for predicting the dichotomous failure and spurious result responses.

6.4.1 Linear Regression

Linear regression models can be used as approximations of the functional relationship between a response variable and a set of predictor variables [MP82]. We use linear regression to build our predictive models of analysis time based on the set of metrics selected for inclusion using the preprocessing discussed above.

In linear regression, the form of the predictive model is $y = \beta_0 + \beta_1 x_1 + \ldots + \beta_i x_i + \varepsilon$, where y is the predicted value, each x_i is a metric, each β_i is a coefficient calculated using linear regression, and ε is an error term. The regression coefficients are calculated using a linear least squares fit to the data. To make the regression coefficients comparable among the metrics, we use standardized variables, which in essence puts each x_i on the same scale. The magnitudes of the resulting standardized regression coefficients can then be used to consider the relative predictive powers of each metric. Note that larger coefficients (positive or negative) indicate stronger predictive power.

To select which variables to include in the linear model, we first check for multicollinearity as described above. We then have a choice of a number of methods for selecting from the remaining variables [DS66]. One alternative is to include all the remaining variables in the linear regression. Another method, called backward elimination, starts with all the variables and iteratively removes variables that have a small effect on the predictive model. A third method, called forward selection, starts with a single variable and iteratively adds variables until the remaining variables have an insignificant effect on the predictive model. Finally, stepwise regression, an improved version of forward selection, can be used to iteratively add variables and reconsider those included in the model at each step.

The idea behind careful variable selection is to generate a parsimonious model that still captures a large amount of the variance in the data. Because our metrics are automatically calculated, we are not concerned with the cost of collecting variable information for use in the predictive model. It is possible, however, to overfit the model to the data by using more variables than are necessary. In an overfitted model, the coefficients can be numerically unstable and can change significantly with the inclusion of additional data points. The overfitted model is thus very good for predicting the relationships in the data from which it is built, but may not be as useful as a general predictive model. We therefore apply all of the above model building techniques to try to build a reasonable model. For the backward elimination technique, we use a probability of 0.10 for removal from the model; for forward selection and stepwise regression, we use a probability of 0.05 for inclusion in the model.

6.4.2 Logistic Regression

While linear regression is a widely used for predicting continuous response variables, it is not appropriate for predicting dichotomous response variables [Agr84]. Because linear regression assumes that the response variable has a continuous range of values, it can not be applied when the response variable can only have two values (true and false, for instance). Logistic regression is the proper technique for these variables, so we use logistic regression to build our predictive models for failure and presence of spurious results. The description below is largely based on information in [HL89].

In logistic regression, the *logit* is given by $g(x) = \beta_0 + \beta_1 x_1 + \ldots + \beta_k x_k$. The logit is transformed into $\pi(x) = \frac{e^{g(x)}}{1 + e^{g(x)}}$, which is used for coefficient calculation. To calculate the coefficients in the equation, a maximum likelihood function is used to calculate the effect of each data point and iterative methods are used to solve the resulting nonlinear equations. The form of the resulting predictive model is $y = \pi(x) + \varepsilon$.

To select the variables for inclusion in the logistic regression model, we preprocess the data as described above. Some statistical analysis philosophies claim that all variables that have scientific significance should be included in the model. Since it is unclear at this time which of the predictor variables (i.e., metrics) are important, we build our logistic regression models starting with all the (preprocessed) predictor variables. We build these models using three techniques: forcing all variables to be included, using backward stepwise elimination of variables, and using forward stepwise selection of variables. The stepwise techniques are analogous to those described above for linear regression. Again, we use p=0.10 for elimination and p=0.05 for inclusion. We then compare the resulting models to each other and select a reasonable model based on the criteria discussed in Section 6.5.

6.5 Analyzing the Models

After using linear or logistic regression to generate our predictive models, we examine those models in several ways. We consider goodness of fit to determine how well the model fits the data, we examine the residuals to check our assumptions about errors in the model, and we check for outliers using the residuals.

6.5.1 Goodness of Fit

To determine how well a predictive model fits the data used, we need some measure of how well the model captures the variance in the data. For linear regression, the standard measure of this is the Multiple Correlation Coefficient Squared, or R^2. R^2 is given by $R^2 = 1 - \frac{SS_E}{S_{yy}}$. The residual sum of squares, SS_E, is given by $\sum_{i=1}^{N} (y_i - \hat{y}_i)^2$, which squares the difference between the actual and predicted value of the response variable (called the *residual*) at each data point. S_{yy} is a measure of the total variability in the response variable. Thus, R^2 measures how much of the variance in the response variable is captured by the predictive model. R^2 ranges from 0 to 1, with a magnitude near 1 indicating that the model explains most of the variance in the data.

In logistic regression, the deviance can be used to measure the amount of deviation captured by the fitted model. Deviance is given by

$$D = -2\sum_{i=1}^{n}\left[y_i \ln\left(\frac{\hat{\pi}_i}{y_i}\right) + (1-y_i)\ln\left(\frac{1-\hat{\pi}_i}{1-y_i}\right)\right],$$

with the observed value at data point i given by y_i, and the estimated value of π for that point given by $\hat{\pi}_i$. The deviance in logistic regression is analogous to the residual sum of squares in linear regression. Because the deviance quantifies how much of the variance in the data is captured by a specific model (with a smaller deviance indicating a better fit), we use this value as one of our considerations when choosing between the models. We believe the percent of the predictions by the model that are correct to be an even more important consideration, so we use these values as our primary consideration when selecting a logistic regression model.

For both of the regression techniques, unrealistically large coefficients or standard errors of the coefficients are indicative of numerical problems in the analysis. They can indicate multicollinearity that was not removed by our preprocessing, and they can also support the inference that the model has been overfit to the data.

6.5.2 Residual Analysis

The above regression techniques assume that the errors (i.e., residuals) in the model are independent, have zero mean, constant variance, and follow a normal distribution [DS66]. These assumptions can be checked using plots of the standardized residuals against the predicted response values ($\hat{Y}_i = \sum_{i=1}^{N} \hat{y}_i$).

The structure we would expect to find in these plots, given our assumptions about the errors in the model, is essentially a horizontal line with residual values scattered randomly above and below zero. If we find that the "spread" of the residuals increases (or decreases) as the value of the response or predictor variable increases, we should suspect that the variance is not constant. Techniques exist to account for this problem - for

instance, using a weighted least squares fit rather than the standard least squares fit. If we find structure in the residual plots, such as an obvious quadratic component, additional quadratic or cross-product terms in the model can be used to remove this structure. Our analysis stops at recognition of such problems - we do not apply the more advanced regression techniques.

We note that visual inspection of the residual plots is a somewhat informal technique for checking our assumptions. While more formal statistical techniques have been proposed for checking these assumptions, the informal techniques are generally sufficient for recognizing serious violations of the assumptions [DS66].

6.5.3 Identifying Outliers

Outliers in our data can have a significant effect on the resulting model, particularly for linear regression. We would therefore like to recognize such outliers so we can investigate them further. It is not generally prudent to eliminate an outlier simply for statistical reasons, and we are unlikely to eliminate any outliers from the data since we don't know if the outliers are in fact more representative of "real" concurrent programs than the more normal points in our data. We do, however, want to recognize the outliers in our data to further examine them to gain insight into why these particular points are outliers.

One method for recognizing outliers is by doing so visually on the residual plots described above. Points that are significantly separated from the other points are indications of outliers, and should be investigated further. Another, more formal method, uses *studentized residuals*. A studentized residual is a residual that has basically been standardized by dividing by the square root of the variance of the residual. Given a studentized residual, we can check a table of threshold values (for a given probability) to determine if the point should be considered an outlier [DW80]. We apply both these methods to try to identify outliers in our data.

6.6 Summary of Statistical Analysis

Statistical analysis techniques are thus used in a variety of ways to process the data gathered from the experiment. Two sample and paired-sample t-tests are used to check for biases that might have been introduced by our methodology. Randomization tests are used to preprocess the data, removing extraneous collinear variables from the regressions. Linear regression is used to build predictive models for analysis time and logistic regression is used to build predictive models for failure and spurious results. The resulting models are analyzed for goodness of fit, and the residuals from the models are examined to check the assumptions of the regression techniques and to identify outliers in the data.

CHAPTER 7

EMPIRICAL RESULTS

This chapter describes the results of our experiment. We describe our experimental environment, provide the empirical comparisons of the tools in terms of analysis time, failures, and accuracy, and present the results of the statistical analysis described in Chapter 6. We close with remarks about the validity of our predictive models and the practical significance of our results.

7.1 Experimental Environment

The tools used in the experiment were SPIN version 2.7.3, SPIN+PO version 3.1 (with SPIN version 1.6.5), TRACC dated 11/29/95, SMV version 2.4.4 (with upgrade for Alphas, dated 10/11/95), INCA version 3.2, and FLAVERS dated 11/10/95.

SPIN, SPIN+PO, and SMV accept command line options that can affect the performance of these tools. In SPIN and SPIN+PO, the default depth of the reachability graph generation can be increased with the -m option. We needed to increase this depth for some of the larger problem sizes. To select a value for a given program, we selected the smallest value (within 100,000) that would let us check all properties on that program. SPIN+PO also provides a -DDEADLOCK flag that can be used when freedom from deadlock is being checked. We used this flag for all SPIN+PO runs that were checking for freedom from deadlock. For those programs with more than 32 tasks, we needed to modify a variable in one of the SPIN+PO files to allow more than 32 processes; we set this variable to 64 for those programs. When using assertions to check no_w1w2 in the presence of deadlock, we used the -c0 flag for SPIN and SPIN+PO. This forces the tools to search the entire state space; without this flag, the tools terminate on detection of the deadlock. Unfortunately, this also means the tools do not terminate on an assertion violation, but there is no way to instruct the tools to ignore deadlocks and terminate on assertion violations. Because SMV version 2.4.4 automatically enforces weak fairness,

the specification "EX 1" is always true; with CMU's assistance, we removed one line from the SMV code to allow checking "EX 1". We used the revised version of SMV for all SMV runs. We also used the SMV -f option, which calculates the reachable states of the system before checking the SPEC formula, for all SMV runs.

The experimental platform was an AlphaStation 200 4/233 with 128 MB of real memory. Virtual memory limits were set to 131072 KB for data, 2048 KB for the stack, and 121800 KB for program memory. We ran SPIN, TRACC, SMV, INCA, and FLAVERS on this platform. We were unable to build SPIN+PO on the Alpha, so we ran SPIN+PO on a SPARCstation 10 Model 40 with 32 MB of memory. Total virtual memory on the Sparc was set to 2105343, with 8192 KB for the stack and "unlimited" memory for data and program memory.

To allow comparison of SPIN+PO with the other tools, we calculated a multiplication factor for the SPARC analysis times relative to the Alpha analysis times and multiplied all SPIN+PO analysis times by this factor. To calculate the factor, we ran the first three sizes of SPIN and SMV (without -REORDER) and all sizes of SPIN+PO on the SPARC; we could not build the other tools on the SPARC. For each configuration/size/property for these tools, we calculated the ratio (Alpha Time)/(SPARC Time). We then averaged these ratios and used the result (0.376) as the multiplication factor. We use the original SPARC times when we build the predictive models and convert to Alpha time after using the models to generate a predicted (SPARC) analysis time. We also note that, because the SPARC has less memory than the Alpha, SPIN+PO could run out of memory on the SPARC on an analysis run for which the Alpha would have had sufficient memory.

7.2 Checking for Bias Statistically

Recall that we identified a number of ways in which we could inadvertently bias the experimental results based on the program representation or property specifications we used as input to the tools. Specifically, we suspected that the variable ordering in the

SMV input could introduce bias, that adding variables to check properties could introduce bias against SMV, that modeling properties as assertions rather than never claims in PROMELA could bias the results for SPIN, and that the INCA input might introduce bias because there are no accept bodies, because the tasks are uniquely specified, or because the query was specified with two intervals rather than with additional inequalities. To check for these biases, we executed the t-tests described in Section 6.2. Specifically, we executed two sample t-tests to check the possibility of bias for each program/size/property and paired sample t-tests to check for bias over all the programs, sizes and properties.

The symbolic model checking method implemented in SMV is sensitive to the size of the OBDDs generated, which is in turn sensitive to the variable ordering presented in the SMV input. To check if we introduced bias against SMV with the variable ordering in our SMV input, we ran the analysis cases both with and without the REORDER option. Our null hypothesis is that analysis times without using the REORDER option are equal to analysis times using the REORDER option. For our alternative hypothesis, we check whether analysis times using the REORDER option are smaller, because it seems to us that this option should provide a performance improvement. The result is a one-tailed test. For the two sample t-tests, in 146 cases we could reject the null hypothesis (implying that using REORDER was significantly faster than not using REORDER), in 99 cases the difference was not significant, and in 28 cases there was statistically significant evidence that not using the REORDER option was faster than using the REORDER option. For the paired sample t-test, the difference in analysis times using REORDER and not using REORDER were not statistically significant. This result was surprising, but further investigation indicated that, on programs with a ring structure (such as cyclic, dining philosophers, and so on), using the REORDER option led to larger analysis times. These larger analysis times reduced the significance of the other (smaller) analysis times enough that we do not have sufficient statistical evidence to reject the null hypothesis. Because the analysis times are not significantly different, however, and

because the two sample t-tests indicate that there are many cases in which using the REORDER option yields smaller analysis times, the analysis times included in the models below are for SMV runs using the REORDER option.

We used two different styles for specifying and checking SMV properties - embedding additional variables in the transition relation and checking properties based on those values, and developing an alternate CTL specification (without adding additional variables). We note that this choice only applies to some of the properties; many of the properties can be checked without using additional variables. The properties for which we use both styles are presented in Chapter 4. We only execute the t-tests for these properties. Our null hypothesis is that analysis times using the additional variables are equal to analysis times using the alternate CTL specification. For our alternative hypothesis, we check whether analysis times are statistically different (i.e., that they are unequal). We selected this alternative because we did not have any preliminary insight about which style would yield better performance. The result is a two-tailed test, since we are simply checking for a difference in analysis times. For the two sample t-tests, in 42 cases we had statistically significant evidence that using the additional variables was faster, in 14 cases the difference was not significant, and in 24 cases there was statistically significant evidence that using the alternate CTL specification was faster than using additional variables. For the paired sample t-test, we had statistically significant evidence ($p < 0.04$) that using the additional variables led to smaller analysis times. These t-test results indicate that for these programs, sizes and properties using additional variables is faster than using the alternate CTL specifications. All SMV analysis times included in the models below are for SMV runs using additional variables (for those properties on which this style is appropriate).

SPIN allows the user to specify properties as never claims or as assertions embedded in the PROMELA program. We specified the properties as assertions to allow comparison with the SPIN+PO results and as never claims to ensure we were not biasing

our results against SPIN by using assertions. Our null hypothesis is that analysis times using assertions are equal to analysis times using never claims. For our alternative hypothesis, we check whether analysis times are statistically different (i.e., that they are unequal). We selected this alternative because we did not have any preliminary insight about which form of property specification would yield better performance. The result is a two-tailed test. For the two sample t-tests, in 83 cases we had statistically significant evidence that using assertions was faster, in 52 cases the difference was not significant, and in three cases we had statistical evidence that using never claims was faster. For the paired sample t-test, the difference between using never claims and assertions was not statistically significant. Although using assertions was not statistically better than using never claims, it was also not statistically worse. This indicates that we have not introduced bias against SPIN by using assertions, but we must treat these results with caution. Examination of the data indicates that, for some properties, using assertions to check the property can yield significantly larger analysis times than using never claims. This is a result of our use of the -c0 flag as described above. We therefore build predictive models for both SPIN using never claims and SPIN using assertions.

There are several areas where our methodology could introduce bias against INCA. To check for these biases, we executed analysis runs for inputs with and without accept bodies, for inputs consisting of unique tasks and also with arrays of tasks, and, for some properties, with both a query with multiple intervals and a query with a single interval and an additional inequality.

For the two sample t-test for inputs with and without accept bodies, in 13 cases we had statistically significant evidence that inputs without accept bodies yield smaller analysis times, in 123 cases the difference was not statistically significant, and in 11 cases we had statistical evidence that using accept bodies was faster. For the paired sample t-test, the difference between the analysis times with and without accept bodies was not statistically significant. These results indicate that we have not introduced bias against

INCA by not using accept bodies. For the two sample t-test for inputs containing unique tasks compared to inputs containing arrays of tasks, in 66 cases we had statistically significant evidence that using unique tasks was faster, in 160 cases the difference was not statistically significant, and in four cases we had statistically significant evidence that using arrays was faster. For the paired sample t-test, the difference between the analysis times using the unique tasks and the times including these tasks in the array was not statistically significant. These results indicate that we have not introduced bias against INCA by specifying unique tasks rather than arrays of task. For the two sample t-test for inputs using multiple intervals as opposed to an additional inequality, in 23 cases we had statistically significant evidence that using multiple intervals was faster, in 3 cases the difference was not statistically significant, and in 32 cases we had statistical evidence that using the additional inequality was faster. For the paired sample t-test, the difference between the analysis times using multiple intervals as opposed to an additional inequality was not statistically significant. These results indicate that we have not introduced bias against INCA by using multiple intervals. The analysis times included in the models below are for INCA input with no accept bodies, unique tasks, and properties specified using multiple intervals rather than additional inequalities (where appropriate).

7.3 Experimental Data

The data from our experiment is too voluminous to provide here; it is, however, available from the author. Analysis times ranged from hundredths of seconds to several hours. All the tools failed on some runs, and all the tools generated some spurious results. Counts of failure and spurious results are provided explicitly in Sections 7.6 and 7.7, respectively.

It is instructive to consider briefly the input domain in terms of the metrics described in Chapter 5. The predictive models are likely to provide more predictive power within the domain in which they were developed. A user of the predictive models may thus be able to gain additional insight into the accuracy of the predictions through comparison of

the metrics for the program and property to be analyzed and the input domain of the experiment. Statistical summaries of the program and property metrics are provided in Tables 7.1 and 7.2. In the tables, we provide the minimum and maximum values for each metric to indicate the range of that metric's values. We provide the median, which is the middle value in the data, the mean, and the standard deviation to provide insight into the shape of the distribution of the metric's values.

Table 7.1. Program Metric Data for Experiment

	Mimimum	Maximum	Median	Mean	Std. Deviation
T	3	61	9	13.03	11.08
C	2.08	19.00	3.5	5.19	4.23
MaxC	4	120	6	14.22	19.24
Alpha	0	87	0	9.10	17.72
Alpha'	0	5.72E+07	0	1.00E+06	7.36E+06
Beta	0	84	9	15.57	17.83
Beta'	0	2.81E+14	44	7.06E+12	4.41E+13
Gamma	0	171	13	24.67	32.79
Gamma'	0	2.81E+14	44	7.06E+12	4.41E+13
Cnd	0	171	13	24.67	32.79
Cnd'	0	1.85E+08	32	6.92E+06	3.22E+07
Cif	0	5.69E+09	0	5.74E+07	5.27E+08
N	2.33	212.22	4.39	14.79	33.90
MaxN	3	1814	11	82.97	273.06
TRANS	3.00	1129.67	10.04	54.61	135.83
MaxTRANS	4	10045	40.50	532.87	1466.43
WFSA	3.56	1.39E+21	6118.33	2.21E+19	1.71E+20
TN	3.08	20.78	4.75	6.87	5.00
MaxTN	5	121	7.50	16.32	20.25
TE	3.68	89.78	7.43	14.62	16.71
MaxTE	5	672	20	61.73	114.25
WTIG	7.02	1.39E+21	5156.35	2.32E+19	1.79E+20
Cgt	5	631	40	76.22	104.02
SMVSt	7	1910	57	140.01	285.12
SMVTr	5	10087	98	580.58	1457.09
Vars	0	24	0	2.05	4.37

Table 7.2. Property Metric Data for Experiment

	Mimimum	Maximum	Median	Mean	Std. Deviation
QRE Alphabet	3	89	5	11.68	14.87
QRE States	3	5	4	3.87	0.66
QRE Trans	1	265	13	27.32	37.83
Query Events	2	13	2	2.81	2.08
Query Intervals	1	2	1	1.16	0.37
Never States	3	6	3	3.23	0.61
Never Trans	4	10	4	4.45	1.22
Assertions	1	20	2	3.19	4.82
Assignments	1	34	4	5.39	7.67

7.4 Analysis Time Comparisons

In this section we present the results of the analysis time comparisons, both when analysis time is measured from the native input of each tool and when total analysis time is measured.

7.4.1 Native Input Analysis Times

In this section, we provide a comparison of the analysis times starting with the native input specification for each tool. We begin with a comparison of the mean analysis times for the tools, shown in Table 7.3. We also provide standard deviations to show how much the data varies and medians to give some insight into how much outliers affect the mean.

Table 7.3. Mean Native Input Analysis Times

	Deadlock			Other Properties		
	Mean	Std Dev	Median	Mean	Std Dev	Median
SPIN, Never Claims	37.95	255.93	0.33	65.34	342.29	0.87
SPIN, Assertions	-	-	-	55.58	304.71	0.85
SPIN+PO	10.65	51.11	0.18	37.13	155.11	0.51
TRACC	13.12	13.38	6.88	18.94	37.20	4.51
SMV	106.31	534.08	0.75	46.09	202.47	1.02
INCA	40.97	209.39	2.86	11.40	30.29	2.56
FLAVERS	-	-	-	333.68	1006.66	45.58

For checking deadlock, SPIN+PO has the smallest mean analysis time, followed by TRACC. We must use this result with caution, however, because TRACC detects spurious deadlocks much more often than the other tools. Because TRACC terminates the analysis on detection of the spurious deadlock, the TRACC analysis times are reduced because of the inaccurate results. SPIN and INCA provide approximately equivalent mean analysis times checking for deadlock. For checking other properties, INCA, TRACC, and SPIN+PO have the lowest mean analysis times. As with deadlock, the TRACC analysis times are small due to detection of spurious property violations. In addition, because TRACC fails on relatively small sizes of all the programs, there are no large analysis times that act as outliers.

We note that many of the standard deviations in the table are very large. This indicates that there are large amounts of variability in the analysis times for most of the tools. We also note that the median values are significantly less than the means, in some cases several orders of magnitude smaller. We view this as an indication that outliers are having a significant effect on the mean analysis times. We therefore consider an alternate approach for analysis time comparison.

Another way to compare the analysis times is by counting the number of cases for which each tool had the fastest analysis time and comparing these counts. The results are provided in Table 7.4.

Table 7.4. Fastest Case Counts, Native Input Analysis Time

	Deadlock	Other Properties
SPIN, Never Claims	28	23
SPIN, Assertions	-	24
SPIN+PO	47	25
TRACC	0	0
SMV	40	66
INCA	9	49
FLAVERS	-	2

For checking deadlock, SPIN+PO and SMV provide the largest number of cases for which they yield the fastest analysis times. We note that SMV had the largest mean analysis time for checking deadlock, so this comparison technique yields significantly different results from a mean analysis time comparison. For checking other properties, SMV and INCA provide the largest number of cases for which they yield the fastest analysis times.

These results must be considered with care, however. Because we restricted the maximum size of each program based on the tool that performs worst on that program, some of the other tools may be able to scale to much larger sizes of that program. The above table also does not show the magnitude of difference in analysis times. For example, a tool might not have the fastest analysis time for a particular case, but the analysis time for that tool on that case might only be 0.01 seconds longer than the fastest

time. This difference is probably not significant to the analyst, but is reflected in the counts in the table.

As discussed in Chapter 3, we believe that simply counting the fastest cases for each tool might bias the results against a tool that consistently does well but is seldom the fastest. We propose using the average ranking for each tool for the comparison; these average rankings are provided in Table 7.5.

Table 7.5. Average Rankings, Native Input Analysis Time

	Deadlock	Other Properties
SPIN, Never Claims	2.11	3.07
SPIN, Assertions	-	2.59
SPIN+PO	1.85	2.73
TRACC	4.60	6.00
SMV	2.23	2.15
INCA	3.31	3.27
FLAVERS	-	4.99

For checking deadlock, SPIN+PO, SPIN, and SMV have the best average rankings. Note that SPIN had significantly fewer fastest analysis cases than SMV, but it has a slightly better average ranking than SMV. For checking other properties, SMV, SPIN using assertions, and SPIN+PO have the best average rankings. Although INCA has the second largest number of fastest analysis cases, it has the fifth best average ranking.

7.4.2 Total Analysis Times

To gain more insight into the true cost of using these tools to analyze Ada programs, we also collected timing information for all the translation steps in the analysis process and for the compilation of the PROMELA programs. We then recalculated the total analysis times for each tool, including all times from input of the Ada program to output of the analysis results.

We first compare the mean analysis times for each tool. These times are provided in Table 7.6. For checking deadlock, TRACC has the smallest mean analysis time, followed by SPIN+PO and INCA. We note (again) that the mean analysis times for TRACC are low because of its detection of spurious deadlocks. For checking other properties,

TRACC, INCA, and SMV have the lowest mean analysis times. As discussed above, TRACC analysis times are small because spurious results and a large number of failures.

