AUTOMATING AND EVALUATING ASSUME-GUARANTEE REASONING

A Dissertation Presented

by

JAMIESON M. COBLEIGH

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

DOCTOR OF PHILOSOPHY

February 2007

Computer Science

© Copyright by Jamieson M. Cobleigh 2007

All Rights Reserved

AUTOMATING AND EVALUATING ASSUME-GUARANTEE REASONING

A Dissertation Presented

by

JAMIESON M. COBLEIGH

Approved as to style and content by:

____

Lori A. Clarke, Chair

____

George S. Avrunin, Member

____

Maciej Ciesielski, Member

____

Neil Immerman, Member

____

Leon J. Osterweil, Member

____

W. Bruce Croft, Department Chair
Computer Science

To my wife Rachel for all of her help and support.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Lori Clarke. Lori has supported me and supplied advice throughout my graduate school career. She has constantly pushed me to improve and I am grateful for her holding me to high standards.

I am also indebted to George Avrunin for his input on my research. His eye for detail caught places where my arguments needed more support or were unclear. I would also like to thank Lee Osterweil for being part of many fruitful conversations about my research.

I would like to acknowledge Matt Dwyer and Gleb Naumovich for their work on FLAVERS. Without the solid foundation they laid, the work I did would not have been possible.

The two summers I spent at the NASA Ames Research Center were important to my work. Dimitra Giannakopoulou and Corina Păsăreanu were instrumental in getting me interested in assume-guarantee reasoning. Alex Groce pointed me at the L* algorithm and provided help in understanding it. Willem Visser, Peter Mehlitz, and Owen O'Malley patiently explained the internals of Java PathFinder to me which enabled me to modify it to suit my needs.

At the University of Massachusetts, Heather Conboy has provided a significant amount of work and advice regarding the implementation and design of FLAVERS. She also supplied us with a copy of The Advocate crossword puzzle every week, which provided a necessary break from work. Sandy Wise was always willing to answer whatever questions I came up with, supplying helpful advice, arcane technical knowledge, and many bad puns.

Barbara Ryder at Rutgers University provided helpful guidance when I was an undergraduate. Without her advice, I never would have considered becoming a graduate student.

My parents Robert and Mary Ann have always supported me in everything I have tried to do and for that I am grateful. My wife Rachel has also been invaluable, particularly as I was finishing. Her help enabled me to spend the time I needed to complete my dissertation.

Lastly, I would like to thank God for watching over me now and always. Everything I have accomplished is due to His provision and blessing.

ABSTRACT

AUTOMATING AND EVALUATING ASSUME-GUARANTEE REASONING

FEBRUARY 2007

JAMIESON M. COBLEIGH

B.Sc., RUTGERS UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lori A. Clarke

Software systems are taking on an increasingly important role in society and are being used in critical applications where their failure could result in human casualties or substantial economic loss. Thus, it is important to validate software systems to ensure their quality. One technique for validating software systems is finite-state verification, in which a finite model of a system is analyzed to ensure that it satisfies a property that specifies a desired system behavior. Unfortunately, the cost of finite-state verification can be exponential in the size of the system being analyzed.

Compositional analysis is a "divide-and-conquer" approach to verification that aims to reduce the cost of verification. One proposed compositional technique is assume-guarantee reasoning. With this technique a system is decomposed into subsystems and these subsystems are analyzed individually. By composing the results of these analyses, it can be

determined whether or not a system satisfies a property. Because each subsystem is smaller than the whole system, analyzing each subsystem individually may reduce the overall cost of verification. Often the behavior of a subsystem is dependent on the subsystems with which it interacts, and thus it is usually necessary to provide assumptions about the environment in which a subsystem executes. Because developing assumptions has been a difficult manual task, the evaluation of assume-guarantee reasoning has been limited.

In this thesis we present an algorithm that automatically learns assumptions. Using this algorithm, we undertook a study to determine if assume-guarantee reasoning provides an advantage over monolithic verification. Using two different verifiers, we considered all two-way decompositions for a set of systems and properties. By increasing the number of repeated tasks in these systems, we evaluated the decompositions as they were scaled. We found that in only a few cases can assume-guarantee reasoning verify properties on larger systems than monolithic verification can and in these cases the systems that can be analyzed are only a few sizes larger. Although these results are discouraging, they provide insight about research directions that should be pursued and highlight the importance of experimental evaluation.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT vii

LIST OF TABLES xiii

LIST OF FIGURES xvii

CHAPTER

1. INTRODUCTION 1

2. RELATED WORK 7

2.1 Finite-State Verification 7

2.1.1 Reachability Based 7
2.1.2 Data-Flow Based 13
2.1.3 SAT Based 14
2.1.4 Integer Necessary Condition Based 15

2.2 Compositional Analysis 16

2.2.1 Compositional Construction 17
2.2.2 Compositional Reasoning 19

3. BACKGROUND 24

3.1 FLAVERS 24

3.1.1 Specifying Properties 25
3.1.2 System Model 26
3.1.3 Verifying Properties 31
3.1.4 Improving Precision 32

3.2 LTSA 36

3.2.1 Specifying Properties 36
3.2.2 System Model 37
3.2.3 Checking Properties 42

3.3 The L* Algorithm 42

4. USING THE L* ALGORITHM TO LEARN ASSUMPTIONS FOR ASSUME-GUARANTEE REASONING 46

4.1 Implementing the Teacher 46

4.1.1 Answering Queries 46
4.1.2 Answering Conjectures 47
4.1.3 Correctness and Termination 48

4.2 Implementing the Teacher for LTSA 48

4.2.1 The Alphabet of the Assumption 48
4.2.2 Answering Queries 50
4.2.3 Answering Conjectures 50

4.3 Implementing the Teacher for FLAVERS 51

4.3.1 The Model 51
4.3.2 The Alphabet of the Assumption 53
4.3.3 Answering Queries 53
4.3.4 Answering Conjectures 55
4.3.5 Environment Generation 56

5. EXPERIMENTAL METHODOLOGY AND RESULTS 58

5.1 Does Assume-Guarantee Reasoning Save Memory for Small System Sizes? 61
5.2 Does Assume-Guarantee Reasoning Save Memory for Larger System Sizes? 64
5.3 Can Assume-Guarantee Reasoning Verify Properties of Larger Systems than Monolithic Verification 71
5.4 Are the Generalized Decompositions the Best Decompositions? 75

5.4.1 Comparing the Best Known Decompositions to the Generalized Decompositions 76
5.4.2 Generalizing Decompositions from the Best Known Decomposition at Larger System Sizes 79
5.4.3 Discussion 81

5.5 Does Assume-Guarantee Reasoning Save Time? 82

5.6 What is the Cost of Using the L* Algorithm? 85

5.6.1 What is the Memory Cost of Using the L* Algorithm? 86
5.6.2 What is the Time Cost of Using the L* Algorithm? 88
5.6.3 Reducing the Cost of Using the L* Algorithm 89

5.7 Threats to Validity 90

6. ASSUME-GUARANTEE REASONING OF SOFTWARE USING DESIGN-LEVEL ASSUMPTIONS 93

6.1 Checking Assume-Guarantee Triples Using JPF 94

6.1.1 Background 94
6.1.2 Instrumenting the Java Software 95
6.1.3 Modeling Environments 97

6.2 Case Study 98

6.2.1 Description of the K9 Mars Rover 98
6.2.2 Analysis of the Design of the Rover 99
6.2.3 Analysis of the Implementation of the Rover 100

6.2.3.1 Environment Modeling 100
6.2.3.2 Results 102

6.3 Discussion 104

7. CONCLUSIONS 106

APPENDICES

A. DESCRIPTION OF EXAMPLES 110

A.1 Chiron 110
A.2 Gas Station 111
A.3 Peterson 112
A.4 Relay 112
A.5 Smokers 113

B. SUBJECT NUMBERS 114
C. DETAILED DATA 117

C.1 FLAVERS Data 118

C.1.1 Chiron Single 118

C.1.2 Chiron Multiple . . . 121
C.1.3 Gas Station . . . 123
C.1.4 Peterson . . . 126
C.1.5 Relay . . . 127
C.1.6 Smokers . . . 127

C.2 LTSA Data . . . 129

C.2.1 Chiron Single . . . 129
C.2.2 Chiron Multiple . . . 132
C.2.3 Gas Station . . . 134
C.2.4 Peterson . . . 136
C.2.5 Relay . . . 136
C.2.6 Smokers . . . 136

BIBLIOGRAPHY . . . 139

LIST OF TABLES

Table **Page**

5.1 Number of two-way decompositions examined for systems of size 2 for FLAVERS . . . 61

5.2 Number of two-way decompositions examined for systems of size 2 for LTSA . . . 62

5.3 Generalized decompositions compared to monolithic verification with respect to scaling . . . 73

5.4 System sizes at which the best decomposition is known . . . 76

6.1 Results of verifying the Mars K9 Rover . . . 103

B.1 Mapping from subject number to subject . . . 115

B.2 Mapping from subject to subject number . . . 116

C.1 Chiron Single property 1 with FLAVERS . . . 118

C.2 Chiron Single property 2 with FLAVERS . . . 118

C.3 Chiron Single property 3 with FLAVERS . . . 119

C.4 Chiron Single property 4 with FLAVERS . . . 119

C.5 Chiron Single property 5 with FLAVERS . . . 119

C.6 Chiron Single property 6 with FLAVERS . . . 119

C.7 Chiron Single property 7 with FLAVERS . . . 120

C.8 Chiron Single property 8 with FLAVERS . . . 120

C.9 Chiron Single property 9 with FLAVERS . . . 120

C.10 Chiron Multiple property 1 with FLAVERS . . . 121
C.11 Chiron Multiple property 2 with FLAVERS . . . 121
C.12 Chiron Multiple property 3 with FLAVERS . . . 121
C.13 Chiron Multiple property 4 with FLAVERS . . . 122
C.14 Chiron Multiple property 5 with FLAVERS . . . 122
C.15 Chiron Multiple property 6 with FLAVERS . . . 122
C.16 Chiron Multiple property 7 with FLAVERS . . . 122
C.17 Chiron Multiple property 8 with FLAVERS . . . 123
C.18 Chiron Multiple property 9 with FLAVERS . . . 123
C.19 Gas Station property 1 with FLAVERS . . . 123
C.20 Gas Station property 2 with FLAVERS . . . 124
C.21 Gas Station property 3 with FLAVERS . . . 125
C.22 Gas Station property 4 with FLAVERS . . . 126
C.23 Peterson property 1 with FLAVERS . . . 126
C.24 Relay property 1 with FLAVERS . . . 127
C.25 Smokers property 1 with FLAVERS . . . 127
C.26 Smokers property 2 with FLAVERS . . . 127
C.27 Smokers property 3 with FLAVERS . . . 128
C.28 Smokers property 4 with FLAVERS . . . 128
C.29 Smokers property 5 with FLAVERS . . . 128
C.30 Smokers property 6 with FLAVERS . . . 128
C.31 Smokers property 7 with FLAVERS . . . 128
C.32 Smokers property 8 with FLAVERS . . . 129

C.33 Chiron Single property 1 with LTSA . . . 129
C.34 Chiron Single property 2 with LTSA . . . 130
C.35 Chiron Single property 3 with LTSA . . . 130
C.36 Chiron Single property 4 with LTSA . . . 130
C.37 Chiron Single property 5 with LTSA . . . 130
C.38 Chiron Single property 6 with LTSA . . . 131
C.39 Chiron Single property 7 with LTSA . . . 131
C.40 Chiron Single property 9 with LTSA . . . 131
C.41 Chiron Multiple property 1 with LTSA . . . 132
C.42 Chiron Multiple property 2 with LTSA . . . 132
C.43 Chiron Multiple property 3 with LTSA . . . 132
C.44 Chiron Multiple property 4 with LTSA . . . 133
C.45 Chiron Multiple property 5 with LTSA . . . 133
C.46 Chiron Multiple property 6 with LTSA . . . 133
C.47 Chiron Multiple property 7 with LTSA . . . 133
C.48 Chiron Multiple property 9 with LTSA . . . 134
C.49 Gas Station property 1 with LTSA . . . 134
C.50 Gas Station property 2 with LTSA . . . 135
C.51 Gas Station property 3 with LTSA . . . 135
C.52 Gas Station property 4 with LTSA . . . 135
C.53 Peterson property 1 with LTSA . . . 136
C.54 Relay property 1 with LTSA . . . 136
C.55 Smokers property 1 with LTSA . . . 136

C.56 Smokers property 2 with LTSA 137

C.57 Smokers property 3 with LTSA 137

C.58 Smokers property 4 with LTSA 137

C.59 Smokers property 5 with LTSA 137

C.60 Smokers property 6 with LTSA 138

C.61 Smokers property 7 with LTSA 138

C.62 Smokers property 8 with LTSA 138

LIST OF FIGURES

Figure		Page
1.1	Simplest assume-guarantee rule	3
2.1	State compression in SPIN	10
2.2	Addition for abstracted integers	11
2.3	CEGAR phases	12
2.4	Typical compositional construction algorithm	16
2.5	Alternative assume-guarantee rules	19
3.1	Elevator system in Ada	25
3.2	Example property for FLAVERS	26
3.3	CFG for task car	27
3.4	CFG for task controller	27
3.5	Refined CFG for task car	28
3.6	Refined CFG for task controller	28
3.7	TFG for the elevator system	29
3.8	Variable automaton for x	32
3.9	TFG with events relating to the variable x	34
3.10	TA for task car	35
3.11	Example property for LTSA	36
3.12	LTS for task car	38

3.13 LTS for task controller 38

3.14 LTS for variable x 38

3.15 LTS for the composed elevator system, (car $\|$ controller $\|$ x) 40

3.16 Minimal LTS for the elevator system with respect to the property shown in Figure 3.11 41

3.17 The L* algorithm 43

4.1 LTS for $S_1 \| P$, with a counterexample for the query $\langle$sync, controller.x==false, open, move$\rangle$ highlighted 49

4.2 CFG for the environment of S_1 53

4.3 TFG for S_1, without MIP edges 54

4.4 Feasibility constraint for the query $\langle$sync, s2.x==false, open, move$\rangle$ 55

5.1 Memory used by the best decomposition of size 2 for FLAVERS 62

5.2 Memory used by the best decomposition of size 2 for LTSA 63

5.3 Process for generalizing decompositions 65

5.4 Memory used by the generalized decompositions for FLAVERS up to dize 10 68

5.5 Memory used by the generalized decompositions for FLAVERS for just those subjects where the largest system size that could be verified is greater than 10 69

5.6 Memory used by the generalized decompositions for LTSA 70

5.7 Memory used by the generalized decomposition compared to the best decomposition for FLAVERS 77

5.8 Memory used by the generalized decomposition compared to the best decomposition for LTSA 77

5.9 States explored on property 1 of Gas Station with FLAVERS 79

5.10 States explored on property 1 of Relay with FLAVERS 80

5.11 Time used by the generalized decompositions for FLAVERS up to size 10 83

5.12 Time used by the generalized decompositions for FLAVERS for just those subjects where the largest system size that could be verified is greater than 10 84

5.13 Time used by the generalized decompositions for LTSA 85

5.14 Percentage of time spent learning for FLAVERS 87

5.15 Percentage of time spent learning for LTSA 88

6.1 Verification at the design level and software level 94

6.2 Method `event` of class `AG_Monitor` 96

6.3 Class `AG_Assumption` (excerpt) 97

6.4 The executive of the K9 Mars Rover 98

6.5 Environment for the ExecCondChecker 101

CHAPTER 1

INTRODUCTION

Software systems are taking on an increasingly important role in society and are being used in critical applications where their failure could result in human casualties or substantial economic loss. Thus, it is important to validate such software systems to ensure their quality. This task is becoming more difficult, however, as software systems continue to increase both in size and in complexity. There are many techniques that can be used to validate software systems, one of which is finite-state verification (FSV). FSV techniques work by analyzing a finite model of a system to ensure that it satisfies a property that specifies a desired system behavior. Since FSV techniques examine all paths through the system model, they can be used to determine whether or not the property being verified is violated. If the property is violated, FSV techniques usually provide a counterexample, a path through the model that reveals this violation.

A more commonly used technique to validate software systems is testing (e.g., [1]). With testing, a system is executed and reasoning is performed based on the observed behaviors of that system. Since the number of possible ways that a system can execute is often prohibitively large, it is usually infeasible to test all possible executions of a system. Thus, testing can only show the existence of errors, not the absence of errors. This limitation is exacerbated for concurrent systems where a test case can produce different results for the same input depending on how actions in the threads are scheduled. Although testing is important to validate a software system in its actual runtime environment, for critical systems it needs to be supplemented with techniques that can provide more definitive results.

Another validation technique that can provide such definite results is theorem proving. With theorem proving, a formal model of the system is built and mathematical reasoning is employed to prove properties about the model (e.g, [68]). While some tools have been developed to help automate the writing of proofs (e.g., [92]), automated theorem provers are usually not guaranteed to terminate. Furthermore, theorem provers often require significant human interaction and expertise to use. While FSV techniques cannot prove as wide a range of properties as theorem provers, FSV techniques are usually considered easier to use.

FSV techniques, therefore, seek a middle ground between testing and theorem proving based verification. FSV techniques, however, are limited in the size of the system that they can evaluate since the cost of verification can be exponential in the size of the system being verified, a problem known as state explosion.

Compositional analysis techniques have been proposed as one way to address the state-explosion problem. These techniques use a "divide-and-conquer" approach to verification. One of the most frequently advocated compositional analysis techniques is assume-guarantee reasoning [78, 97] in which a system under analysis is decomposed into subsystems and these subsystems are analyzed individually. By composing the results of these analyses, it can be determined if a system satisfies a property. By individually analyzing each subsystem, which is smaller than the whole system, the effect of the state-explosion problem may be reduced. Often the behavior of a subsystem is dependent on the subsystems with which it interacts and thus it is usually necessary to provide assumptions about the environment in which a subsystem executes.

In assume-guarantee reasoning, a verification problem is represented as a triple, $\langle A \rangle\ S\ \langle P \rangle$, where:

- S is the subsystem being analyzed,
- P is the property to be verified, and
- A is an assumption about the environment in which S is used.

$$\begin{array}{ll} \text{Premise 1:} & \langle A \rangle\ S_1\ \langle P \rangle \\ \text{Premise 2:} & \langle true \rangle\ S_2\ \langle A \rangle \\ \hline & \langle true \rangle\ S_1 \parallel S_2\ \langle P \rangle \end{array}$$

Figure 1.1: Simplest assume-guarantee rule

Note that although this notation resembles a Hoare triple [68], A is not a precondition and P is not a postcondition. Instead, A is a constraint on the behavior of S. If S, as constrained by A, satisfies P, then the formula $\langle A \rangle\ S\ \langle P \rangle$ is true.

Consider a system that is decomposed into two subsystems, S_1 and S_2 (which may then be further decomposed). Figure 1.1 shows the simplest assume-guarantee rule that can be used to verify that a property P holds on the system composed of S_1 and S_2 running in parallel, denoted $S_1 \parallel S_2$. This rule states that if a subsystem S_2 satisfies an assumption A and that if under assumption A subsystem S_1 satisfies property P, then the system $S_1 \parallel S_2$ satisfies property P. This allows a property to be verified on $S_1 \parallel S_2$ without ever having to examine a monolithic model for the entire system.

In addition, assume-guarantee reasoning can be applied at different phases of the software lifecycle. For example, if a design for a software system has been written in a formalism with well-defined semantics, then that design can be analyzed using assume-guarantee reasoning to detect property violations. By detecting such violations before they can be implemented in a software system, the overall cost of developing that software system can be reduced. Furthermore, assumptions found during verification of the design of a software system can aid in the verification of the actual software system.

There are several issues that make using the assume-guarantee rule shown in Figure 1.1 difficult. First, if the system under analysis is made up of more than two subsystems, which is often the case, then S_1 and S_2 may each need to be made up of several of these subsystems. How this decomposition is done can have a significant impact on the time and memory needed for verification, but it is not clear how to select an effective decomposition. In fact, we have found that the memory usage between two different decompositions can

vary by over an order of magnitude. Second, once a decomposition is selected, it can be difficult to manually find an assumption A that can be used to complete an assume-guarantee proof because the assumption must:

1. be strong enough to sufficiently restrict the behavior of S_1 so that $\langle A \rangle\ S_1\ \langle P \rangle$ holds, and

2. be weak enough to not overly restrict the behavior of S_2 so that $\langle true \rangle\ S_2\ \langle A \rangle$ holds.

Because selecting a decomposition and developing an assumption are difficult tasks, it had not been practical previously to undertake an empirical evaluation of assume-guarantee reasoning, although several case studies have been reported (e.g., [49, 67, 83]).

Recent work on automatically computing assumptions for assume-guarantee reasoning [5, 14, 23, 36, 52, 65] eliminates one of the obstacles to empirical evaluation by making it feasible to examine a large number of decompositions without having to manually produce a suitable assumption for each one. Using one algorithm that learns assumptions [36], we undertook such a study to evaluate the effectiveness of assume-guarantee reasoning.

We initially undertook a study to gain insight into how to best decompose systems and to learn what kind of savings could be expected from assume-guarantee reasoning. We began by using FLAVERS [43], a finite-state verifier, to verify properties of several systems written in Ada, but the results of these experiments were not as promising as the results seen in [36], which used LTSA [80], another finite-state verifier. Although the two tools use different models and verification methods, this discrepancy was surprising to us. As a result, we translated these systems into FSP, the input language of LTSA, to see if our choice of tool affected our results. In the study reported here, we applied both tools to a small set of scalable systems and properties.

Initially, we selected several decompositions for each example at the smallest reasonable system size based on our understanding of the system, the property, and of assume-guarantee reasoning. We expected that in most cases assume-guarantee reasoning would

save memory over monolithic verification. In these experiments, we were surprised to discover that in over half of the subjects we verified the decompositions we selected did not use less memory than monolithic verification.

Based on these initial results, we undertook a more comprehensive study in which, for each property of each system at that system's smallest size, we examined *all* of the ways to decompose the subsystems of that system into S_1 and S_2 to find the best decomposition in the sense that assume-guarantee reasoning explores the fewest states. Because examining all decompositions at larger system sizes quickly becomes infeasible due to the explosion in the number of decomposition that need to be considered and the increased cost for evaluating each decomposition, we generalized the best decompositions found for smaller system sizes and used those generalized decompositions when applying assume-guarantee reasoning for larger system sizes. To evaluate these generalized decompositions we tried to explore all two-way decompositions for a few larger system sizes, although we were not always able to find the best decomposition in all cases because of the time required. In total, we examined over 43,500 two-way decompositions and used over 1.54 years of CPU time.

The results of our experiments are not very encouraging and raise concerns about the effectiveness of assume-guarantee reasoning. For the vast majority of decompositions, more states are explored using assume-guarantee reasoning than are explored using monolithic verification. If we restrict our attention to just the *best* decomposition for each property, we found that in about half of these cases our automated assume-guarantee reasoning technique explores fewer states than monolithic verification for the smallest system size. When we used generalized decompositions to scale the systems, assume-guarantee reasoning often explores fewer states than monolithic verification. This memory savings, however, is rarely enough to increase the size of the systems that can be verified beyond what can be done with monolithic verification. Although these results are discouraging, they provide

insight about research directions that should be pursued and highlight the importance of experimental evaluation.

This thesis begins with a discussion of related work and then chapter 3 provides background information about the two finite-state verifiers and the learning algorithm we used. Chapter 4 describes how the learning algorithm can be used for automated assume-guarantee reasoning with these two finite-state verifiers. Chapter 5 describes our experimental methodology and results. Chapter 6 discusses another application of assume-guarantee reasoning, using assumptions learned while verifying the design of a software system to complete assume-guarantee reasoning proofs on the actual software system. Chapter 7 presents our conclusions and discusses future work.

CHAPTER 2

RELATED WORK

Both finite-state verification and compositional analysis are active areas of research with long histories and in this chapter we first focus on related work in the area of finite-state verification and then turn to the area of compositional analysis.

2.1 Finite-State Verification

Finite-state verification (FSV) techniques work by analyzing a finite model of a system to ensure that it satisfies a property that specified a desired system behavior. Properties are usually classified into two categories: safety and liveness [3]. Intuitively, a safety property is one that states that some “bad thing” never happens, for example, that an elevator never moves while its doors are open. A liveness property is one that states that some “good thing” will eventually happen, for example, that if a call button is pushed on a floor then the elevator will eventually stop at that floor. If the property being verified is violated, FSV techniques usually provide a counterexample, a path through the model that reveals this violation. FSV techniques, however, are limited in the size of the system that they can verify since the cost of verification can be exponential in the size of the system being verified, a problem known as state explosion. Many different approaches to FSV have been proposed and in this section we will discuss four of these: reachability based, data-flow based, SAT based, and integer necessary condition based.

2.1.1 Reachability Based

Reachability based finite-state verifiers attempt to prove properties about systems by building a reachability graph for the system under analysis. The reachability graph of a

system contains a node for every state in the system being analyzed, where a state consists of information about the values of variables and the program counters for each thread in the system. A property can then be checked by analyzing the reachability graph, looking for paths on which that property is violated.

One of the first FSV tools used a reachability-based approach and was called APPROVER [59]. APPROVER was capable of verifying a predefined set of safety properties, a predefined set of liveness properties, and deadlock using a heuristic search. APPROVER was used to check correctness properties of several protocols and was capable of verifying programs written in Algol. Because of the state-explosion problem, APPROVER was limited to verifying properties of relatively small systems.

Later FSV tools based on building reachability graphs have adopted a wide range of optimizations to lessen the effect of the state-exposition problem. In this section, we will look at some of the optimization techniques that have been developed and discuss some of the tools that employ them.

Symbolic Approaches Instead of building the reachability graph explicitly, symbolic approaches represent the states (and transitions) of the reachability graph using Ordered Binary Decision Diagrams (OBDDs), an efficient representation for Boolean functions [19]. The most well-known FSV tool that uses a symbolic representation of the reachability graph is SMV [82], which can check systems for freedom from deadlock and properties specified in CTL [35]. NuSMV [31] is a reimplementation and reengineering of SMV, which adds several new features, including the ability to check properties specified in LTL [35].

Although OBDDs can often compactly represent very large sets, saving memory over explicit approaches, one challenge in using them is that they require that the Boolean variables they operate over be ordered. As seen by Chan et al. in their analysis of the TCAS II specification, the amount of memory used by two different variable orderings for a given system and property can vary by over an order of magnitude [24]. In addition,

there are some sets for which the smallest representation using OBDDs is be exponentially large [18]. Still, symbolic approaches have been very successful at verifying hardware systems, although their success with software systems has been more limited.

Partial Order Approaches Partial order approaches are another way to reduce the effect of the state explosion problem (e.g., [55, 95, 110]). These approaches work by noticing that if certain interleavings of events in different tasks are equivalent with respect to the property being checked, then only one interleaving from each equivalence class needs to be explored during verification. While partial order methods can reduce the time and memory needed for verification, they are not always effective and can sometimes increase the time needed for verification because of the overhead of computing equivalence classes [37, 87]. Still, this overhead is usually minimal and many reachability-based FSV tools, including SPIN [70] and LTSA [80], make use of partial-order approaches when building reachability graphs.

State Compaction FSV tools based on building and analyzing reachability graphs often employ optimizations to reduce the amount of memory needed to store each state in the reachability graph. For example, if it is known that an integer variable in a system only ranges in value from zero to seven, then only three bits are needed to store the state of this variable, not the number of bits needed to store an integer. Even early FSV tools, such as APPROVER, made use of this technique and tried to use the fewest number of bits to store the state of the system.

A more sophisticated optimization called state compression is used by SPIN to reduce the amount of memory needed to store a state of the system. Holzmann observed that often a task in a system only takes on a small number of "local" states and that the exponential number of "global" states in the entire system can be attributed to the large number of possible combinations of these local states [69]. Thus, SPIN stores the local states of each task separately and a global state is constructed by pointing to the local states from which

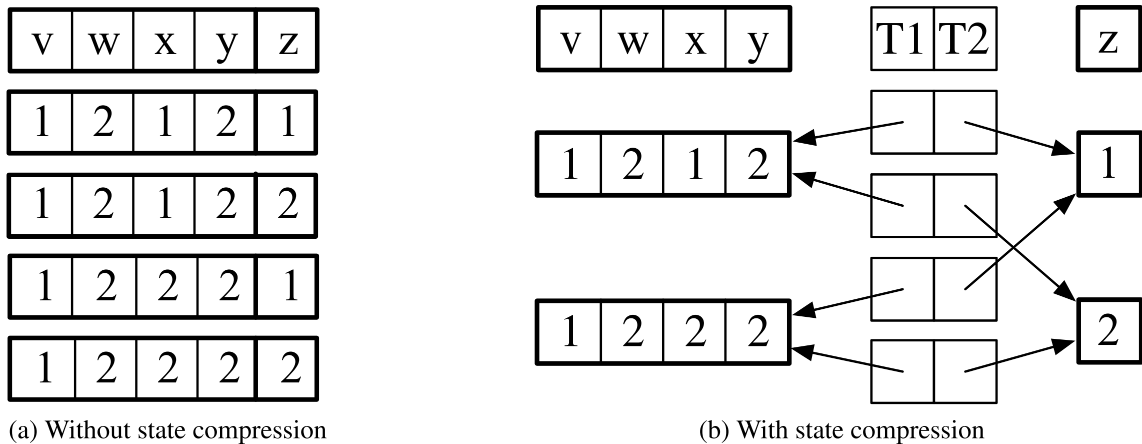

Figure 2.1: State compression in SPIN

it is composed. Consider a task T1 with integer variables v, w, x, and y and a second task T2 with an integer variable z. Four possible states of this system are shown in Figure 2.1a.[1] Figure 2.1b shows what happens when state compression is employed. In this Figure, the local states of T1 are shown on the left, the local states of T2 are shown on the right, and the global states of the system are shown in the middle. Each global state has a pointer to a local state of T1 and to a local state of T2. Assuming a pointer is the same size as an integer, then without state compression 20 integers are needed to store these four states, while with state compression only 18 integers are needed. With more states, this memory savings can be significant.

Even with aggressive optimization techniques, state explosion can still result in properties that cannot be verified on a system because of the amount of memory needed to store all of the states of that system. Fingerprinting is another technique that can be used to reduce the memory needed to store the states of a system. With fingerprinting, a fingerprint for a state in the system is stored rather than the entire state. Let the function *fingerprint* compute the fingerprint of a state. Then, for states s_1 and s_2:

[1]For simplicity, we are assuming that a system state is made up of just the values of variables. We are ignoring, for example, the program counter for each task.

	NEG	ZERO	POS
NEG	NEG	NEG	NEG, ZERO, POS
ZERO	NEG	ZERO	POS
POS	NEG, ZERO, POS	POS	POS

Figure 2.2: Addition for abstracted integers

- The number of bits needed to store $\mathit{fingerprint}(s_1)$ should be less than the number of bits needed to store s_1,
- If states s_1 and s_2 are equivalent then $\mathit{fingerprint}(s_1) = \mathit{fingerprint}(s_2)$, and
- If $\mathit{fingerprint}(s_1) = \mathit{fingerprint}(s_2)$, then s_1 and s_2 are equivalent with a high probability.

This means that two states may be judged to be equivalent even when they are not equivalent, thus FSV tools that use this technique may miss exploring some states and report unsound results. Still, because the fingerprint of a state is smaller than the actual state, using fingerprinting can allow an FSV tool to explore more states of a system than could otherwise be explored [69]. Some FSV tools, like SPIN, make fingerprinting an option.[2] Other FSV tools, such as ZING [6], only use fingerprinting and are therefore inherently unsound.

Abstraction Another technique that can be used to reduce the number of states that need to be generated during reachability analysis is abstraction [40]. Abstraction works by mapping the values of a program data type to a set of abstract values and operations over that data type to operations over the set of abstract values. For example, the integer data type could be abstracted to three values: NEG, which represents a negative integer, ZERO, which represents zero, and POS, which represents a positive integer. Figure 2.2 shows the addition operator for this abstraction. For some values the result of adding two abstract

[2]In SPIN, fingerprinting is called bit-state hashing or the supertrace technique.

1) Abstract: An abstract model of the system is built.
2) Verify: The model is analyzed to determine if the desired property holds. If the property holds, then this is reported and the CEGAR algorithm stops. Otherwise, a counterexample is produced.
3) Refine: The counterexample is analyzed to determine if it is a feasible or an infeasible counterexample. If the counterexample is feasible, this is reported and the CEGAR algorithm stops. If the counterexample is infeasible, the model is refined automatically to eliminate the infeasible counterexample and the CEGAR algorithm goes back to the abstraction phase, where a new, more precise model of the system is built.

Figure 2.3: CEGAR phases

integers is unique. For example, adding two positive integers always results in a positive integer. For other values, the result of adding two abstract integers is not unique. For example, adding a positive integer to a negative integer may result in a positive integer, zero, or a negative integer depending on the original unabstracted integer values. During verification, if an operation over an abstracted data type results in multiple values, then to be conservative the FSV tool must nondeterministically choose between all possible results of that operation.

Some tools require that abstractions be manually applied to a system. For example, abstractions in SPIN need to be written by hand and are not automatically checked for correctness [71]. Other tools provide more automated support. For example, Bandera [39] provides support for automatically checking abstractions for correctness and applying abstractions to program data types [44]. The use of abstractions in Bandera has allowed them to verify some properties using Java PathFinder (JPF) [111] that could not be verified on the original unabstracted system [44]. Still, the abstraction mechanism in Bandera requires that the analyst manually select which data types to abstract.

An even more automated abstraction approach is Counterexample Guided Abstraction Refinement (CEGAR) [33]. One tool that uses CEGAR is the Berkeley Lazy Abstraction

Software verification Toolkit (BLAST) [66], which starts with an abstract model of the system and iteratively adds precision to that model. The refinements BLAST makes to the model to add precision are performed automatically based on the counterexamples that were previously returned by the verifier. To be more specific, CEGAR proceeds in three phases, shown in Figure 2.3. Unlike other CEGAR tools, in which a new model is built each time the abstract phase is performed, BLAST performs abstraction lazily. This means that in the refinement phase if the counterexample is infeasible, then a state in the model called the pivot state is identified. The pivot state is the first state in the infeasible counterexample which does not have a concrete counterpart in the unabstracted system, in other words, the state in the counterexample in which the counterexample first becomes infeasible. Then, when abstraction is performed, the model before the pivot state is kept unchanged, but the model after the pivot state is discarded and made more precise using the automatically selected refinements. This allows different parts of the model to have different levels of precision and allows better reuse of artifacts and information between the different phases of CEGAR.

2.1.2 Data-Flow Based

In data-flow analysis, a model of the system is analyzed to determine what facts about that system are true at each node in the model [63]. Data-flow analysis is often used for software optimization, but also has a long history in the realm of verification. Early systems were limited to checking for a fixed set of properties over sequential code [77,91,101]. The Cecil/Cesar system generalized this approach and was capable of checking user-defined properties of sequential Fortran software [89, 90]. FLAVERS builds upon this work, extending it to handle concurrent Ada [43] and concurrent Java [86], and also adding in the ability to improve the precision of the model through constraints. FLAVERS also makes use of some optimizations described in the previous section, including partial-order optimization [87] and symbolic representation of the states [106].

SLAM [13] uses data-flow analysis to check user-defined properties of sequential Boolean programs, programs with only Boolean variables, using the Bebop model checker [12]. It was designed to verify C software, which can be automatically translated into Boolean programs [11]. SLAM uses CEGAR in its analyses and uses a tool called Newton to determine how the model should be refined during the CEGAR process. Although the domain of Boolean programs is limited, SLAM has been successful for checking properties of Windows device drivers.

ESP performs data-flow analysis of user-defined properties of sequential Java systems [41]. It reduces the complexity of analysis by only accurately modeling branches of a system when the state of the property differs on the different arms of the branch. This approach, however, can lead to ESP reporting that a property does not hold when it actually does. Unfortunately, there is no support in ESP for improving the precision of the model to remove such incorrect results.

More recently, BLAST has been modified to use data-flow analysis to verify properties of sequential systems [46]. Often the refinement performed in CEGAR is based on Boolean predicates. For example, if to prove a property the verifier needs to know whether or not a variable x was equal to another variable y, then the predicate "x==y" is added to the model. Then, in the next verify phase, the verifier would keep track of whether or not the value of x equaled the value of y. Rather than just determining which Boolean predicates should be added to the model during the refinement phase, BLAST now learns which pointers and which parts of the heap need to be modeled without expressing these refinements as Boolean predicates. By not forcing refinement to just be based on Boolean predicates, an improvement in efficiency can be seen over CEGAR based just on Boolean predicates.

2.1.3 SAT Based

SAT based verifiers translate the system under analysis and the property to be verified into a Boolean formula. These verifiers then use a SAT solver to determine whether or

not the formula has a solution. From this, it can be determined whether or not the model satisfies the property being verified.

The Alloy Analyzer [74] checks systems written in Alloy [75]. Systems in Alloy are translated into a Boolean formula which is analyzed by a SAT solver to determine whether or not the formula has a solution. The model is bounded during translation so that the Boolean formula only allows a fixed number of objects to be allocated and only allows each loop to be iterated a fixed number of times. Since the Alloy Analyzer uses a bounded model, it may not find a property violation in a system if the violation only exists in a system configuration that exceeds the specified bounds, but the Alloy Analyzer is guaranteed to find a property violation if one exists within the specified bounds.

MAGIC (Modular Analysis of proGrams in C) is a tool that uses CEGAR to verify user-defined properties of sequential C software [22]. MAGIC uses Labeled Transition Systems as its model and checks for weak simulation [84] between a system and the desired properties of that system. This check is done by creating a Boolean formula that has a solution if the system weakly simulates the properties being verified. To obtain scalability, MAGIC requires that the analyst provide procedure abstractions (PAs) to summarize the effects of procedure calls. PAs allow MAGIC to analyze recursive software and are used in two ways. First, PAs are used as properties to ensure that each PA accurately summarizes the procedure it corresponds to. Second, when a procedure call is encountered, the PA corresponding to that call is used to determine the effect of the call on the state of the system.

2.1.4 Integer Necessary Condition Based

INCA (Integer Necessary Condition Analyzer) does not represent the states of a system explicitly, but rather models the system and property as a set of integer linear equations [8]. An integer linear programming solver is then used to verify the property by determining whether or not the set of equations has a solution. Since the reachability graph for

Input: A system R made up of subsystems $S_1, S_2, \ldots, S_n$

Output: A model M that is semantically equivalent to R

Algorithm:

- Let G be the set $\{S_1, S_2, \ldots, S_n\}$
- While $|G| > 1$
  1. Select $F \subseteq G$, where $|F| \geq 2$
  2. Compute $Q = \|_{T \in F} T$
  3. Hide events in Q that are "local" to Q
  4. Minimize Q, maintaining semantic equivalence
  5. Let $G = (G \setminus F) \cup \{Q\}$
- Return as M the only element left in G

Figure 2.4: Typical compositional construction algorithm

the system is never built, INCA can sometimes verify properties of large systems very quickly [9, 10].

2.2 Compositional Analysis

While all of the previously described FSV techniques use different approaches and optimizations to make checking properties tractable, the state-explosion problem can still result in verifications that require a significant amount of time and memory. To reduce the cost of verification, compositional analysis techniques advocate a "divide-and-conquer" approach to verification. We classify compositional analysis techniques into two categories, as proposed in [56], which we call compositional construction and compositional reasoning. Compositional construction techniques are those in which a model is built for the system that is (hopefully) smaller than the full reachability graph, but is semantically equivalent to the full reachability graph with respect to the property being verified. Compositional reasoning techniques are those in which a model for the entire system is not built.

2.2.1 Compositional Construction

Most compositional construction techniques share a common core algorithm, shown in Figure 2.4 [56]. In this algorithm, a model M is built for a system R made up of subsystems $S_1, S_2, \ldots, S_n$. These subsystems are put into a set G, and a while loop is iterated until the size of G is one. Once there is only one element in G, the model M being built is the only element left in G. The while loop iterates selecting some subsystems in G (line 1), building a model of those subsystems[3] (line 2), hiding events local[4] to just the selected subsystems (line 3), minimizing the model[5] built while maintaining semantic equivalence with respect to the property being checked (line 4), and finally replacing the selected subsystems with the newly created model for those subsystems (line 5).