Table 7.6. Mean Total Analysis Times

	Deadlock			Other Properties		
	Mean	Std Dev	Median	Mean	Std Dev	Median
SPIN, Never Claims	71.35	282.03	25.95	101.86	365.47	27.67
SPIN, Assertions	-	-	-	89.34	326.72	27.37
SPIN+PO	57.49	66.33	35.22	84.79	161.38	40.86
TRACC	23.33	18.56	14.67	25.17	37.65	11.03
SMV	133.23	537.63	21.72	76.00	208.89	26.56
INCA	60.48	211.32	20.92	31.94	33.69	21.90
FLAVERS	-	-	-	344.82	1010.50	55.25

When we use the means from the total analysis times rather than from the native input analysis times, we find that SPIN and SPIN+PO do not provide as good performance relative to the other tools. The analysis times for SPIN and SPIN+PO are increased both by the conversion of the Ada program to PROMELA and by the compilation of the generated C program.

As with the native input analysis times, the standard deviations in the table are very large. This indicates that there are large amounts of variability in the analysis times for most of the tools. We also note that the median values are significantly less than the means, though this difference is not nearly as pronounced as it is for native input analysis times. The difference still provides evidence, however, that outliers are having a significant effect on the mean analysis times.

Because we believe that a comparison of average rankings provides a more meaningful comparison than counts of the fastest cases for each tool, we next consider these rankings. The average rankings are provided in Table 7.7.

For checking deadlock, INCA, TRACC, and SMV provide the best average rankings. For checking other properties, these same three tools provide the best average rankings. TRACC rankings are better than we would expect because of the large numbers of spurious results and failures for TRACC.

Table 7.7. Average Rankings, Total Analysis Time

	Deadlock	Other Properties
SPIN, Never Claims	3.21	4.16
SPIN, Assertions	-	3.43
SPIN+PO	4.08	5.15
TRACC	1.89	3.17
SMV	1.99	2.11
INCA	1.71	1.43
FLAVERS	-	4.28

The most noticeable difference between these results and those for native input analysis times is that INCA moves from the fourth best average ranking to the best average ranking for checking deadlock and from the fifth best average ranking to the best average ranking for checking other properties. One reason for this is that the time for building the FSAs for the program is included in INCA's native input analysis time but is not included in the SPIN, SPIN+PO, or SMV native input analysis times, even though the FSAs must be built to generate the input for these tools. Because this time is often non-trivial, including it in the total analysis time for all the tools has a noticeable effect. Also, the time to compile the C programs generated by SPIN and SPIN+PO has an adverse affect on the average rankings for these tools.

7.5 Failure Comparisons

The counts and percentages of failures for each tool are provided in Table 7.8. Note that the total number of cases for TRACC is less than for the other tools. Because the current implementation of TRACC only allows modeling boolean variables, we could not model the variables in 48 of the cases. Also, because it is necessary to write a separate program for each property TRACC checks and because we had preliminary indications that TRACC is not a viable static analysis tool, at least compared to the others in the experiment, we only checked some of the non-deadlock properties using TRACC.

As the table indicates, the TRACC tool had the worst failure percentages by far, followed by SPIN+PO, which seemed to do better checking for deadlock as opposed to checking other properties. This may be due in part to our use of the -c0 to check some of

these properties in the presence of deadlock. The SPIN tool using assertions may have experienced a large number of failures for the same reason; using never claims seemed to be more effective, at least in the context of failures. SMV and FLAVERS had a small number of failures, and INCA was the best tool for avoiding failures, with only 1 failure out of 300 cases.

Table 7.8. Counts for Failures

	Not Failure	Failure	Total	% Failures
SPIN, Never Claims				
Deadlock	105	15	120	12.5
Other Properties	149	31	180	17.2
SPIN, Assertions				
Other Properties	141	39	180	21.7
SPIN+PO				
Deadlock	110	10	120	8.3
Other Properties	146	34	180	18.9
TRACC				
Deadlock	45	27	72	37.5
Other Properties	18	30	48	62.5
SMV				
Deadlock	109	11	120	9.2
Other Properties	167	13	180	7.2
INCA				
Deadlock	119	1	120	0.8
Other Properties	180	0	180	0.0
FLAVERS				
Other Properties	167	13	180	7.2

7.6 Spurious Result Comparisons

We provide the counts and percentages of spurious results for each tool in Table 7.9. We note that the table only includes analysis runs that did not fail, since these runs do not provide any results.

As the table indicates, TRACC analyses clearly yield the largest percentage of spurious results, followed by FLAVERS and INCA checking for deadlock. SPIN, SPIN+PO, and SMV provide nearly equivalent percentages of spurious results checking for deadlock. Using SPIN with assertions or SPIN+PO provides the smallest percentage of spurious results for checking properties other than deadlock.

Table 7.9. Counts for Spurious Results

	Not Spurious	Spurious	Total	% Spurious
SPIN, Never Claims				
Deadlock	70	35	105	33.3
Other Properties	128	20	148	13.5
SPIN, Assertions				
Other Properties	130	11	141	7.8
SPIN+PO				
Deadlock	72	38	110	34.5
Other Properties	134	12	146	8.2
TRACC				
Deadlock	6	39	45	86.7
Other Properties	13	5	18	27.8
SMV				
Deadlock	75	34	109	31.2
Other Properties	150	17	167	10.2
INCA				
Deadlock	65	54	119	45.4
Other Properties	157	23	180	12.8
FLAVERS				
Other Properties	78	83	161	51.6

Because the table includes all analysis runs that do not fail, it also includes cases for which a spurious result is impossible. For example, if a property is in fact violated in a given program (no_r1w in our readers/writers program, for example), a spurious result is not possible. If a tool answers that the property is violated, this is an accurate result. If a tool answers that the property is not violated, this is not a spurious result, it is an indication that the analysis is not conservative. We did not have any cases for which a tool was not conservative, but in 18 of the 300 cases (6%) a property violation occurs and spurious results are therefore not possible. These cases are counted as accurate results in the table above and the logistic regressions that follow.

7.7 Successful Analysis Case Comparisons

Because the failure and spurious result percentages are calculated for a different number of cases for each tool, it is difficult to immediately discern the percentage of successful analysis cases for each tool. We define a successful analysis cases as a case that runs to completion (does not fail) and yields the correct answer (does not give a spurious result). The results of these calculations are provided in Table 7.10.

Table 7.10. Successful Analysis Percentages

	Failure	Spurious	Successful	Total	% Successful
SPIN, Never Claims					
Deadlock	15	35	70	120	58.3
Other Properties	31	20	129	180	71.7
SPIN, Assertions					
Other Properties	39	11	130	180	72.2
SPIN+PO					
Deadlock	10	38	72	120	60.0
Other Properties	34	12	134	180	74.4
TRACC					
Deadlock	27	39	6	72	8.3
Other Properties	30	5	13	48	27.1
SMV					
Deadlock	11	34	75	120	62.5
Other Properties	13	17	150	180	83.3
INCA					
Deadlock	1	54	65	120	54.2
Other Properties	0	23	157	180	87.2
FLAVERS					
Other Properties	13	83	84	180	46.7

For checking deadlock, SMV and SPIN+PO have the highest successful analysis percentages, though all of the tools except TRACC are in a fairly small (8%) range. For checking other properties, INCA and SMV have significantly better successful analysis percentages than the rest of the tools. The percentages for SPIN using never claims, SPIN using assertions, and SPIN+PO are within 3% of each other. The percentage for FLAVERS is significantly lower than most of the other tools, and TRACC has the worst percentage by far.

7.8 Preprocessing the Data

High linear correlation between our predictor variables (i.e., the metrics) can cause numerical stability problems in both linear and logistic regression. Before applying these regressions, we preprocess our data to remove collinear variables. To gain some assurance that we are not removing too many variables, we conduct randomization tests to check the null hypothesis that variable pairs are collinear by chance. We then select which variables to elide from the models based on collinearity, and use the remaining variables in the regressions.

We begin the preprocessing of the data by calculating the pairwise Pearson's correlation coefficient for each pair of program metrics and for each pair of property metrics. For metrics pairs in which the magnitude of the coefficient is greater than 0.75, we run a randomization test on that pair as described in Section 6.3. In all such cases, we can reject the null hypothesis (with $p < 0.001$) that the two metrics are actually not linearly correlated.

Based on these results, we build sets of collinear variables and select which variables to elide from the regression models. The selection of the single variable to include from each set is somewhat arbitrary and should not affect the results of the regressions, but we tend to select variables that are measures of the program rather than the internal representations, or internal representation measures that correspond to the FSAs rather than the TIGs. We believe that this guideline for variable selection will lead to more intuitive predictive models. The resulting sets of collinear variables, and the variables we select for inclusion in the regression models, are shown in Table 7.11.

Table 7.11. Collinear Sets of Metrics

Set of Collinear Metrics	Selected Metric
{ N, MaxN, TRANS, TE, MaxTE, Cgt }	N
{C, Alpha, Gamma, Cnd, TN }	C
{ Cnd', Beta', Gamma' }	Cnd'
{ Cnd, Beta, Gamma }	Beta
{ TRANS, MaxTRANS, SMV Trans }	MaxTRANS
{ MaxC, MaxTN }	MaxC
{ WFSA, WTIG }	WFSA
{ T }	T
{ Vars }	Vars
{ Alpha' }	Alpha'
{ Cif }	Cif
{ QRE Alphabet, QRE Trans }	QRE Alphabet
{ Never States, Never Trans }	Never States
{ Assertions, Assignments }	Assertions
{ QRE States }	QRE States
{ Query Events }	Query Events
{ Query Intervals }	Query Intervals

7.9 Predictive Models for Analysis Time

We use linear regression to build the predictive models for analysis time, where analysis time is measured from the native input for each tool. Because analysis time is not meaningful for those analysis cases that failed, the regression only includes analysis cases that did not fail. We would expect an analyst to use the predictive models for failure first to check whether or not the analysis will fail, then use the predictive models for analysis time if the analysis is not predicted to fail. Because most of the tools provide automatic checking for deadlock (or, for INCA, a "pre-canned" query), the property metrics are not meaningful for checking deadlock. We therefore generate two models for each tool - one for deadlock, using only the program metrics as independent variables, and one for the other properties, using both the program and property metrics as independent variables.

As described in Chapter 6, we use four different linear regression methods for each model. The methods are the enter method, backward elimination, forward selection, and stepwise regression. Because the R^2 value quantifies how much of the variance in the data is captured by a specific model (with an R^2 greater than 0.800 indicating a good fit), we use this value as one of our primary considerations when choosing between the models. The R^2 values for each of the linear regressions are provided in Table 7.12. More detailed examination of each model is provided in the following sections. The selected models for analysis time, failures, and spurious results are provided in the Appendix.

7.9.1 SPIN, Never Claims

This section provides the results of our linear regressions for analysis runs using SPIN with properties specified using never claims. Although checking for deadlock does not actually require a never claim, we include it here rather than with the assertions; this choice is arbitrary, since assertions are not used to check for deadlock either.

Table 7.12. R^2 Values for Analysis Time Models

	Enter Method	Backward Elimination	Forward Selection	Stepwise Regression
SPIN, Never Claims				
Deadlock	0.415	0.387	0.387	0.387
Other Properties	0.350	0.333	0.333	0.333
SPIN, Assertions				
Other Properties	0.468	0.452	0.426	0.426
SPIN+PO				
Deadlock	0.218	0.178	0.156	0.156
Other Properties	0.128	0.052	0.052	0.052
TRACC				
Deadlock	0.955	0.951	0.951	0.951
Other Properties	0.999	0.998	0.996	0.996
SMV				
Deadlock	0.176	0.109	0.109	0.109
Other Properties	0.223	0.178	0.076	0.076
INCA				
Deadlock	0.572	0.537	0.537	0.537
Other Properties	0.902	0.897	0.897	0.897
FLAVERS				
Other Properties	0.960	0.958	0.959	0.957

7.9.1.1 Predictive Model for Deadlock

The results of the linear regressions indicate that for SPIN checking for deadlock, the Cnd' (a measure of nondeterminism in the program) and MaxTRANS (the maximum number of transitions in the set of FSAs for the program) metrics have the largest effect on analysis time. We see evidence of this both in the coefficients from the enter method, where these two metrics have the largest coefficients, and from the fact that the other methods excluded all but these two metrics from their models. We also note that the backward elimination, forward selection, and stepwise regression methods all generated the same model. This is not always the case, but is not uncommon.

Although there are no indications of numerical instability or overfitting (i.e., extremely large coefficients or standard errors) in the enter method model, we select the model generated by the other methods instead. The R^2 value for these models is only slightly smaller than the R^2 for the enter method model (representing a 7% reduction), and the inclusion of significantly fewer variables may make the model slightly more

general. The R^2 value of 0.387 indicates that the model does not fit the experimental data very well, which in turn implies that it probably will not provide much predictive power for real programs either.

As we reviewed the selected model, we noted that the predicted values of analysis time can be negative; this occurs because the regression simply performs a least-squares fit to the data without considering the "meaning" of time. While a negative predicted time has no practical meaning, it could still be used for comparison to the (potentially negative) predicted analysis times for the other tools. Unfortunately, once a tool was selected, a negative predicted analysis time would give no insight into how long the analysis might actually take.

To check our assumptions about the errors (i.e., residuals) in the model (see Section 6.5.2), we plot the standardized residuals against the predicted analysis times. Example plots are shown in Figures 7.1 and 7.2.

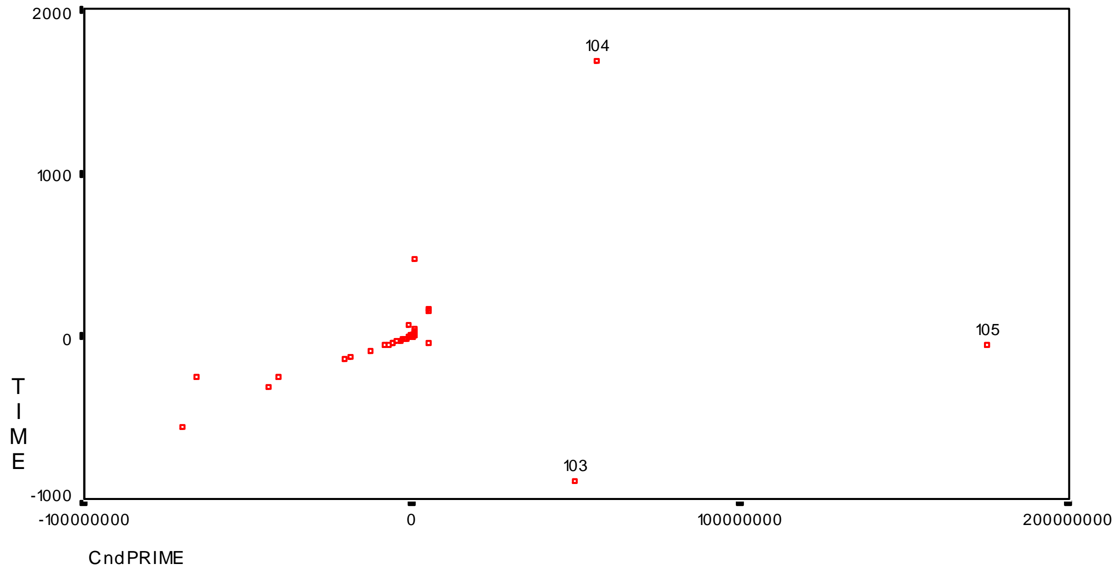


Figure 7.1. Plot of Standardized Cnd' Residuals vs Predicted Time

The plot of standardized MaxTRANS residuals seems to support our assumptions about the residuals, because the residuals appear to be scattered randomly about the 0 line (with the exception of the outliers, discussed below). The plot of standardized Cnd'

residuals, however, seems to indicate that there is an additional linear effect of Cnd' that has not been included in the model. This could occur because of an error in the analysis software (unlikely, given the maturity of the SPSS software) or, more likely, an indication that additional cross-product terms (i.e., terms of the form $x_i x_j$) would lead to a better model. As stated in Chapter 6, our analysis stops at recognition of this problem - we do not add cross-product terms or use other more advanced regression techniques.

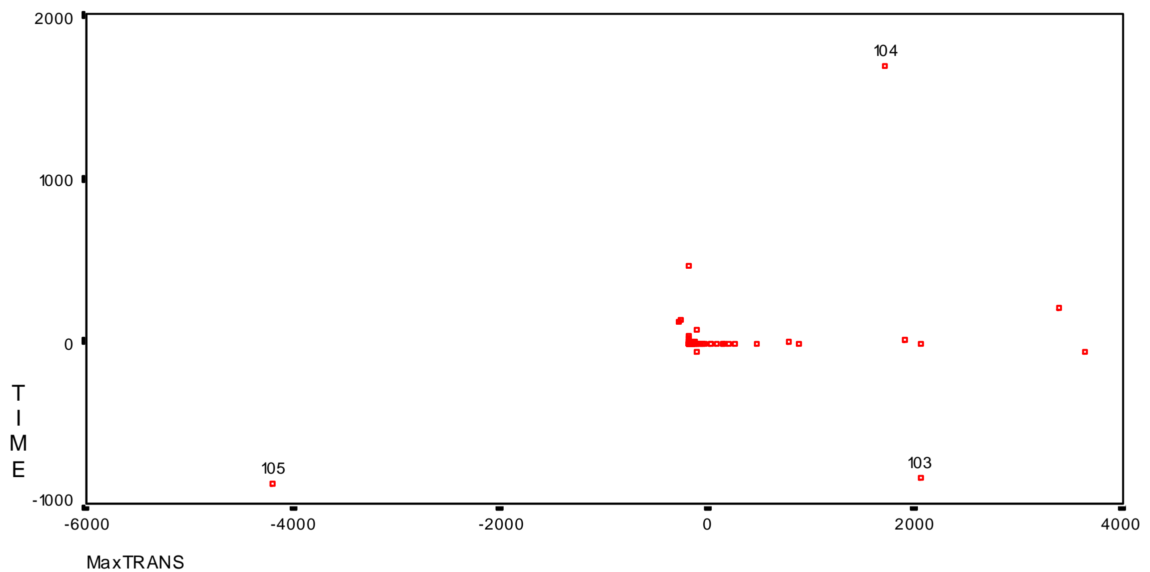


Figure 7.2. Plot of Standardized MaxTRANS Residuals vs Predicted Time

We have identified several outliers in the residual plots above. These are marked as analysis cases 103, 104, and 105. These cases correspond to checking for deadlock on the readers/writers program with 12 readers and writers, modeling no variables (103), the Writer variable (104), and both the Writer and Readers variables (105). These cases include the largest values of Cnd' in the dataset, and the MaxTRANS values are well into the upper quartile for the dataset. The analysis time for case 103 is in the upper quartile of analysis times, case 104 yields the largest analysis time in the dataset, but the analysis time for case 105 is fairly small. We do not exclude these outliers from our analysis, since they may be more representative of real program properties than the other data points, but believe there is still some value in identifying them.

Another technique for identifying outliers is to use the studentized residuals. Threshold values for studentized residuals for various p values, numbers of observations (cases in the dataset), and independent variables are included in [DW80]. The threshold value for $p < 0.05$, 105 cases, and 11 independent variables is 3.42; any studentized residual magnitude above this value represents an outlier. The only analysis cases in our dataset with values above this threshold are cases 103 (-6.45), 104 (9.70), and 105 (-4.08). We have therefore identified the same outliers using both informal examination of the residual plots and the more formal studentized residual method.

7.9.1.2 Predictive Model for Other Properties

The results of the four regressions indicate that for SPIN using never claims and checking properties other than deadlock, the MaxTRANS and Query Events (number of events in the INCA query) metrics have the largest effect on analysis time.

The backward elimination, forward selection, and stepwise regression models are all equivalent. We select the backward elimination model over the enter method model because 13 variables are removed from the model at the cost of a 5% reduction in the R^2 value. We note that the R^2 value of 0.333 is low, and the model is therefore unlikely to provide good predictive power.

In the interest of brevity, we do not provide the standardized residuals plots for any of the remaining linear regressions; our technique for visually identifying higher order trends and outliers is as demonstrated above. The plots for this model do not indicate any problems with our assumptions about the distribution of the residuals.

From the plots of standardized residuals against predicted analysis times and the studentized residuals, we identify cases 143 and 144 as outliers. These cases are for the readers/writers program with 12 readers and writers, no variables (143) and the Writer variable (144) modeled, checking the no_w1w2 property. For case 143, the model significantly overestimates the analysis time, since a (spurious) property violation is found. For case 144, the model significantly underestimates the analysis time; we

believe this is simply a result of a poor fit of the model to this point, which represents the largest analysis time in the dataset.

7.9.2 SPIN, Assertions

This section provides the results of our linear regressions for analysis runs using SPIN with properties specified using assertions. Because the regressions for checking for deadlock were included in the previous section, we only include regressions for checking other properties in this section.

The results of the four regressions indicate that for SPIN using assertions and checking properties other than deadlock, the MaxTRANS and Query Events metrics have the largest effect on analysis time.

We select the backward elimination model over the enter method model, since the removal of 10 variables from the model only results in a 3% reduction in the R^2 value. We do not select the forward selection or stepwise regression models because the reduction in R^2 is 9%, while only 4 more variables are removed than for the backward elimination model. The R^2 value of 0.452 indicates that the model is not likely to provide much predictive power.

The plots of standardized residuals against predicted analysis times do not indicate any higher-order effects; the residuals seem randomly scattered around the 0 line. From these plots and the studentized residuals, we identify cases 137, 138, 139, and 140 as outliers. These cases correspond to the readers/writers program with 12 readers and writers. Case 137 checks no_w1w2 with only the Writer variable modeled; this case yields the largest analysis time for SPIN using assertions. Case 138 checks no_w1w2 with both variables modeled; despite the small value of MaxTRANS for this case, it takes significantly longer than predicted. Cases 139 and 140 check no_r1w with the no variables (139) and only the Writer variable (140) modeled. The value of MaxTRANS for these cases is very large, but the actual analysis time is small because a property violation is quickly detected.

7.9.3 SPIN+PO

This section provides the results of our linear regressions for analysis runs using SPIN+PO. The section includes a model for checking for deadlock and a model for checking other properties.

7.9.3.1 Predictive Model for Deadlock

The results of the four regressions indicate that for SPIN+PO, checking deadlock, the C (average number of communications per task), Alpha' (a measure of nondeterminism), Beta (another measure of nondeterminism), Cnd', and MaxTRANS metrics have the most significant effect on analysis time. It is interesting to note that, while the Cnd' metric has a relatively large coefficient for the enter method model, it is not selected by any of the more advanced regression techniques.

Again, there are no indications of numerical instability in the enter method model. Because using the next best model (generated by the backward elimination method) results in a reduction of almost 20% in the R^2 value, we select the enter method model as our predictive model. The reduction in the R^2 value is caused by the fact that, once the backward elimination model contains three variables, none of the remaining variables has a sufficient effect on the predictions to be included in the model (recall our significance threshold for adding additional variables to the model is 0.10). However, including all these remaining variables, as the enter method model does, apparently allows the model to capture more of the variance in the data. We note that we again have a very low R^2; for SPIN+PO checking deadlock, our predictive model only accounts for slightly more than 20% of the variance in our experimental data. Such a weak model is not likely to be of practical use for predicting analysis times for real programs.

The standardized residuals plots seem to indicate a missing linear term for both Cnd' and MaxTRANS. We suspect that adding cross-product terms might help with this problem.

Analysis cases 59 and 105 appear to be outliers. The threshold value for the studentized residuals in this dataset is 3.44. Only case 105 has a studentized residual (9.23) above this threshold (the studentized residual for case 59 is 2.94). Case 105 is for the readers/writers program with 8 readers and writers and only the Writer variable modeled. This generates a very large state space (with no deadlock possible), which in turn yields the largest analysis time in the dataset.

7.9.3.2 Predictive Model for Other Properties

The results of the four regressions indicate that for SPIN+PO, checking properties other than deadlock, the Beta, Vars, and QRE Alphabet (number of events in the QRE alphabet) metrics have the largest effect on analysis time.

The backward elimination, forward selection, and stepwise regression models are all equivalent. We select the enter method model as our predictive model, since using the more advanced techniques results in a reduction of 59% in the R^2 value. This occurs because all the variables but Alpha' are eliminated from the backward elimination model because the significance of their effects is less than 0.10 when they are considered individually. Similarly, Alpha' is the first variable selected for inclusion by the forward selection and stepwise regression techniques, and none of the other variables have sufficient effect individually (threshold for adding variables is 0.05) to be included in the model. Including all the variables in the enter method model, despite the minimal individual effects of each of them, results in a model that captures a much larger (though still small) portion of the variance in the data. The R^2 value of 0.128 is very low, indicating that this is probably a very weak predictive model.

From the studentized residuals and plots of standardized residuals, we identify cases 8, 33, and 133 as outliers. All these cases have large analysis times (including the largest for this dataset), and in all these cases we had to use the -c0 option to check the property in the presence of deadlock. We believe this "unusual" configuration (we did not have to use the -c0 option most of the time) leads to the poor fit of the model to these points.

7.9.4 TRACC

This section provides the results of our linear regressions for analysis runs using TRACC. The section includes a model for checking for deadlock and a model for checking other properties. We point out that, because TRACC had such a high number of failures and because we did not write custom property checkers for the majority of the other properties, the datasets in this section are much smaller than those for the other tools.

7.9.4.1 Predictive Model for Deadlock

The results of the four regressions indicate that for TRACC, checking deadlock, the T (number of tasks) and C metrics have the most significant effect on analysis time.

We select the backward elimination model as our predictive model, since it yields only a slight reduction (less than 1%) in R^2 over the enter method model while removing six variables from the model. The forward selection and stepwise regression models are equivalent to the backward elimination model. We note that our R^2 value (0.951) is much higher than we are typically finding in our regressions, and such a high value implies that the model may provide good predictive power.

There appears to be a mild linear component in the plot of the standardized T residuals, but the other plots look like the random distribution of points around the 0 line that we expect.

Analysis cases 17, 35, and 39 appear to be outliers. The threshold value for the studentized residuals in this dataset is 3.11. Only case 17 has a studentized residual (3.84) above this threshold. Case 17 is for the dining philosophers with dictionary program with six philosophers. For this case, the value of T is relatively small, but the analysis time is large.

7.9.4.2 Predictive Model for Other Properties

The results of the four regression indicate that for TRACC, checking properties other than deadlock, the Wampler (FSA) metric has the greatest effect on analysis time. It is

interesting to note that the enter method model excludes some of the variables. These variables are excluded because they are constant (or nearly so); essentially, they do not have sufficient variance, given the size of the dataset (18 cases), to have an effect on the regression.

We select the forward selection model (the stepwise regression model is equivalent) as our predictive model. The reduction in R^2 over the enter method model is less than 1%, and 12 variables are removed from the model. The R^2 value is very high (0.996), indicating that this model may provide good predictive power.

There appears to be a mild negative linear component in the plot of the standardized Cnd' residuals and a stronger positive linear component in the plot of the standardized Wampler (FSA) residuals. The other plots do not indicate any problems.

Using the studentized residuals and standardized residuals plots, we identify case 11 as the only outlier. Some of the plots indicated that cases 2 or 6 might also be outliers, but the studentized residuals for these cases were significantly less than the threshold. Case 11 is for the dining philosophers with host program with 3 philosophers and no variables modeled. Both values of the Wampler (FSA) metric and the analysis time for this case are relatively large, but only just in the top quartile, so it is not clear why this case is not predicted well by the model.

7.9.5 SMV

This section provides the results of our linear regressions for analysis runs using SMV. The section includes a model for checking for deadlock and a model for checking other properties. Because our statistical analysis above indicates that whether or not we use the REORDER option has no statistically significant effect on analysis time, we build the models below for runs using the REORDER option.

7.9.5.1 Predictive Model for Deadlock

The results of the four regressions indicate that for SMV, checking for deadlock, the C, Beta, N, and Vars variables have the largest effect on analysis time.

We select the enter method model as our predictive model, because choosing any of the other models would results in a reduction of 38% in the R^2 value. This reduction occurs because only the Vars metric has a sufficient individual effect to be retained (backward elimination) or added to (forward selection, stepwise regression) the model. The combination of the effects of all the variables in the enter method model allows it to capture more of the variance in the data. We note, however, that the R^2 value for the enter method model is very low, so the model is not likely to provide significant predictive power.

From the studentized residuals and plots of the standardized residuals, we identify cases 90 and 93 as outliers. We also initially identified case 21 as a potential outlier from the plots, but the studentized residual for this case is well below the threshold. Case 90 is for the ring program with 6 servers and masters and no variables modeled. Case 93 is for the ring program with 10 servers and masters and all 20 variables modeled. The values for C and N are fairly small for these cases, but the analysis times are large. We believe this occurs because the ring problem has the ring structure for which the REORDER option does not tend to work well.

7.9.5.2 Predictive Model for Other Properties

The results of the four regressions indicate that for SMV, checking properties other than deadlock, the N, MaxC, and Beta metrics have the largest effect on analysis time.

We select the enter method model as our predictive model; using the backward elimination model would result in a reduction of 20% in the R^2 value, and using the forward selection of stepwise regression models would result in a reduction of 66% in R^2. The reduction for the backward elimination model occurs because the technique eliminates 10 of the variables from the model because of their small individual effects. The forward selection and stepwise regression techniques both select the N metric for inclusion in the model, then do not include any other variables because they do not have sufficient individual effects. The combination of all the variables in the enter method

model, however, captures more of the variance in the data. The R^2 value of 0.223 implies that the model is unlikely to provide much predictive power.

The plots of the standardized residuals do not indicate any problems. From these plots and the studentized residuals, we identify cases 16 and 135 as outliers. Case 16 is for the cyclic program with 10 customer and scheduler tasks and no variables modeled, checking no_c2ss. The value of T for this case is in the upper quartile but the value of n is not large, so the model significantly underestimates the analysis time. This case represents the longest analysis time for SMV. Case 135 is for the ring program with 10 servers and masters and all variables modeled, checking no_m1m2. This case represents the second largest analysis time for SMV, and again the model significantly underestimates the analysis time.

7.9.6 INCA

This section provides the results of our linear regressions for analysis runs using INCA. The section includes a model for checking for deadlock and a model for checking other properties. Our statistical analysis above indicates that whether we use arrays or unique tasks is not statistically significant, whether or not accepts have bodies is not statistically significant, and whether we use multiple intervals or use additional constraints is not statistically significant. We therefore build the models below for runs using unique tasks, no accept bodies (except where required by the program), and multiple interval queries (where necessary).

7.9.6.1 Predictive Model for Deadlock

The results of the four regressions indicate that for INCA, checking for deadlock, the C and N metrics have the largest effect on analysis time.

We select the backward elimination model as our predictive model, since it yields a fairly small reduction (6%) in R^2 over the enter method model while removing nine variables from the model. The forward selection and stepwise regression models are

equivalent to the backward elimination model. We note that the R^2 value (0.537) is lower than our informal threshold for a good fit.

The plots of the standardized residuals do not indicate any problems. From these plots and the studentized residuals, we identify cases 69 and 70 as outliers. We also identified case 71 as an outlier from the plots, but the standardized residual for this case is well below the threshold. Cases 69 and 70 are for the gas station problem with 5 customers, without variables modeled (69) and with all variables modeled (70). The values for C and N are large for these cases. Case 69 detects a spurious deadlock, so the observed analysis time is significantly less than predicted. Case 70 represents the largest INCA analysis time, and the effect of the other 118 cases causes the model to under-predict this analysis time.

7.9.6.2 Predictive Model for Other Properties

The results of the four regressions indicate that for INCA, checking properties other than deadlock, the MaxTRANS metric has the largest effect on analysis time.

The backward elimination, forward selection, and stepwise regression models are equivalent. We select the backward elimination model over the enter method model as our predictive model, because it removes 11 variables from the model and results in a reduction of only 1% in the R^2 value. The R^2 value of 0.897 indicates that the model may have strong predictive power.

The plot of the standardized MaxTRANS residuals demonstrates a moderate linear component that is not accounted for by the model. The other plots did not provide any evidence of problems.

From the studentized residuals and plots of standardized residuals, we identify cases 101 and 103 as outliers. These cases are for the gas station program with 6 customers and no variables modeled, checking no_c1c2 (101) and no_c1p2 (103). The values for MaxTRANS are very large for these cases, as are the analysis times, but the effects of the other 178 points in the regression cause the model to significantly overestimate the

analysis time for case 101 and to significantly underestimate the analysis time for case 103.

7.9.7 FLAVERS

This section provides the results of our linear regressions using FLAVERS. Because FLAVERS does not currently support checking for deadlock, we only include regressions for checking other properties in this section.

The results of the four regressions indicate that for FLAVERS, checking properties other than deadlock, the C, MaxC, and Alpha' metrics have the largest effect on analysis time.

We select the stepwise regression model over the others as our predictive model. It removes the most variables (9) from the model, and has an R^2 value less than 1% smaller than the R^2 value for the enter method. The R^2 value of 0.957 implies that this model may provide strong predictive power.

The plot of the standardized C residuals indicates a moderate linear component not accounted for by the model and the plot of the standardized Alpha' residuals indicates a stronger linear component. The other plots do not indicate any problems.

Using the studentized residuals and plots of standardized residuals, we identify cases 112, 114, and 116 as outliers. Cases 112 and 114 are for the memory management program with 5 users and all variables modeled, checking no_u1u2 (112) and no_sdu1a (114). Case 116 is for the memory management program with 6 users and all variables modeled, checking no_u1u2. The values of the C and Alpha' metrics are very high for these cases, as are the analysis times, but the effect of the other 158 cases cause a poor fit to these cases.

7.10 Predictive Models for Failures

We use logistic regression to build the predictive models for failure. The regression (obviously) includes all analysis cases, both those that did and did not fail. As discussed above, the property metrics are not meaningful for checking deadlock. We therefore

generate two models for each tool - one for deadlock, using only the program metrics as independent variables, and one for the other properties, using both the program and property metrics as independent variables.

As described in Chapter 6, we use three different logistic regression methods for each model. The methods are the enter method, backward elimination, and forward selection. Because the deviance quantifies how much of the variance in the data is captured by a specific model (with a smaller deviance indicating a better fit), we use this value as one of our considerations when choosing between the models. We believe the percent of the predictions by the model that are correct to be an even more important consideration, so we provide these values as well. The deviance and percent correct values for each of the logistic regressions are provided in Table 7.13. More detailed examination of each model is provided in the following sections.

Table 7.13. Deviances and Percents Correct for Failure Models

	Enter Method		Backward Elimination		Forward Selection	
	Deviance	% Correct	Deviance	% Correct	Deviance	% Correct
SPIN, Never Claims						
Deadlock	-	-	-	-	33.993	95.83
Other Properties	50.093	95.56	51.535	94.44	57.540	94.44
SPIN, Assertions						
Other Properties	81.298	89.44	82.311	90.00	63.634	91.67
SPIN+PO						
Deadlock	6.819	98.33	6.819	98.33	22.565	97.50
Other Properties	24.719	98.89	25.493	98.33	48.404	94.44
TRACC						
Deadlock	-	-	-	-	64.567	84.72
Other Properties	-	-	-	-	-	-
SMV						
Deadlock	25.614	95.83	25.765	95.83	59.654	93.33
Other Properties	-	-	-	-	65.580	96.11
INCA						
Deadlock	-	-	-	-	3.450	98.33
Other Properties	-	-	-	-	-	-
FLAVERS						
Other Properties	-	-	-	-	3.450	99.44

As we tried to run these regressions, we often encountered numerical problems, especially with the enter and backward elimination methods. We do not have sufficient

statistical analysis experience to determine what caused these numerical problems, but we developed a process that worked around them in most cases. When we encountered numerical problems as we were building a model for deadlock, we simply selected one of the models that was successfully created. When we encountered numerical problems as we were building a model for the other properties, we removed the property metrics from the regression; this often solved the problem.

7.10.1 SPIN, Never Claims

This section provides the results of our logistic regressions to predict failure of analysis runs using SPIN with never claims. As for the predictive models for analysis times, we include deadlock in this section as well.

7.10.1.1 Predictive Model for Deadlock

We had numerical problems using the enter and backward elimination methods on this dataset. The results of the forward selection regression indicate that, while the Alpha' and Cif (information flow) metrics are included in the model, the N (average number of FSA states) metric has the strongest influence.

Although the coefficients and deviance for the predictive model are of statistical interest, more insight about the predictive power of the model can be gained through consideration of a classification table of predicted vs observed failures. Such a table is provided in Table 7.14. A "0" row or column in the table indicates no failure, and a "1" row or column indicates a failure. This table shows that, for the 105 analysis cases that did not fail, the predictive model predicts that 104 will not fail and 1 will fail. Of the 15 analysis cases that did fail, the predictive model predicts that 4 will not fail and that 11 will fail. Overall, the predictive model predicts 95.83% of the analysis cases correctly, and thus will hopefully provide good predictive power for real programs as well.

For our residual analysis, we plot the standardized residuals against the failure variable; the resulting plot is shown in Figure 7.3. We originally plotted the standardized

residuals against the predicted failures, but it was more difficult to detect potential outliers in the resulting plot.

Table 7.14. SPIN Failure Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	104	1	99.05 %
1	4	11	73.33 %
		Overall :	95.83 %

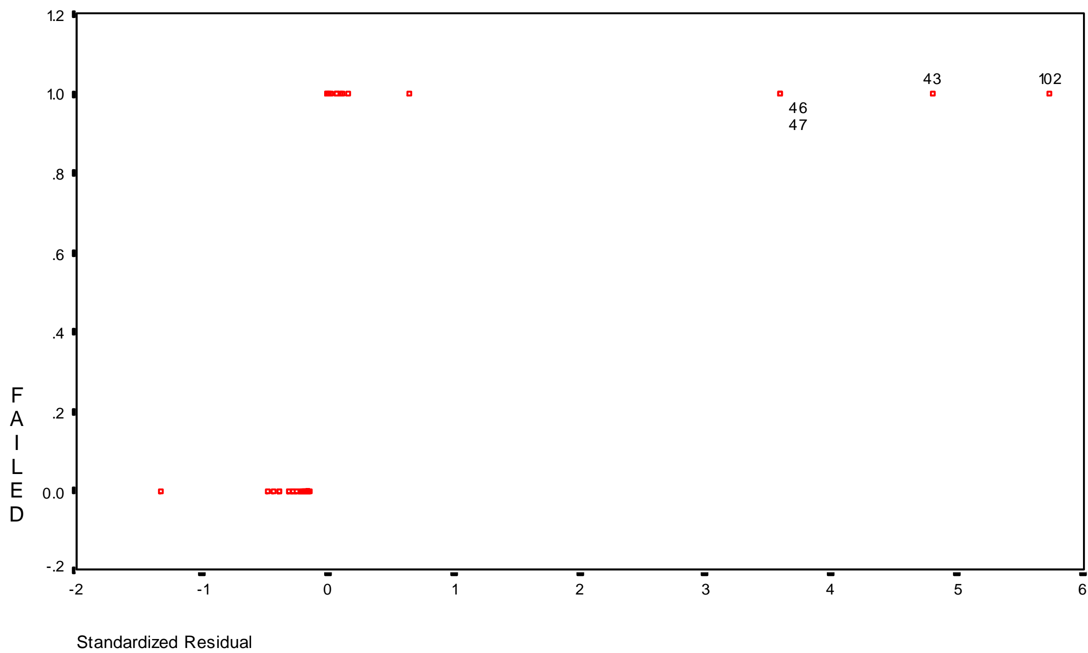


Figure 7.3. Plot of Standardized Residuals vs Failures

We identify the residuals associated with analysis cases 43, 46, 47, and 102 as outliers. The SPSS software also identifies these four points as outliers, using a threshold value of 2.00 on the studentized residuals. These cases are for the dining philosophers with fork manager program with: 6 philosophers, no variables modeled (43), 7 philosophers, no variables modeled (46), and 7 philosophers with only fork_2 modeled

(47), and for the size 12 ring program with all variables modeled (102). These cases all have a fairly low value of n, but all 4 cases failed.

7.10.1.2 Predictive Model for Other Properties

We had numerical problems with the enter method and backward elimination regressions when we included the property metrics, so we perform these regressions including only the program metrics. The results of the three regressions indicate that for SPIN using never claims and checking properties other than deadlock, the C and T metrics have the largest effect on whether or not the analysis will fail.

We select the enter method model as our predictive model because it provides the highest percent correct value; the classification table for this model is shown in Table 7.15.

Table 7.15. SPIN, Never Claims, Failure Classification Table

Observed	Predicted 0	Predicted 1	Percent Correct
0	147	2	98.66 %
1	6	25	80.65 %
		Overall :	95.56 %

In the interest of brevity, we do not provide the plots of the standardized residuals against failures for this or any of the following logistic regressions; our outlier analysis is as described above. From the standardized residuals plot, we identify cases 9, 13, and 17 as outliers; SPSS identifies the same set of cases. These cases are for the cyclic program checking no_c2ss with no variables modeled for sizes 6 (9), 8 (13), and 10 (17). For these cases, C and T are close to their mean values, but all three cases fail. We note that we had to use the -c0 option for these cases to check no_c2ss in the presence of deadlock, and suspect this contributed to the failures.

7.10.2 SPIN, Assertions

This section provides the results of our logistic regressions to predict failure of analysis runs using SPIN with assertions. Because the regressions for checking for deadlock were included in the previous section, we only include regressions for checking other properties in this section.

We had numerical problems with the enter method and backward elimination regressions when we included the property metrics, so we perform these regressions including only the program metrics. The results of the three regressions indicate that for SPIN using assertions and checking properties other than deadlock, the C and NeverStates (the number of states in the never claim) have the largest effect on whether or not the analysis will fail.

We select the forward selection model as our predictive model, since it provides the best percent correct value. The classification table for this model is shown in Table 7.16. Despite the fairly high overall percent correct, this model incorrectly predicts successful analysis runs for 10 of the cases that actually fail.

Table 7.16. SPIN, Assertions, Failure Classification Table

Observed	Predicted 0	Predicted 1	Percent Correct
0	136	5	96.45 %
1	10	29	74.36 %
		Overall :	91.67 %

Examination of the standardized residuals plot indicates that cases 30 and 65 are outliers. The SPSS software also identifies cases 31 and 144 as outliers. Case 30 is for the dac program with 40 solvers, checking no_s3f. The NeverStates value is large enough that the predictive model predicts failure, but the case actually completes successfully, Case 31 is for the dac program with 50 solvers, checking no_s1js3j. The values of C and

NeverStates are small, so the model predicts successful completion, but the case fails because the generated C program can not be compiled. Case 65 is for the dining philosophers with fork manager program with 6 philosophers and no variables modeled, checking no_p1p2, and case 144 is for the ring program with 12 servers and masters and all variables modeled, checking no_m1m2. For both these cases, the values of C and NeverStates are small enough that the model predicts successful completion, but the cases actually fail.

7.10.3 SPIN+PO

This section provides the results of our logistic regressions to predict failure of analysis runs using SPIN+PO. The section includes a model for checking for deadlock and a model for checking other properties.

7.10.3.1 Predictive Model for Deadlock

The results of the three regressions indicate that for SPIN+PO, checking for deadlock, the Beta and N metrics have the largest effect on whether the analysis case will fail.

Because some of the coefficients are very large in the enter and backward elimination models, we need to investigate further for numerical problems. Specifically, these coefficients may indicate an overfitting of the model to the data. This inference is supported by the fact that the standard error of several of the coefficients is very large. We therefore reject the enter and backward elimination models as overfitted, and select the forward selection model as our predictive model.

The classification table of predicted against observed failures is shown in Table 7.17. Although the classification tables for the other models indicate 98.33% correct predictions, we accept the slight decrease in predictive accuracy to gain numerical stability in the model.

In our examination of the plot of the standardized residuals we identify cases 46, 116, and 118 as outliers. The SPSS software identifies cases 46 and 118 as outliers. Case 46

is for the dining philosophers with fork manager problem with 7 philosophers and no variables modeled. The predictive model does not predict failure for this case because N and Beta are small, but the case actually fails. Case 118 is for the readers/writers problem with 12 readers and writers and no variables modeled. The predictive model predicts failure for this case because Beta is large, but the case does not fail (deadlock is detected).

Table 7.17. SPIN+PO Failure Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	109	1	99.09 %
1	2	8	80.00 %
		Overall :	97.50 %

7.10.3.2 Predictive Model for Other Properties

The results of the three regressions indicate that for SPIN+PO, checking properties other than deadlock, the C and QREAlpha (number of events in the QRE alphabet) metrics have the largest effect on whether the analysis case will fail.

We again discover evidence (i.e., large coefficients and standard errors) of overfitting in the enter and backward elimination models. We therefore reject the enter and backward elimination models as overfitted, and select the forward selection model as our predictive model.

The classification table of predicted against observed failures is shown in Table 7.18. As for predicting deadlock failures, we accept a slight decrease in predictive accuracy to gain numerical stability in the model.

In our outlier analysis we identify cases 81 and 163 from the plot of the standardized residuals. The SPSS software also identifies cases 157 as an outlier. Case 81 is for the dining philosophers with host program with 7 philosophers and no variables modeled, checking no_p1p2. The C and QREAlpha metrics are close to their means for this case,

but the case fails because the state space is too large. Cases 157 and 163 are for the readers/writers program with 6 (157) and 8 (163) readers and writers and no variables modeled, checking no_w1w2. The model predicts failure for these two cases, but because the Writer variable is not modeled, SPIN+PO detects a (spurious) property violation and successfully completes the analysis.

Table 7.18. SPIN+PO Failure Classification Table for Other Properties

Observed	Predicted 0	Predicted 1	Percent Correct
0	142	4	97.26 %
1	6	28	82.35 %
		Overall :	94.44 %

7.10.4 TRACC

This section provides the results of our logistic regressions to predict failure of analysis runs using TRACC. The section includes a model for checking for deadlock and a model for checking other properties.

7.10.4.1 Predictive Model for Deadlock

We had numerical problems with the enter method and backward elimination regressions, so we select the forward selection model as our predictive model. This model indicates that the Beta variable has the most effect on whether or not the analysis will fail; in fact, the other variables do not have large enough individual effects to be included in the model.

The classification table for this model is provided in Table 7.19. Note that the percent correct percentage for predicting cases that actually fail is low, leading to an overall percent correct value smaller than those for our other predictive models for failure.

Our outlier analysis and the SPSS software indicate that cases 22, 23, 24, 30, and 83 are outliers. Cases 22, 23, and 24 are for the standard dining philosophers program with 8 (22), 10 (23), and 12 (24) philosophers. The value of the Beta metric is 0 for these cases, so the model predicts that the analysis will not fail, but the cases actually do fail. Case 30 is for the dining philosophers with dictionary program with 7 philosophers. Again, Beta is 0 so the model does not predict failure, but the case actually does fail. Case 83 is for the memory management program with 3 users and no variables modeled. The value of the Beta metric is 42 for this case, so the model predicts that the analysis will fail, but the case actually does not fail.

Table 7.19. TRACC Failure Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	43	2	95.56 %
1	9	18	66.67 %
		Overall :	84.72 %

7.10.4.2 Predictive Model for Other Properties

We had numerical problems with all three regression methods when we included the property metrics, so we performed these regressions including only the program metrics. Although we could get all three regression methods to build models using only the program metrics, all of the models had several terms with very high coefficients and standard errors. Because all three models appear to be overfitted to the data and are therefore probably not general enough for use as predictive models, we do not select any of them. We thus do not provide a predictive model for failures of TRACC checking properties other than deadlock.

7.10.5 SMV

This section provides the results of our logistic regressions to predict failure of analysis runs using SMV. The section includes a model for checking for deadlock and a model for checking other properties.

7.10.5.1 Predictive Model for Deadlock

The results of the three regressions indicate that, for SMV, checking for deadlock, the C and MaxC metrics have the largest effect on whether or not the analysis will fail.

We select the enter method model because it provides the lowest deviance and the highest percent correct; the backward elimination model is equivalent. The classification table for this model is provided in Table 7.20. We note that, despite the high overall percent correct value, the model is not as accurate as we would like for predicting failed cases.

Table 7.20. SMV Failure Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	107	2	98.17 %
1	3	8	72.73 %
		Overall :	95.83 %

In our analysis of the plot of the standardized residuals we identify cases 7, 9, 14, and 120 as outliers. The SPSS software does not identify case 9 as an outlier, but since this case is not predicted correctly, we include it as an outlier. Cases 7 and 9 are for the cyclic program with 8 (7) and 10 (9) customers and schedulers with no variables modeled. The values of C and MaxC are small enough that the model does not predict failure for these cases, but the cases actually fail. Case 14 is for the dac program with 20 solvers; the model predicts failure for this case, but the case actually completes successfully. Case 120 is for the readers/writers program with 12 readers and writers and all variables

modeled. The predicted probability of failure is 0.516, which is rounded up to a predicted failure, but the case actually completes successfully.

7.10.5.2 Predictive Model for Other Properties

We had numerical problems with the enter method and backward elimination regressions when we included the property metrics, so we performed these regressions including only the program metrics. We again had numerical problems with these methods; only the forward selection method generated a model (using both program and property metrics). The results of the this regression indicate that, for SMV, checking properties other than deadlock, the T and MaxTRANS metrics have the strongest effect on whether or not the analysis will fail.

The classification table for the forward selection model is provided in Table 7.21. Although the overall percent correct value is over 96%, the model does a poor job predicting cases that failed. It seems more important to us to accurately predict cases that will fail rather than cases that will not fail, so this model is not as good in our view as the overall percent correct value implies.