When compositional construction is used, the order in which the subsystems are selected in line 1 can have significant impact on the amount of memory needed to perform the verification. Several heuristics have been proposed to guide the selection of subsystems. Both Sabnani et al. [102] and Tai et al. [105] proposed using the amount of communication between subsystems as a heuristic. Alternatively, Yeh and Young [112] use the hierarchy of modules in Ada software to guide their process. Unfortunately, there are examples for which there is no good order for selecting subsystem and the size of the intermediate model Q computed in line 3 can be exponentially larger than the size of the original subsystems. Several researchers have proposed using constraints to reduce this intermediate state-explosion problem.

Graf and Steffen proposed an approach which allows the analyst to specify interfaces to different subsystems [56]. These interfaces act as constraints to reduce the size of the intermediate models. If an incorrect interface is written, the analysis will never report that

---

[3] In line 2, the parallel composition operator $\|$ is used to combine two subsystems that run in parallel with each other. It is both associative and commutative.

[4] A local event is an event that occurs only in the subsystems selected in line 1.

[5] Since some events were hidden in line 3, this minimization will hopefully produce a smaller model.

a property holds if it does not hold. The analysis, however, might report that a property does not hold even if it does hold.

To avoid errors introduced by incorrect manually written constraints, Cheung and Kramer show how in LTSA constraints can automatically be computed based on the behavior of other subsystems being analyzed [29]. Later, they showed how user-specified constraints could be incorporated [30]. Unlike the approach of Graf and Steffen, their technique will detect an incorrect constraint and report this as an error. Although originally developed for safety properties, this approach was later extended to support liveness properties [28].

Bultan et al. also proposed building constraints automatically, but instead of basing the constraints on the behavior of other subsystems, they build constraints based upon the negation of the property being verified [20]. Since the property being verified is expected to hold, this approach only models the violating subset of the state space, possibly reducing the size of the model that is built.

Clarke et al. describe a framework for compositional minimization that automatically computes interfaces of subsystems [32]. The interface for a subsystem is computed by hiding all local events to that subsystem and then minimizing the result. An interface for a subsystem can be used to replace that subsystem when verifying properties. While the examples in this paper group subsystems together to compute their interface (akin to line 1 in Figure 2.4), this paper only gives some guidelines on how such grouping is to be done.

Rather than build constraints Cheng et al. propose an approach where the system being analyzed is refactored to make it more amenable to compositional construction [27]. This approach is not always applicable and its applicability can be affected by the data structures used in the system [26]. Furthermore, even if a way to refactor the system can be found, a suitable order for selecting subsystems in line 1 is still needed, something the authors do not address.

$$\frac{\begin{array}{l}\langle true\rangle\ S_1\ \langle A_1\rangle \\ \langle A_1\rangle\ S_2\ \langle A_2\rangle \\ \langle A_2\rangle\ S_1\ \langle A_3\rangle \\ \quad\vdots \\ \langle A_n\rangle\ S_{(n \bmod 2)+1}\ \langle P\rangle\end{array}}{\langle true\rangle\ S_1 \parallel S_2\ \langle P\rangle}$$

(a) Alternating chain

$$\frac{\begin{array}{l}\langle true\rangle\ S_1\ \langle A_1\rangle \\ \langle A_1\rangle\ S_2\ \langle A_2\rangle \\ \langle A_2\rangle\ S_3\ \langle A_3\rangle \\ \quad\vdots \\ \langle A_{n-1}\rangle\ S_n\ \langle P\rangle\end{array}}{\langle true\rangle\ S_1 \parallel \cdots \parallel S_n\ \langle P\rangle}$$

(b) n-subsystem chain

$$\frac{\begin{array}{l}\langle true\rangle\ S_1\ \langle A_1\rangle \\ \langle A_1\rangle\ S_2\ \langle P_1\rangle \\ \langle true\rangle\ S_2\ \langle A_2\rangle \\ \langle A_2\rangle\ S_1\ \langle P_2\rangle\end{array}}{\langle true\rangle\ S_1 \parallel S_2\ \langle P_1 \wedge P_2\rangle}$$

(c) Property conjunction

$$\frac{\begin{array}{l}\langle A_1\rangle\ S_1\ \langle P\rangle \\ \langle A_2\rangle\ S_2\ \langle P\rangle \\ \overline{A_1} \cap \overline{A_2} = \emptyset\end{array}}{\langle true\rangle\ S_1 \parallel S_2\ \langle P\rangle}$$

(d) Circular

Figure 2.5: Alternative assume-guarantee rules

2.2.2 Compositional Reasoning

Unlike compositional construction techniques in which a model is built for the system under analysis, with compositional reasoning techniques a model for the entire system is not built. Assume-guarantee reasoning [78, 97] is the most common compositional reasoning technique. In this section, we will discuss some alternative assume-guarantee rules, some tools that make use of assume-guarantee reasoning, and some techniques for automated assumption generation.

Assume-Guarantee Rules In Figure 1.1 we presented the simplest assume-guarantee rule and other more complicated rules have been developed, some of which are shown in Figure 2.5. Figure 2.5a shows a rule for verifying a property on a system with two subsystem by using multiple assumptions [58, 103]. A similar rule for a system with n subsystems is shown in Figure 2.5b. One challenge with using both of these rules is that they require finding more than one assumption. In [32], a framework for assume-guarantee reasoning is presented as well as several assume-guarantee rules. One of these rules is shown in Figure 2.5c. While it is an interesting generalization of the simple rule shown in

Figure 1.1, it requires that the property being verified is written as the conjunction of two other properties and requires finding two assumptions.

Circular assume-guarantee reasoning rules (sometimes called symmetric rules) have also been suggested. An example of this type of rule is shown in Figure 2.5d [14]. In this rule, $\overline{A_1}$ denotes the complement of A_1. Circular rules are difficult to develop and use, because no circular rule without some side condition can be both sound and complete [81]. Intuitively, a side condition is a premise that breaks the circularity of the premises or is not expressible as an assume-guarantee triple. The rule shown in Figure 2.5d is both sound and complete, but its third premise is a side condition which is not expressible as an assume-guarantee triple.

Tools that Use Assume-Guarantee Reasoning Dwyer shows how FLAVERS can be used for checking assume-guarantee properties of Ada software [42]. This work only deals with interactions resulting from rendezvous, while the approach we used also deals with shared variables. Also, Dwyer's work requires that analysts manually develop the assumptions, unlike our work in which assumptions are learned automatically.

Henzinger et al. performed a case study [67] in which they used assume-guarantee reasoning to verify several protocols using MOCHA [4]. Their use of assume-guarantee reasoning allowed them verify larger systems than could be verified monolithically. This work, however, requires that assumptions be developed manually.

Fournet et al. incorporated assume-guarantee reasoning for message passing systems into ZING [49]. This approach checks that each message that is sent is received and that no subsystem blocks waiting for a message that will never be sent but requires that the assumptions (called contracts) be written manually. They did not compare their approach to monolithic verification, however, so it is unknown if their use of assume-guarantee reasoning offers a benefit over monolithic verification.

CALVIN performs compositional analysis using a theorem prover to verify user-defined properties of concurrent Java software [47]. CALVIN analyzes each thread in the system

individually and requires that the analyst provide an environment assumption to describe the effects of other threads on the state of the system. In order to handle loops, either the analyst needs to provide a loop invariant or CALVIN will unroll each loop a small number of times. The latter approach is unsound, but has been effective in detecting errors in a number of systems. To check properties, they use an automated theorem prover that requires no input from the analysts. However, the theorem prover may fail to terminate.

Automated Assumption Generation One difficulty with applying assume-guarantee reasoning has been finding suitable assumptions to complete assume-guarantee proofs. There has been recent work on addressing this problem and several techniques have been developed to compute assumptions automatically.

Inverardi et al. showed how assumptions could automatically be computed [73] to check for freedom from deadlock for systems specified in CHAM [15]. Assumptions are discharged by finding one or more components that satisfy each generated assumption. This approach has a better worst-case memory bound than the standard reachability analysis but it does not improve upon the worst-case time bound for reachability analysis. This work, however, does not provide an empirical evaluation of the approach, so it is unknown if it offers a benefit over monolithic verification.

Data mining techniques [2] have been used by de la Riva et al. [99] for building assumptions for SA/RT models (which resemble StateCharts [60]). This work was later extended to allow assumptions for multiple components to be generated simultaneously using the assumptions that have already been generated to prune the search space [98]. In the worst case, though, this work still requires building a reachability graph for each subsystem. This approach was applied to one system, but they do not report on how their approach compared to monolithic verification, so it is not known if it provides an advantage.

Giannakopoulou et al. showed how assumptions could be built automatically for LTSA [52]. This approach requires that a model be built for the subsystem for which an assumption is generated. Since this work is for LTSA, this subsystem model can be

built using compositional construction, which can reduce the cost of assumption generation. Still, it can still be expensive to build models for an entire subsystem. In later work we showed that the approach we used, based on the L* algorithm, can reduce the memory cost of assumption generation [36]. This learning approach was later extended to handle circular assume-guarantee rules [14]. Chaki et al. implemented this approach and their experimental results show that while using circular rules can reduce the memory needed compared to using the non-circular rule presented in [36], using circular rules can increase the time needed [21].

Jeffords and Heitmeyer show how an invariant generation tool can be used to generate invariants for subsystems that can be used to complete assume-guarantee proofs [76]. These invariants are "facts" that are guaranteed to be true about a given subsystem. While the assume-guarantee proof rules they use are sound and complete, the invariant generation algorithm they use is not guaranteed to produce invariants that will complete an assume-guarantee proof even if such invariants exist. Still, in their experiments, their approach provided a time savings over monolithic verification. They do not, however, report on memory usage.

BLAST incorporates thread-modular reasoning into its CEGAR process and learns assumptions automatically to detect race conditions in concurrent C software [65]. This approach is based on a technique that was proposed for CALVIN but was never implemented [48]. Thread-modular reasoning in BLAST is incomplete, meaning there are some properties that can be verified by monolithic analysis but not by their compositional approach. Still, they used their approach to successfully prove properties of several systems. They do not report on how their approach compared to monolithic verification, so it is not known if it provides a memory savings or a time savings.

Chaki et al. developed the L^T algorithm, based on the L* algorithm, which can learn deterministic tree automata [23]. They show how this algorithm can be used to learn assumptions to complete assume-guarantee proofs for checking simulation conformance. They

looked at 8 properties of 1 system that had only 2 subsystems and were able to verify simulation conformance compositionally on some examples that could not be checked monolithically. Since the system they verified only had two subsystems, they did not have to address the decomposition problem, as we did in our study.

Alur et al. adapted the learning approach we used for NuSMV [5]. In addition to representing the subsystems symbolically using BDDs, they also represent the learned assumption and the data structures used by the L* algorithm symbolically as well. They found some properties that could be verified using assume-guarantee reasoning but not verified monolithically. Some of these properties were for scalable systems and, on these systems, they were able to increase the size of the system that could be verified by 1 or 2. They did not determine if assume-guarantee reasoning could scale farther than this, but, based on their data, it seems unlikely. They also reported on one property where assume-guarantee reasoning used more time and more memory than monolithic verification.

CHAPTER 3

BACKGROUND

This chapter gives a description of FLAVERS and LTSA, the two verifiers we will use in our study. It also describes the L* algorithm, the algorithm we will use to learn assumptions to complete assume-guarantee proofs. To explain the two verifiers, we will use the example shown in Figure 3.1, which shows an elevator system in Ada. The system has two tasks, a car and a controller, and a variable *x* that is shared by both tasks. In this system, the variable *x* is first set by both tasks and then the tasks rendezvous on sync.[1] This ensures that *x* is set, but uses a race condition so that the variable of *x* could be either true or false when the rendezvous sync occurs. If *x* is true when sync occurs, then, through the rendezvous open_doors and close_doors, the controller instructs the car to first open and then close its doors. Once that is done, the rendezvous move_car occurs, which causes the car to move. If *x* is false when sync occurs, then only the rendezvous move_car occurs. Note that this system assumes that the car's doors are closed to start.

3.1 FLAVERS

FLAVERS is a finite-state verification tool that use data-flow analysis to check user-defined safety properties of systems. In this section, we describe for FLAVERS how properties are specified, how systems are modeled, and how properties are verified.

---

[1]Rendezvous are a form of synchronous communication used in Ada. A task may call a named entry in another task. Execution of the calling task is blocked until the called task accepts the call and the two task complete the rendezvous. During a rendezvous, information may be passed in both directions. Both the calling task and accepting task continue execution after their rendezvous is completed.

```
task body car                 task body controller
  x := true;                    x := false;
  accept sync;                  car.sync;

  if (x) then                   if (x) then
    accept open_doors;            car.open_doors;
  end if;                       end if;

  if (x) then                   if (x) then
    accept close_doors;           car.close_doors;
  end if;                       end if;

  accept move_car;              car.move_car;
end car;                      end controller;
```

Figure 3.1: Elevator system in Ada

3.1.1 Specifying Properties

The properties that FLAVERS verifies need to be expressed as sequences of events that should (or should not) happen on any execution of the system. A property can be expressed in a number of different notations, but is translated into a *Finite-state Automaton* (FSA). Formally, an FSA is a tuple, $F = \langle \Sigma, Q, \Delta, A, q \rangle$ where:

- Σ is a set of events, called the alphabet of the FSA,
- Q is the finite set of states,
- $\Delta : Q \times \Sigma \rightarrow Q$ is the total transition function,
- $A \subseteq Q$ is the set of accepting states, and
- $q \in Q$ is the initial state.

We use $\mathscr{L}(F)$ to denote the set of all strings accepted by an FSA F, meaning the sequences of events that occur on paths from the initial state to an accepting state. FSAs are closed

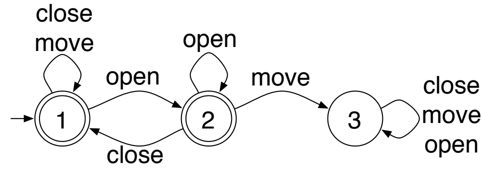

Figure 3.2: Example property for FLAVERS

under complement, meaning if F is an FSA, then there exists an FSA that accepts every string rejected by F and that rejects every string accepted by F. Thus, for simplicity we will assume that properties in FLAVERS always describe event sequences that should happen.

One property that should hold on the elevator system is that the car should never move while its doors are open. The FSA for this property is shown in Figure 3.2, in which the events close, move, and open correspond to the rendezvous close_doors, move_car, and open_doors, respectively. Each of the three states in this FSA is represented by a circle. The initial state, state 1, is denoted by a state with an incoming arrow that does not originate at another state. The accepting states, states 1 and 2, are denoted by states with two concentric circles. State 1 represents the state in which the car's doors are closed, state 2 represents the state in which the car's doors are open. The transition on move from state 2 to state 3 represents the car moving when its doors are open. State 3, the only non-accepting state, represents a violation of the property, since the only way to enter it is having the elevator move while the car's doors are open.

3.1.2 System Model

To verify a property, FLAVERS uses a model of the system based on annotated *Control Flow Graphs* (CFGs). Annotations are placed on nodes of the CFGs to represent events that occur during execution of the actions associated with a node. Formally, a CFG is a labeled directed graph $C = \langle N, n_i, n_f, E, Label \rangle$, where:

- N is the finite set of CFG nodes,

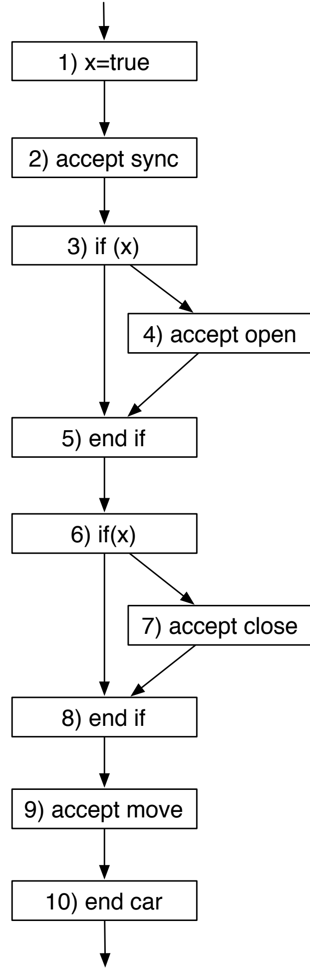

Figure 3.3: CFG for task car

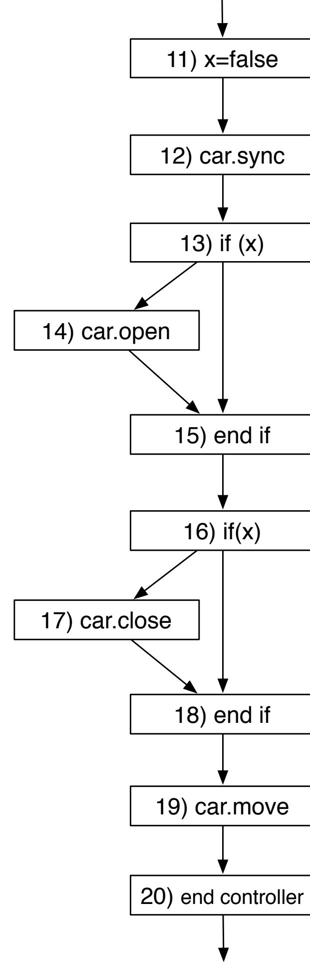

Figure 3.4: CFG for task controller

- $n_i \in N$ is the initial node,
- $n_f \in N$ is the final node,
- $E \subseteq N \times N$ is the set of directed edges, and
- $Label : N \rightarrow \Sigma_{sys} \cup \{\tau\}$ is the function that labels each node with its associated event, where Σ_{sys} is the set of all events for a given system and τ is a special "empty" event that is associated with a node that does not have an event from Σ_{sys} associated with it.

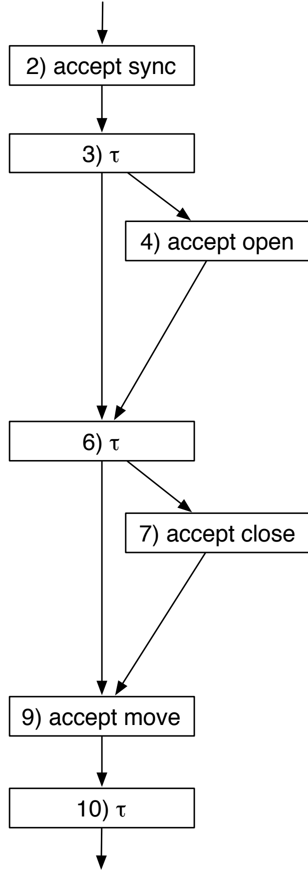

Figure 3.5: Refined CFG for task car

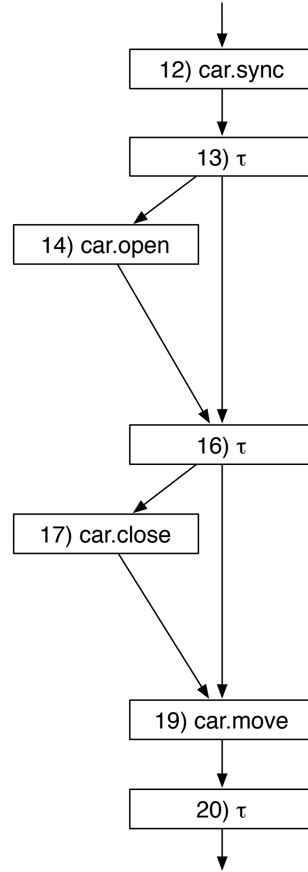

Figure 3.6: Refined CFG for task controller

The CFGs for the car and controller tasks from Figure 3.1 are shown in Figures 3.3 and 3.4, respectively.