Table 7.21. SMV Failure Classification Table for Other Properties

	Predicted 0	Predicted 1	Percent Correct
Observed 0	167	0	100.00 %
Observed 1	7	6	46.15 %
		Overall :	96.11 %

Our outlier analysis indicates that cases 15, 19, 21, 139, 141, 143, and 144 are outliers. Case 15 is for the cyclic program with 10 customers and schedulers and no variables modeled, checking no_c3c2. Cases 19 and 21 are for cyclic program with 12 customers and schedulers and no variables modeled, checking no_c3c2 (19) and no_c2ss (21). For these three cases, the T values are in the top quartile of T values, but the MaxTRANS values are just above the bottom quartile of MaxTRANS values. This causes the model to predict success for these cases, all of which actually fail. Cases 139,

141, and 143 are for the ring program with 8 (139), 10 (141), and 12 (143) customers and no variables modeled, checking no_m1m2. Case 144 is for the ring program with all variables modeled, checking no_m1m2. In all these cases, the T values are in the top quartile but the MaxTRANS values are in the second quartile. The model predict that these cases will not fail, but in fact they do.

7.10.6 INCA

This section provides the results of our logistic regressions to predict failure of analysis runs using INCA. The section includes a model for checking for deadlock and a model for checking other properties.

7.10.6.1 Predictive Model for Deadlock

We had numerical problems using the enter and backward elimination methods on this dataset. The results of the forward selection regression indicate that for INCA, checking for deadlock, the value of N has the strongest influence on whether or not the analysis case will fail.

The classification table for the forward selection model is provided in Table 7.22. Despite the high overall percent correct value, the model does not predict the single failure case correctly. Because there is only one failure in the dataset used to generate the model, we are unsure how useful this model would be in practice.

Table 7.22. INCA Failure Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	118	1	99.16 %
1	1	0	0.00 %
		Overall :	98.33 %

Our examination of the plot of the standardized residuals indicates that cases 71 and 72 are outliers. The SPSS software does not classify these (or any) cases as outliers, but

since they represent the two cases for which the model does not predict failure accurately, we include them as outliers. The cases are for the gas station program with 6 customers and no variables (71) and all variables (72) modeled. The value of N is larger for case 71 and the model predicts failure, but INCA detects a (spurious) deadlock for this case and terminates successfully. The value of N is smaller for case 72 and the model does not predict failure, but for this case the INCA analysis fails.

7.10.6.2 Predictive Model for Other Properties

INCA did not fail on any of the cases for which we checked properties other than deadlock. It is therefore not possible to use logistic regression to build a predictive model for failure of these analysis cases. The simplest model we could use, of course, would be one that ignored all the metrics and predicted that all analysis cases would not fail, but we suspect this a result of our input domain, rather than an indication than INCA never fails checking properties other than deadlock. Experimental data over a wider input domain, leading to at least some INCA failures, would be required to build a predictive model.

7.10.7 FLAVERS

This section provides the results of our logistic regressions to predict failure of analysis runs using FLAVERS. Because FLAVERS does not currently support checking for deadlock, we only include regressions for checking other properties in this section.

We had numerical problems with the enter method and backward elimination regressions when we included the property metrics, so we performed these regressions including only the program metrics. We again had numerical problems with these methods; only the forward selection method generated a model (using both program and property metrics). The results of the this regression indicate that for FLAVERS, checking properties other than deadlock, the Vars metric has the strongest influence on whether or not the analysis will fail. Further investigation of the model, however, shows that all the terms in the model have very high coefficients and standard errors. Because the model therefore appears to be overfitted to the data and is probably not general enough for use as

a predictive model. We therefore do not provide a predictive model for failures of FLAVERS checking properties other than deadlock.

7.11 Predictive Models for Spurious Results

We use logistic regression to build the predictive models for spurious results. Because a result can not be spurious in an analysis case that fails, the regression only includes analysis cases that did not fail. As discussed above, the property metrics are not meaningful for checking deadlock. We therefore generate two models for each tool - one for deadlock, using only the program metrics as independent variables, and one for the other properties, using both the program and property metrics as independent variables.

As described in Chapter 6, we use three different logistic regression methods for each model. The methods are the enter method, backward elimination, and forward selection. Because the deviance quantifies how much of the variance in the data is captured by a specific model (with a smaller deviance indicating a better fit), we use this value as one of our considerations when choosing between the models. We believe the percent of the predictions by the model that are correct to be an even more important consideration, so we provide these values as well. The deviance and percent correct values for each of the logistic regressions are provided in Table 7.23. More detailed examination of each model is provided in the following sections. We experienced the same numerical problems as in the failure regressions, and we followed the same approach to resolve them.

7.11.1 SPIN, Never Claims

This section provides the results of our logistic regressions to predict spurious results for analysis runs using SPIN with never claims. As for the predictive models for analysis times, we include deadlock in this section as well.

7.11.1.1 Predictive Model for Deadlock

The results of the three regressions indicate that for SPIN, checking for deadlock, the Vars metric has the most significant effect on whether or not analysis results are spurious.

This is not particularly surprising, since in the presence of spurious results we modeled additional variables to improve analysis accuracy.

Table 7.23. Deviances and Percents Correct for Spurious Result Models

	Enter Method		Backward Elimination		Forward Selection	
	Deviance	% Correct	Deviance	% Correct	Deviance	% Correct
SPIN, Never Claims						
Deadlock	45.814	87.62	46.600	88.57	50.335	89.52
Other Properties	46.635	91.22	47.643	91.22	51.862	91.22
SPIN, Assertions						
Other Properties	27.636	96.45	30.560	96.45	77.238	92.20
SPIN+PO						
Deadlock	48.391	87.27	49.388	86.36	57.427	84.55
Other Properties	18.314	97.26	-	-	28.240	95.89
TRACC						
Deadlock	-	-	-	-	29.150	88.89
Other Properties	2.773	94.44	2.773	94.44	5.407	94.44
SMV						
Deadlock	-	-	-	-	50.272	87.16
Other Properties	32.774	95.21	37.502	93.41	51.176	92.81
INCA						
Deadlock	-	-	-	-	43.974	94.96
Other Properties	58.770	92.78	61.153	92.22	78.265	87.78
FLAVERS						
Other Properties	158.229	72.05	163.609	73.29	36.078	95.65

Although the enter method model has the smallest deviance (by 2%), we select the backward elimination model as our predictive model. Consideration of the classification tables indicates that the enter method predicts that accurate results will be provided from 7 of the cases that actually yield spurious results, while the backward elimination model only (incorrectly) predicts 5 such accurate results. We believe this to be a more important consideration than a small increase in deviance. The classification table for the selected model is provided in Table 7.24.

From the plot of the standardized residuals, we identify cases 17, 30, 33, and 36 as outliers. The SPSS software only identifies case 17 as an outlier, since the studentized residuals for cases 30, 33, and 36 are below 2.00. Case 17 is the standard dining philosophers problem with 2 philosophers (and no variables). The model predicts a spurious result for this case, but the actual analysis results are accurate. Cases 30, 33, and

36 are all for the dining philosophers with fork manager problem with only fork_2 modeled. The model does not predict spurious results for these cases, but SPIN does detect (spurious) deadlock for these cases.

Table 7.24. SPIN Spurious Results Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	63	7	90.00 %
1	5	30	85.71 %
		Overall :	88.57 %

7.11.1.2 Predictive Model for Other Properties

We had numerical problems with all three regression methods when we included the property metrics, so we performed these regressions including only the program metrics. The results of the three regressions indicate that for SPIN using never claims and checking properties other than deadlock, the Vars metric has the strongest effect on whether or not the analysis results will be spurious, followed by the C metric.

All three regression methods yield the same percent correct values, but we select the forward selection model as our predictive model, despite the fact that it has the largest deviance. We make this selection because this model contains the fewest metrics, and thus may be slightly more general than the other models.

The classification table for the selected model is provided in Table 7.25. Despite the fairly high overall percent correct value, the incorrect predictions for 40% of the cases yielding spurious results is somewhat larger than we would like.

Our examination of the plot of the standardized residuals identifies cases 3, 7, 10, 13, 122, and 128 as outliers. The SPSS software does not identify any outliers (based on the studentized residuals), but since the above 6 cases are also predicted incorrectly by the model, we include them as outliers. Cases 3, 7, 10, and 13 are for the cyclic program

with 4 (3), 6 (7), 8 (10), and 10 (13) customers and schedulers with no variables modeled, checking no_c3c2. The effect of the C metric causes the model to predict accurate results for these case, but in fact they actually yield spurious results. Cases 122 and 128 are for the readers/writers program with 4 (122) and 6 (128) readers and writers and no variables modeled, checking no_r1w. The model predicts spurious results for these two cases, but they actually yield accurate identification of the property violation.

Table 7.25. SPIN, Never Claims, Spurious Results Classification Table

Observed	Predicted 0	Predicted 1	Percent Correct
0	123	5	96.09 %
1	8	12	60.00 %
		Overall :	91.22 %

7.11.2 SPIN, Assertions

This section provides the results of our logistic regressions to predict spurious results for analysis runs using SPIN with assertions. Because the regressions for checking for deadlock were included in the previous section, we only include regressions for checking other properties in this section.

We had numerical problems with the enter method and backward elimination regressions when we included the property metrics, so we perform these regressions including only the program metrics. The results of the three regressions indicate that, as usual, the Vars metric has the strongest effect on whether or not the analysis will yield spurious results, followed by the C metric.

We select the backward elimination model as our predictive model because it provides the highest percent correct value. Although the deviance for this model is slightly higher than for the enter method model (the percent correct values are identical), we accept this growth to reduce the metrics in the model to four.

The classification table for the selected model is shown in Table 7.26. The accuracy of the model for predicting cases yielding spurious results is somewhat low, but the overall percent correct value is high.

Table 7.26. SPIN, Assertions, Spurious Results Classification Table

Observed	Predicted 0	Predicted 1	Percent Correct
0	128	2	98.46 %
1	3	8	72.73 %
		Overall :	96.45 %

Our examination of the plot of the standardized residuals identifies cases 3, 7, 112, and 118 as outliers. The SPSS software only identifies cases 3 and 7 as outliers; the studentized residuals for the other two cases are below the threshold. Cases 3 and 7 are for the cyclic program with 4 (3) and 6 (7) customers and schedulers with no variables modeled, checking no_c3c2. The effect of the C metric causes the model to predict accurate results for these case, but in fact they actually yield spurious results.

7.11.3 SPIN+PO

This section provides the results of our logistic regressions to predict spurious results for analysis runs using SPIN+PO. The section includes a model for checking for deadlock and a model for checking other properties.

7.11.3.1 Predictive Model for Deadlock

The results of the three regression indicate that for SPIN+PO, checking for deadlock, the Vars metric has the largest effect on whether or not an analysis will yield spurious results, followed by the C metric.

We select the enter method model as our predictive model because it provides the highest percent correct value and the lowest deviance. The classification table for this model is provided in Table 7.27. The overall percent correct value for this model is lower

than we have typically found, but it is still high enough to indicate a good fit for the model.

Table 7.27. SPIN+PO Spurious Results Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	65	7	90.28 %
1	7	31	81.58 %
		Overall :	87.27 %

We identify case 19 as an outlier from our examination of the plot of the standardized residuals. The SPSS software also identifies cases 17 and 66 as outliers, but since accurate results are correctly predicted for these cases, we do not include them as outliers. Case 19 is for the standard dining philosophers program with 2 philosophers. The model predicts a spurious result for this case, while SPIN+PO actually correctly identifies the possibility of deadlock.

7.11.3.2 Predictive Model for Other Properties

We had numerical problems with all three regression methods when we included the property metrics, so we performed these regressions including only the program metrics. The backward elimination method still had numerical problems, but we were able to complete the other two regressions. The results of these regressions indicate that for SPIN+PO, checking properties other than deadlock, the Vars metric has the largest effect on whether or not an analysis will yield spurious results, followed by the C metric.

Despite the fact that the enter method model has a higher percent correct value, we select the forward selection model as our predictive model. The coefficient and standard error for the Vars metric in the enter method is extremely high, and we select the forward selection model to avoid choosing a model that is probably overfitted to the data.

The classification table for the selected model is provided in Table 7.28. The accuracy for predicting cases that yield spurious results is somewhat low, but the overall percent correct value is fairly high.

Our examination of the plot of the standardized residuals indicates that cases 3, 121, and 127 are outliers. The SPSS software only identifies cases 3 and 121 as outliers, but since case 127 is not predicted correctly by the model, we include it as an outlier as well. Case 3 is for the cyclic program with 4 customers and schedulers and no variables modeled, checking no_c3c2. As we have seen in the other spurious result models, the C metric has sufficient effect to cause the model to predict an accurate result, but the case actually yields a spurious result. Cases 121 and 127 are for the readers/writers program with 2 (121) and 4 (127) readers and writers and no variable modeled, checking no_r1w. The model predicts spurious results for these cases, but SPIN+PO accurately detects the property violation.

Table 7.28. SPIN+PO Spurious Results Classification Table for Other Properties

Observed	Predicted 0	Predicted 1	Percent Correct
0	131	3	97.76 %
1	3	9	75.00 %
		Overall :	95.89 %

7.11.4 <u>TRACC</u>

This section provides the results of our logistic regressions to predict spurious results for analysis runs using TRACC. The section includes a model for checking for deadlock and a model for checking other properties.

7.11.4.1 <u>Predictive Model for Deadlock</u>

We had numerical problems with the enter method and backward elimination regressions. The results of the forward selection regression indicate that for TRACC,

checking for deadlock, the C metric has the strongest effect on whether or not the analysis will yield spurious results.

The classification table for the forward selection model is provided in Table 7.29. The overall percent correct value for this model is lower than we typically find, but is still high enough to provide a good fit to the data.

Table 7.29. TRACC Spurious Results Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	2	4	33.33 %
1	1	38	92.44 %
		Overall :	88.89 %

Our examination of the plot of the standardized residuals indicates that cases 10, 11, and 12 are outliers. The SPSS software also identifies these cases as outliers. These cases are for the standard dining philosophers program with 2 (10), 4 (11), and 6 (12) philosophers. The model predicts that these three cases will yield spurious results, but TRACC actually correctly detects the possibility of deadlock.

7.11.4.2 Predictive Model for Other Properties

We had numerical problems with all three regression methods when we included the property metrics, so we performed these regressions including only the program metrics. Although we could get all three regression methods to build models using only the program metrics, all of the models had several terms with very high coefficients and standard errors. Because all three models appear to be overfitted to the data and are therefore probably not general enough for use as predictive models, we do not select any of them. We thus do not provide a predictive model for spurious results for TRACC checking properties other than deadlock.

7.11.5 SMV

This section provides the results of our logistic regressions to predict spurious results for analysis runs using SMV. The section includes a model for checking for deadlock and a model for checking other properties.

7.11.5.1 Predictive Model for Deadlock

We had numerical problems with the enter method and backward elimination regressions. The results of the forward selection regression indicate that for SMV, checking for deadlock, the Vars metric has the largest effect on whether or not an analysis will yield spurious results, followed by the C metric.

The classification table for the forward selection model is provided in Table 7.30. The accuracy for the model's predictions for case that yield spurious results is somewhat low, but the overall percent correct value is high enough to indicate a reasonably good fit.

Table 7.30. SMV Spurious Results Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	69	6	92.00 %
1	8	26	76.47 %
		Overall :	87.16 %

Our examination of the plot of the standardized residuals indicates that case 16 is an outlier. The SPSS software also indicates case 67 is an outlier, but because this case is predicted correctly by the model, we do not include it as an outlier. Case 16 is for the standard dining philosophers programs with 2 philosophers. The model predicts a spurious result for this case, but SMV correctly detects the possibility of deadlock.

7.11.5.2 Predictive Model for Other Properties

We had numerical problems with all three regression methods when we included the property metrics, so we performed these regressions including only the program metrics.

The results of these regressions indicate that for SMV, checking properties other than deadlock, the Vars metric has the largest effect on whether or not an analysis will yield spurious results, followed by the C metric.

Despite the fact that it has the lowest overall percent correct value, we select the forward selection model as our predictive model. In both the enter method and backward elimination models, the coefficient and standard error for the Vars metric are very large, providing evidence of overfitting.

The classification table for the selected model is provided in Table 7.31. While the overall percent correct value is fairly high, the model incorrectly predicts accurate results for 41% of the cases that yield spurious results. This model may thus not be as useful for predicting spurious results as the overall percent correct value implies.

Table 7.31. SMV Spurious Results Classification Table for Other Properties

Observed	Predicted 0	Predicted 1	Percent Correct
0	145	5	96.67 %
1	7	10	58.82 %
		Overall :	92.81 %

We identify case 11 as an outlier in the plot of the standardized residuals; the SPSS software identifies case 11 as the only outlier. Case 11 is for the cyclic program with 8 customers and schedulers and no variables modeled, checking no_c3c2. The model predicts an accurate analysis result for this case, but SMV actually detects a spurious property violation.

7.11.6 INCA

This section provides the results of our logistic regressions to predict spurious results for analysis runs using INCA. The section includes a model for checking for deadlock and a model for checking other properties.

7.11.6.1 Predictive Model for Deadlock

We had numerical problems with the enter method and backward elimination regressions. The results of the forward selection regression indicate that for INCA, checking for deadlock, the Vars metric has the largest effect on whether or not an analysis will yield spurious results, followed by the C metric.

The classification table for the forward selection model is provided in Table 7.32. All of the percent correct values in the table are high, indicating a good fit.

Table 7.32. INCA Spurious Results Classification Table for Deadlock

Observed	Predicted 0	Predicted 1	Percent Correct
0	63	2	96.92 %
1	4	50	92.59 %
		Overall :	94.96 %

Examination of the plot of the standardized residuals indicates that cases 19, 61, and 63 are outliers. The SPSS software also identifies case 70 as an outlier, but because this case is predicted correctly by the model, we do not classify it as an outlier. Case 19 is for the standard dining philosophers program with 2 philosophers. The model predicts a spurious result, but INCA correctly detects the possibility of deadlock. Cases 61 and 63 are for the gas station program with 1 (61) and 2 (63) customers and no variables modeled. The model predicts accurate results for these two cases, but INCA detects a (spurious) deadlock.

7.11.6.2 Predictive Model for Other Properties

We had numerical problems with all three regression methods when we included the property metrics, so we performed these regressions including only the program metrics. The results of these regressions indicate that for INCA, checking properties other than

deadlock, the Vars metric has the largest effect on whether or not an analysis will yield spurious results, followed by the C metric.

We select the enter method model as our predictive model because it provides the highest percent correct value and the lowest deviance. The classification table for this model is provided in Table 7.33. Despite the high overall percent correct value, the accuracy of the model predictions for cases that yield spurious results is somewhat low.

Table 7.33. INCA Spurious Results Classification Table for Other Properties

Observed	Predicted 0	Predicted 1	Percent Correct
0	152	5	96.82 %
1	8	15	65.22 %
		Overall :	92.78 %

Examination of the plot of the standardized residuals indicates that cases 154 and 160 are outliers. The SPSS software also indicates that these two cases are the only outliers. Cases 154 and 160 are for the readers/writers program with 4 (154) and 6 (160) readers and writers and no variables modeled, checking no_r1w. The model predicts that these cases will yield spurious results, but INCA correctly detects the property violation.

7.11.7 FLAVERS

This section provides the results of our logistic regressions to predict spurious results for analysis runs using FLAVERS. Because FLAVERS does not currently support checking for deadlock, we only include regressions for checking other properties in this section.

We had numerical problems with the enter method and backward elimination regressions when we included the property metrics, so we perform these regressions including only the program metrics. The results of the three regressions indicate that, as

usual, the QRE States metric has the strongest effect on whether or not the analysis will yield spurious results, followed by the C metric.

We select the forward regression model as our predictive model because it provides a significantly higher percent correct value and significantly lower deviance than the other models. The classification table for the selected model is provided in Table 7.34. All the percent correct values in the table are high.

Table 7.34. FLAVERS Spurious Results Classification Table for Other Properties

Observed	Predicted 0	Predicted 1	Percent Correct
0	74	4	94.87 %
1	3	80	96.39 %
		Overall :	95.65 %

Examination f the plot of standardized residuals indicates that cases 71, 73, and 119 are outliers. The SPSS software also indicates that case 78 is an outlier, but because the model predicts this case accurately, we do not include it as an outlier. Cases 71 and 73 are for the dining philosophers with host program with 6 (71) and 7 (73) philosophers and no variables modeled, checking no_p1p2. The model predicts spurious results for these cases, but FLAVERS accurately checks the property. Case 119 is for the ring program with 2 servers and masters and no variables modeled, checking no_m1m2. The model predicts and accurate analysis result, but FLAVERS detects a (spurious) property violation.

7.12 Validating the Models

We require two characteristics of good predictive models - the models must be correctly generated from valid experimental data, and the models must prove to be useful in actual practice. We have carefully developed a sound empirical methodology to ensure our experimental data is valid, and we have rigorously applied standard statistical analysis

techniques to ensure that the models have been correctly generated from that data. Showing that the models will be useful in practice, however, is much more difficult.

We believe the best way to find out whether the models are useful in practice is to use them on real programs and properties, and determine whether they provide good predictive power. We have begun this effort with the case study programs discussed in Chapter 8.

Another way to try to validate the models (i.e., show that they are correct and useful) is to use them for larger sizes of the academic programs included in the experiment. While this seems intuitively attractive, it will not yield any insight about the predictive power of the models for real programs and properties. The predictive models we generate essentially represent n-dimensional vectors, where n is the number of metrics included in the model. Our hope is that the combination of the academic programs and properties in the experiment will yield a vector that approximates the direction in which the n metrics grow in real programs. None of the academic programs in the experiment follow this n-dimensional vector. We would therefore not expect good predictions for larger sizes of these programs. The best we could learn from such a study is how well the predictive models work for larger academic programs, and since predicting performance on large academic programs is not the goal of our predictive models, the results would not be of practical interest.

The predictive models we have built for failure and spurious results seem to provide reasonable predictive power within the dataset. Of course, these models still need to be validated on real programs. Unfortunately, the predictive models we have built for analysis times generally only capture a small amount of the variance in the dataset. Given that the predictive models will not even work very well within the dataset, it would be unreasonable to expect that they will have good predictive power outside this domain, whether on real programs or larger academic programs. We therefore limit our discussion to the validity of the predictive models within the input domain of the experiment.

We first provide a comparison of the tools based on analysis time. Table 7.35 lists the number of times each tool had the fastest analysis time for an analysis case, as well as the number of times the predictive models predict each tool will have the fastest analysis time. As usual, we separate checking for deadlock from checking the other properties. In cases where two tools had the (same) fastest analysis time, both tools were credited with the fastest time.

Table 7.35. Counts of Fastest Analysis Times

	Deadlock		Other Properties	
	Observed	Predicted	Observed	Predicted
SPIN, Never Claims	28	43	23	13
SPIN, Assertions	-	-	23	12
SPIN+PO	47	10	24	11
TRACC	0	0	0	1
SMV	40	40	67	28
INCA	9	26	49	73
FLAVERS	-	-	2	36

For predicting analysis times checking for deadlock, the predictive models yield optimistic predicted counts for SPIN and INCA and pessimistic predicted counts for SPIN+PO. For predicting analysis times checking other properties, the predictive models yield optimistic predicted counts for INCA, FLAVERS, and TRACC and pessimistic predicted counts for SPIN and SPIN+PO. When we calculate the average magnitude of the optimistic or pessimistic predictions (expressed as a percentage), we find that the average error magnitude is over 250% (ignoring TRACC). We note that a large part of this error is caused by the significant overestimate for FLAVERS, but when we exclude this estimate the average error magnitude is still almost 72%. This result indicates that the predictive models do not provide good predictive power, even in the input domain of the experiment.

The analysis above still excludes an important consideration, however. For instance, for checking deadlock, the observed and predicted counts for SMV are both 40. This does not indicate, however, which cases are observed to be the fastest and which cases are

predicted to be the fastest. The 40 cases predicted to be the fastest by the predictive models may not include any of the 40 cases for which SMV actually provides the fastest analysis time. It is also important, therefore, to consider the correspondence between specific observed and predicted fastest cases.

Toward this end, we have examined the data to determine the number of cases in which the predictive models select the fastest tool. The results are provided in Table 7.36. We have also included the number of times the predictive models select the second and third fastest tools. The total number of analysis cases is 299; 119 for deadlock, and 180 for other properties.

Table 7.36. Specific Case Predictions

	Deadlock	Other Properties
Fastest Tool	30	62
Second Fastest Tool	47	30
Third Fastest Tool	29	24

For checking deadlock, the predictive models select the fastest tool in 25% of the cases, the second fastest tool in 40% of the cases, and the third fastest tool in 24% of the cases. For checking other properties, the predictive models select the fastest tool in 34% of the cases, the second fastest tool in 17% of the cases, and the third fastest tool in 13% of the cases. While the predictive models do not select the fastest tool as often as we had hoped, they do select one of the fastest three tools 89% of the time for deadlock and 64% of the time for other properties. The experiment includes 5 analysis tools that check for deadlock, so a random tool selection would pick one of the three fastest tools 60% of the time. The experiment includes 7 analysis tools for checking other properties (using two property specification styles for SPIN), so a random selection would pick one of the three fastest tools 43% of the time. These results indicate that the predictive models may be able to provide some useful guidance to an analyst trying to select an analysis tool, despite the weaknesses in the models discussed above.

As a final examination of the validity of the predictive models over the input domain of the experiment, we quantify the effect of using the predictive models. To do so, we use the average ranking measure we use to compare the tools in Section 7.4. We originally considered using mean analysis time for comparison, but the cases requiring significant analysis time overwhelmed the much more numerous cases requiring less time.

The results are provided in Table 7.37. For comparison purposes, we have also included the effect of randomly selecting a tool for each case and the effect of using each tool for all the cases.

Table 7.37. Effect of Using Predictive Models

	Deadlock	Other Properties
Predictive Models	2.21	2.92
Random Selection	2.40	3.11
SPIN, Never Claims	2.11	3.07
SPIN, Assertions	-	2.59
SPIN+PO	1.85	2.73
TRACC	4.60	6.00
SMV	2.23	2.15
INCA	3.31	3.27
FLAVERS	-	4.99

As always, the results in the table must be considered with care. Comparison between using the predictive models and randomly selecting a tool for each case is straightforward, and this comparison indicates that at least the predictive models provide better tool selection than random selection does. When we compare using the predictive models to using specific tools for all cases, however, the comparison is not as straightforward. The predictive models never select a tool that fails (in the experiment), but all the tools fail on at least one case. For checking deadlock, using SPIN+PO or SPIN for all cases provides better performance than using the predictive models (ignoring failures). For the other properties, using SPIN (with assertions), SPIN+PO, or SMV provides better performance than using the predictive models, again ignoring failures.

7.13 Summary

We use a variety of statistical techniques to analyze our experimental data. In this section, we summarize the results presented above.

We use two sample t-tests and paired sample t-tests to statistically test for biases we may have introduced by our methodology. Of the six areas of potential bias identified, we find that in five of those areas there was no statistical evidence that our methodology introduced bias. For specifying SMV properties, we discover that, when appropriate, it is generally better to specify properties using additional variables in the transition relation than to use an alternate CTL specification. All the data in the dataset therefore represents using additional variables (when necessary).

We preprocess our data to remove metrics that are collinear with others, since this collinearity can cause problems in both the linear and logistic regression techniques. This preprocessing reduces the number of program metrics included from 26 to 11, and reduces the number of property metrics from 9 to 6. We conduct randomization tests to ensure we have not removed metrics with apparent (but not real) collinearity; the results of these tests indicate that we have only removed metrics that are truly collinear in this dataset.

The results of our linear regressions are disappointing. We use threshold of 0.800 for the R^2 value to indicate a good fit, and 8 out of the 12 linear models we build have R^2 values less than 0.54. Because these models do not capture much of the variance in the experimental data, they are unlikely to provide good predictive power for real programs. We also check to see if one or more of the metrics commonly appear in the models, indicating that there are certain characteristics of the program or property that affect the analysis times for all the tools. We find no such common characteristics in the linear regression models.

The results of our logistic regressions to predict failure of analysis runs are more encouraging. For all our selected predictive models for failure, the overall percent correct

value is greater than 90%. This indicates that these models may provide reasonable predictive power for real programs. We again do not find any common characteristics that appear in all the models.

The results of our logistic regressions to predict spurious results for analysis runs are also encouraging, with all our selected predictive models having overall percent correct values greater than 87%. Again, this implies that these models may provide reasonable predictive power for real programs. All but one of these models had the strongest effect on the results from the number of variables modeled. This is not surprisingly, because when an analysis run yielded a spurious result, we added additional variable modeling to try to improve the accuracy of the analysis. The average number of communications in the tasks in the program also had a noticeable effect in all these models.

We discuss several approaches for validating the models, but because the linear regression models appear to be weak, we restrict our attention to the validity of these models over the input domain of the experiment. The models do not predict the fastest tool for a given program and property very well (24% of the time for deadlock, 34% of the time for other properties). They do, however, select one of the three fastest tools a significant percentage of the time, which may be somewhat useful. We quantify the impact of using the predictive models, comparing to random tool selection and selecting one tool to use for all programs and properties. Using the predictive models was better than random tool selection, but was worse than selecting certain tools for all the analyses.

Finally, because our real interest is in how long each of the tools takes to analyze Ada programs, we also analyze timing data that includes all times from input of the Ada program to output of the analysis results.

CHAPTER 8

CASE STUDIES

In this chapter we describe the results of our preliminary examination of several programs we have acquired from government and academic sources. To be most useful, concurrency analysis tools need to be applicable to programs of realistic size, containing realistic communication structures. In almost all cases, including the experiment we have conducted, static concurrency analysis tools have been demonstrated using programs from the concurrency analysis literature. It is not clear that these academic programs are representative of concurrent programs in general. Most tasks in these programs are relatively small, and the program constructs used in these programs are relatively simple.

To begin gathering information about how the concurrency analysis tools will fare when applied to real concurrent programs, we have acquired several real concurrent programs and examined various characteristics of those programs. Our examination includes discussion of the program constructs and language features used in the programs and observations about program characteristics that are likely to affect the applicability of static concurrency analysis tools to these programs. The programs and the results of our examination are described below.

8.1 Programs Considered

The programs we examined were acquired from academic and government sources. To find these programs, we monitored the newsgroup comp.lang.ada, discussed our need for real programs at conferences and demonstrations, and reviewed the concurrency analysis literature for previous work with real concurrent programs.

We actually had a surprising amount of difficulty gaining access to real concurrent programs. Our sample therefore does not represent a careful selection from a large set of programs; rather, it consists of all the real programs to which we could gain access. We believe our difficulties arose for a number of reasons. For example, we believe a large

number of concurrent programs are written under government contract, and our access to this group of programs was severely curtailed for contractual and security reasons. In addition, we require source code to perform the static concurrency analysis. Many commercial and government agencies are hesitant to provide source code for their products. Of course, there are potentially many other factors that also make it difficult for the academic community to gain access to real concurrent programs.

8.1.1 Border Defense System (BDS)

The Border Defense System (BDS) code was written by T. Griest and M. Sperry of LabTek Corporation in 1988/89. The code was designed to simulate a system in which incoming targets are detected and tracked, rockets are assigned to those targets and launched, and damage assessment is carried out to determine whether the rockets destroy their targets.

The system consists of approximately 4K lines of code, contained in 58 files (25 package specifications and 33 package and procedure bodies). After appropriate inlining has been accomplished (see Section 8.2.1), the system consists of 14 tasks.

8.1.2 Train Control Program

The train control code was written by a group of students at SUNY/Plattsburgh for a real-time class; it was provided to us by John McCormick. The code was designed to control a model railroad train system, in which the system senses the locations of multiple trains on the track, provides access to sections of track in a manner that avoids collisions, and processes commands for the trains on the track.

The system consists of approximately 5K lines of code, contained in 31 files (17 package specifications and 14 package and procedure bodies). After appropriate inlining has been accomplished, the system consists of 46 tasks.

8.1.3 ALSP Common Module (ACM)

The ALSP Common Module (ACM) code was written by a group at Mitre Corporation; it was provided to us by Richard Weatherly. The code was designed to coordinate multiple interacting simulations, providing communication between the simulations and management of the simulation objects.

The system consists of approximately 30K lines of code, contained in 262 files (50 package specifications and 212 package and procedure bodies). We have not yet attempted inlining on this system, so we do not know exactly how many tasks will be in the system, but we are estimating approximately 59 tasks.

8.2 Conversion to Control Flow Graphs

As the first step in our experimental methodology, we convert the Ada program to be analyzed into a set of CFGs. There are a number of characteristics of the programs considered here that adversely affect this conversion. These "problem areas" include task interactions in called procedures, separate packages, generic definitions and instantiations, use of Ada attributes, use of pragmas, use of compiler-dependent packages, use of discriminated types, and use of exception handlers for control flow.

8.2.1 Task Interactions in Called Procedures

When an Ada program is converted to a set of CFGs, a CFG is created for each function, procedure, task, and exception handler in the system. This approach causes problems, however, when one procedure calls another and the called procedure contains a task interaction. For analysis purposes, the calling procedure needs to include this interaction, and simply using the set of CFGs created for the program does not explicitly provide this information. This problem can occur through an arbitrary number of procedure calls, so it is not limited to the simple "one call level" example given above. Ensuring that task interactions contained in called procedures are considered by the analysis implies that some sort of interprocedural analysis needs to be performed to gather this information. We discuss several alternatives for solving this problem below,

but examine the scope of this problem for the three programs examined here before we do so.

In the BDS code, 17 procedures and tasks contain task interactions, either directly or through procedure calls. Of these, three are procedures that are called by other procedures and tasks. One call is the maximum depth of procedure calls required to reach a procedure containing an interaction. In the train control code, 85 procedures and tasks contain task interactions, either directly or through procedure calls. Of these, 39 are procedures that are called by other procedures and tasks. Four calls is the maximum depth of procedure calls required to reach a procedure containing an interaction. In the ACM code, 404 procedures and tasks contain task interactions, either directly or through procedure calls. Of these, 298 are procedures that are called by other procedures and tasks. We have not yet determined the maximum depth of procedure calls required to reach a procedure containing an interaction. Clearly, the above data indicates that the problem of called procedures containing task interactions is a pervasive one, and must be addressed.

There are several ways to handle the requirement for interprocedural analysis to address this problem. We have implemented a rudimentary inlining tool that performs structural inlining on the CFGs. Essentially, a CFG node representing a call on a procedure to be inlined is replaced by the CFG of the inlined procedure. This is essentially a brute-force approach to the interprocedural analysis, and it is easy to see that this approach can explode the size of the CFGs for the system. The largest CFG we have produced using this technique (on the train code) contains 2,887 nodes and 3,508 edges. We have not yet attempted to inline procedures in the ACM code.

We believe there are much more elegant solutions to this problem than structural inlining; certainly, the compiler community uses more advanced techniques. We are currently working to develop a more elegant solution that will provide the necessary interprocedural analysis without incurring the size explosion of structural inlining.

We also note that we originally tried to identify the procedures that needed to be inlined (because they contained task interactions) manually on the BDS code. We then developed a tool to identify these procedures automatically when we started to generate the CFGs for the train control code. As part of our testing of this new tool, we used it to check our manual inlining results for the BDS code. Even for this relatively simple system, we had overlooked one procedure call to a procedure containing a task interaction. It therefore seems to us that extensive automated tool support is absolutely critical when we undertake analysis of real programs.

8.2.2 Separate Packages

One of the language features provided by Ada is the ability to declare procedure and package bodies as *separate*. These separate bodies represent distinct compilation units, allowing iterative large-scale development of systems. Certainly, none of the programs in the concurrency analysis literature use this language feature.

We have discovered, however, that this feature is commonly used in our real programs. The BDS code contains 10 compilation units that are declared to be separate, the ACM code contains 127 such compilation units; the train control code does not contain any. While not every real program uses this language feature, it is clear that it is certainly not uncommon in real programs.

Our tools were not originally robust enough to handle a large number of the separate compilation units. We have made modifications to our tools to make them more robust, but there are still several separate compilation units we can not process correctly. Unfortunately, we have not yet found a standard way to identify "problematic" separates, so the processing of separate compilation units is still a trial-and-error process. Our current workaround for the separate compilation units we can not process correctly is to manually modify the code to include the separate unit in its parent unit. This defeats the original purpose of the separate unit, but lets us build the CFGs for the programs.

8.2.3 Generics

Another feature provided by Ada is the *generic*. A generic is a package or procedure that performs specified functions on whatever types and/or procedures are provided in an instantiation of that generic.

The BDS code does not use any generics. The train control code uses two very simple generics; we had no trouble processing these with our tools. The ACM code makes extensive use of generics, including nested generic instantiations (i.e., instantiation of a generic that instantiates another generic). Our tools were originally unable to handle nested generic instantiations, but we have since modified them to process these structures correctly.

8.2.4 Use of Attributes

Ada provides a set of *attributes* that let the user discover or set properties of certain types and variables. For example, the storage_size attribute can be used to specify how much storage is allocated for variables declared to be of a certain type. The BDS code uses several attributes that caused problems for our tools. Our workaround in these cases was to delete the use of the attribute. We feel that this workaround is reasonable, especially since the attributes are used to specify characteristics of the operational environment, which we ignore in our static analysis anyway.

8.2.5 Use of Pragmas

Ada allows *pragmas* as another means of giving the compiler instructions for the compilation. The BDS code, train control code, and ACM code all use pragmas that caused problems for our tools. Our workaround for these was to remove the troublesome pragmas, using the same rationale as for attributes.

8.2.6 Use of Compiler-Dependent Packages

Both the BDS code and the train control code use compiler-dependent packages. These packages were not provided with the code for licensing reasons, so these uses can not be processed correctly by our tools. Our workaround for these was to build shell

packages to provide the interface to the missing package without providing the actual functionality. While we recognize that this changes the semantics of the program, we do not believe tasking-related operations are included in these packages. Our changes should therefore not affect the results of static concurrency analysis.

8.2.7 Use of Discriminated Types

Like many high-level languages, Ada provides the capability to declare variant records; in Ada this is accomplished using *discriminated types*. The ACM code contains several uses of discriminated types that caused problems for our tools. In one case, our tools could not correctly process an implicit dereference of a pointer to a discriminated type. The workaround for this was to explicitly dereference the pointer before using it. In another case, our tools could not correctly process a derivation of a discriminated type, where the discriminated type was defined in a separate compilation unit. This problem has been corrected by a modification to our tools.

8.2.8 Use of Exception Handlers for Control Flow

Exception handlers are an Ada construct designed to provide special processing in the event of unusual program behavior. For example, if a divide by zero occurs in the program, Ada raises the Constraint_Error exception. An exception handler that traps this exception can provide special processing to recover from the error or to allow graceful degradation of the program behavior.

The ACM code contains three procedures in which exception handlers are used to detect the exit condition for a loop. This is problematic, since the exception can be raised at any statement within the loop. Our workaround for this was to add a conditional exit statement after every statement in the loop (based on a dummy condition), but this does not seem to be a feasible approach if exception handlers are used to detect normal (for instance, loop exit) conditions. In fact, determining how to sensibly model exceptions and exception handlers and their effect on control flow in a program appears to be rich topic for extensive research.

8.3 Characteristics Affecting Analysis

After we have converted the Ada code for a program into a set of CFGs, we then either apply an analysis tool directly (FLAVERS, for instance) or convert to a set of FSAs and from there to the input language of an analysis tool. Through our examination of the BDS code, train control code, and ACM code, we have discovered several characteristics of these programs that are likely to affect our ability to perform static analysis on them. These characteristics include dynamic allocation of tasks in the system, exception handlers that contain task interactions, complex individual tasks, and inclusion of task types in complicated data structures.

8.3.1 Dynamic Task Allocation

Ada provides the capability to declare task types and then to declare variables of those types or pointers to those types. Pointers to a task type can be allocated at run time, which essentially creates a new task during execution of the program. Because the static concurrency analysis techniques examined here use a static (i.e., constant) model of the tasks in the system, these techniques can not analyze programs containing dynamic task allocation.

Several of the tasks in the ACM code are dynamically allocated, so this is a real barrier to our ability to analyze this code. It turns out, however, that there is a static bound on the number of tasks present at any given time, so we can model the dynamically allocated tasks with static tasks. To do so, we replace the points of allocation with a call on a (new) start entry in the static tasks, and replace points of deallocation with calls on a (new) stop entry in the static tasks. This does not exactly capture the semantics of the dynamic task allocation, since the pointer could be deallocated when the task pointed to is at any point in its execution, while accepts can only occur at set points in the task. We believe, however, that this approach may provide sufficiently close semantics to allow useful analysis of the modified code.

The dynamic allocation of a statically bounded number of tasks brings up an interesting design question - why would a developer dynamically allocate tasks whose number is bounded (in some cases in the ACM code, the pointer is simply for a local variable that is allocated once)? A task that is inactive does not get a time slice in the Ada run-time environments we are familiar with, so there does not appear to be a valid time concern. We asked the developer of the ACM code about this, and were told that they needed to use task types so they could abort those tasks if necessary. This explains why task types were used, but does not explain why the developers used dynamically allocated pointers to those task types rather than variables of those task types. Given the problems that it causes, dynamic task allocation should be avoided whenever possible if the programs is to be subjected to static concurrency analysis.

8.3.2 Task Interactions Within Exception Handlers

We mentioned above that exception handlers are problematic for static analysis techniques. This problem is exacerbated when the exception handlers contain task interactions. If we ignore the exception handlers, we could miss potential program behaviors, implying that our analysis is no longer conservative. On the other hand, it is difficult to see how to sensibly model exceptions and exception handling so that the size of the graph structure of the program does not increase drastically.

The BDS code does not contain any exception handlers with task interactions, but the train control code does contain one such exception handler. The ACM code contains 442 exception handlers that contain task interactions. Clearly, this is a problem that will need to be addressed if we are to perform analysis of real code. Our current approach is to ignore the exception handlers in a program, but we would like to eventually capture the full semantics of the program in our analysis.

8.3.3 Complexity of Individual Tasks

One of the reasons that we believe the concurrent programs from the concurrency analysis literature may not be representative of real concurrent programs is that the tasks

in the academic programs tend to be fairly simple. We have noted that, in the three real programs we have examined, the number of tasks in a given real program may not be much larger than the number of tasks in a program from the literature, but the individual tasks can be much more complex. For example, the ACM code contains a task with 58 entries; we have yet to discover a task with this complexity in the set of programs typically analyzed in the literature.

8.3.4 Task Types in Complicated Data Structures

Because Ada allows the definition of task types, pointers to task types can be contained in arbitrarily complicated data structures. This does not appear to be a significant problem in the BDS or train control code. The ACM code, however, contains pointers to task types in complicated data structures. The worst case in the ACM code is a variable that is a pointer to an array of records, where two of the fields of the record are pointers to task types. Because in many cases the possible interactions in the system are determined by matching fully qualified entry names, building the entry names for tasks in such a structure is difficult but necessary.

8.4 Discussion

Our examination of the BDS code, train control code, and ACM code has led to the identification of a number of issues that arise when we try to analyze real code rather than the academic code from the concurrency analysis literature.

The real concurrent programs discussed above tend to use more advanced Ada features than the academic programs. Use of these features often causes problems for our tools, even those tools that have been extensively used for a number of years. Real concurrent programs also seem to have certain characteristics that will make them difficult to analyze. The most notable of these is the dynamic allocation of tasks, but other constructs, such as exception handlers that contain task interactions, may also have a significant impact.

It is difficult to draw general conclusions about real programs based on the three programs considered in this chapter, especially since we can not make any claims about how well these programs represent real concurrent programs in general. Even within this very limited dataset we see wide variations in the usage of language features and the characteristics that are likely to make the programs difficult to analyze. The BDS code and train control code have several troublesome areas, but seem like they should be amenable to analysis given some minor changes. The ACM code, on the other hand, has a large number of characteristics that will make this code extremely difficult to analyze. Because we do not know which of these programs are more like real concurrent code in general, we are unwilling to ignore any of them in our observations. We caution, however, that the observations above may give bleaker predictions about how amenable real concurrent code will be to analysis than is actually the case, especially if the ACM code represents an outlier. Further examination of a larger number of real concurrent programs will be required before we start getting a sense of what a typical concurrent program "looks like".

We have suggested workarounds for most of the problems we have encountered, and believe we are approaching the point where we will be able to attempt to prove properties on at least some of these real programs. Given the differences between the real programs and academic programs, analysis of these real programs is liable to yield significant insight into the applicability of static concurrency analysis in practice.

CHAPTER 9

IMPROVING PETRI NET-BASED STATIC ANALYSIS ACCURACY

This chapter presents an approach for improving the accuracy of Petri net-based static analysis methods by eliminating some spurious results from the analysis report. Usually, an analysis method produces a spurious result as a consequence of considering paths that can never be executed in the program (commonly called *infeasible paths*) or of considering aliasing that can never occur in the program. For an example of an infeasible path, consider the program in Figure 9.1. In the caller2 task, the path through the true branch of the first conditional and the false branch of the second conditional is infeasible, assuming the value of BranchCond does not change between the two conditionals. Infeasible paths are natural phenomena of the internal representations we use for analysis and are usually not indicative of a fault in the code.

```
task body caller1 is
begin
   accepter.entry2;
end caller1;
```

```
task body accepter is
begin
   accept entry1;
   accept entry2;
end accepter;
```

```
task body caller2 is
   BranchCond : boolean;
begin
         ...
   if BranchCond then
      accepter.entry1;
   else
      null;
   end if;
         ...
   if BranchCond then
      null;
   else
      accepter.entry2;
   end if;
end caller2;
```

Figure 9.1. Example Program

We conjecture a scenario in which an analyst submits a program and property to a static analysis tool and then examines the anomaly report that results from the analysis. Since some of the reported anomalies might be spurious, due to consideration of infeasible paths or imprecise alias resolution, the analyst must examine each anomaly to determine if it is a spurious result or not. If a large number of the results are spurious, weeding these out might overwhelm the analyst, causing results that actually do

correspond to erroneous program behavior to be discarded. If the number of spurious results is extremely large, the analyst may lose confidence in the analysis tool altogether and forego using it.

It has been our experience that, after looking at an anomaly report, an analyst easily recognizes certain infeasible paths that are the cause of at least some of the spurious results. Early experience with static analysis tools indicated that analysts identified *impossible pairs* of statements after examining anomaly reports. Using information about these impossible pairs to recognize spurious results was shown to be intractable for analyses based on control flow graph representations of a program [GMO76]. The approach presented in this chapter for improving accuracy is based on a Petri net model of a concurrent program. We describe how certain kinds of infeasible path information can be effectively captured in this model, improving the accuracy of the analysis results without degrading the performance of the analysis.

Thus, the basic idea is that an analyst would apply the static analysis method to the Petri net model of the program. Through examination of the anomaly report, certain infeasible paths that are causing spurious results to be reported become apparent. The analyst, using our approach, refines the Petri net model of the program with this information and reapplies the analysis. Of course, if the analyst knew of infeasible paths before running the initial analysis, that information could be incorporated immediately. In our experience, however, analysts do not tend to think about infeasible paths until after examining an anomaly report with some obvious spurious results. The new anomaly report typically contains fewer spurious results than the previous report, since the additional information should have eliminated the cause of some inaccuracies. Frequently, the new report is significantly smaller since additional, as yet undetected, spurious results are eliminated as well. This smaller report may not be so overwhelming to evaluate, perhaps allowing the analyst to recognize additional spurious results more easily. The effect is an iterative process in which the analyst examines an anomaly report,

adds additional information to the analysis, and reapplies the analysis repeatedly until the desired accuracy is achieved.

Our approach allows the analyst to include selected control and/or data information in the Petri net model of the program. The basic idea is to introduce information about the states that the program being analyzed can enter during execution; this information may be in the form of sequences of program statements or in the form of variable values. Petri nets are used because including additional program state information in the net and using that information to control the transitions in the net is relatively straightforward. We hypothesize that, by including additional program state information in the Petri net, we can generate a more accurate estimate of the program state space. Analysis of this more accurate state space considers fewer infeasible paths, potentially reducing the number of spurious results reported by the analysis and increasing the value of the analysis results.

The following section describes the program representations we use to analyze concurrent programs with our approach, and Section 9.2 explains how we represent certain state information to improve the accuracy of those representations. Section 9.3 presents our empirical results, and Section 9.4 offers some conclusions based on those results and some pointers to future work.

9.1 Program Representations

Because Ada is one of the few commonly used languages supporting concurrency, we use Ada examples to explain our static analysis method and our accuracy-improving approach. The approach, however, is applicable to any language using rendezvous-style communication, and could be extended to most other communication styles as well. In Ada programs, potentially concurrent activities occur in *tasks*[3]. Ada tasks typically communicate with each other using a *rendezvous*. In a rendezvous, the calling task

[3]Concurrent activities in Ada programs can also occur in procedures; for simplicity, we call them tasks in this paper.