Since the efficiency of FLAVERS' verification is dependent on the size of the model it analyzes, CFGs are refined to remove nodes that are not of interest to the property being analyzed. This refinement is safe so long as there is a weak bisimulation relationship [84] between each original CFG and its corresponding refined CFG. Let Σ_I be the alphabet of interest, which must at least contain the events in the alphabet of the property and events related to intertask communication via rendezvous. Since the property in Figure 3.2 only uses the events close, move, and open, and these events correspond to rendezvous, the

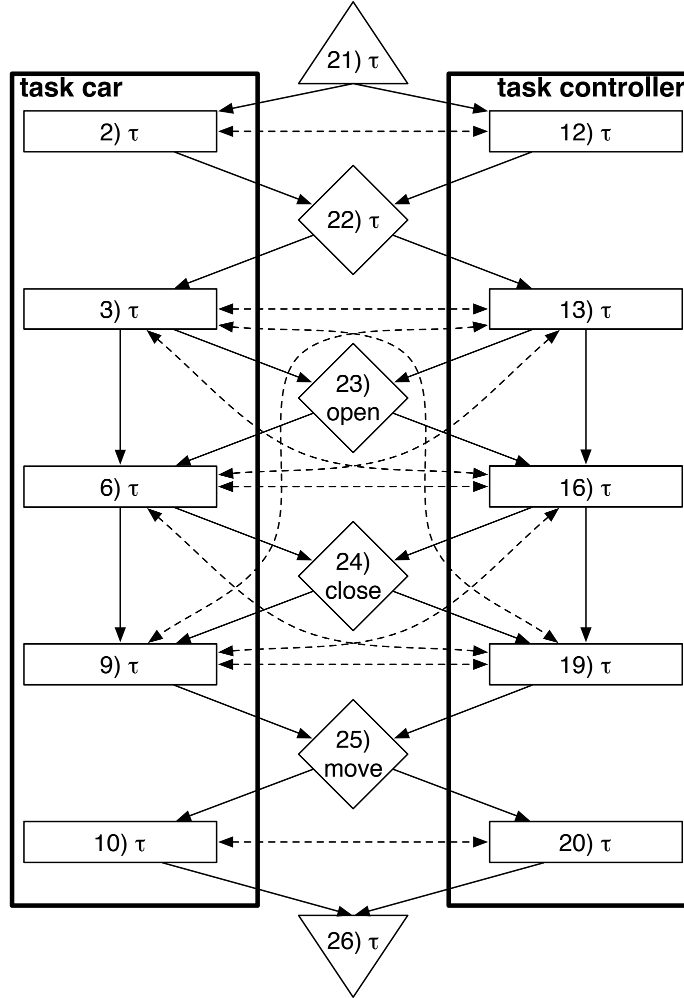

Figure 3.7: TFG for the elevator system

only nodes in the CFGs that are needed are those representing intertask communication. Figures 3.5 and 3.6 show the CFGs for the car and controller tasks refined with respect to the property shown in Figure 3.2. Note that the refinement algorithm also relabels with τ those nodes that do not have an event label in Σ_I but are necessary for control flow reasons.

Since a CFG corresponds to the control flow of a sequential system, this representation is not sufficient for modeling a concurrent system. FLAVERS uses a *Trace Flow Graph* (TFG) to represent concurrent systems. The TFG consists of a collection of CFGs with additional nodes and edges to represent intertask control flow. Formally, a TFG is a labeled directed graph $T = \langle N, n_i, n_f, E, \Sigma_I, Label \rangle$, where:

- N is the finite set of TFG nodes,
- $n_i \in N$ is the initial node,
- $n_f \in N$ is the final node,
- $E \subseteq N \times N$ is the set of directed edges,
- Σ_I is the set of events of interest, and
- $Label : N \rightarrow \Sigma_I \cup \{\tau\}$ is a function that labels each node with its associated event.

For a TFG T, let $\mathscr{L}(T)$ be the language of T, where each element of $\mathscr{L}(T)$ is in $(\Sigma_I)^*$ and is a sequence of non-τ events that occurs on a path from the initial node of T to the final node of T.

The TFG that is built from the CFGs in Figures 3.5 and 3.6 is shown in Figure 3.7. In this figure, nodes 21 and 26 are the unique initial node and final node of the TFG, respectively. To represent intertask communication via rendezvous, extra nodes and edges are added to the TFG. For example, node 23 and its incident edges represent the rendezvous open_doors and replace nodes 4 and 14 from the refined CFGs. Additional edges are needed to represent the possible flow of control between nodes in different tasks due to task interleaving. These *May Immediately Precede* (MIP) edges are computed by the *May Happen in Parallel* (MHP) algorithm [85] and are shown as dashed edges in Figure 3.7. Note that not every pair of nodes from the car and controller tasks are connected by a MIP edge. For example, the MHP algorithm can determine that nodes 9 and 20 cannot happen in parallel because the rendezvous car_move (node 25) must happen between them.

A TFG is an over-approximation of the sequences of events that can occur when executing a system. Every sequence of events that can occur on an execution of the system has a corresponding path in the TFG. To help keep the size of the TFG small, there usually are paths in the TFG that do not correspond to any actual execution of the system. Full details of TFG construction and the proof of TFG conservativeness are given in [43].

3.1.3 Verifying Properties

FLAVERS uses an algorithm called state-propagation to determine if a property represented by an FSA $P = \langle \Sigma_P, Q_P, \Delta_P, A_P, q_P \rangle$ holds on a TFG $T = \langle N, n_i, n_f, E, \Sigma_I, Label \rangle$. Note that by construction, the alphabet of the property is a subset of the alphabet of the TFG, meaning, $\Sigma_P \subseteq \Sigma_I$. P holds on T if

$$\forall \rho \in \mathscr{L}(T), \rho|_{\Sigma_P} \in \mathscr{L}(P)$$

where $\rho|_{\Sigma_P}$ denotes the projection of ρ onto Σ_P, which is defined as follows: let $\rho = \sigma_0, \sigma_1, \ldots, \sigma_k$ where each $\sigma_i \in \Sigma_I$, then the projection operator retains only those elements of ρ that are in Σ_P, meaning $\rho|_{\Sigma_P} = \sigma_{j_0}, \sigma_{j_1}, \ldots, \sigma_{j_m}$ where

1. $\sigma_i \in \Sigma_P$ if and only if $\exists j_k$ such that $j_k = i$, and
2. the order of elements in the projected sequence is preserved, meaning $j_k < j_{k+1}$.

FLAVERS analyses are conservative, meaning FLAVERS will only report that the property holds when the property holds for all TFG paths. If FLAVERS reports that the property does not hold, this can be because at least one of the violating traces through the TFG corresponds to an actual execution of the system and thus there is an error in the system, in the property, or in both. Alternatively, the property may only be violated on *infeasible paths* through the TFG, paths that do not correspond to any possible execution of the system but are an artifact of the imprecision of the model.

FLAVERS would report that the property shown in Figure 3.2 does not hold on the TFG shown in Figure 3.7 because the property is violated on the path $21 \rightarrow 2 \rightarrow 22 \rightarrow 3 \rightarrow 23 \rightarrow 6 \rightarrow 9 \rightarrow 25 \rightarrow 10 \rightarrow 26$, which corresponds to the event sequence $\langle$open, move$\rangle$. This counterexample is infeasible, however, because the variable x must be true for the edge $3 \rightarrow 23$ to be taken and false for the edge $6 \rightarrow 9$ to be taken. Since the value of x is not changed in the system between nodes 23 and 6, this path cannot occur on an actual execution of the elevator system and is therefore infeasible.

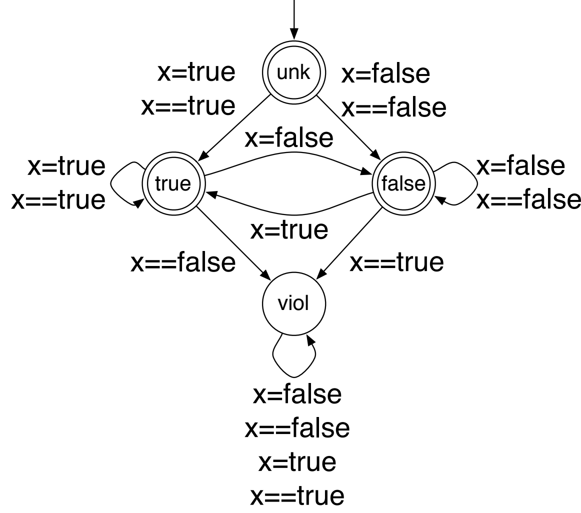

Figure 3.8: Variable automaton for x

3.1.4 Improving Precision

This infeasible path was introduced because all information related to the variable x was removed during CFG refinement. In order to improve the precision of the model and remove some infeasible paths from consideration, the analyst can introduce *feasibility constraints*, also represented as FSAs, to the analysis. An analyst might need to iteratively add feasibility constraints and observe the analysis results several times before determining whether or not a property holds. Feasibility constraints give analysts some control over the analysis process by letting them determine exactly which parts of a system need to be modeled in order to prove a property. To verify a property $P = \langle \Sigma_P, Q_P, \Delta_P, A_P, q_P \rangle$ on a TFG $T = \langle N, n_i, n_f, E, \Sigma_I, Label \rangle$ with feasibility constrains $C_1, \ldots, C_k$ where $C_j = \langle \Sigma_{C_j}, Q_{C_j}, \Delta_{C_j}, A_{C_j}, q_{C_j} \rangle$, state-propagation can still be used, although it has to be modified to track the state of the property and the feasibility constraints at each node of the TFG [88]. P holds on T as constrained by $C_1, \ldots, C_k$ if

$$\forall \rho \in \mathscr{L}(T), \text{ if } \left(\forall 1 \le j \le k, \rho|_{\Sigma_{C_j}} \in \mathscr{L}\left(C_j\right)\right) \text{ then } \rho|_{\Sigma_P} \in \mathscr{L}(P)$$

One type of feasibility constraint is a *Variable Automaton* (VA), which can be used to track a small number of values for a variable. In the elevator system example, the value of the variable x is important to the property being verified, so a VA can be introduced to keep track of its value. In the VA for x, shown in Figure 3.8, events with "=" represent an assignment to x, while the events with "==" represent a test of the value of x. The three accepting states of the VA represent the unknown, true, and false values of x. The one non-accepting state is the *violation state* and is entered when a path is explored that is infeasible because of an operation on x. For example, if x is known to be true and a branch is taken where the value of x is false (i.e., the event x==false occurs), then the violation state would be entered.

To make use of this VA, the TFG from Figure 3.7 needs to be modified so it has nodes with events corresponding to operations on x, as shown in Figure 3.9. In this TFG,[2] each node corresponding to a branch (nodes 3, 6, 13, and 16 from Figure 3.7) has been split into two nodes, one for the true branch (the nodes labeled x==true) and one for the false branch (the nodes labeled x==false). The previously reported counterexample path maps to the path 21 → 1 → 2 → 22 → 3a → 23 → 6b → 9 → 25 → 10 → 26 in the modified TFG and corresponds to the event sequence ⟨x=true, x==true, open, x==false, move⟩. Because the VA for x would transition from the unknown state to the true state on node 1, remain in the true state on node 3a, and transition from the true state to the violation state on node 6b, this path would not be considered during state propagation since it is infeasible. With just the VA for x, FLAVERS would report that the property shown in Figure 3.2 holds and conclude that the elevator's car cannot move while its doors are open.

It is important to note that even when this VA is used in an analysis, there are still infeasible paths in this model that will be considered during state-propagation. For example, any path that starts 21 → 11 → 2 is infeasible because it visits node 2 without having first

[2]In our implementation, nodes 2 and 12 would be refined away, but they have been left in Figure 3.9 for clarity of the presentation.

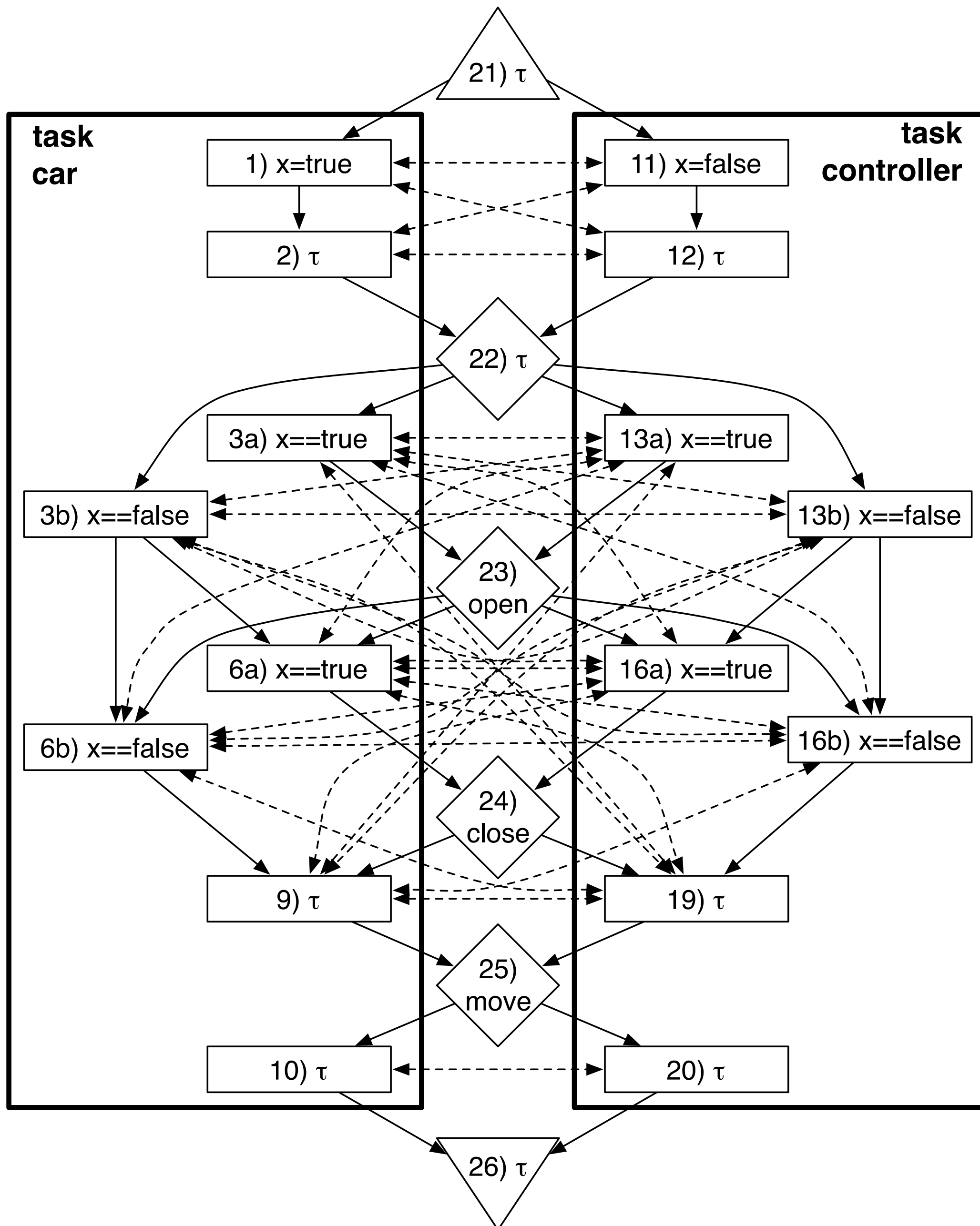

Figure 3.9: TFG with events relating to the variable *x*

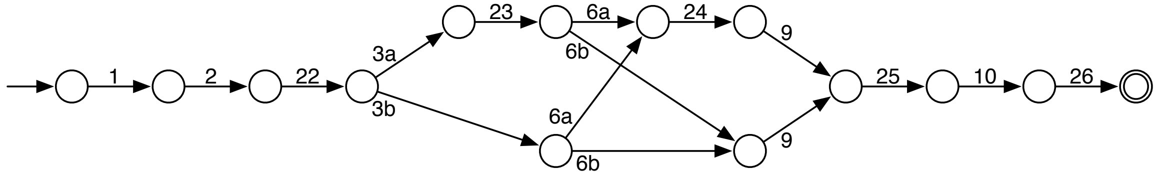

Figure 3.10: TA for task car

visited node 1. *Task automata* (TAs) are another type of feasibility constraint that can be used to remove such infeasible paths from consideration. A task automaton enforces the flow of control within a single task of a system. Unlike VAs, which have transitions based on the annotations on the TFG nodes, TAs have transitions based on the IDs of the TFG nodes. The alphabet of a TA is the set made up of the IDs of every node in the task that the TA models.[3] The TA for the car task is shown in Figure 3.10. Unlike previous FSAs which have been total,[4] we omit transitions from this FSA that cannot ever lead to an accepting state for simplicity. This TA would prevent paths starting $21 \rightarrow 11 \rightarrow 2$ from being considered during verification since, from the initial state of the TA, if any node from the car task other than node 1 is visited, the TA will not accept that sequence. Since the first node visited from the car task on these paths is node 2, this TA would not accept these paths.

The state-propagation algorithm used by FLAVERS has worst-case complexity that is $O\big(|N|^2 \cdot |S_P| \cdot |S_{C_1}| \cdots |S_{C_k}|\big)$, where $|N|$ is the number of nodes in the TFG, $|S_P|$ is the number of states in the property, and $|S_{C_j}|$ is the number of states in the j-th feasibility constraint. If a large number of feasibility constraints are needed, than this worst-case complexity could become large resulting in a high cost for analysis. In our experience, however, properties can often be proven using only a small number of feasibility constraints. Experimental evidence shows that the performance of FLAVERS is often sub-cubic in the size

[3]The initial and final nodes of the TFG are considered to be in every task. Nodes corresponding to rendezvous are considered to be in the two tasks that are participating in the rendezvous.

[4]A total FSA is one which has a transition from every state on every event in its alphabet.

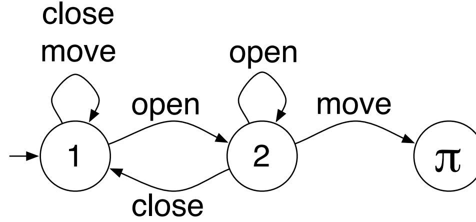

Figure 3.11: Example property for LTSA

of the system [43] and that the performance of FLAVERS is good when compared to other finite-state verifiers [9, 10].

3.2 LTSA

LTSA (Labeled Transition Systems Analyzer) is a finite-state verifier that can prove user-specified properties of sequential and concurrent systems [80]. LTSA can check both safety and liveness properties. Since the assume-guarantee algorithm we use can only handle safety properties, when we talk about properties in LTSA we will henceforth mean only safety properties. In this section, we describe for LTSA how properties are specified, how systems are modeled, and how properties are verified.

3.2.1 Specifying Properties

Properties in LTSA are specified in Finite State Process (FSP), which is described in detail in [80], and are represented using Labeled Transition Systems (LTSs). LTSs are similar to FSAs, but with several differences. Formally, let *States* be the universal set of states, let $\pi \in States$ be the designated error state, let Σ be the universal set of labels, and let $Act = \Sigma \cup \{\tau\}$ where τ denotes an internal action that cannot be observed by the environment of an LTS. Then an LTS L is a four-tuple, $\langle Q, A, \Delta, q \rangle$ where:

- $Q \subseteq States$ is a finite set of states,
- $A = \Sigma_L \cup \{\tau\}$ is the set of actions, where $\Sigma_L \subseteq \Sigma$ denotes the alphabet of L,

- $\Delta \subseteq \{Q \setminus \{\pi\}\} \times A \times Q$ is the transition relation, and
- $q \in Q$ is the initial state of L.

The only LTS that is allowed to have the error state as its initial state is denoted Π and is defined as $\Pi = \langle\{\pi\}, Act, \{\}, \pi\rangle$. An LTS $L = \langle Q, A, \Delta, q\rangle$ transits with action $a \in A$ into an LTS L', denoted as $L \xrightarrow{a} L'$, if:

- $L' = \langle Q, A, \Delta, q'\rangle$ where $q' \neq \pi$ and $(q, a, q') \in \Delta$, or
- $L' = \Pi$ and $(q, a, \pi) \in \Delta$.

The LTS in Figure 3.11 shows the LTS that captures the property that the elevator's car should never move while its doors are open. Since LTSs do not have accepting and non-accepting states as FSAs do, every state except the error state can be thought of as an accepting state in an FSA. Thus, LTSA can only check properties that are prefix closed, i.e. if an event sequence is allowed by a property, then every prefix of that event sequence must also be allowed by that property.

3.2.2 System Model

LTSA also uses LTSs to represent the system being analyzed. Thus, unlike FLAVERS, in which the nodes of the model are labeled with the events of interest, in LTSA the edges (or transitions) of the LTSs are labeled with the events of interest. Figures 3.12 and 3.13 give the LTSs for the car and controller task for the Ada example in Figure 3.1. To model the shared variable x, a third LTS is needed,[5] shown in Figure 3.14.

To build a model for the entire system, individual LTSs are combined using the parallel composition operator ($\|$). The parallel composition operator is a commutative and associative operator that builds an LTS that captures the behavior of two LTSs running in

[5]Unlike the VA shown in Figure 3.8 for the variable x, there is no state in the LTS for the variable x corresponding to the violation state. In LTSA, transitions that are not explicitly shown cannot occur, so, only three states are needed to model this variable.