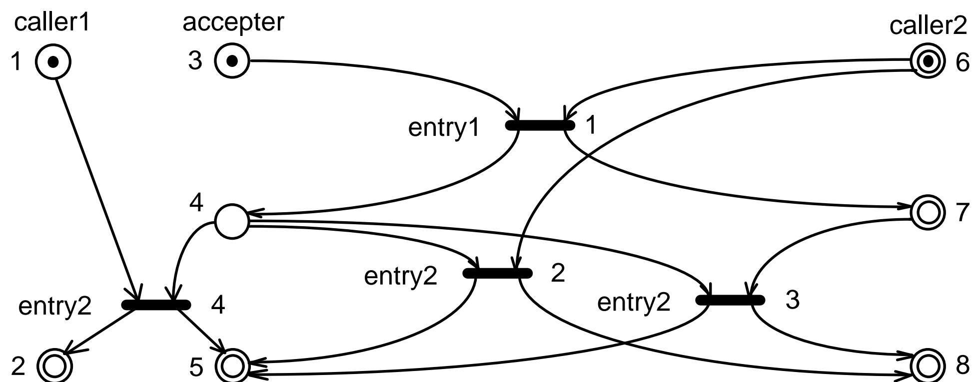
makes an *entry call* on a specific *entry* in the called task; the calling task then suspends execution until the called task terminates the rendezvous. The called task executes any statements contained in the *accept body* for the entry, then terminates the rendezvous and continues execution.

Our static analysis method builds upon a variety of internal representations of a concurrent Ada program to capture information about the program. First, we represent each task with a Task Interaction Graph (TIG) [LC89], which abstracts sequential regions of control flow into single nodes. The nodes in the TIG for a task are connected by edges representing possible interactions (entry calls/accepts) between that task and other tasks in the program. We then combine the set of TIGs for all the tasks in a program into a Petri net [DCN95] to model the system as a whole. Finally, we use the Petri net to generate a reachability graph to represent an estimate of all states the program can enter when started in the initial program state. Petri nets and reachability graphs are central to the techniques we use for improving accuracy, so these representations are described more fully below.

9.1.1 Petri Nets

Petri nets have been proposed as a natural and powerful model of information flow in a system [Pet77]. A Petri net can be represented as a 5-tuple (P, T, I, O, M0). P is the set of places in the Petri net, where a place can hold zero or more tokens. If a place holds one or more tokens, the place is said to be *marked*. T is the set of transitions in the Petri net. Tokens are moved between places in the net by the *firing* of transitions. A transition can only be fired if it is *enabled*; for a transition to be enabled, each of the input places for the transition must contain at least one token. I is a function mapping places in P to inputs of transitions in T. When a transition fires, a token is removed from each of the places that are inputs to the transition, and a token is deposited in each of the output places of the transition; O is a function mapping places in P to outputs of transitions in T. M0 is a list of all the places in the net that are initially marked.

Petri nets appear to be a valuable representation for modeling concurrent software [SC88]. In our analysis method, we use a Petri net representation generated from the set of TIGs for the concurrent program. Each place in the Petri net corresponds to a sequential region of code in one of the tasks in the program, and each transition represents a possible interaction (entry call/accept) between two tasks in the program. For an example Petri net, based on the TIGs generated for the program in Figure 9.1, see Figure 9.2. In Figure 9.2, the places representing a task's states are displayed in a column under the task name and each transition, which represents an inter-task communication, is displayed between the two interacting tasks[4]. Places that represent potential termination points for a task are represented with double circles. For example, the caller2 task could potentially terminate at place 6 (by taking the false branch of the first conditional and the true branch of the second), place 7 (by taking the true branch of both conditionals), or place 8 (by taking the true branch of the first conditional and the false branch of the second). We use TIG-based Petri Nets (TPNs) because it has been shown that TPNs substantially reduce the size of the Petri net, thereby increasing the size of the programs that can be successfully analyzed [DCN95]. Although this example is small, in general Petri nets can be extremely complex and are not usually visualized.



[4]Because of the optimized representation used in a TIG, two transitions are used to represent the interaction between the accepter and caller2 tasks for the entry2 entry. Transition 2 represents the interaction occurring after caller2 takes the false branch in the first conditional and transition 3 represents the interaction occurring after caller2 takes the true branch in the first conditional.

Figure 9.2. Petri Net

A Petri net is called *safe* if each place in the Petri net can contain at most one token. Safety is a desirable property, because safe Petri nets are guaranteed to have a finite number of reachable states. It has been shown that TPNs are safe [Cha95b].

9.1.2 Reachability Graphs

Often, developers want to determine whether or not the concurrent program being analyzed could potentially enter a state in which a specified property is violated; for instance, is it possible for the program to enter a state in which it deadlocks. One method for answering such questions is to enumerate all possible program states and check the property at each state. A reachability graph can be used to represent the program state space.

A reachability graph for a Petri net consists of a set of nodes, $N = \{n_i\}$, and a set of arcs, $A = \{a_i\}$. Nodes in the reachability graph correspond to markings of the Petri net; the root node of the reachability graph corresponds to the initial marking (M0) of the Petri net. An arc goes from n_i to n_j if and only if the marking of the Petri net can change from n_i to n_j with the firing of a single transition. Although in actuality several interactions, represented by fired transitions, can take place concurrently, we can capture all possible execution sequences by firing a single transition at a time; we use this approach, because the resulting graph is greatly simplified. We note that only markings reachable from the initial marking by some sequential combination of transition firings are included in the reachability graph. It is helpful to observe that a marking of a Petri net simply represents the states of all the tasks being modeled by the Petri net; we therefore consider nodes in the reachability graph as states the program can reach when started from the initial program state. Figure 9.3 provides the reachability graph for the Petri net in Figure 9.2. Each node in the figure is annotated with the Petri net places that are marked in the corresponding program state.

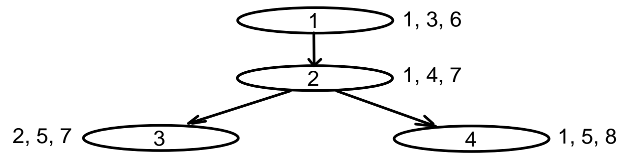


Figure 9.3. Reachability Graph

9.2 Improving Accuracy

In this section we examine an approach for improving the accuracy of static analysis without adding significantly to the cost of such analysis. To improve accuracy, we include additional program state information in the Petri net. Although we describe the approach in terms of TPNs, the approach is also applicable to other Petri net representations, such as those from [SC88]. The reachability graph generated from this enhanced Petri net representation provides a more accurate estimate of the program state space than the original reachability graph. Analysis of the revised reachability graph is thus more accurate, and the number of spurious results reported by the analysis should be less than or, in the worst case, the same as the number of spurious results reported for the original reachability graph. Since we propose a scenario where an analyst introduces additional information in response to discovering spurious results in the anomaly report, we would expect the number of such results to decrease. The increase in cost to gain this accuracy improvement includes the cost of incorporating the additional program state information in the Petri net and the cost of analyzing the resulting reachability graph.

Our approach can incorporate additional control flow or data flow information in the Petri net. The first technique, enforcing impossible pairs, retains information about past program states to eliminate some infeasible paths from consideration by the analysis; this technique may be suitable when conditionals are controlled by complicated conditions or when interactions between certain program statements are easily recognized by the analyst. The second technique, representing variable values, eliminates some infeasible paths by modeling variable values. This technique is suitable when conditionals are controlled by a small number of boolean or enumerated variables. We would expect an

analyst to select the technique that seems most appropriate or natural for the problem at hand.

For either technique, it is important that the enhanced Petri net continue to be an accurate representation of the program under analysis; in other words, adding the additional control or data information must not hide errors that would have been exposed through analysis of the original Petri net. Although not presented here, to ensure our techniques are error-preserving we have verified that the new Petri net is still an accurate representation of the program. Since the new Petri net is actually a more accurate representation than the original Petri net, it can be shown that the only program states removed from the reachability graph are those that are reached through infeasible paths.

9.2.1 Enforcing Impossible Pairs

Impossible pairs [GMO76] are pairs of program statements that can not both execute in the same execution of the program. In the mid-seventies, impossible pairs were recognized as an intuitive concept that developers could potentially exploit to improve the accuracy of their results. It was demonstrated in [GMO76], however, that deciding whether or not a path exists that does not include any impossible pairs is an NP-complete problem. Rather than explicitly solving the above problem to improve accuracy, we implicitly remove some infeasible paths from consideration by adding information about impossible pairs to the Petri net.

In this chapter, we use a less restrictive definition of impossible pairs than the one given in [GMO76], since we believe our definition more accurately captures the restriction that an analyst would want to include. In our definition, executing the first member of the impossible pair inhibits execution of the second member, but executing the second member of the impossible pair has no impact on the executability of the first

member[5]. In an extension of our technique, we also account for cases in which the second member of an impossible pair should only be disabled temporarily; this can occur if the condition that causes the second member to be disabled can subsequently change. Finally, we restrict our attention here to cases in which the impossible pair consists of two interaction (entry call or accept) statements, since the majority of concurrency analysis is concerned with communication events.

We observe that statements in an impossible pair are conceptually different from statements that Can't Happen Together (CHT) [MR93]. Impossible pairs identification is concerned with identifying invalid sequences of statements, whereas CHT analysis is concerned with identifying statements that can not execute concurrently.

The technique described below involves representing additional program state information to eliminate infeasible paths that contain both members of an impossible pair. For an example of when this technique is useful, consider the program in Figure 9.1, and assume for the moment that the conditions in the if statements are much more complicated than the value of a boolean variable. If the condition in the first conditional in the caller2 task evaluates to true, leading to the entry call on entry1 in the first conditional, the call on entry2 in the second conditional is impossible because the truth value of the condition does not change. Note that, similar to symbolic model checking, we could try to encode the possible values of the complicated condition in the Petri net. For general boolean expressions, however, the encoding of the condition in the Petri net could be quite large. Instead, we use information about this impossible pair to improve the accuracy of the Petri net and the corresponding reachability graph.

There are three distinct activities associated with enforcing impossible pairs: recognizing the impossible pairs in a program, recognizing which regions in the program

[5]Of course, using our definition an analyst could represent two statements a and b as an impossible pair as described in [GMO76] by specifying two impossible pairs, [a,b] and [b,a].

re-enable second members of the impossible pairs, and including information about the impossible pairs in the Petri net. Although sophisticated methods, such as symbolic evaluation [CR81], could be used to recognize impossible pairs and regions re-enabling them, we assume that these are relatively easy for an analyst to manually identify after examining the anomaly report. We would expect that after discovering several spurious results in the report, the analyst would introduce specific impossible pair information to improve the accuracy of the results. In any case, for this presentation we assume that some method has been used to recognize the impossible pairs and the regions re-enabling them, so our discussion below focuses on including information about these impossible pairs in our Petri net.

To simplify our explanation, we assume a single impossible pair in the program but note that the technique can be extended to multiple impossible pairs [Cha95b]. Also note that, using the same basic technique, more complicated flow constraints than impossible pairs could be incorporated given Petri net representations of those constraints.

To illustrate the ideas presented here, we modify the Petri net given in Figure 9.2. Transition 1, which corresponds to the accepter.entry1 statement in the caller2 task, is the first member of the impossible pair. Transitions 2 and 3, which correspond to the accepter.entry2 statement in the caller2 task, represent the second member of the impossible pair. The enhanced Petri net is shown in Figure 9.4.

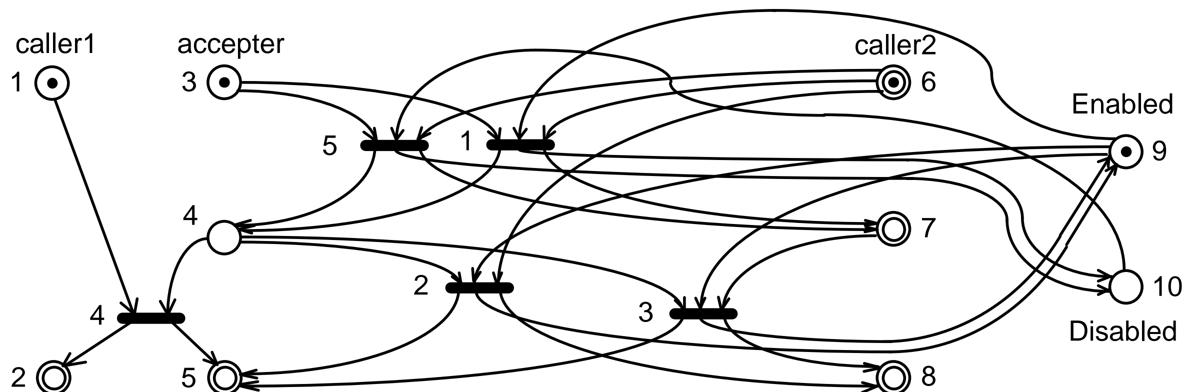


Figure 9.4. Petri Net With Impossible Pairs Represented

In general, to include impossible pair information in our Petri net we add two new places that control firing of the transitions corresponding to the second member of the impossible pair in the program, and also add duplicates of the transitions corresponding to the first member of the impossible pair. The first new place, called the Enabled place for the second member, is used to enable execution of the second member; the second new place, called the Disabled place for the second member, is used to inhibit execution of the second member. Because we restrict our attention here to impossible pairs of interaction statements, the first member and second member of the impossible pair are each represented by one or more transitions in the Petri net. We connect the Enabled place as an input to all transitions that correspond to the task statement for the second member, which ensures the statement can only execute when the Enabled place contains a token (transitions 2 and 3 in Figure 9.4) . We also connect the Enabled place as an output of these transitions, which lets the task statement execute multiple times. Since executing the first member of the impossible pair prohibits the second member from executing, we must ensure that firing the transition corresponding to the first member of the impossible pair results in an unmarked Enabled place and a marked Disabled place for the second member of the impossible pair. Because the second member may be enabled or disabled before executing the first member, we copy the transition corresponding to the first member, including all inputs and outputs of the transition. We then use the original transition (transition 1 in Figure 9.4) to change the second member from enabled to disabled when the first member is executed and the duplicate transition (transition 5 in Figure 9.4) to keep the second member disabled if it is already disabled when the first member is executed; we call these *disabling transitions*.

To ensure that the second member is enabled or disabled (but not both), we have connected the new places to the net such that exactly one of the Enabled place/Disabled place pair for the second member is marked at any given time. The Enabled place is initially marked, and the Disabled place is initially unmarked (see Figure 9.4).

In an extension of the technique described above, we also consider the possibility that the second member of an impossible pair should only be disabled temporarily. For example, if the first member of an impossible pair is contained within a loop and the condition is changed at the end of the loop, the second member of the impossible pair should be re-enabled at the end of the loop. Because the statement changing such a condition will typically not be an interaction statement, this statement is contained within the TIG region corresponding to a place in the Petri net; we call this region a *re-enabling region*, since it re-enables execution of a statement. To re-enable the second member, we modify transitions into the place corresponding to the re-enabling region. Because the statement to be re-enabled may be enabled or disabled before we reach the transition to be modified, we copy the transition, including all inputs and outputs of the transition. We then use the original transition to change the statement from disabled to enabled and the duplicate transition to keep the second member enabled if it is already enabled; we call these *re-enabling transitions*. In our example program the second member of the impossible pair is never re-enabled, so these transition modifications are not required for the Petri net in Figure 9.4.

In our example, the Petri net without impossible pair information is shown in Figure 9.2, and the corresponding reachability graph is shown in Figure 9.3. Node 4 in the reachability graph represents a deadlock of the caller1 task. The transition fired to enter this node, however, represents an interaction that is not possible, because the true branch is traversed in the first conditional in the caller2 task to reach node 2, and the condition is not changed before the second conditional. Therefore, an analysis result that reports deadlock for this program is a spurious result, since the program can not actually execute the path required to reach the deadlocked node. Using the technique for impossible pairs described above, we add impossible pairs information to the Petri net as shown in Figure 9.4; the corresponding reachability graph is shown in Figure 9.5. Note that in Figure 9.5 we have retained the reachability graph node numbering from Figure 9.3 to facilitate

comparison. For this example the spurious result has been removed by the additional information included, and thus analysis of the resulting graph can yield more accurate results.

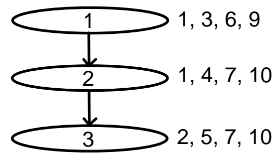


Figure 9.5. Reachability Graph With Impossible Pairs Represented

9.2.2 Representing Variable Values

When we include representation of impossible pairs information in our Petri net, we eliminate some infeasible paths from consideration by explicitly representing information about paths in the program execution. We can also implicitly eliminate some infeasible paths by representing the values of selected variables in the program. This technique is applicable when conditions in the program conditionals are relatively simple and include a small number of boolean or enumerated variables whose values can be statically determined in at least some regions of the program. As with the impossible pairs technique, we modify the Petri net to capture additional information about the program states. In this case, however, the state information is in the form of variable values. We can use this additional information to exclude interactions that are infeasible based on those values, thereby excluding some infeasible paths from our analysis.

For an example of when this technique is useful, consider again the program in Figure 9.1 and assume that BranchCond is set to true at the beginning of caller2. Thus, caller2 makes the entry call on entry1, but the entry call on entry2 is impossible, based on the value of BranchCond. If we modify the corresponding Petri net to include information about values of the variable BranchCond, we can improve the accuracy of the reachability analysis by eliminating consideration of the entry call on entry2.

There are four activities to be considered when we represent variable values in a Petri net: recognizing the interactions that are controlled by specific variable values, recognizing the regions that change the variable's value (and how they change it), building the representation for the variable, and connecting it to the existing Petri net. We believe that this is often straightforward in practice, particularly when a boolean

variable is used to control communication in the program. For these cases, an analyst should easily be able to identify such controlling variables and could specify those variables for inclusion in the Petri net. In this chapter, we assume the first two actions have been accomplished and focus on the actual representation and inclusion of the variable value information.

We represent a variable in the program for which we want to maintain value information with a *variable subnet*. This subnet contains two kinds of places: value places and operation places. The subnet includes a value place for each possible value of the variable, plus an "Unknown" place to account for those occasions on which we can not statically determine the variable's value. To simplify the presentation, we describe a variable subnet for a boolean variable. The variable subnet for a Boolean variable would have a "True" place, a "False" place, and an "Unknown" place. When the "Unknown" place is marked, the variable could be true or false; based on the connections described below, both possibilities are considered during generation of the reachability graph. The "Unknown" place is marked in the initial marking of the Petri net. The variable subnet also includes operation places for the valid operations on a variable of the given type; for example, the valid operations on a boolean variable are "Assign True", "Assign False", and "Not". For each operation, we connect the corresponding operation place to transitions between the appropriate value places. For example, the Boolean variable subnet contains a transition with "Assign True" and "False" as inputs and "True" as an output. The variable subnet is effectively a finite state machine for the variable, with transitions between the states (values) of the variable controlled by operations on the variable.

To make the resulting subnet safe, we modify the Petri net to ensure the operation places can never contain more than one token, using transformations similar to those described by Peterson [Pet81]. For every operation place for the variable, we add an operation prime place, yielding two places for each possible operation on the variable.

For each transition with an operation place as an output, we add the corresponding operation prime place as an input. For each transition with an operation place as an input, we add the corresponding operation prime place as an output. This transformation yields a safe subnet, with the additional property that only one of the operation place/operation prime place pair for a given operation can be marked at any given time. If none of the regions corresponding to marked places in the initial marking of the original Petri net modify the modeled variable, all operation prime places are marked in the initial marking of the Petri net; otherwise, the appropriate operation places are marked, with the corresponding operation prime places left unmarked. We also note that, since it is possible for the program to exit a region in which the value of a variable is statically determinable into a region in which the value is not statically determinable, we need to provide an "Assign Unknown" operation as well. The resulting variable subnet for a Boolean variable is as shown in Figure 9.6, but the subnet shown has not yet been connected to the Petri net for a program.

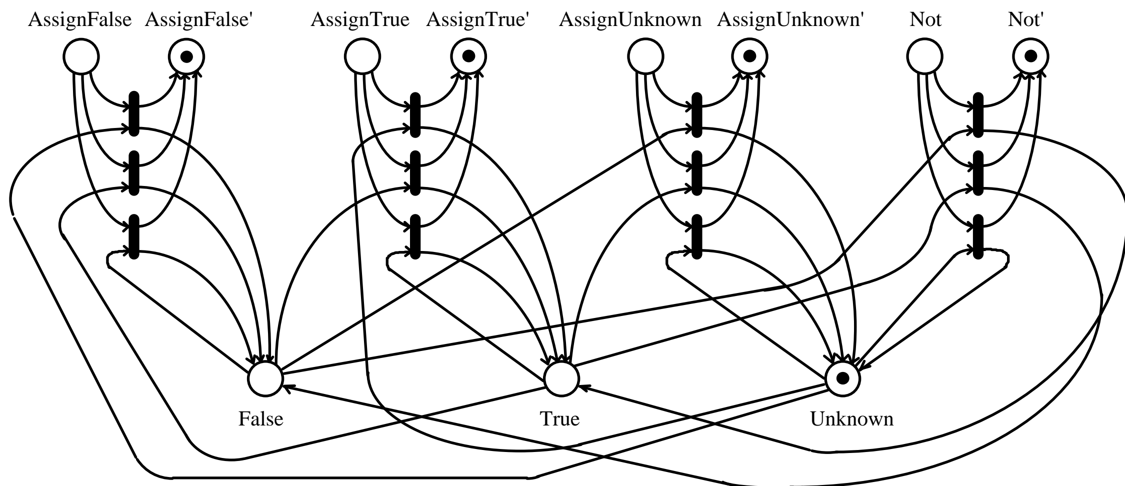


Figure 9.6. Boolean Variable Subnet

To use the additional information provided by the variable subnet, we need to connect the variable subnet to the Petri net. Figure 9.7 illustrates the revisions to the Petri net using the example shown in Figures 9.1 and 9.2. The variable subnet for the

BranchCond variable is abstracted to facilitate understanding. In Figure 9.7, a T, F, or U on an arc represents a connection to the True, False, or Unknown value place in the BranchCond Subnet. Also, connections between transitions and operation prime places are as described below, but are omitted from this figure for clarity.

A variable subnet is connected to the Petri net for a program in two cases: at transitions controlled by the variable and at transitions leading into or out of places corresponding to regions that modify the value of the variable. In the first case, a transition is controlled by a variable if the transition can only occur if the variable has a certain value. In this case, we copy the transition. The appropriate value place for the variable is connected as an input to the original transition (transitions 1, 2, and 3 in Figure 9.7), and the same value place is connected as an output of the transition to preserve the value of the variable. We add the Unknown value place as an input and output for the duplicated transition (transitions 5, 6, and 7 in Figure 9.7) to represent the fact that the interaction may be possible in the case where the variable's value is currently undetermined. In addition, we add all operation prime places for the variable as inputs and outputs for the original and duplicate transitions to ensure any required modifications to the variable have been completed before we use the variable's value. In this manner, we exclude all markings from the reachability graph that include firing this transition when the variable does not have the required value, thereby improving the accuracy of the analysis.

In the second case, to effect changes to the variable values, we need to account for regions from the program (places in the Petri net) in which the variable is changed (by assignment, for instance); we call these regions *modifying regions*. If we assign BranchCond the value true initially in the caller2 task then the corresponding place (place 6 in Figure 9.7) corresponds to a modifying region. For each of these regions, we add the appropriate operation place as an output and the corresponding operation prime place as an input of all transitions leading into the modifying region; this initiates modification of

the variable on entry into the modifying region. We also add the operation prime place as an input and output of all transitions exiting the modifying region; because the operation prime place will not be marked until the operation on the variable is completed, this ensures the modification is complete before the program exits the modifying region. Since the operation prime places have already been added to transitions 1 and 5 as described above, no further changes are required in Figure 9.7.

Note that a single region can potentially modify a given variable in several different ways. To simplify the description we assume a simpler model here, in which a single region modifies a given variable in one specific way. Note that more complicated modeling can be used to handle the more general case. Also note that since the region represented by place 6 in the Petri net would contain `BranchCond := true`, in our initial marking the AssignTrue place is marked (and the AssignTrue' place is unmarked).

Using a variable subnet as described above yields the Petri net shown in Figure 9.7. The corresponding reachability graph is shown in Figure 9.8, where the reachability graph nodes are annotated with the marked Petri net places as well as the marked value, operation, and operation prime places in the BranchCond Subnet. Again we see that the spurious result is no longer reported.

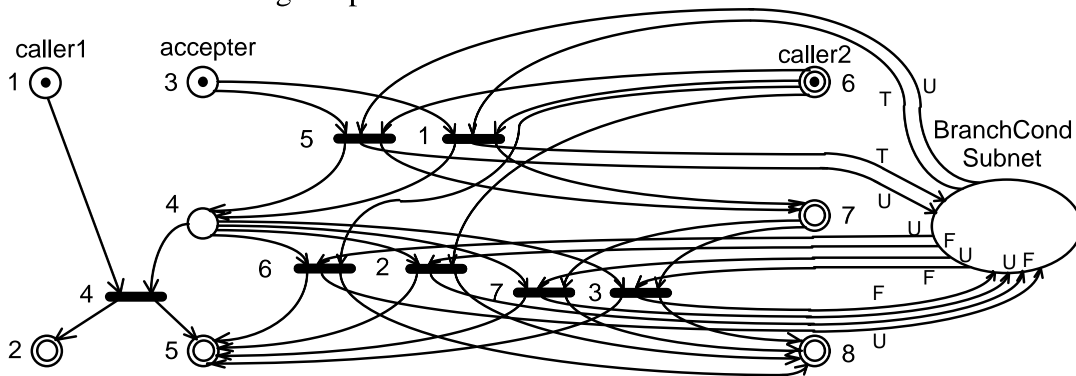


Figure 9.7. Petri Net With Variable Subnet Added

Information about variable values could also be incorporated using an FSM, with states of the FSM representing variable values and transitions in the FSM representing operations on the variable. While the FSM would certainly be easier to understand than

Figure 9.6, the difficulty comes when incorporating the FSM into the model. An FSM can not be "connected" to the Petri net as our variable subnets are, so the FSM would need to be used during reachability graph generation, potentially slowing down the generation process significantly. Representing variables with variable subnets provides the same accuracy improvements as would be provided with FSMs, while retaining a standard Petri net as the program model.

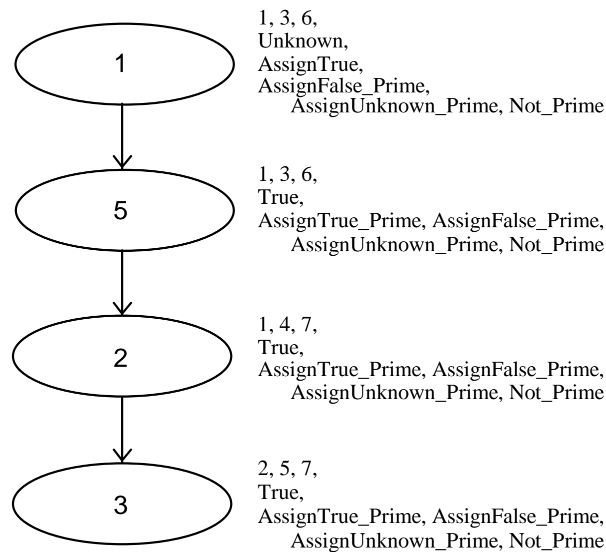


Figure 9.8. Reachability Graph Using Variable Subnet

9.2.3 Choosing Between the Two Techniques

The two techniques described above give the analyst flexibility when determining what kind of additional information to include to improve analysis accuracy. In general, we expect the analyst to choose whichever technique appears more natural given the program being analyzed and the property of interest.

The impossible pairs technique seems particularly attractive when static information about the impossible pairs in the program is readily available and transitions correspond to members of a single impossible pair. If the control flow decisions in the program are complicated, the impossible pairs technique may be more suitable than the variable values technique. The impossible pairs technique will tend to be expensive for programs for

which the Petri net contains transitions that affect multiple members of impossible pairs, since the number of these transitions grows exponentially in the number of impossible pairs affected.

In the variable values technique, efficient algorithms for recognizing the regions that affect a variable's value are available. An analyst may also be able to easily identify those variables that are used in the program to control communications. If the control flow decisions on those variables are not extremely complicated, recognizing the transitions controlled by the variable values and making the appropriate connections is relatively straightforward. The additional information added to the Petri net is based on the variable type, so the variable subnet for a variable with relatively few values (such as a boolean variable), used in relatively few locations, does not increase the Petri net size significantly. Limitations of this technique include the requirement to be able to statically determine variable values to gain accuracy improvement, the difficulties determining the proper connections to account for complicated conditions, and the rapid growth of the size of the variable subnet as the number of possible values of the represented variable grows.

9.3 Empirical Results

We have run experiments on a small set of programs to gather information about how the application of our approach affects the sizes of the Petri nets and reachability graphs for these examples. We hypothesize that our accuracy-improving approach can improve analysis accuracy without significantly impacting performance.

In each of the techniques presented, the size of the Petri net is increased by the places and transitions added to model the additional semantic information. On one hand, we expect the size of the reachability graph to grow as the size of the Petri net grows, since the upper bound on the size of the reachability graph is exponential in the number of Petri net places. On the other hand, we would expect the additional modeling in the Petri net to remove some infeasible paths from consideration, thereby reducing the size of the

reachability graph. We perform the experiments to acquire preliminary indications of which scenario is more common and also to gain experience applying the approach.

Whenever the approach is applied, the resulting reachability graph more accurately represents the program state space. However, this does not necessarily guarantee that the number of spurious results in the anomaly report will be reduced. For instance, if the states removed from the reachability graph are independent of the property being checked, the number of spurious results in the anomaly report will stay the same. For that reason, we consider our accuracy improvements as improvements in the reachability graph as a representation of the program state space, rather than as reductions in the number of spurious results in the anomaly report. While we expect that improving the accuracy of the reachability graph will commonly reduce the number of spurious results, whether or not this occurs in practice depends on the property being checked.

To perform the experiments below we modified an existing tool set. Tools to convert an Ada program to a TIG and a set of TIGs to a Petri net were already available. We developed a general tool to generate the reachability graph from a Petri net, and also built several specialized tools to include impossible pair information and variable subnets in the Petri net.

For the experiments described here, we used various sizes of the readers/writers problem and the gas station problem. The notation rwXY indicates an instance of the readers/writers problem with X readers and Y writers. The code for readers/writers programs is fairly standard, with a Boolean variable WriterPresent used to track the presence of a writer. The notation gasXY indicates an instance of the standard gas station problem [HL85] with X customers and Y pumps.

For the impossible pair technique, identifying the impossible pairs in the program to be analyzed is done manually. Once we have identified which regions correspond to impossible pairs, we provide this information to a tool that scans the transitions in the Petri net and automatically modifies the transitions as described in the previous section.

When we use the variable subnet technique, we provide the name of the variable to be modeled to the Petri net toolset. The toolset then automatically generates a variable subnet with the appropriate value and operation places. Currently, we only automatically build Boolean variable subnets. We then take the resulting variable subnet and manually connect it to the original Petri net by recognizing interactions that are controlled by the variable value and also identifying regions in which an operation is performed on the variable. This activity could be automated by scanning for the variable name in branches and select guards and by collecting information about operations on the variable for each region.

The effects of using these techniques for the sample programs can be found in Table 9.1. In the table, NA means that no additional information is included in the Petri net for the program. Imp specifies a Petri net that includes information about impossible pairs and Var specifies a Petri net that includes one or more variable subnets.

Table 9.1. Effects of Approach on Petri Nets and Reachability Graphs

		Petri Net		Reachability Graph	
Program	Refinement	Places	Transitions	Nodes	Arcs
rw21	NA	17	48	41	119
	Imp	25	183	31	71
	Var	28	105	52	94
rw22	NA	20	66	175	692
	Imp	28	306	98	276
	Var	31	138	166	348
rw23	NA	23	84	609	3,031
	Imp	31	429	248	794
	Var	34	171	426	978
rw32	NA	23	81	579	2,884
	Imp	31	336	308	1,097
	Var	34	168	502	1,295
rw25	NA	29	120	6,229	43,571
	Imp	37	675	1,320	4,888
	Var	40	237	2,330	5,908
rw52	NA	29	111	5,811	40,660
	Imp	33	638	2,972	14,955
	Var	40	228	4,678	16,665
gas31	NA	39	75	493	987
	Imp	45	111	931	1,773
	Var	87	224	559	885
gas51	NA	59	163	9,746	26,785
	Imp	64	463	22,841	57,655

For the Imp version of the Petri net for readers/writers problems, we model the impossible pairs resulting from whether or not a writer is present. These pairs were easy to recognize given the simple guards in the control task. Including this information improves the accuracy of the analysis by eliminating consideration of some infeasible paths through the program and reduces the size of the reachability graph as well.

For the Imp version of the gas station problems, we use impossible pairs to reflect the fact that if a customer enters an empty pump queue, then that customer gets their change before any other customer. Including information about impossible pairs in gas31 and gas51 yields reachability graphs with approximately twice as many nodes and arcs as the original reachability graph.

Including impossible pairs information in the Petri net can cause an increase in the reachability graph size because we encode not just the current program state, but also information about the path leading to that state. For example, consider the state in which customer 1 and customer 2 have both pre-paid the operator. Without impossible pairs information, this state is represented by a single node in the reachability graph. When we include impossible pairs information, the reachability graph contains one node for this state in which customer 1 entered the (empty) queue first, one node in which customer 2 entered the (empty) queue first, and one state in which neither entered an empty queue. In such cases, the improvement in accuracy comes at the cost of a larger reachability graph to be analyzed.

For the Var version of readers/writers, we model the WriterPresent variable that is included in the guards of the main select statement. Selecting this variable to be modeled and recognizing the appropriate connection points for the variable subnet were straightforward because of the basic operations on the variable and the simplicity of the guards containing the variable. We observe that, for instances of readers/writers larger than rw21, the technique yields two benefits: it improves the accuracy of the analysis by eliminating consideration of some infeasible paths through the program and it reduces the

size of the reachability graph. For rw21, this technique increases the size of the reachability graph. This occurs because of the possible interleavings of firing transitions that change the variable value and firing transitions that are independent of the variable value. As the problem is scaled, the affect of these interleavings seems to decrease, and we see reduction in the reachability graph size instead of growth.

For the Var version of gas31, we implement a variable subnet for each element of the customer queue, in addition to the counter for the number of active customers. Because our tools don't currently automatically build subnets for enumerated or subrange types, we manually built the subnets for this version. Modeling the customer queue and number of active customers yields a slight increase in the number of reachability graph nodes, so simply checking for a property at each node would take somewhat longer. In addition, we note that manually building the variable subnets was tedious. Although building the subnet for each queue element is straightforward, the difficulty comes in recognizing where the gas31 code moves the queue forward and representing that movement with the subnets. In any case, the analysis is more accurate, since using the variable subnets ensures that change is always given to the correct customer. Developing the model of the customer queue was sufficiently time-consuming that we did not attempt this for the gas51 program.

For the readers/writers problem, the impossible pairs and variable value techniques implicitly model the "same" information (the value of the WriterPresent variable). It is therefore valid to directly compare the sizes of the resulting reachability graphs (since they have the same accuracy), and to note that the impossible pairs technique is more effective at reducing the size of the graph. On the other hand, the Imp Petri nets contain many more transitions than the Var Petri nets for this problem, so it may take longer to actually generate the (smaller) Imp reachability graphs. With both techniques, the accuracy of the reachability graph is improved; the reduction in size is a beneficial side effect.

For the gas station problem, our impossible pairs results are not comparable to the Var version, since we are not capturing the same information in our Petri net. The Var version captures a significant amount of state information for only a slight increase in reachability graph size, but manually adding the required variable value modeling was difficult. The Imp version captures less information than the Var version, and yields a large increase in reachability graph size, but including the modeling was straightforward.

Table 9.2 lists several properties of each program considered. Entries is the number of unique entries in the program and Entry Calls is the total number of calls on those entries. Variables provides the number of variables modeled in the Var version of the Petri net, with the number of possible variable values (including unknown) following in parentheses. For instance, for the Var version of the gas31 Petri net, we model 3 variables with 4 possible values and 1 variable with 5 possible values. Impossible Pairs provides the number of impossible pairs modeled in the Imp version of the Petri net. For the readers/writers programs, the numbers of variables and impossible pairs modeled stay constant as the problem is scaled. This occurs because the additional modeling is applied to the control task, which does not change as the problem is scaled. For the gas station problems, the number of impossible pairs modeled grows as the problem is scaled because the modeling is applied in the operator task, which grows as the problem size grows.

Table 9.2. Program Properties

Program	Entries	Entry Calls	Variables	Impossible Pairs
rw21	4	6	1 (3)	7
rw22	4	8	1 (3)	7
rw23	4	10	1 (3)	7
rw32	4	10	1 (3)	7
rw25	4	14	1 (3)	7
rw52	4	14	1 (3)	7
gas31	10	17	3 (4), 1 (5)	6
gas51	14	27	-	20

9.4 Conclusions

Static analysis can be used to answer questions about properties of concurrent programs, although often with the inclusion of spurious results. We have identified an approach that can be used to improve the accuracy of Petri net-based analysis of concurrent programs. In several cases that we examined, the approach reduced the size of the reachability graph for the system as well. The impossible pairs technique retains additional program state information in the form of the impossible pair transitions that are currently enabled and disabled, and the variable subnet technique retains additional program state information in the form of the current values of selected variables.

The cost of using the above techniques can vary considerably from program to program. To effectively use variable subnets, we must first recognize which variables affect the control flow of the program and identify the regions in which those variables are modified. We must also determine how the represented values should be connected to the transitions of the Petri net to accurately reflect how the values influence the interactions of the program. The difficulty of doing this ranges from very easy (for control flow decisions based on a Boolean variable's value only, for example) to very difficult (for control flow decisions containing complicated conditions). Alternatively, we can sometimes account for complicated conditions by including impossible pairs information instead. The complexity of adding the information for the impossible pairs is linear in the number of original transitions in the Petri net; the difficulty comes in recognizing the regions of the program that represent impossible pairs. Ultimately, the decision about which technique to use will fall on the analyst. For some programs, the impossible pairs may be easily recognized by the analyst, whereas for other programs, representing key variables that control communications in the program may seem more straightforward.

In several of the programs examined, the reachability graph size or complexity was reduced as a side effect of the improved accuracy. Static analysis models generally include infeasible as well as feasible paths through the program; the state space which

needs to be searched for the property is therefore larger than the actual possible state space of the program. Because our goal was to improve accuracy by eliminating impossible program states from the reachability graph, it is reasonable to expect a smaller reachability graph to result. On the other hand, in some cases our modeling of the additional state information leads to larger graphs, because we add possible interleavings between activities on our variable subnets or Enabled/Disabled impossible pair places and the original Petri net. In all cases, the generated reachability graph represents more accurately the possible states of the program because of the additional information modeled.

We have examined how to incorporate accuracy-improving semantic information into Petri nets. It is not as easy to modify the semantics of other internal representations that are commonly used for analysis, such as control flow graphs, abstract syntax trees, and program dependency graphs. A complementary and somewhat similar approach is explored in [DC94], but instead of modifying the internal representation, the approach incorporates the additional semantic constraints in the analysis algorithms. Similarly, information about impossible pairs or variable values could be incorporated in the reachability graph generation algorithm rather than in the Petri net representation of the program. It is not clear how this would affect the size of the resulting reachability graph, but the added complexity in the algorithm might lead to a significant increase in reachability graph generation time. It is too early to determine when one approach might be superior to the other.

Because of various limitations, we have only demonstrated the viability of our approach on a small sample of programs. It is doubtful, however, that these programs are representative of the population of "real" concurrent programs. To more accurately quantify how well these techniques work in general, more experiments need to be run on a larger sample of programs. Our future plans include performing a series of experiments using this approach on a wider range of program sizes and complexities.

For the programs examined here, we have manually detected variables and impossible pairs to model, then added them to the Petri net using partially automated tools. More support could be provided to the analyst through automatic recognition of variables that control interaction patterns in the program; these variables could then be automatically included in the Petri net or recommended as useful variables to model. Automatically detecting impossible pairs in the program may not be feasible except in simple cases, but further automating the process of modeling variables and impossible pairs is a potential area for future research.

It would also be interesting to make the tool interactive to determine the effects on analysis accuracy of representing other user-supplied information. If the analysis yields spurious results that are not easily eliminated using the above techniques, it may be possible to include additional information from the user to refine the Petri net to improve accuracy. Other constraints on the control flow, such as sequences of certain statements that can never occur or must always occur, can be modeled with subnets and attached appropriately. More generally, any constraints that can be expressed with a subnet could be used to improve the accuracy of analysis results, as long as the analyst or an enhanced tool could determine how to attach the subnet appropriately. To ensure conservativeness, the modifications would need to be error-preserving, at least for the property being checked.

The results above support our hypothesis that modeling specific kinds of program state information in the Petri net can lead to cost-effective improvements in the accuracy of the corresponding reachability graph, and for some programs reduce the size of the reachable state space as well. Further work needs to be done to more accurately quantify the benefits of these techniques, and the tools should be made more robust to allow additional investigation of these and other techniques for improving static analysis accuracy.

CHAPTER 10

CONCLUSION

Static concurrency analysis techniques can be used to check that the behavior of concurrent systems meet specified requirements. A variety of these methods have been proposed, including reachability analysis, symbolic model checking, integer programming, and data flow analysis. Given the variety of tools available, analysts need assistance when selecting which tools to use for a specific program and property. Empirical tool comparisons can provide useful insight into which tool would be most suitable for a given program and property.

The main contribution of this dissertation is the methodology we have developed to gather experimental data and analyze that data. We believe that this methodology can be used to conduct sound empirical comparisons of concurrency analysis tools and to provide valuable assistance to analysts selecting a tool for analysis of a concurrent system. In describing our methodology, we identify many of the concerns and tradeoffs that must be considered. We believe that the description will be informative to those considering similar such investigations.

To ensure that an empirical comparison is fair, a careful comparison methodology must be employed. For the comparison to be fair, the tools should be used on the same input domain of programs and properties and the methodology should not introduce bias against one or more of the tools. A valid comparison methodology should therefore ensure that each tool is analyzing the same program and property, or recognize and identify cases in which this is not possible. Such a methodology should also try to minimize bias introduced by the methodology.

To ensure each tool analyzes the same program, the methodology presented above uses an Ada program as a canonical model of the concurrent system and carefully translates this model to the inputs for each of the tools. This translation process has been

carefully developed and automated, but because of differences in the semantics of the analysis tools it may not be possible to force them to analyze identical programs. Instead, we view our process as (at least close to) the best we can do for this set of analysis tools. To ensure each tool checks the same property, the properties of interest are carefully created. This task can be difficult given the variety of property specification formalisms. Because we manually convince ourselves that the properties are the same, there is always some question whether or not we have specified identical properties.

The program and property translations in our methodology can inadvertently introduce bias against one or more of the tools. This bias can be introduced by the form of the inputs generated for each tool, by the configuration in which each tool is run, by the form of the property specification, and by other unknown factors. The methodology attempts to recognize possible areas of bias and, when possible, executes analysis runs to ensure such bias is not introduced by the methodology.

We know that our methodology introduces some bias through our selection of program sizes. Specifically, the sizes for a specific program and property are selected based on the performance of the tool that does worst (in terms of analysis time and failure) on that program and property. In many cases, this restricts some of the analysis tools to an input domain that is much smaller than they could actually analyze. The positive effect of this choice is that the comparison is performed for the same input domain of programs, sizes, and properties. The negative effect is that some of the tools are forced to analyze programs in only a small portion of their domain of applicability. An alternative would be to select different sizes for each tool, based on the point at which that tool fails. This would potentially give a clearer picture of each tool's performance, especially in terms of failures, but would preclude direct comparison of analysis times and failure percentages because of the differences in the input domain.

The choice of what to measure for analysis time for the comparison is a difficult one. Using each tool's native input as the starting point for the time measurement seems the

fairest, but may not give a true picture of analysis cost, at least for Ada programs, given the translations required to generate the native input. Starting the analysis time measurement with the input of the Ada program may give better insight into the true cost of the analyses, but this time also includes potential inefficiencies contained in our translation tools.

We also note that the measured analysis times ignore a very interesting, and almost always significant, time factor - the amount of time it takes an analyst to specify the property of interest. Our informal observations below indicate problems that we encountered with each of the tools specifying the properties. While many of these problems are probably caused by our inexperience with the tools and their specification formalisms, we believe that the property specification time would be non-trivial even for experienced users. Developing an experiment to take this time into account, however, would be a difficult undertaking, because many factors involving human behavior (i.e., analysis experience, training effects, and so on) would need to be accounted for in the experimental design.

A second contribution of this dissertation is the application of the methodology to conduct an empirical comparison of six concurrency analysis tools. As we applied this methodology, we gained valuable experience using each of the tools in the experiment. Because we have the perspective of a user, rather than a developer, of these tools, we believe these experiences provide interesting insights about the tools.

One of the key differences between the tools (from the user's perspective) is whether they are *state-based* or *event-based*. We classify a tool as state-based if properties are typically specified in terms of states of the program being analyzed; SPIN, SPIN+PO, TRACC, and SMV are state-based tools. We classify a tool as event-based if properties are typically specified in terms of events that occur during execution of the program being analyzed; INCA and FLAVERS are event-based tools. We make a similar distinction between state properties and path properties. State properties can be checked by

considering each state of the system in isolation. Freedom from deadlock and no_w1w2 (from readers/writers) are examples of state properties. Path properties require consideration of paths through the program, often in terms of events along those paths. The no_r1w property (from readers/writers) is an example of a path property. We have found that using state-based tools to check path properties can be somewhat difficult. For example, in many cases we found it necessary to add additional variables to the system specification being analyzed by the state-based tools to let us recognize the events of interest for the property. We did not seem to experience the same difficulty using event-based tools to check state properties because it was usually possible to identify the events leading to the states of interest and to formulate the property in terms of those events. Of course, we had more experience using event-based tools before conducting the experiment described here, so this might simply be a result of our prior experience. We provide more specific comments about the tools below.

SPIN provides two different methods for specifying properties. Never claims essentially represent a Finite State Automaton representation of the property, while assertions are embedded in the program being analyzed. Our biggest difficulty with SPIN was caused by the fact that, even with the processes in the program specified as FSAs, we do not get a "true" transition on communication events. SPIN evaluates the guards for the alternatives (typically the guards are communication events) in one step of the evaluation, but does not execute the action associated with the selected alternative until some later step. This was particularly problematic when we wanted to check mutual exclusion properties. Consider the case where one user of a resource releases that resource (through a communication), but is not transitioned to its new state because SPIN has not yet executed the action associated with that communication. If a second user starts using the resource, examination of the states of the processes in the system indicates that both users are using the resource (i.e., mutual exclusion is violated). The evaluation of an alternative in one step and execution of the action for that alternative in a later step also

made it difficult to specify properties as never claims. We were able to work around this characteristic with careful specification of the never claim or embedding of assertions, but the resulting properties were often less intuitive than those we originally formulated. Because it is based on SPIN, SPIN+PO has this same characteristic.

Because we specify the system for SMV using the transition relation of the system, we were able to more easily identify events of interest than with the other state-based tools. Because our events of interest are often communications in the program, which are represented by transitions in the transition relation, we can identify these events by using additional variables to identify when certain transitions occur. Adding these variables to potentially large transition relations was initially a painful, manual process, but we quickly developed a tool that automatically makes most of the changes. We had more difficulty when we tried to avoid adding additional variables by specifying the properties as alternate CTL formulae instead, because these formulae are in terms of states rather than events. On the other hand, because the transition relation provides true state transitions on the communications, we did not experience the same problems we had with SPIN.

We included TRACC as an additional reachability analysis tool for comparison, but its performance, in terms of both analysis time and accuracy, indicates that it is not a viable tool for static concurrency analysis. In addition, a special program must be written to check each property, an effort we would not expect an analyst to undertake each time a new property of interest is developed.

INCA is one of the two event-based tool in the experiment, and identification of the communications in the program is automatically provided by the tool. We initially had some difficulty determining when multiple intervals were required to check properties, but discussions with the developers of the tool clarified this issue. We also had some difficulty determining the semantics of certain query constructs (:ends-with, for instance),

but view this as a documentation problem rather than a weakness of the tool. Finally, we found the Lisp syntax of the queries somewhat inconvenient.

FLAVERS is also an event-based tool, and identification of rendezvous accepts is automatically provided. Because the tool does not identify specific communications (accepts of entry calls from two different tasks are marked with the same event), we occasionally had to add annotations to capture the events of interest. This characteristic also led us to manually add annotations to check the mutual exclusion properties, for which we encountered a problem similar to that for SPIN. FLAVERS annotations can only be specified to occur just before or just after a communication, while we wanted the annotations to be exactly at certain communications. Our workaround for this was similar to the one used for SPIN. Properties in FLAVERS are specified as Quantified Regular Expressions (QREs). Given our familiarity with regular expressions, we found this an intuitive way to specify properties. We note, however, that we developed a process in which we created a QRE for our property of interest and then converted it to an FSA to confirm that it specified the property we intended. On several occasions, this process indicated that our property was not quite specified correctly, so we made modifications to the QRE one or more times before achieving the property we wanted to check. This experience implies that, for FLAVERS, FSAs may be a more useful property specification formalism than QREs.

To allow the use of statistical tests to check for bias and to gain confidence in the analysis times collected, we ran each analysis case five times. In an effort to remove caching effects from these runs, we randomized the order in which the analysis cases were run. While we believe this approach is reasonable, there are some practical difficulties with it. For example, a change to one tool's input requires that the entire set of analysis cases be rerun, since the analysis cases are randomized across multiple tools. An alternative would be to somehow clear the cache before each analysis case. We would still run each analysis case five times to allow the statistical testing for bias, but would no

longer need to randomize the order of those runs. We could then run the analysis cases for each tool as they became available, and would no longer have to rerun the entire set of cases when one tool's input changed. Because all of these tools are regularly updated, an additional benefit of using the new approach would be that we could run the analysis cases for a new version of one of the tools without having to rerun the analysis cases for the other tools as well.