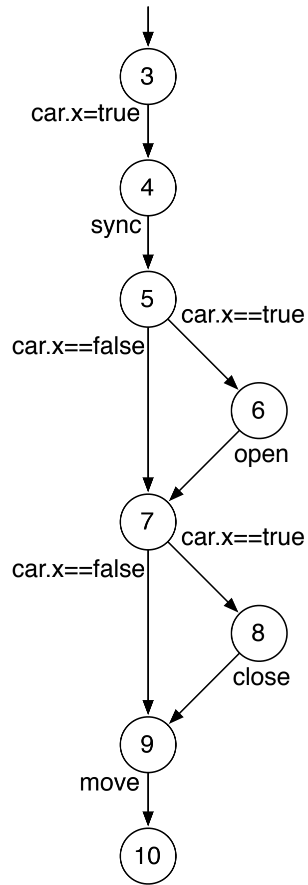

Figure 3.12: LTS for task car

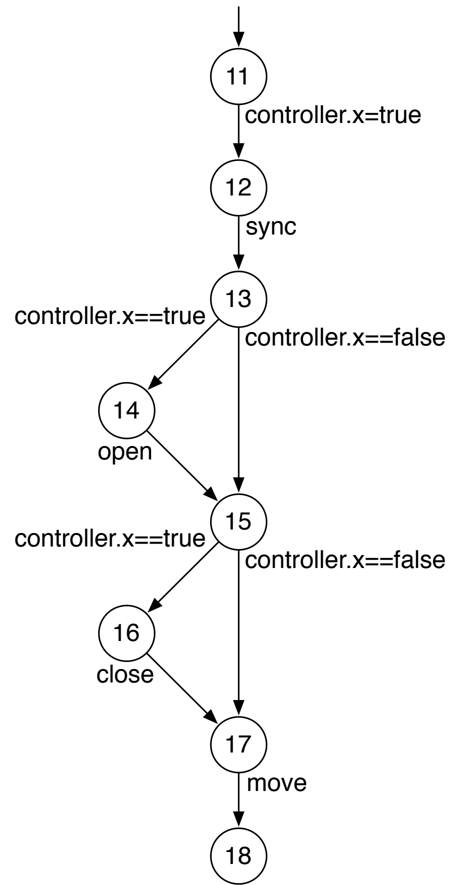

Figure 3.13: LTS for task controller

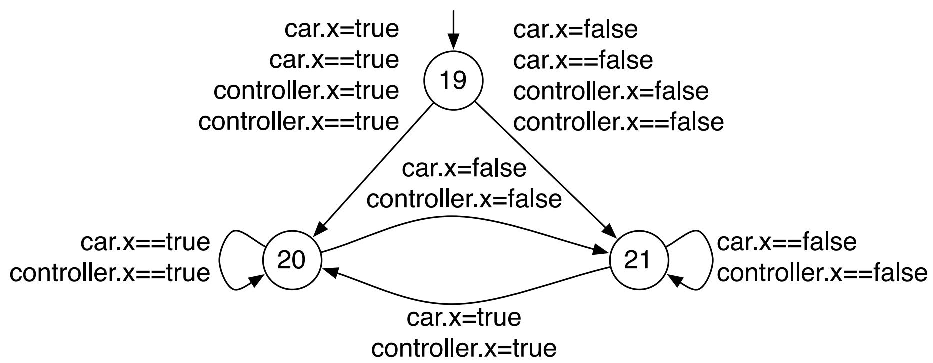

Figure 3.14: LTS for variable x

parallel by synchronizing the events common to both and interleaving the remaining events. The parallel composition of two LTSs $L = \langle Q_L, A_L, \Delta_L, q_L \rangle$ and $M = \langle Q_M, A_M, \Delta_M, q_M \rangle$, denoted $L \parallel M$, is defined as follows. If $L = \Pi$ or $M = \Pi$, then $L \parallel M = \Pi$. Otherwise $L \parallel M = \langle Q_L \times Q_M, A_L \cup A_M, \Delta, (q_L, q_M) \rangle$, where Δ is the smallest relation satisfying the following rules where $a \in Act$:

$$\frac{L \xrightarrow{a} L'}{L \parallel M \xrightarrow{a} L' \parallel M} \quad \text{where } a \notin \Sigma_M$$

$$\frac{M \xrightarrow{a} M'}{L \parallel M \xrightarrow{a} L \parallel M'} \quad \text{where } a \notin \Sigma_L$$

$$\frac{L \xrightarrow{a} L' \text{ and } M \xrightarrow{a} M'}{L \parallel M \xrightarrow{a} L' \parallel M'} \quad \text{where } a \neq \tau$$

Figure 3.15 shows the result of composing the LTSs for the car, controller, and x together using the parallel composition operator. Each state in the composed LTS is labeled with states it corresponds to in each of the three original LTSs. Notice that events (such as sync) which are common to multiple LTSs cause transitions in all LTSs in which those events occur. The state $(2,10,19)$ has an edge to $(3,11,19)$ since there is an edge $2 \to 3$ in the car LTS and there is an edge $10 \to 11$ in the controller LTS that are each labeled with the event sync. Since sync does not occur in the LTS for variable x, it remains in state 19 when this transition occurs. This synchronization mechanism is the reason that the events related to the variable x have been prefixed with the name of the task in which they occur. If these prefixes were not added, edges $3 \to 4$, $5 \to 6$, $11 \to 12$, and $13 \to 14$ would all be labeled with “x==true”. As a result of this, the event “x==true” could only occur in the composed system in a state that corresponded to the car being either in state 3 or 5 and the controller being either in state 11 or 13. Since, in the original Ada system, no such synchronization requirement exists, the prefixes are necessary to make the model accurate.

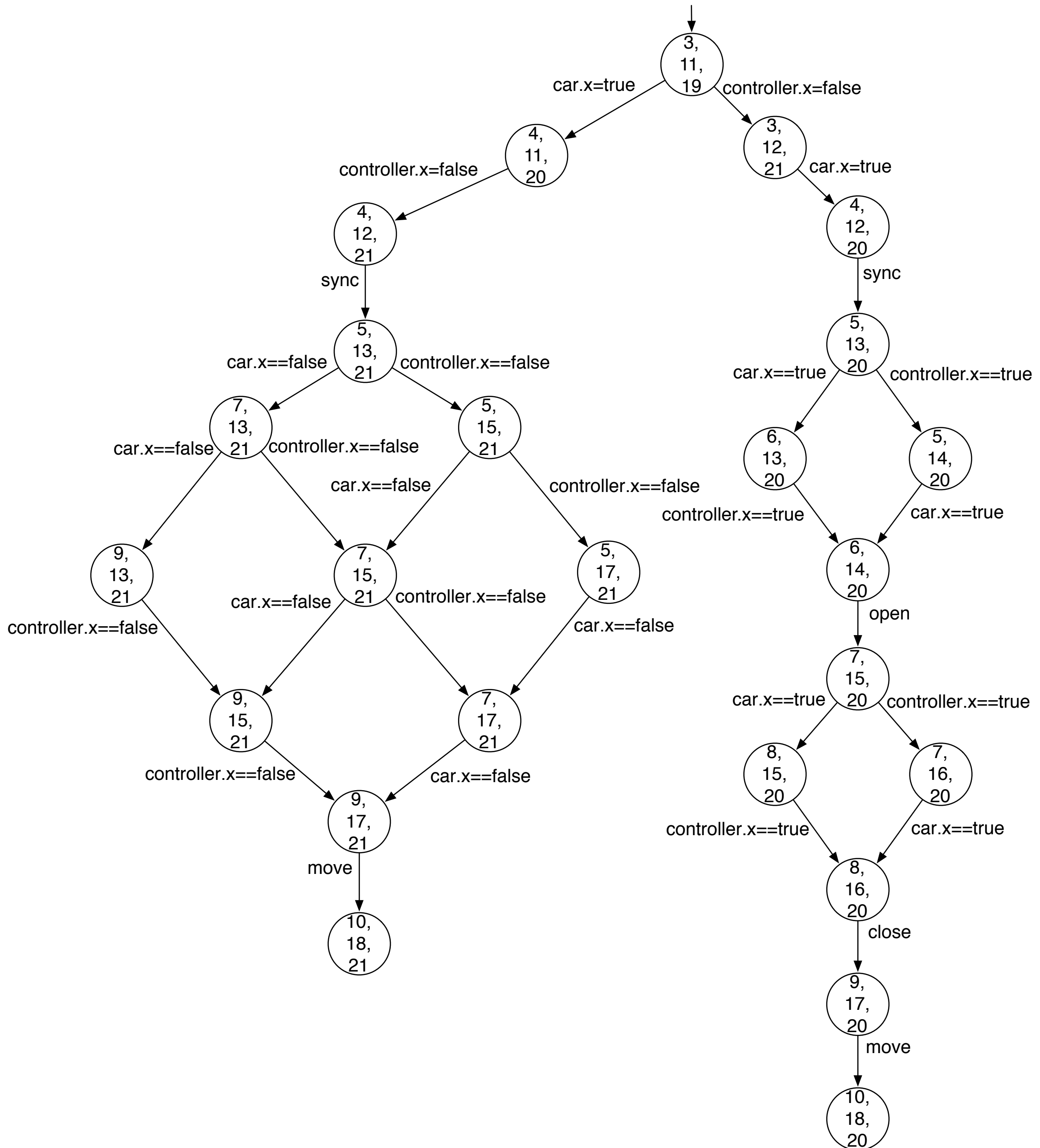

Figure 3.15: LTS for the composed elevator system, (car || controller || x)

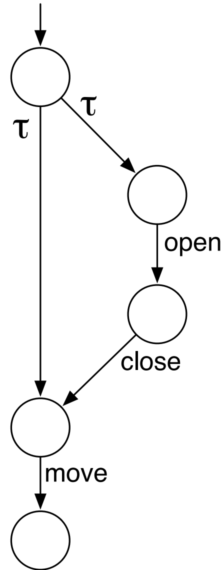

Figure 3.16: Minimal LTS for the elevator system with respect to the property shown in Figure 3.11

LTSA uses a compositional construction technique called *Compositional Reachability Analysis* (CRA), which is based on the hierarchical structure of a system. CRA incrementally computes and abstracts the behavior of composite components using the architecture of the system as a guide to the order to perform the composition [50]. If several LTSs form a subsystem of a larger system and the system is well designed, then those LTSs will likely have many events in common, but have few events in common with other LTSs in the system. With CRA, the LTSs of the subsystem are first composed using the parallel composition operator. Then, the events common to just the LTSs of that subsystem can be hidden, which replaces then with τ. Finally, the resulting system can be minimized. Since the events that are replaced with τ are local to just the LTSs in the subsystem, those events have no effect on other LTSs in the system, so the resulting LTS is a minimal model that captures the interactions of that subsystem with the rest of the system.

Figure 3.16 shows the LTS that would result from taking the LTS in Figure 3.15, hiding all events other than close, move, and open, and performing minimization. This LTS has only 5 states, while the original (shown in Figure 3.15) has 25 states. If the subsystem made

up of the car, controller, and x were used in a larger system that only interacted with this subsystem using the events close, move, and open, the minimal LTS shown in Figure 3.16 could be used instead of the LTS shown in Figure 3.15. Since the former is smaller than the latter, using the former would reduce the cost of verification. While CRA can reduce the cost of verification, state explosion can still occur when several LTSs are composed together.

3.2.3 Checking Properties

The algorithm which computes the parallel composition of two LTSs is the same algorithm used to check properties in LTSA. To check a property, the property LTS is composed into the system via the parallel composition operator. The property holds if, in the final model, there is a not a path from the initial state to the error state π. If the LTS for the property shown in Figure 3.11 and the LTS for the elevator system, either the full version shown in 3.15 or the minimized version shown in Figure 3.16, were combined using the parallel composition operator, the error state would not be reachable. Thus, LTSA would conclude that the property holds on the elevator system.

3.3 The L* Algorithm

The L* algorithm was developed by Angluin [7] to learn an FSA for an unknown regular language and was later improved by Rivest and Schapire [100]. In this work, we use the L* algorithm to learn assumptions to complete assume-guarantee proofs. We used Rivest and Schapire's version of the L* algorithm because it has a better worst-case running time. Let U be an unknown regular language over an alphabet Σ. The L* algorithm learns an FSA that recognizes U by building an observation table through its interactions with a *minimally adequate teacher*, henceforth referred to as a *teacher*. The teacher needs to answer two types of questions, queries and conjectures. A *query* consists of a string $\sigma \in \Sigma^*$ and the teacher returns *true* if $\sigma \in U$ and *false* otherwise. A *conjecture* consists of an

```
1)  let S = E = {λ}
    loop {
2)      Update T using queries
        while (S,E,T) is not closed {
3)          Make S closed by adding an element to S
4)          Update T using queries
        }
5)      Construct candidate DFA C from (S,E,T)
6)      Make the conjecture C
        if C is correct
7)          return C
        else
8)          Add e ∈ Σ* that witnesses the counterexample to E
    }
```

Figure 3.17: The L* algorithm

FSA, C, that the L* algorithm believes will recognize U. The teacher returns *true* if C is correct. Otherwise, the teacher returns *false* and a counterexample, a string in Σ^* that is in the symmetric difference of the language of the conjectured automaton and the language being learned, meaning a string that is one language but not the other. The *observation table* built by the L* algorithm is a tuple, $\langle S, E, T \rangle$, where

- S is a set of prefixes, each in Σ^*,
- E is a set of suffixes, each in Σ^*, and
- T is a function mapping every element in $((S \cup S \cdot \Sigma) \cdot E)$ to either *true* or *false*.

In the definition of T, the operator "$\cdot$" is defined as follows: given two sets of events sequences X and Y, $X \cdot Y = \{xy \mid x \in X \text{ and } y \in Y\}$, where xy represents the concatenation of the event sequences x and y.

The L* algorithm is shown in Figure 3.17. Initially, the L* algorithm sets S and E to $\{\lambda\}$ (line 1), where λ represents the event sequence of length zero. Next, it updates the function T by making queries so that it has a value for every event sequence in

$((S \cup S \cdot \Sigma) \cdot E)$ (line 2). It then checks whether or not the observation table is *closed*. The observation is closed if and only if

$$\forall s \in S, \forall a \in \Sigma, \exists s' \in S, \forall e \in E, T(sae) = T(s'e)$$

If the observation table is not closed, then sa is added to S where $s \in S$ and $a \in \Sigma$ are elements for which there is no $s' \in S$ (line 3). Once sa has been added to S, T needs to be updated (line 4). Lines 3 and 4 are repeated until the observation table is closed.

Once the observation table is closed, a candidate DFA $C = \langle \Sigma, Q, \Delta, A, q \rangle$ is constructed (line 5), with states $Q = S$ and initial state $q = \lambda$. The alphabet of C, Σ, is the alphabet of U. The set A consists of the states $s \in S$ such that $T(s) = \text{true}$. The transition relation Δ is defined as $\Delta(s,a) = s'$ where $\forall e \in E : T(sae) = T(s'e)$. Such an s' is guaranteed to exist when the observation table is closed. The DFA C is presented as a conjecture to the teacher (line 6). If the conjecture is correct, the L* algorithm returns C as correct (line 7), otherwise it receives a counterexample $c \in \Sigma^*$ from the teacher.

The counterexample c is analyzed using a process described below to find a suffix e of c that witnesses a difference between the language accepted by C and the language being learned (line 8). e must be such that adding it to E will cause the next conjectured automaton to reflect this difference. Once e has been added to E, the L* algorithm repeats the entire process by looping back to line 2.

As stated previously, in line 8 the L* algorithm must analyze the counterexample c to find a suffix e of c that witnesses a difference between the language accepted by C, the conjectured automaton, and U, the language being learned. This is done by finding the earliest point in e at which the conjectured automaton and the automaton that would recognize the language U diverge in behavior. This point found by determining where $\alpha_i \neq \alpha_{i+1}$, where α_i is computed as follows:

1. Let p be the event sequence made up of the first i events in c. Let r be the event sequence made up of the events after the first i events in c. Thus, $c = pr$.

2. Run C on p. This moves C into some state q. By construction, this state q corresponds to a row $s \in S$ of the observation table.

3. Perform a query on the event sequence sr.

4. Return the result of the membership query as α_i.

By using binary search, the point where $\alpha_i \neq \alpha_{i+1}$ can be done in $\log(|c|)$ queries, where $|c|$ is the length of c.

To learn a regular language, Rivest and Schapire's version of the L* algorithm performs at most $l-1$ conjectures and $\mathrm{O}\left(kl^2 + l\log m\right)$ queries, where k is the size of Σ, the alphabet of the FSA being learned, l is the number of states in the minimal deterministic FSA that recognizes the language being learned, and m is the length of the longest counterexample returned when a conjecture is made.

CHAPTER 4

USING THE L* ALGORITHM TO LEARN ASSUMPTIONS FOR ASSUME-GUARANTEE REASONING

As described in [36], the L* algorithm can be used to learn an assumption to verify a property P with assume-guarantee reasoning. In this approach, the system under analysis needs to be divided into two subsystems, S_1 and S_2. The L* algorithm is then used to learn an assumption to complete an assume-guarantee proof using the assume-guarantee rule given in Figure 1.1. To do this, a teacher capable of answering the queries and conjectures made by the L* algorithm must be provided. Conceptually, this teacher is the same for both FLAVERS and LTSA, however, differences in the models used by these two verifiers necessitate differences in the implementation of their teachers. In this chapter, a high level of the teacher is presented, followed by a detailed description of its implementation for both LTSA and FLAVERS.

4.1 Implementing the Teacher

4.1.1 Answering Queries

A query posed by the L* algorithm consists of a sequence of events, where each event is in Σ. The teacher must answer true if this sequence is in the language being learned and false otherwise. To answer a query, the model of S_1 is examined to determine if the given sequence results in a violation of the property P. If this results in a violation of the property P, then the assumption needed to make $\langle A \rangle\ S_1\ \langle P \rangle$ true should not allow the event sequence in the query and false will be returned to the L* algorithm. Otherwise, the event sequence is permissible and true will be returned to the L* algorithm.

4.1.2 Answering Conjectures

A conjecture posed by the L* algorithm consists of an FSA that the L* algorithm believes recognizes the language being learned. To answer a conjecture, the teacher needs to find an event sequence in the symmetric difference of the conjectured FSA and the language being learned, if such an event sequence exists. Since the conjectured FSA is a candidate assumption to be used to complete an assume-guarantee proof, conjectures are answered by determining if the candidate assumption makes the two premises of the assume-guarantee proof rule true.

First, the conjectured automaton, A, is checked to see if it satisfies Premise 1. To check this, the model of S_1, as constrained by the assumption A, is verified. If this verification reports that P does not hold, then the counterexample returned by the verifier represents an event sequence permitted by A that also causes S_1 to violate P. Thus, the conjecture is incorrect and the counterexample is returned to the L* algorithm. If the verification reports that the property does hold, then A is good enough to satisfy Premise 1 and Premise 2 can be checked.

Premise 2 states that $\langle true \rangle\ S_2\ \langle A \rangle$ should be true. To check this, the model for S_2 is verified to see if it satisfies A. If this verification reports that A holds, then both Premise 1 and Premise 2 are true, so it can be concluded that P holds on $S_1 \parallel S_2$. If this verification reports that A does not hold, then the resulting counterexample is examined to determine what should be done next.

First, a query is made to see if the event sequence of the counterexample leads to a violation of the property P on S_1. If a property violation results, then the counterexample is a behavior that occurs in S_2 that will result in a property violation when S_2 interacts with S_1, so it can be concluded that P does not hold on $S_1 \parallel S_2$. If a property violation does not occur, then the counterexample is a behavior that occurs in S_2 that will not result in a property violation when S_2 interacts with S_1 and, thus, A is restricting the behavior of S_2

unnecessarily. The counterexample is then returned to the L* algorithm in response to the conjecture.

4.1.3 Correctness and Termination

This approach to assume-guarantee reasoning is correct and will terminate [36]. The learning algorithm will converge on the weakest assumption [52] under which S_1 satisfies P. The weakest assumption, A_w, is the assumption such that:

1. $\langle A_w \rangle\ S_1\ \langle P \rangle$ is true, and

2. If A is an assumption such that $\mathscr{L}(A_w) \subset \mathscr{L}(A)$, then $\langle A \rangle\ S_1\ \langle P \rangle$ is false.

4.2 Implementing the Teacher for LTSA

In this section the implementation of the teacher for LTSA is described. Consider the elevator system with the LTSs car, controller, and x (shown in Figures 3.12, 3.13, and 3.14, respectively). Let P be the property that the elevator's car cannot move while its doors are open, shown in Figure 3.11. Suppose that $S_1 = \text{car} \parallel x$ and $S_2 = \text{controller}$.

4.2.1 The Alphabet of the Assumption

In order to use the L* algorithm to learn an assumption, the L* algorithm must be supplied with an alphabet, Σ. In the assume-guarantee proof, the learned assumption will act as a bridge between S_1 and S_2, so it must contain every event that S_1 and S_2 have in common. In addition, the alphabet must contain all the events that S_2 has in common with the property, so that those events can be suitably constrained by the assumption. Thus, the alphabet for the assumption, denoted Σ_A, is $(\Sigma_{S_1} \cup \Sigma_P) \cap \Sigma_{S_2}$. For the elevator example, $\Sigma_A = \{$close, controller.x==false, controller.x==true, controller.x=false, controller.x=true, move, open, sync$\}$.

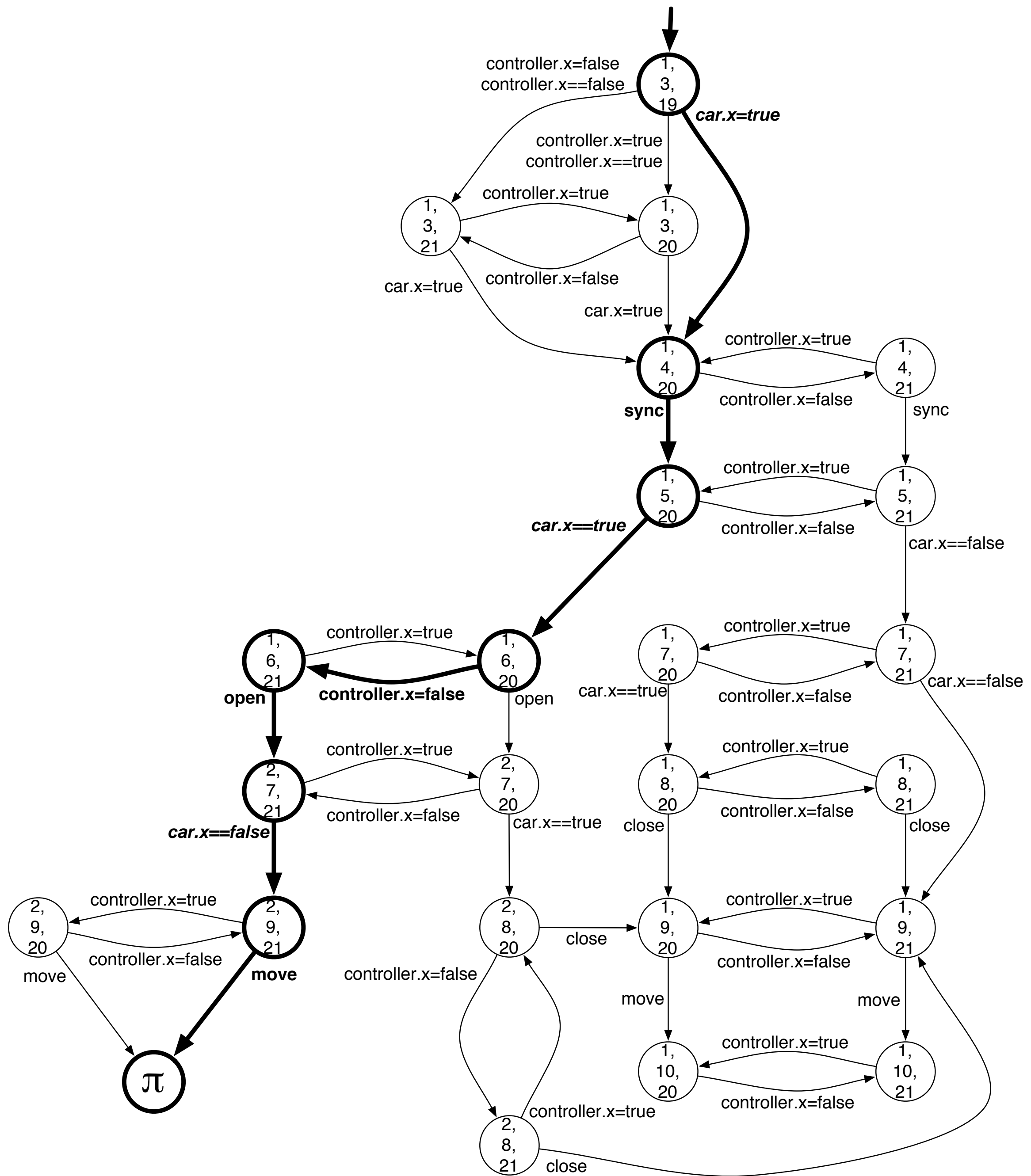

Figure 4.1: LTS for $S_1 \parallel P$, with a counterexample for the query ⟨sync, controller.x==false, open, move⟩ highlighted

4.2.2 Answering Queries

To answer a query, the model of S_1 is examined to determine if the given sequence results in a violation of the property P. If so, then false will be returned to the L* algorithm, otherwise true will be returned to the L* algorithm. To answer a query, $S_1 \parallel P$ can be computed, as shown in Figure 4.1 for the elevator example.[1] Then, it is a matter of determining if the sequence of events in the query can cause the error state π to be reached. Consider the query $\langle$sync, controller.x==false, open, move$\rangle$. The highlighted path in Figure 4.1 shows how that sequence of events can lead to the error state. Note that while there are other events in the path, those shown in italics and prefixed with "car", these events are not part of the query. Since these events are not in the alphabet of the assumption being learned, the assumption cannot put any constraints on the occurrence of these events. Thus, they are free to occur at any point during a search for a path to the error state in response to a query.

4.2.3 Answering Conjectures

There are several steps that need to be performed to answer a conjecture, as described in Section 4.1.2. First, the conjectured automaton, A, is checked in Premise 1, $\langle A \rangle\ S_1\ \langle P \rangle$. Since LTSs resemble FSAs, the conjectured automaton is turned into an LTS in the obvious way. Then, the LTS $A \parallel S_1 \parallel P$ is constructed. If the error state is reachable, the conjecture is incorrect and the sequence of events in the counterexample is returned to the L* algorithm.[2] If the error state is not reachable, then the assumption is sufficient to satisfy Premise 1 and Premise 2 can be checked.

---

[1]For simplicity, all self-loop transitions have been omitted from Figure 4.1. In particular, every state that has an 20 in its label has self-loop transitions on controller.x=true and controller.x==true. Similarly, every state that has a 21 in its label has self-loop transitions on controller.x=false and controller.x==false.

[2]When a counterexample is returned to the L* algorithm, it must be projected onto the alphabet of the assumption being learned.

Premise 2 states that $\langle true \rangle\ S_2\ \langle A \rangle$ should be true. To check this, the conjectured automata can be converted into a property LTS by making the non-accepting trap state of the automata into the error state of a property LTS.[3] Let this property LTS be designated A_P. Then, to check Premise 2, the LTS $S_2 \parallel A_P$ is constructed. If the error state is not reachable, then both Premise 1 and Premise 2 are true, so it can be concluded that the property P holds on $S_1 \parallel S_2$. If the error state is reachable, then, as described previously, a query can be performed to determine if P does not hold on $S_1 \parallel S_2$ or if the L* algorithm needs to continue learning a new assumption.

4.3 Implementing the Teacher for FLAVERS

In this section the implementation of the teacher for FLAVERS is described. Consider the elevator example with the two tasks car and controller shown in Figures 3.3 and 3.4, respectively. Let P be the property that the elevator's car cannot move while its doors are open, shown in Figure 3.2. Suppose that $S_1 =$ car and $S_2 =$ controller.

4.3.1 The Model

To answer queries and conjectures, we need to build TFG models for S_1 and S_2, the two subsystems used in the assume-guarantee proof rule. The TFGs for S_1 and S_2 are similar to the TFGs that FLAVERS would normally create, but must be extended to simulate the environment in which each subsystem will execute. If the TFG for S_1 were built just from the CFG for the car task, then this TFG would not have any entry calls made to the rendezvous it accepts. Thus, to model S_1 in the context of the whole system, an environment needs to be constructed to represent interactions between S_1 and S_2.

[3] Because of the semantics of LTSs and the way queries are answered, we are guaranteed that the conjectured automata will have only a single non-accepting state and that that non-accepting state will be a trap state (i.e., have only self-loop transitions).

The environment for S_1 needs to have accept statements[4] from S_2 that are called by S_1 and entry calls made by S_2 to accept statements in S_1. In this example, these would be the entry calls close_doors, move_car, open_doors, and sync.

Additionally, the environment for S_1 needs to contain events from S_2 that can affect the property or feasibility constraints. To determine these events, each property and feasibility constraint automaton that contains events in both S_1 and S_2 is examined. For these automata, events that occur in S_2 need to be added to the environment of S_1. In this example, the VA for the variable x has events in both the car and controller tasks, so the events from S_2, x=false, x==true, and x==false need to be in the environment for S_1. Now, the events x==true and x==false also occur in S_1. When checking $\langle A \rangle\ S_1\ \langle P \rangle$, for example, we want the assumption A to only constrain the behavior of the environment of S_1, not the behavior of S_1. Thus, we need to relabel the property and the feasibility constraints that have events in both S_1 and S_2 so that events common to S_1 and S_2 can be distinguished based on the subsystem in which they occur. We prefix common events in S_1 with "s1" and events in S_2 with "s2". The events on the nodes of the CFGs for tasks in S_1 and S_2 need to be similarly relabeled.

Finally, the environment needs to be able to perform, in any order, zero or more accepts, entry calls, and events that can affect the property or the feasibility constraints. All of the information needed to construct the environment can be gathered from an automated analysis of the CFGs, the properties, and the feasibility constraints. Figure 4.2 shows the CFG for the environment for S_1. Notice that nodes 29 and 30, which correspond to events common to S_1 and S_2 are prefixed with "s2". To build the TFG for S_1, the environment for S_1 needs to be combined with the CFGs for every task in S_1. Figure 4.3 shows the TFG for S_1 in the elevator example, with MIP edges removed for clarity.[5]

---

[4]Ada also has accept statements with bodies. To support this construct, a rendezvous with a body is converted into two rendezvous: one that occurs before the body and one that occurs after the body.

[5]In this TFG, there would be a MIP edge between every node in the car task and the environment for S_1, not counting the initial node, final node, and nodes corresponding to rendezvous.

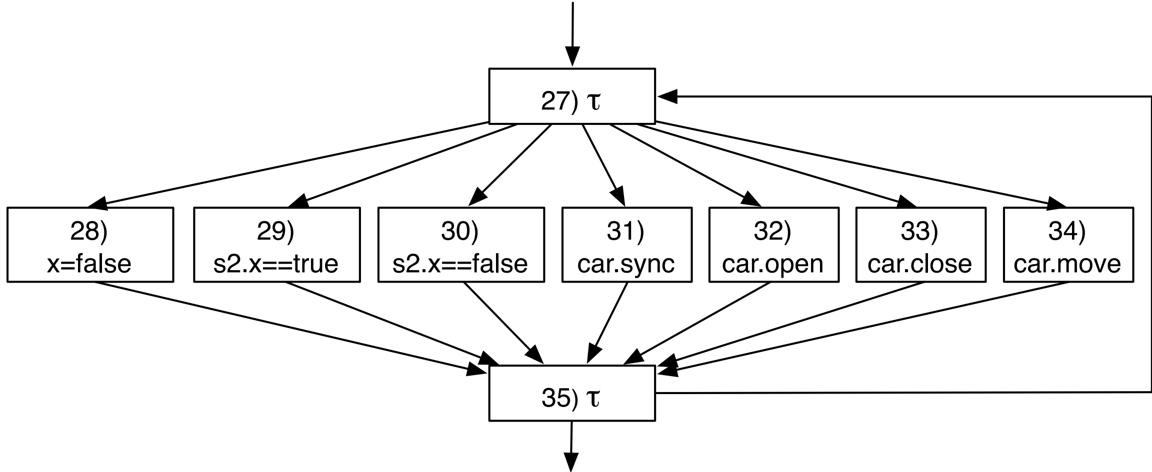

Figure 4.2: CFG for the environment of S_1

4.3.2 The Alphabet of the Assumption

The L* algorithm learns an FSA over an alphabet Σ. In order to use the L* algorithm for assume-guarantee reasoning, it needs to be given Σ. For FLAVERS, the alphabet consists of the labels on all rendezvous[6] that occur between S_1 and S_2, and the labels on the non-τ nodes that do not correspond to rendezvous in the environment of S_1. For the elevator example, $\Sigma = \{$close, s2.x==false, s2.x==true, x=false, move, open, sync$\}$.

4.3.3 Answering Queries

A query posed by the L* algorithm consists of a sequence of events, where each event is in Σ. The teacher must answer true if this sequence is in the language being learned and false otherwise. To answer a query in FLAVERS, S_1 is represented as a TFG and the query is represented as a feasibility constraint. We then use FLAVERS to determine if the property is consistent with the TFG model as constrained by the query. If this results in a violation of the property P, then the assumption needed to make $\langle A \rangle\ S_1\ \langle P \rangle$ true should

[6]This includes rendezvous that are not mentioned in either the property or the feasibility constraints. This is why, for example, node 37 in Figure 4.3 is labeled with sync instead of τ.

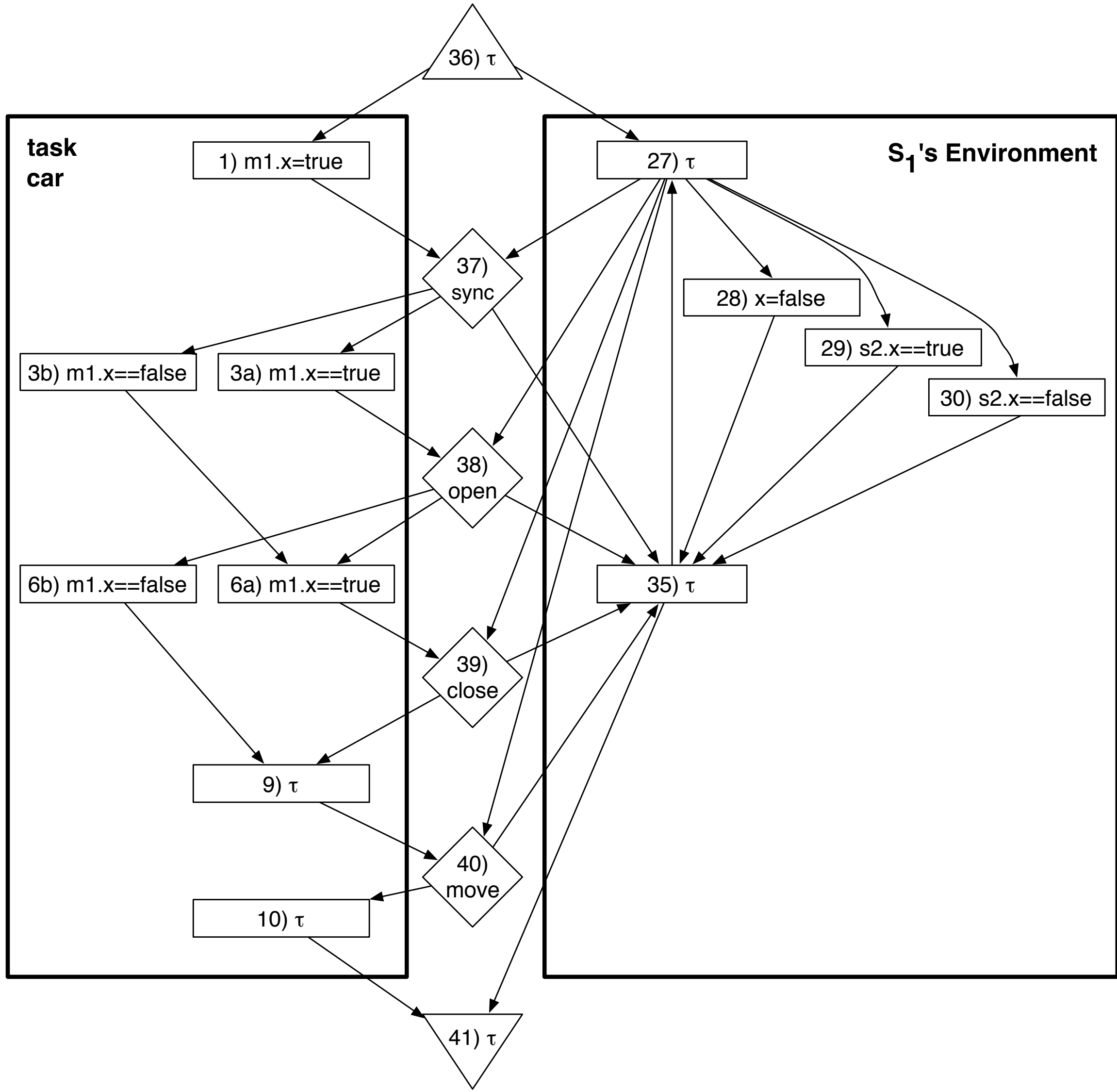

Figure 4.3: TFG for S_1, without MIP edges

not allow the event sequence in the query and false will be returned to the L* algorithm. Otherwise, the event sequence is permissible and true will be returned to the L* algorithm.

More specifically, a TFG is first constructed using the CFGs for tasks in S_1 and the CFG for the environment of S_1. The property to be checked is P. Of the feasibility constraints provided by the analyst, this verification uses the TAs for the tasks in S_1 and the VAs that contain events in S_1. The CFGs, VAs, and the property P are relabeled as described previously to allow events in S_1 and S_2 to be distinguished. Additionally, a query

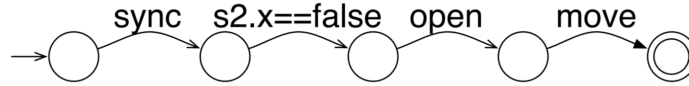

Figure 4.4: Feasibility constraint for the query ⟨sync, s2.x==false, open, move⟩

constraint is used to restrict FLAVERS to only look at paths through the TFG that correspond to the event sequence specified by the query. For example, if the query were ⟨sync, s2.x==false, open, move⟩, then the feasibility constraint shown in Figure 4.4 would be used.[7] State-propagation is then applied to check the property; if the property is violated then false is returned to the L* algorithm, otherwise true is returned.

4.3.4 Answering Conjectures

A conjecture posed by the L* algorithm consists of an FSA that the L* algorithm believes recognizes the language being learned. To answer a conjecture, the teacher needs to find an event sequence in the symmetric difference of the conjectured FSA and the language being learned, if such an event sequence exists. Since the conjectured FSA is the candidate assumption to be used to complete an assume-guarantee proof, it is necessary to determine if the conjectured assumption makes the two premises of the assume-guarantee proof rule true.

First, the conjectured automaton, A, is checked in Premise 1, $\langle A \rangle\ S_1\ \langle P \rangle$. To check this in FLAVERS, a TFG is constructed using the CFGs for tasks from S_1 and the CFG for the environment of S_1. The property to be checked is P and the feasibility constraints used consist of the TAs for tasks in S_1 and VAs that contain events in S_1. The CFGs, VAs, and the property P are relabeled as described previously to allow events in S_1 and S_2 to be distinguished. In addition, the assumption is used as a feasibility constraint. If this verification results in a property violation, then the counterexample returned represents an

[7]This is the same query that is answered for LTSA in Figure 4.1. The events are different because the second event is prefixed by “controller” for LTSA and “s2” for FLAVERS.

event sequence permitted by A but violating P. Thus, the conjecture is incorrect and the counterexample is returned to the L* algorithm. If the property is not violated, then A is good enough to satisfy Premise 1 and Premise 2 can be checked.

Premise 2 states that $\langle true \rangle\ S_2\ \langle A \rangle$ should be true. To check this in FLAVERS, a TFG is constructed using the CFGs for tasks from S_2 and the CFG for the environment of S_2. Unlike the environment for S_1, which consists of events, entries (calls to rendezvous), and accepts from S_2, the environment for S_2 only consists of entries and accepts from S_1. Rather than treat A as a feasibility constraint as was done in checking Premise 1, A is treated as a property. The feasibility constraints used consist of the TAs for tasks in S_2 and VAs that contain events only in S_2. Even though the environment does not have any events from S_1 (those prefixed with "s1"), the VAs, CFGs, and property need to be relabeled as described previously so that the alphabet of this subsystem is consistent with the alphabet of the assumption. If this verification does not result in a property violation, then both Premise 1 and Premise 2 are true, so it can be concluded that P holds on $S_1 \parallel S_2$. If this verification results in a property violation, then the returned counterexample is examined to determine what should be done next. A query is made based on this counterexample, as described previously, to determine if P does not hold on $S_1 \parallel S_2$ or if the L* algorithm needs to continue learning a new assumption.