A third contribution of this dissertation is the demonstration of careful statistical analysis to check for bias and to develop predictive models for analysis time, failures, and spurious results. Unfortunately, the linear regression models for analysis time did not generally capture much of the variance in the experimental data, so they are not likely to provide much predictive power for real programs. For some of the metrics, we occasionally identified additional linear components that were not accounted for by the regression model. It is possible that adding additional cross-product terms to the model or using more sophisticated regression techniques will yield better predictive models, but we stopped our analysis at identification of these problems. It is also possible that the metrics we have chosen do not capture those characteristics that actually do affect analysis time, and that a different set of metrics would yield better predictive models. Of course, it may also be the case that there does not exist a set of metrics that will yield good predictive models, but we believe additional experimentation should be performed before we reach this conclusion. We note that the results of the logistic regressions for failure and spurious result predictive models yielded much better results than the linear regressions, though these models still need to be validated on real programs.

We have also noticed that minor changes in the Ada source can have significant effects on analysis performance. For example, the Ada program we used for readers/writers contained several unguarded select alternatives. INCA yielded spurious results when checking for freedom from deadlock because of this. However, guards can be added to these alternatives without changing the semantics of the program. When we

included these guards and modeled both the **Writer** and **Readers** variables we were able to eliminate the spurious results from INCA. Thus, even differences in programming style in the Ada program can lead to variations in analysis tool performance. These style differences do not affect the values of the metrics we use, so the variations in tool performance caused by these style differences will not be captured in the predictive models built using our metrics.

To be most useful, the analysis tools need to be applicable to programs of realistic size, containing realistic communication structures. In almost all cases, including the experiment conducted for this dissertation, concurrency analysis tools have been demonstrated using programs from the concurrency analysis literature. It is not clear that these academic programs are representative of concurrent programs in general. Most tasks in these programs are relatively small, for instance, and the program constructs used in these programs are relatively simple. A fourth contribution of the work presented here is the preliminary examination of several "real" programs. The examination includes quantification of the communication structure of the programs, discussion of the program constructs used in the programs, and observations about program characteristics that are likely to affect the applicability of static concurrency analysis tools to these programs.

Performing fair experimental comparisons of concurrency analysis tools is difficult given the variety of tool semantics and property specification formalisms. We believe that the methodology presented in this dissertation can be used as a basis for such comparisons. The methodology attempts to ensure the tools are analyzing the same programs and properties, and it provides a method for statistically checking various assumptions about biases that may be introduced by the methodology. The methodology has been developed so it can be used on real programs as well as those from the concurrency analysis literature, so it is applicable to case studies as well as experiments. Through continued use of this methodology, we should be able to conduct additional experiments that broaden our understanding of various static concurrency analysis

techniques and provide analysts with useful insights about which tools would be most appropriate for specific programs and properties of interest.

APPENDIX

PREDICTIVE MODELS

This appendix provides the equations for the predictive models we selected to predict analysis time, failures, and spurious results for each of the tools.

A.1 Analysis Time Predictive Models

This section provides the equations for the predictive models we selected to predict analysis time. The equation for SPIN checking deadlock is

Analysis Time = -5.285194 + 2.58366E-06*Cnd' + 0.085944*MaxTRANS.

The equation for SPIN using never claims to check other properties is

Analysis Time = 1.99480E-04*Alpha' + 2.52515E-06*Cnd' + 0.104129*MaxTRANS - 54.090492*Query Events + 140.704582.

The equation for SPIN using assertions to check other properties is

Analysis Time = 28.023946 + 4.470356*T - 2.808051*MaxC + 2.809042*Beta + 2.07503E-06*Cnd' + 0.156220*MaxTRANS - 83.595626*Query Events + 119.416482*Query Intervals.

The equation for SPIN+PO checking deadlock is

Analysis Time = 5.561660 + 0.957373*T - 6.444770*C - 0.290362*MaxC + 1.17370E-04*Alpha' + 2.579100*Beta - 1.29418E-06*Cnd' - 3.17055E-09*Cif - 0.044143*N + 0.031457*MaxTRANS - 2.43181E-21*WFSA + 0.245484*Vars.

The equation for SPIN+PO checking other properties is

Analysis Time = 122.064648 +7.021475*T - 13.844693*C - 6.028394*MaxC + 1.86244E-04*Alpha' + 7.899061*Beta - 3.16718E-06*Cnd' - 9.83386E-08*Cif + 1.893762*N + 0.033242*MaxTRANS + 1.57110E-19*WFSA + 25.766125*Vars -11.576069*QRE Alphabet + 1.910704*QRE States - 0.761156*Query Events + 134.807495*Query Intervals - 49.258551*Never States - 6.273414*Assertions.

The equation for TRACC checking deadlock is

Analysis Time = 0.397474 + 0.541131*T + 0.135893*MaxC + 0.008972*Alpha' + 0.181751*N + 4.58857E-21*WFSA.

The equation for TRACC checking other properties is

Analysis Time = 18.987413 - 1.852495*T - 3.345392*C - 0.007447*Cnd' - 0.043212*Cif + 0.216304*WFSA.

The equation for SMV checking deadlock is

Analysis Time = -11.131395 + 10.712972*T - 41.659575*C - 4.234177*MaxC + 2.65252E-06*Alpha' + 9.744026*Beta - 3.75022E-06*Cnd' - 1.08421E-07*Cif + 9.161629*N - 0.096725*MaxTRANS + 4.89367E-20*WFSA + 30.185195*Vars

The equation for SMV checking other properties is

Analysis Time = -110.172226 + 6.936632*T - 13.241819*C - 4.913227*MaxC + 8.70360E-07*Alpha' + 4.326865*Beta - 1.29216E-06*Cnd' - 7.25095E-08*Cif + 4.075190*N - 0.040636*MaxTRANS + 9.98048E-20*WFSA - 4.940134*Vars + 1.995971*QRE Alphabet + 19.152749*QRE States - 9.523060*Query Events + 157.192641*Query Intervals - 41.925466*Never States - 1.473770*Assertions

The equation for INCA checking deadlock is

Analysis Time = 4.592838 -7.889957*C + 5.806953*N.

The equation for INCA checking other properties is

Analysis Time = 21.303073 + 7.24986E-08*Cnd' + 0.017519*MaxTRANS - 6.985987*QRE States - 2.814540*Query Events + 3.342872*Never States + 0.577033*Assertions.

The equation for FLAVERS checking other properties is

Analysis Time = -343.063823 + 130.849429*C - 48.007135*MaxC + 7.39592E-05*Alpha' + 7.344377*Beta + 8.29521E-07*Cif + 5.960648*N - 2.33986E-18*WFSA + 16.000030*QRE Alphabet

A.2 Failure Predictive Models

This section provides the equations for the predictive models we selected to predict failures. The form of the predictive equations is $\Pr(Failure) = \frac{e^{g(x)}}{1 + e^{g(x)}}$; for readability, we provide equations for g(x) below. The equation for SPIN checking deadlock is

g(x) = -4.0399 + 1.26E-07*Alpha' + 7.95E-09*Cif + 0.0830*N.

The equation for SPIN using never claims to check other properties is

g(x) = -4.4229 + 0.1321*T - 0.3799*C - 0.0721*MaxC + 0.0012*Alpha' + 0.0805*Beta - 0.0006*Cnd' + 1.92E-09*Cif + 0.0397*N + 0.0090*MaxTRANS - 2.6E-21*WFSA + 0.0102*Vars.

The equation for SPIN using assertions to check other properties is

g(x) = 6.6110 + 0.2749*T - 1.4242*C - 0.1312*MaxC + 5.92E-07*Alpha' - 1.8E-08*Cnd' + 0.3675*N - 0.2309*Vars - 2.8209*Never States.

The equation for SPIN+PO checking deadlock is

g(x) = -6.7785 + 0.0710*Beta + 0.0459*N + 0.0007*MaxTRANS.

The equation for SPIN+PO checking other properties is

g(x) = 7.3872 - 1.4322*C + 7.34E-07*Alpha' + 0.2331*Beta + 0.2134*N - 1.3656*QRE Alphabet - 0.8042*Query Events.

The equation for TRACC checking deadlock is

g(x) = -2.0140 + 0.1147*Beta.

Because we had indications that all the failure models we built for TRACC checking other

properties were overfit to the data, we do not provide an equation for TRACC checking

other properties. The equation for SMV checking deadlock is

g(x) = -14.5847 + 0.4086*T + 3.6889*C - 2.0163*MaxC - 0.0001*Alpha' + 0.1014*Beta - 4.1E-09*Cnd' - 6.7E-07*Cif + 0.3898*N + 8.96E-05*MaxTRANS + 1.58E-19*WFSA - 0.1367*Vars.

The equation for SMV checking other properties is

g(x) = -4.2911 + 0.0536*T + 0.0006*MaxTRANS.

The equation for INCA checking deadlock is

g(x) = -11.0889 + 0.0527*N.

INCA did not fail on any of the cases for which it was used to check properties other than deadlock, so we do not provide an equation for INCA checking other properties. Because we had indications that all the failure models we built for FLAVERS checking other properties were overfit to the data, we do not provide an equation for FLAVERS checking other properties.

A.3 Spurious Result Predictive Models

This section provides the equations for the predictive models we selected to predict failures. The form of the predictive equations is $\Pr(\mathit{Spurious\ Results}) = \frac{e^{g(x)}}{1+e^{g(x)}}$; for readability, we provide equations for g(x) below. The equation for SPIN checking deadlock is

g(x) = 8.0896 - 0.2590*T - 1.8541*C - 0.1526*Alpha' + 0.2539*Beta + 0.3956*N - 0.0015*MaxTRANS - 5.6727*Vars.

The equation for SPIN using never claims to check other properties is

g(x) = 6.9290 - 3.3254*C + 0.7151*N - 0.0004*MaxTRANS - 10.7109*Vars.

The equation for SPIN using assertions to check other properties is

g(x) = 8.1618 - 0.2886*T - 2.9286*C + 0.5816*N - 11.1929*Vars.

The equation for SPIN+PO checking deadlock is

g(x) = 6.6263 - 0.2370*T - 1.0433*C - 0.0980*MaxC - 0.2057*Alpha' + 0.2141*Beta - 3.5E-09*Cnd' + 2.25E-09*Cif + 0.2264*N - 0.0010*MaxTRANS - 8.4E-18*WFSA - 4.7549*Vars.

The equation for SPIN+PO checking other properties is

g(x) = 10.1997 - 0.2419*T - 3.5706*C + 0.4826*N - 12.2021*Vars.

The equation for TRACC checking deadlock is

$$g(x) = 3.6589 - 0.3487*C.$$

Because we had indications that all the spurious result models we built for TRACC checking other properties were overfit to the data, we do not provide an equation for TRACC checking other properties. The equation for SMV checking deadlock is

$$g(x) = 8.3817 - 0.4975*T - 1.2482*C + 0.1847*Beta + 0.1081*N - 4.1590*Vars.$$

The equation for SMV checking other properties is

$$g(x) = 7.1481 - 0.1515*T - 2.1992*C + 0.1435*N - 10.3828*Vars.$$

The equation for INCA checking deadlock is

$$g(x) = 9.1374 - 0.2867*T - 1.9769*C + 0.3537*Beta + 0.0095*MaxTRANS - 3.2146*Vars.$$

The equation for INCA checking other properties is

$$g(x) = 3.9281 - 0.0991*T - 1.6297*C + 0.0243*MaxC - 1.9E\text{-}07*Alpha' + 0.1987*Beta + 2.59E\text{-}08*Cnd' - 6.1E\text{-}09*Cif + 0.1292*N - 0.0019*MaxTRANS - 6.0E\text{-}21*WFSA - 10.7992*Vars.$$

The equation for FLAVERS checking other properties is

$$g(x) = -31.1782 - 2.7071*C + 0.6871*MaxC + 0.5250*Beta - 0.3649*N + 1.0691*Vars + 10.1329*QRE\ States - 2.0786*Query\ Events - 0.4609*Assertions.$$

BIBLIOGRAPHY

[Agr84] Agresti, Alan, *Analysis of Ordinal Categorical Data*, John Wiley & Sons, New York, 1984.

[AS87] Alpern, Bowen and Schneider, Fred B. Recognizing safety and liveness. *Distributed Computing*, 2:117-126, 1987.

[AK84] Avizienis, A. and Kelly, J.P.J. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67-80, Aug 1984.

[ABC+91] Avrunin, George S., Buy, Ugo A., Corbett, James C., Dillon, Laura K., and Wileden, Jack C. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204-1222, November 1991.

[ACD+94] Avrunin, G.S., Corbett, J.C., Dillon, L.K., and Wileden, J.C. Automatic derivation of time bounds in uniprocessor concurrent systems. *IEEE Transactions on Software Engineering*, 20(9):708-719, 1994.

[BDF92] Balbo, Gianfranco, Donatelli, Susanna, and Franceschinis, Giuliana. Understanding parallel program behavior through petri net models. *Journal of Parallel and Distributed Computing*, 15(3):171-187, July 1992.

[BSH86] Basili, Victor R., Selby, Richard W., and Hutchens, David H. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733-743, July 1986.

[BS87] Basili, Victor R. and Selby, Richard W. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278-1296, December 1987.

[BW84] Basili, Victor R. and Weiss, David M. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728-738, November 1984.

[Bla70] Blalock, Hubert M., Jr. Correlated Independent Variables: The Problem of Multicollinearity. In Edward R. Tufte, editor, *The Quantitative Analysis of Social Problems*. Addison-Wesley Publishing Company, Massachusetts, 1970.

[BCM+90] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., and Hwang, L.J. Symbolic model checking : 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428-439, 1990.

[Cha95a] Chamillard, A.T. An exploratory study of program metrics as predictors of reachability analysis performance. In *Proceedings of the 1995 European Software Engineering Conference*, pages 343-361, Barcelona, Spain, September 1995.

[Cha95b] Chamillard, Albert Timothy. Improving static analysis accuracy on concurrent Ada programs: Complexity results and empirical findings. Technical Report TR 95-49, University of Massachusetts, Amherst, 1995.

[CC96] Chamillard, A.T., and Clarke, Lori A. Improving the accuracy of Petri net-based analysis of concurrent programs. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 24-38, San Diego CA, January 1996.

[CK93] Cheung, S.C. and Kramer, J. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the Software Engineering Conference*, 1993.

[CK94] Cheung, S.C. and Kramer, J. An integrated method for effective behaviour analysis of distributed systems. In *Proceedings of the 16th International Conference on Software Engineering*, pages 309-320, Soreno Italy, May 1994.

[CK95] Cheung, S.C. and Kramer, J. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 140-150, Washington DC, October 1995.

[CES86] Clarke, E.M., Emerson, E.A., and Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244-263, April 1986.

[CR81] Clarke, Lori A. and Richardson, Debra J. Symbolic evaluation methods - implementations and applications. In *Computer Program Testing*, pages 65-102. Chandrasekaran and Radicchi, editors, North-Holland Publishing Company, 1981.

[Coh95] Cohen, Paul R., *Empirical Methods for Artificial Intelligence*, The MIT Press, Massachusetts, 1995.

[CW90] Compton, B. Terry and Withrow, Carol. Prediction and control of Ada software defects. *Journal of Systems and Software*, 12(3):199-207, July 1990.

[Cor93] Corbett, James C. Identical tasks and counter variables in an integer programming based approach to verification. In Martin Feather and Axel van Lamsweerd, editors, *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 100-109, Los Alamitos, California, December 1993.

[Cor94] Corbett, James C. An empirical evaluation of three methods for deadlock analysis of Ada tasking programs. In *Proceedings of the 1994 International Sympoosium on Software Testing and Analysis (ISSTA)*, pages 228-239, Seattle WA, August 1994.

[CA94] Corbett, James C. and Avrunin, George S. Towards scalable compositional analysis. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62-75, New Orleans, Louisiana, December 1994.

[CA95] Corbett, James C. and Avrunin, George S. Using integer programming to verify general safety and liveness properties. *Journal of Formal Methods in System Design*, 6(1):97-123, 1995.

[DS92] Damerla, Srinivasarao and Shatz, Sol M. Software complexity and Ada rendezvous: Metrics based on nondeterminism. *Journal of Systems and Software*, 17(2):119-127, February 1992.

[DW80] Daniel, Cuthbert and Wood, Fred S., *Fitting Equations to Data: Computer Analysis of Multifactor Data*, John Wiley & Sons, New York, 1980.

[DO88] DeMillo, Richard and Offutt, J. An experimental evaluation of automatic test data generation. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pages 209-232, Portland, Oregon, September 1988.

[DS66] Draper, N.R. and Smith, H., *Applied Regression Analysis*, John Wiley & Sons, New York, 1966.

[Dun86] Dunham, Janet R. Experiments in software reliability: Life-critical applications. *IEEE Transactions on Software Engineering*, SE-12(1):110-123, January 1986.

[DN84] Duran, Joe W. and Ntafos, Simeon C. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438-444, July 1984.

[DBD+94] Duri, S., Buy, U., Devarapalli, R., and Shatz, S.M. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Transactions on Software Engineering and Methodology*, 3(4):340-380, October 1994.

[DC94] Dwyer, Matthew B. and Clarke, Lori A. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62-75, New Orleans, Louisiana, December 1994.

[DCN95] Dwyer, Matthew B., Clarke, Lori A., and Nies, Kari A. A compact petri net representation for concurrent programs. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995.

[FPG94] Fenton, Norman, Pfleeger, Shari Lawrence, and Glass, Robert L. Science and substance: A challenge to software engineers. *IEEE Software*, pages 86-95, July 1994.

[For88] Ford, Ray. Concurrent algorithms for real-time memory management. *IEEE Software*, pages 10-23, September 1988.

[FW93] Frankl, Phyllis G. and Weiss, Stewart N. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774-787, August 1993.

[GMO76] Gabow, Harold N., Maheshwari, Shachindra N., and Osterweil, Leon J. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, SE-2(3):227-231, September 1976.

[GW91] Godefroid, Patrice and Wolper, Pierre. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Third Workshop on Computer Aided Verification*, pages 417-428, July 1991.

[Hal76] Halstead, M.H., *Elements of Software Science*, North-Holland, Amsterdam, 1977.

[HL85] Helmbold, D. and Luckham, D.C. Debugging Ada tasking programs.

IEEE Software, pages 47-57, March 1985.

[Hol91] Holzmann, Gerard J., *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[HL89] Hosmer, David W., Jr. and Lemeshow, Stanley, *Applied Logistic Regression*, John Wiley & Sons, New York, 1989.

[HFG+94] Hutchins, Monica, Foster, Herb, Goradia, Tarak, and Ostrand, Thomas. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191-200, Soreno Italy, May 1994.

[IMO+84] Iannino, Anthony, Musa, John D., Okumoto, Kazuhira, and Littlewood, Bev. Criteria for software reliability model comparisons. *IEEE Transactions on Software Engineering*, SE-10(6):687-691, November 1984.

[IR85] Iyer, Ravishankar K. and Rossetti, David J. Effect of system workload on operating system reliability: A study on IBM 3081. *IEEE Transaction on Software Engineering*, SE-11(12):1438-1448, December 1985.

[KL86] Knight, John C. and Leveson, Nancy G. An experimental evaluation of the assumption of indpendendence in multivarsion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96-109, January 1986.

[LT93] Levine, David L. and Taylor, Richard N. Metric-driven reengineering for static concurrency analysis. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 40-50, Cambridge MA, June 1993.

[LC87] Li, H.F. and Cheung, W.K. An empirical study of software metrics. *IEEE Transactions on Software Engineering*, SE-13(6):697-708, June 1987.

[Lit91] Littlewood, Bev. Software reliability modelling: achievements and limitations. In *Proceedings of the 5th Annual European Computer Conference*, pages 336-334, Bologna Italy, May 1991.

[LC89] Long, Douglas L. and Clarke, Lori A. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering*, pages 44-52, Pittsburgh PA, May 1989.

[LC91] Long, Douglas and Clarke, Lori A. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification (TAV4)*, Victoria, Canada, pages 236-250, October 1991.

[MR91] Masticola, Stephen P. and Ryder, Barbara G. A model of Ada programs for
static deadlock detection in polynomial time. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 97-107, May 1991.

[MR93] Masticola, Stephen P. and Ryder, Barbara G. Non-concurrency analysis. In *Proceedings of the ACM Symposium on Principles and Practices of Parallel Programming (PPOPP)*, 1993.

[McC76] McCabe, Thomas J. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308-320, December 1976.

[McD89] McDowell, Charles E. A practical algorithm for static analysis of programs.
Journal of Parallel and Distributed Computing, 6:515-536, 1989.

[McM93] McMillan, Kenneth L., *Symbolic Model Checking*, Kluwer Academic Publishers, Boston, MA, 1993.

[Mil80] Milner, R., *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[MP82] Montgomery, Douglas C. and Peck, Elizabeth A., *Introduction to Linear Regression Analysis*, John Wiley & Sons, New York, 1982.

[MSS89] Murata, Tadao, Shenker, Boris, and Shatz, Sol M. Detection of Ada static deadlocks using petri net invariants. *IEEE Transactions on Software Engineering*, 15(3):314-326, March 1989.

[MO84] Musa, John D. and Okumoto, Kazuhira. A comparison of time domains for
software reliability models. *Journal of Systems and Software*, 4(4):277-287, November 1984.

[NCO96] Naumovich, Gleb N., Clarke, Lori A., and Osterweil, Leon J. Verification of communication protocols using data flow analysis. Technical
Report TR 96-27, University of Massachusetts, Amherst, 1996.

[NWK85] Neter, John, Wasserman, William, and Kutner,Michael H., *Applied Linear Statistical Models*, Richard D. Irwin, Inc, Illinois, 1985.

[OC92] Osterweil, Leon and Clarke, Lori A. A proposed testing and analysis research initiative. *IEEE Software*, pages 89-96, September 1992.

[Pet77] Peterson, James L. Petri nets. *Computing Surveys*, 9(3):223-252, September 1977.

[Pet81] Peterson, James L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[Pfl94] Pfleeger, Shari Lawrence. Design and analysis in software engineering part 1: The language of case studies and formal experiments. *ACM SIGSOFT Software Engineering Notes*, 19(4):16-20, October 1994.

[PS90] Porter, Adam A., and Selby, Richard W. Evaluating techniques for generating metric-based classification trees. *Journal of Systems and Software*, 12(3):209-218, July 1990.

[PST+95] Porter, A., Siy, H., Toman, C.A., and Votta, L.G. An experiment to assess the cost-benefits of code inspections in large scale software development. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 92-103, Washington DC, October 1995.

[PV94] Porter, A.A. and Votta, L.G. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the Sixteenth International Conference on Software Engineering*, Soreno Italy, pages 103-112, May 1994.

[RS90] Reif, John H. and Smolka, Scott A. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1-30, 1990.

[Rus91] Russell, Glen W. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25-31, January 1991.

[Sca89] Scanlan, David A. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, pages 28-36, September 1989.

[SMT92] Schneider, G. Michael, Martin, Johnny, and Tsai, W.T. An experimental study of fault detection in user reuirements documents. *ACM Transactions on Software Engineering and Methodology*,

1(2):188-204, April 1992.

[SC88] Shatz, S.M. and Cheng, W.K. A petri net framework for automated static analysis of Ada tasking behavior. *Journal of Systems and Software*, 8(5):343-359, December 1988.

[SMM+77] Shneiderman, Ben, Mayer, Richard, McKay, Don, and Heller, Peter. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, 20(6):373-381, June 1977.

[Sho75] Shooman, Martin L. Software reliability: Measurement and models. In *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, Washington DC, pages 485-491, January 1975.

[Tay83a] Taylor, Richard N. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362-376, May 1983.

[Tay83b] Taylor, Richard. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57-84, 1983.

[TO80] Taylor, Richard N. and Osterweil, Leon J. Anomaly detection in concurrent
software by static data flow analysis. *IEEE Transaction on Software Engineering*, SE-6(3):265-277, May 1980.

[TLP+95] Tichy, Walter F., Lukowicz, Paul, Prechelt, Lutz, and Heinz, Ernst A. Experimental evaluation in computer science: A quantitative study. *The Journal of Systems and Software*, 28(1):9-18, January 1995.

[Val90] Valmari, A. A stubborn attack on state explosion. In E.M. Clarke and R.P. Kurshan, editors, *Computer-Aided Verification '90*, number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 25-41, Providence, Rhode Island, 1991.

[VW84] Vessey, Iris and Weber, Ron. Research on structured programming: An empiricist's evaluation. *IEEE Transactions on Software Engineering*, SE-10(4):397-407, July 1984.

[YY91] Yeh, Wei Jen and Young, Michal. Compositional reachability analysis using
process algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification (TAV4)*, Victoria, Canada, pages 49-59, October 1991.

[YT88] Young, Michal and Taylor, Richard N. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software*

Engineering, 14(10):1499-1511, October 1988.

[YTF+89] Young, Michal, Taylor, Richard N., Forester, Kari, and Brodbeck, Debra. Integrated concurrency analysis in a software development environment. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis and Verification (TAV3)*, pages 200-209, 1989.