4.3.5 Environment Generation

To accurately model a subsystem for FLAVERS, we needed to generate an environment for each subsystem being analyzed. In our context, this was not too difficult a problem, still, tools such as BEG (Bandera Environment Generator) [108,109], could be used for this task. Since BEG is designed to solve the more general problem of environment generation for Java software, using it would likely be more expensive than the simple analysis we used.

Păsăreanu et al. studied the problem of environment generation for assume-guarantee reasoning of Ada software [93]. They compared two ways of generating environments

when checking assume-guarantee triples. The first approach is to generate a universal environment and have its actions constrained by the assumption A, which is the approach we took. The second approach is to convert the assumption A into an environment that only allows the actions permitted by the assumption. Their experiments used SPIN and SMV and produced inconclusive results since neither approach for environment generation outperformed the other approach consistently.

We used the first approach in our study, that is we generated a universal environment and constrained it with an assumption. We chose this approach because it results in better reuse of artifacts since the TFG does not need to be regenerated for each conjecture. Of course, our results may have been different had we used the other approach although we do not expect that they would be significantly different based on the results seen in [93].

CHAPTER 5

EXPERIMENTAL METHODOLOGY AND RESULTS

As stated previously, the lack of automated assumption generation has made it difficult to evaluate assume-guarantee reasoning. Using the approach described in Chapter 4, we undertook a study to gain a sense of whether or not assume-guarantee reasoning provides an advantage over monolithic verification. For this study, we further restricted our evaluation to assume-guarantee reasoning using the rule shown in Figure 1.1 and to two finite-state verifiers, FLAVERS and LTSA. There are several different ways that assume-guarantee reasoning could provide an advantage over monolithic finite-state verification:

1. Does assume-guarantee reasoning use less time than monolithic verification?
2. Does assume-guarantee reasoning use less memory than monolithic verification?
3. If assume-guarantee reasoning uses less memory than monolithic verification, is there enough of a memory savings to allow assume-guarantee reasoning to verify properties on larger systems than monolithic verification?

Since finite-state verification techniques are often limited more by memory than by time, we focused our study on points 2 and 3 rather than point 1.

To evaluate the usefulness of this automated assume-guarantee reasoning technique, we tried to verify properties that were known to hold on a small set of scalable systems: the Chiron user interface system [79] (both the single and the multiple dispatcher versions as described in [10]), the Gas Station problem [64], Peterson's mutual exclusion protocol [96], the Relay problem [104], and the Smokers problem [94]. These systems were specified in Ada and use rendezvous for intertask communication. Except for Peterson's mutual

exclusion protocol, which also uses shared variables for intertask communication, these systems all have a client-server architecture where the server has an interface made up of a small number of rendezvous that may be called by the clients. The properties we checked on these systems are all safety properties that describe a legal (or illegal) sequence of events in each system. Descriptions of the systems and the properties we verified are given in Appendix A. Since we used two versions of the Chiron system and proved the same properties on both versions, we use the term *subject* to refer to a property-system pair. Thus, each Chiron property is used in two subjects while all other properties are used in one subject.

Each of the systems we used was scaled by creating more instances of one particular task, and the size of the system is measured by counting the number of occurrences of that task in the system. For the Chiron systems we counted the number of artists, for the Gas Station system we counted the number of customers, for the Peterson system we counted the number of tasks trying to gain access to the critical section, for the Relay system we counted the number of tasks accessing the shared variable, and for the Smokers system we counted the number of assemblers.

For both FLAVERS and LTSA we considered a task to be an indivisible subsystem. Thus, a decomposition was an assignment of each task in the system to either S_1 or S_2. Note that each scalable system we looked at had more than two subsystems (i.e., tasks), even at size 2.

Both FLAVERS and LTSA prove that a property holds by exploring all of the reachable states in an abstracted model of a system. On properties that do not hold, these two tools stop as soon as a property violation is found. As a result, their performance on properties that do not hold is more variable. Although using only properties that hold restricts the scope of our study, including properties that do not hold would have made it more difficult to meaningfully compare the performance of monolithic verification to assume-guarantee reasoning.

For LTSA, INCA [38] was used to translate the Ada systems into FSAs, which are then easily translated into LTSs. Because INCA was used to generate models for LTSA, we did not make use of the CRA capabilities of LTSA. Also, there is one fewer Chiron property for LTSA than for FLAVERS. The events needed to express property 8 of the Chiron system are removed from the model when INCA generates the FSAs. Since this property states that those events cannot occur, this property is shown to hold during model construction, making verification using LTSA unnecessary. This means there are two fewer subjects for LTSA than for FLAVERS, because this property occurs in two subjects, one for each version of Chiron.

We did not use the most recent version of LTSA, which is based on plugins [25], because the plugin interface does not provide direct access to the LTSs. An implementation of this assumption generation technique exists for the plugin version of LTSA [51], but verification takes significantly longer because all LTSs must be created by writing appropriate FSP, necessitating parsing the entire model for each query and conjecture, even for the parts of the model that do not change between different queries and conjectures.

We used the version of FLAVERS that directly accepts Ada systems. Since FLAVERS uses feasibility constraints to control the amount of precision in a verification, we used a minimal set of feasibility constraints when verifying properties.[1]

---

[1]A minimal set of feasibility constraints is one such that removal of any feasibility constraint causes FLAVERS to report that the property may not hold. While these sets are minimal for each property, they may not be the smallest possible set of feasibility constraints with which FLAVERS can prove the property holds nor the best set with respect to the memory or time cost for FLAVERS. While the worst-case complexity of FLAVERS increases with each feasibility constraint that is added, sometimes adding more feasibility constraints can improve the actual performance of FLAVERS. Since we did not consider all possible combinations of all possible feasibility constraints, we can not be certain that the selected minimal feasibility constraint set is either the smallest minimal set or the set that uses the least time or memory. On the other hand, a process that might be applied by analysts, described in [43], was used select the sets of feasibility constraints used in our study.

Table 5.1: Number of two-way decompositions examined for systems of size 2 for FLAVERS

System	Properties	Decompositions	Total
Chiron single	9	62	558
Chiron multiple	9	254	2,286
Gas Station	4	30	120
Peterson	1	6	6
Relay	1	6	6
Smokers	8	14	112
Total	**32**		**3,088**

5.1 Does Assume-Guarantee Reasoning Save Memory for Small System Sizes?

To determine the amount of memory used by monolithic verification, we counted the number of states explored during verification. While the artifacts created by the verifiers (e.g. TFGs and FSAs in FLAVERS, LTSs in LTSA) use memory, we did not count them when determining memory usage since the amount of memory needed to store them is usually small when compared to the amount of memory needed to store the states explored during verification. Similarly, to determine the amount of memory used by assume-guarantee reasoning, we looked at the maximum number of states explored by the teacher when answering a query or a conjecture of the L* algorithm. We say one decomposition is better than another decomposition if the maximum number of states explored when the teacher answers a query or conjecture using the first decomposition is smaller than the maximum number of states explored when the teacher answers a query or conjecture using the second decomposition.

For each subject in our study, we examined *all* two-way decompositions to find the best decomposition for that subject with respect to memory. For systems for size 2 Tables 5.1 and 5.2 list, for FLAVERS and LTSA respectively, the number of properties for

Table 5.2: Number of two-way decompositions examined for systems of size 2 for LTSA

System	Properties	Decompositions	Total
Chiron single	8	62	496
Chiron multiple	8	254	2,032
Gas Station	4	30	120
Peterson	1	6	6
Relay	1	6	6
Smokers	8	14	112
Total	**30**		**2,772**

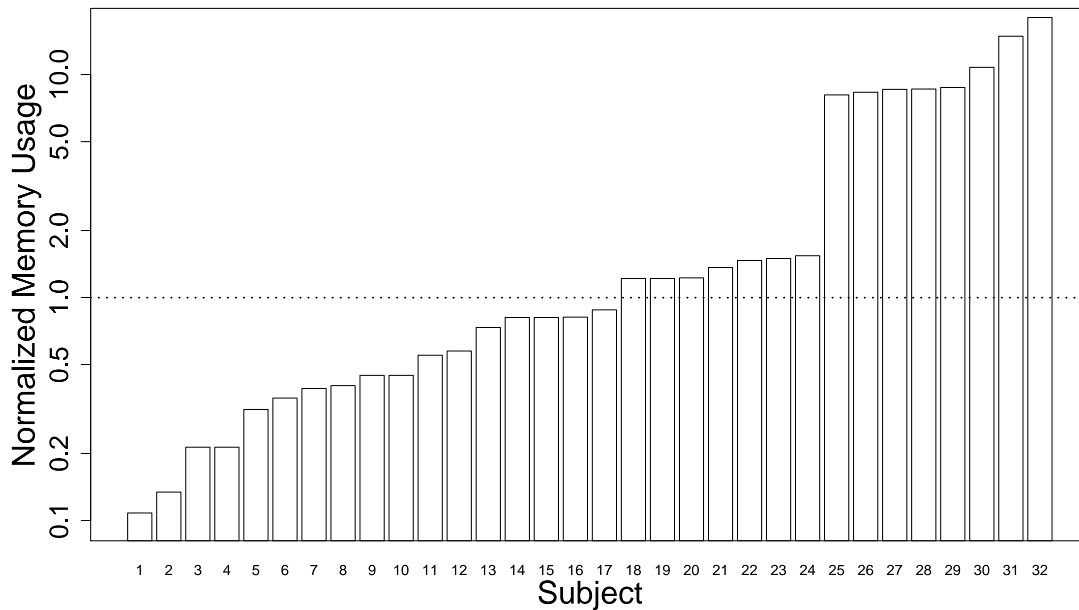

Figure 5.1: Memory used by the best decomposition of size 2 for FLAVERS

each system, the number of two-way decompositions[2] examined for each subject, and the total number of decompositions examined on each system.

Figures 5.1 and 5.2 show, for FLAVERS and LTSA respectively, the amount of memory used by the best decomposition at size 2 normalized by dividing it by the amount of memory used by monolithic verification. For reference, a line at 1.0 has been drawn. Points below this line represent subjects on which the best decomposition is better than monolithic

[2]Note that the number of two-way decompositions examined for each subject is always two fewer than a power of two because the two-way decompositions where either S_1 or S_2 are empty are not checked.

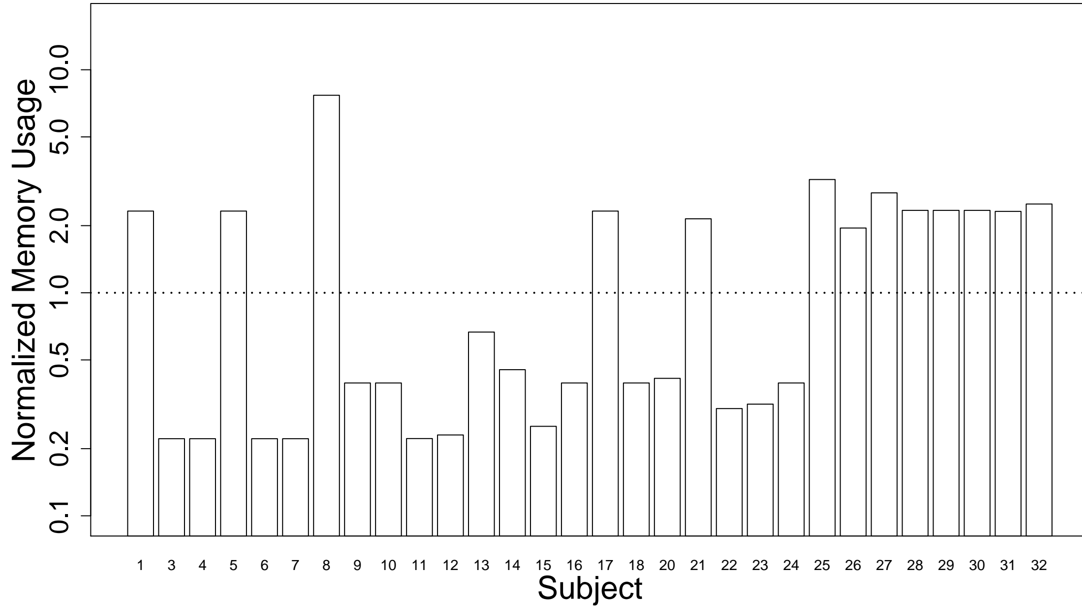

Figure 5.2: Memory used by the best decomposition of size 2 for LTSA

verification, while points above this line represent subjects on which the best decomposition is worst than monolithic verification. In Figure 5.1 each subject is given a number from 1 to 32, in increasing order of normalized memory usage. These mapping between these numbers and subjects is given in Appendix B. In Figure 5.2, each subject is numbered with the same subject number it was assigned in Figure 5.1. Note that in Figure 5.2 there are no bars numbered 2 or 19. These subjects are the two that are shown to hold during model construction.

For FLAVERS, the best decomposition is better than monolithic verification on 17 of the 32 subjects. For these 17 subjects, on average the best decomposition uses 48.5% of the memory used by monolithic verification. For the 15 subjects where the best decomposition is worse than monolithic verification, on average the best decomposition uses 654.1% of the memory used by monolithic verification.

For LTSA, the best decomposition is better than monolithic verification on 17 of the 30 subjects. For these 17 subjects, on average the best decomposition uses 33.6% of the memory used by monolithic verification. For the 13 subjects where the best decomposition

is worse than monolithic verification, on average the best decomposition uses 281.7% of the memory used by monolithic verification.

While there are 17 subjects on which assume-guarantee reasoning is better than monolithic for both FLAVERS and LTSA, these are not exactly the same 17 subjects. There are a total of 12 subjects for which the assume-guarantee approach is better than monolithic verification for both FLAVERS and LTSA. In fact, the subject on which assume-guarantee reasoning with FLAVERS saves the most memory compared to monolithic verification, is a subject on which assume-guarantee reasoning with LTSA does not save memory compared to monolithic verification. Still, if assume-guarantee reasoning saved memory on a given subject with one verifier, it was likely to save memory on the same subject with the other verifier.

It is important to note that the vast majority of decompositions are not better than monolithic verification. Even for the subjects where the best decomposition is better than monolithic verification, most of the decompositions we examined for those subjects are not better than monolithic verification. Thus, randomly selecting decompositions would likely not yield a decomposition better than monolithic verification. Furthermore, our intuition on how to select decomposition was not good. On the subjects where it is possible to select a decomposition that uses less memory than monolithic verification, we did not select a decomposition that saves memory most of the time. While it might be possible to develop heuristics to aid in finding such a decomposition, when we examined the decompositions that saved memory in our experiments, however, we did not see any patterns that could be used as the basis for such heuristics.

5.2 Does Assume-Guarantee Reasoning Save Memory for Larger System Sizes?

Although assume-guarantee reasoning using learned assumptions saves memory in only about half of the subjects we looked at when the best decomposition is used and finding

- For each non-repeatable task, put the task into S_1 if the task was put into S_1 in the best decomposition at size 2. Otherwise, put the task into S_2.
- For each repeatable task:
  - If the best decomposition for size 2 had both repeatable tasks in S_1, put the repeatable task in S_1. Otherwise, put the repeatable task in S_2.
  - If the best decomposition for size 2 had one of the repeatable tasks in S_1 and the other in S_2, look to see if the property treated one of the repeatable tasks in a different way than all the other repeatable tasks.
    + If one of the repeatable tasks is treated in a different way in the property, then:
      - If this repeatable task is the one that is treated differently, then put this repeatable task into S_1 if its corresponding task in the best decomposition at size 2 was put into S_1. Otherwise, put this task into S_2.
      - If this repeatable task is not the one that is treated differently, then put this repeatable task into S_1 if the repeatable task that is treated differently was in S_2 on the best decomposition at size 2. Otherwise, put this task into S_1.
    + If none of the repeatable tasks are treated in a different way in the property, then:
      - If this repeatable task is the repeatable task with the smallest ID, put this repeatable task into S_1 if the repeatable task with the smallest ID was put into S_1 in the best decomposition at size 2. Otherwise, put this repeatable task into S_2.
      - If this repeatable task is not the repeatable task with the smallest ID, put this repeatable task into S_2 if the repeatable task with the smallest ID was put into S_2 in the best decomposition at size 2. Otherwise, put this repeatable task into S_1.
    + If two of the repeatable tasks are treated in a different way in the property, then handle the tasks treated in a different way as in the "one" case above and handle the tasks that are not treated in a different way as in the "none" case above.

Figure 5.3: Process for generalizing decompositions

these best decompositions was expensive, the overall approach was not too onerous. On average at size 2 it required about two minutes to examine one decomposition with FLAVERS and about half a minute to examine one decomposition with LTSA. For larger size systems, however, it would be infeasible to evaluate all two-way decompositions because the number of decompositions to be evaluated increases exponentially and the cost of evaluating each decomposition increases as well. For example, we have several instances where evaluating a single decomposition on a system of size 4 takes over 1 month. Thus, if memory is a concern in verifying a specific system and if it is important to verify it for a larger size, a reasonable approach might be to examine all decompositions for a small system size and then to generalize the best decomposition for that small system size to a larger system size. We used this generalization approach to evaluate the memory usage of assume-guarantee reasoning for larger system sizes. Our algorithm for generalizing decompositions from the best decomposition for size 2 is shown in Figure 5.3.[3] At a high level, this algorithm assigns each task into one of two categories, either a task is repeatable (e.g., a customer task in the gas station system) or non-repeatable (e.g., all non-customer tasks in the gas station system). A non-repeatable task is put in S_1 (S_2) if the best decomposition has the corresponding task in S_1 (S_2). For repeatable tasks, a similar process is followed, but since there are more repeatable tasks at larger system sizes, determining the corresponding task is more complicated but is still based on whether each repeatable task at size 2 was in S_1 or S_2.

With FLAVERS, if the best decomposition at size 2 is better than monolithic verification, the associated generalized decompositions are usually better than monolithic verification, as seen with 16 of the 17 such subjects. In addition there are 2 subjects on which the best decomposition at size 2 is worse than monolithic verification, but the generalized decomposition is better than monolithic verification at the largest size such a comparison

[3] We considered several other ways to generalize the decomposition and discuss them in Section 5.7.

could be made. Thus, with FLAVERS assume-guarantee reasoning using generalized decomposition is better than monolithic verification on 18 of the 32 subjects.

With LTSA, of the 17 subjects on which the best decomposition at size 2 is better than monolithic verification, on only 5 of these is the generalized decomposition better than monolithic verification at the largest size such a comparison could be made. With LTSA, there are two subjects worth noting. The first is property 4 of the Gas Station system and on this subject assume-guarantee reasoning is better than monolithic verification at size 7 and worst at all other sizes. On this subject, assume-guarantee reasoning ran out of memory at size 9 but monolithic verification was able to verify the property. The second subject is property 3 of the Chiron multiple system and on this subject assume-guarantee reasoning is worst than monolithic verification at size 4 and better at all other sizes. Unfortunately, we could not generate the model for this system at size 6 and we will discuss this more in Section 5.3

Figures 5.4, 5.5, and 5.6 show the amount of memory used by assume-guarantee reasoning with generalized decompositions normalized by dividing by the amount of memory used by monolithic verification as the size of the systems is increased. Each solid line represents a single subject. A dotted line at 1.0 has been provided for reference. Figures 5.4 and 5.5 show data for FLAVERS. The former shows the data for system sizes less than or equal to 10 while the latter shows the data for only those subjects that could scale above size 10. Note that each line in Figure 5.5 corresponds to a line that is shown in Figure 5.4. Figure 5.6 shows all of the data for LTSA. For 8 subjects with FLAVERS and 3 subjects with LTSA there are single points at size 2. On these subjects, the generalized decompositions runs out of memory at size 3. Note that each line stops at the largest system on which both monolithic verification and assume-guarantee reasoning can verify the corresponding subject. For some subjects either assume-guarantee reasoning or monolithic verification can verify that subject at even larger sizes.

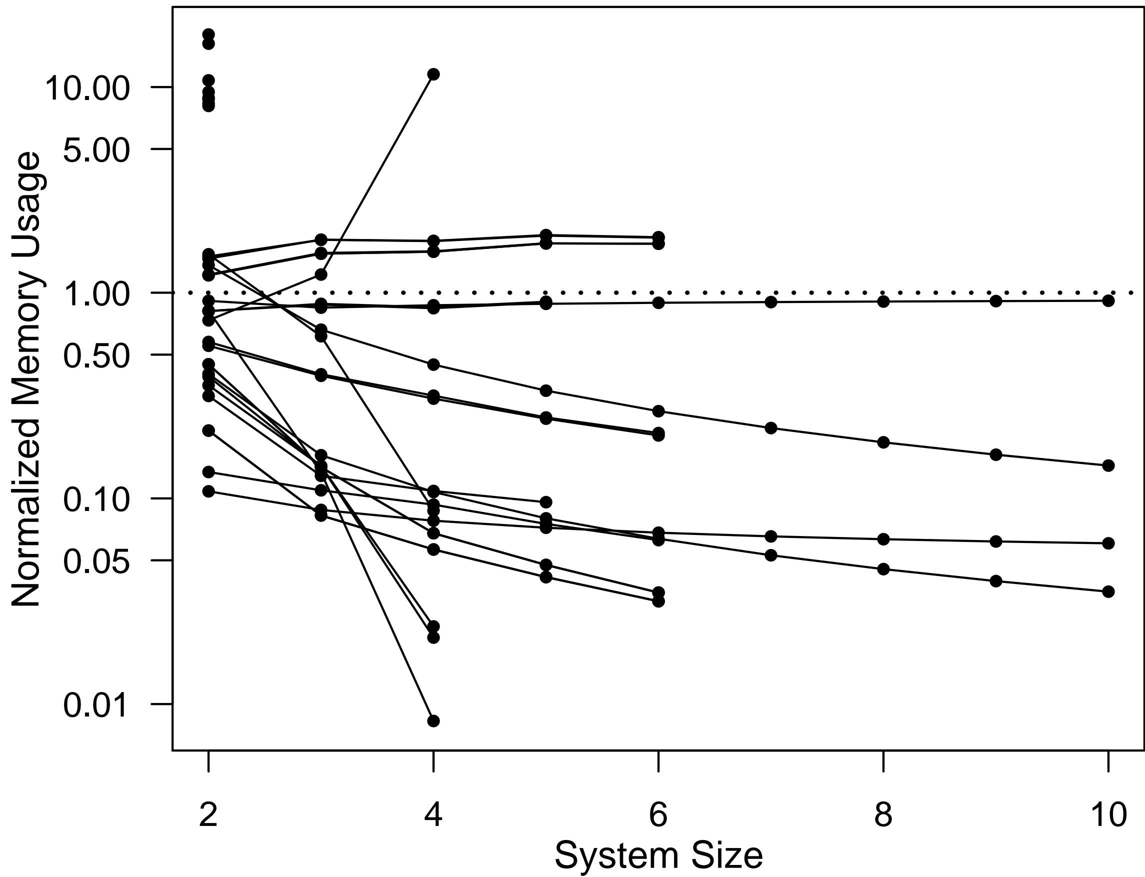

Figure 5.4: Memory used by the generalized decompositions for FLAVERS up to dize 10

Although these figures are difficult to read, they illustrate the significant difference in the performance of the generalized decompositions between FLAVERS and LTSA. With FLAVERS, on the subjects where the best decomposition at size 2 is worst than monolithic verification, assume-guarantee reasoning tends to use increasingly larger amounts of memory, when compared to monolithic, as the system size increases. In other words, the normalized memory usage increases as the system size increases. With FLAVERS, on the subjects where the best decomposition at size 2 is better than monolithic verification, assume-guarantee reasoning tends to save more memory, when compared to monolithic verification, as system size increases. In other words, the normalized memory usage tends to decrease as the system size increase.

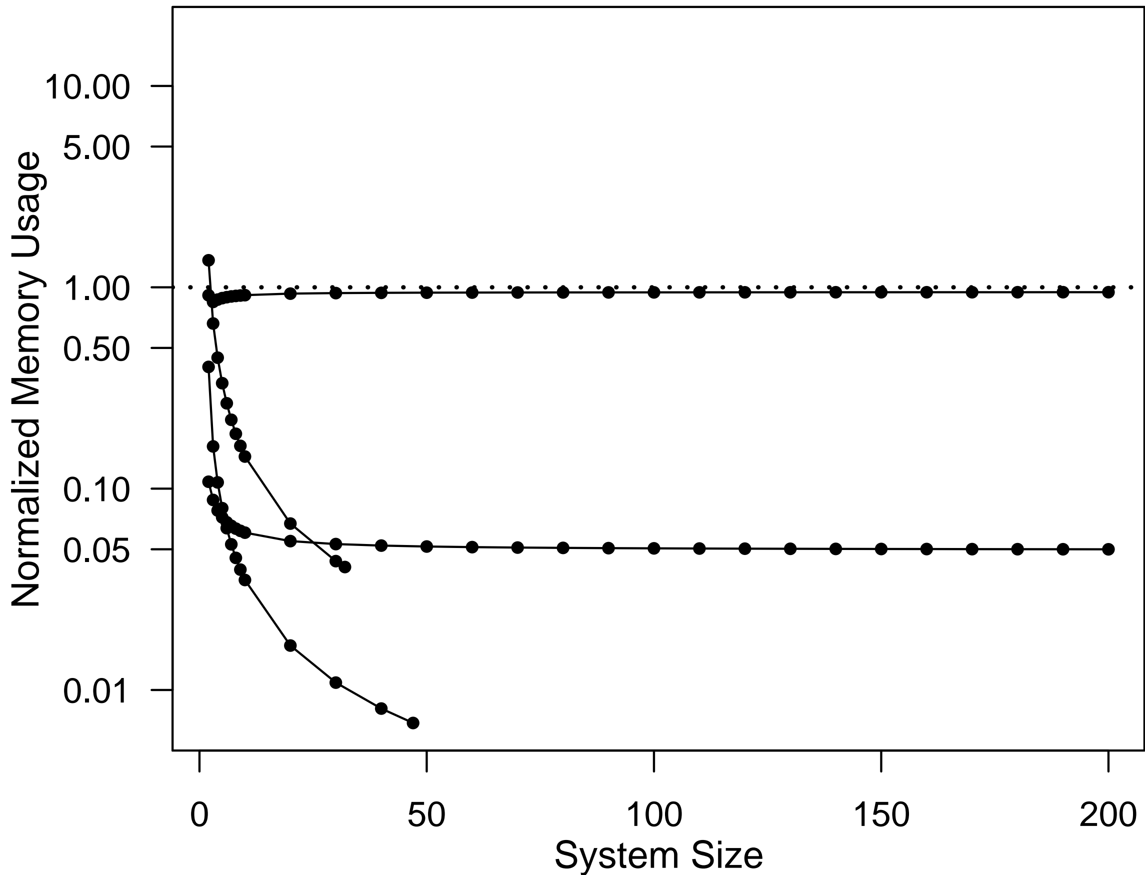

Figure 5.5: Memory used by the generalized decompositions for FLAVERS for just those subjects where the largest system size that could be verified is greater than 10

This is not true, however, with LTSA. With LTSA, assume-guarantee reasoning tends to use more memory, when compared to monolithic verification, as the system size increases. In other words, the normalized memory tends to increase as the system size increases, regardless of the performance of assume-guarantee reasoning at size 2, although this increase was more pronounced on subjects where the memory used by the best decomposition at size 2 was worst than monolithic verification.

We believe that the difference in the performance of assume-guarantee reasoning on the two verifiers is mostly due to how the models for the two verifiers are built. For LTSA, each thread is represented by a thread reachability graph. For FLAVERS, each thread is represented by a control flow graph, which is usually smaller and more abstract than a

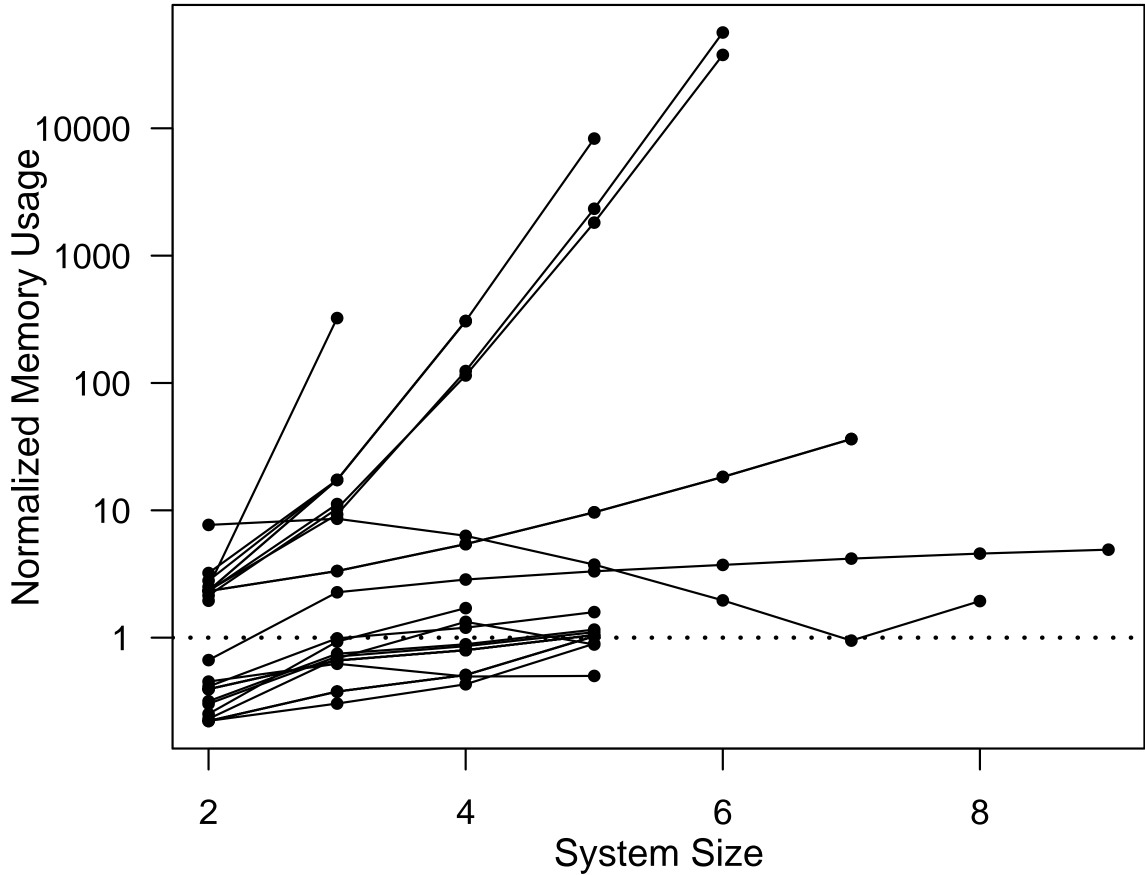

Figure 5.6: Memory used by the generalized decompositions for LTSA

thread reachability graph. FLAVERS adds precision into the model through the use of feasibility constraints. Since the number of feasibility constraints does not usually increase as the system size increases [43], this means that the model used by FLAVERS does not usually increase in size as quickly as the model generated for LTSA. We believe that this difference resulted in the performance difference between the two verifiers.

In summary, with FLAVERS the best decomposition at size 2 is better than monolithic verification on 17 of the 32 subjects, or about 53% of the time, and the generalized decomposition is better than monolithic verification on 18 of the 32 subjects, or about 56% of the time. With LTSA the best decomposition at size 2 is better than monolithic verification on 17 of the 30 subjects, or about 56% of the time, and the generalized decomposition is better

than monolithic verification on 3 of the 30 subjects, or 10% of the time. Thus, the automated assume-guarantee reasoning technique we used was able to save memory on larger size systems for a bit more than half the subjects with FLAVERS and for one tenth of the subjects with LTSA.

5.3 Can Assume-Guarantee Reasoning Verify Properties of Larger Systems than Monolithic Verification

Although using generalized decompositions for assume-guarantee reasoning uses less memory than monolithic verification in some cases, this memory savings might not be sufficient to overcome the state-explosion problem. Thus, we tried to determine, for each subject, whether or not assume-guarantee reasoning using generalized decompositions would allow us to verify larger systems than monolithic verification.

Because the language processing toolkit [107] used by FLAVERS[4] for generating its models and by INCA for generating the models for LTSA cannot handle the Chiron and Relay systems at larger sizes, we were unable to determine for each subject whether or not assume-guarantee reasoning using generalized decompositions would allow us to verify larger systems than monolithic verification. Thus, we assigned each subject to one of five categories:

1. Assume-guarantee reasoning can verify a larger system than monolithic verification.
2. It is unknown if assume-guarantee reasoning can verify a larger system than monolithic verification. We consider it likely, however, because assume-guarantee reasoning is better than monolithic verification for the largest system size such a comparison can be made.

---

[4]This toolkit is very old and not easily modifiable. We are working on building models for FLAVERS from Java software using Bandera [39] and, thus, expect to remove some of the limitations of this language processing toolkit.

3. Assume-guarantee reasoning cannot verify a larger system than monolithic verification.

4. It is unknown if assume-guarantee reasoning can verify a larger system than monolithic verification. We consider it unlikely, however, because assume-guarantee reasoning is worse than monolithic verification for the largest system size such a comparison can be made.

5. Assume-guarantee reasoning is better than monolithic verification at the largest size such a comparison can be made, but monolithic verification can verify the subject on systems with size 30 or more. While assume-guarantee reasoning might be able to verify a larger system, we think verifying these subjects on larger systems will not be of much use.[5]

Table 5.3 shows the number of subjects in each category for FLAVERS and LTSA. We consider using generalized decompositions to be a potential success in verifying a larger system than monolithic verification if the subject is in category 1 or 2. We consider using generalized decomposition to be a likely failure in verifying a larger system than monolithic verification if the subject is in category 3 or 4. We consider assume-guarantee reasoning to not be needed if the subject is in category 5. As mentioned previously, for LTSA property 3 of the Chiron multiple system is hard to classify since assume-guarantee reasoning is worst than monolithic verification at size 4 and better at all other sizes. Because of language processing issues, we could not build a model for this system at size 6. Since assume-guarantee reasoning is better than monolithic verification at size 5, the largest size such a comparison can be made, we conservatively assigned this subject to category 2.

[5]The somewhat arbitrary cutoff of 30 represents a substantial size for the systems under consideration. All of the subjects we examined that can be verified on systems larger than size 10 can be verified on systems of size 30.

Table 5.3: Generalized decompositions compared to monolithic verification with respect to scaling

		FLAVERS		LTSA	
		Number of Subjects	Percentage	Number of Subjects	Percentage
Potential Success	(1) Generalized can scale farther than monolithic	6	18.8%	0	0.0%
	(2) Don't know, but generalized appears better than monolithic	7	21.9%	2	6.7%
	Subtotal	**13**	**40.6%**	**2**	**6.7%**
Likely Failure	(3) Generalized cannot scale farther than monolithic	13	40.6%	14	46.7%
	(4) Don't know, but generalized appears worse than monolithic	2	6.3%	14	46.7%
	Subtotal	**15**	**46.9%**	**28**	**93.3%**
Both Scale Well	(5) Monolithic scales well, so generalized unlikely to be of much use	4	12.5%	0	0.0%
	Subtotal	**4**	**12.5%**	**0**	**0.0%**
	Total	**32**	**100.0%**	**30**	**100.0%**

Table 5.3 shows a potential success rate of about 40% for FLAVERS and about 7% for LTSA. Note that this rate is the upper bound of the success rate. By looking at just the subjects where we could demonstrate that assume-guarantee reasoning could scale farther, meaning those in category 1, we obtain the lower bound of the success rate: about 19% for FLAVERS and 0% for LTSA.

Although we could demonstrate that assume-guarantee reasoning scales farther than monolithic verification on six subjects for FLAVERS, it is also important to look at how much farther assume-guarantee reasoning scales. On five of these six subjects, assume-guarantee reasoning can verify the subject on a system one size larger, but not two sizes

larger, than monolithic verification can. On the sixth subject, assume-guarantee reasoning can verify the subject on a system two sizes larger, but not three sizes larger, than monolithic verification can. In addition, there were two subjects, counted in category 5, where assume-guarantee reasoning can verify the system at least three sizes larger than monolithic verification can. In one case, this allowed us to increase the size of the subject from 32 to 35 and in the other case this allowed us to increase the size of the subject from 47 to 50. Since monolithic verification can scale to at least size 30, a fairly large system size, we do not believe verifying the system on larger sizes is particularly useful and do not count these subjects as successes. While there were eight subjects where we demonstrated that assume-guarantee reasoning can scale farther than monolithic verification, assume-guarantee reasoning can verify these subjects on systems only slightly larger than the size of the system on which these subjects can be verified monolithically.

In summary, if we had not encountered model building issues, we believe that assume-guarantee reasoning could verify a larger system size than monolithic verification on at most 41% of the subjects for FLAVERS and 7% of the subjects for LTSA. While a 41% success rate may look encouraging, assume-guarantee reasoning using generalized decompositions did not significantly increase the size of the systems that could be verified. Considering the amount of time that was spent to find the best decomposition at size 2, it is questionable if the benefit of verifying a subject on a slightly larger system size is worth the necessary investment of time.

Although these results are discouraging, we also tried to determine if there was some way to classify the subjects to predict the subject for which assume-guarantee reasoning would likely produce a significant memory savings. Unfortunately, we could not find such a classification, although we do have some observations. One type of feasibility constraint used by FLAVERS is a Task Automaton (TA). It appears that when the number of TAs needed to prove a property increases as the system size increases, assume-guarantee reasoning based on generalized decompositions tends to use more memory than monolithic

verification. Of the 14 subjects where the number of TAs needed to prove the property increases as the system size increases, 10 of them are classified as failures in Table 5.3. Of the 13 subjects where only one TA is needed to prove the property regardless of system size, 4 are classified as failures in Table 5.3 and 3 others are in category 5 where assume-guarantee reasoning based on generalized decompositions does better than monolithic verification but assume-guarantee reasoning is not likely to be of much use since monolithic verification scales well. On the remaining 5 subjects, we cannot find a pattern to determine whether or not assume-guarantee reasoning based on generalized decompositions will perform better than monolithic verification. For LTSA, the picture is even less clear. On some of the subjects where assume-guarantee reasoning for FLAVERS works well, assume-guarantee reasoning for LTSA does not work well. Conversely, there was 1 subject where assume-guarantee reasoning for FLAVERS works poorly, but assume-guarantee reasoning for LTSA works well. While our observations may provide some guidance to help determine whether or not assume-guarantee reasoning is likely to save memory, more experimentation is needed before any conclusions can be drawn.

5.4 Are the Generalized Decompositions the Best Decompositions?

These discouraging results were obtained using decompositions that were generalized from the best decomposition on problems of size 2. It is possible that the generalized decompositions we selected are not the best decompositions to use on the larger systems sizes. To investigate this issue, we tried to find the best decomposition for some larger system sizes.

Table 5.4: System sizes at which the best decomposition is known

	FLAVERS		LTSA	
Size	Attempted	Best Known	Attempted	Best Known
2	32	32	30	30
3	32	23	30	30
4	24	18	21	21
5	2	2	1	1

5.4.1 Comparing the Best Known Decompositions to the Generalized Decompositions

In performing these experiments, we encountered a number of two-way decompositions where it took more than a month to learn an assumption.[6] As a result, we imposed an upper bound on the amount of time we spent evaluating a single two-way decomposition to be the maximum of 1 hour and 10 times the amount of time needed to verify that subject monolithically. On every subject where the upper bound on time was reached, we were able to find at least one decomposition that is better than the generalized decomposition. Table 5.4 gives the number of subjects for which we attempted to find the best decomposition at a given system size. It also gives, in the Best Known column, the number of subjects for which we were able to find the best decomposition at a given system size.

Figures 5.7 and 5.8 compare, for FLAVERS and LTSA respectively, the memory usage of the generalized decomposition, the best known decomposition, and monolithic verification at the largest size such a comparison could be made. In Figures 5.7 and 5.8 the height of the bars shows the amount of memory used by the generalized decomposition, normalized by dividing by the amount of memory used by monolithic verification. The height of the dots shows the amount of memory used by the best known decomposition, normalized by dividing by the amount of memory used by monolithic verification. A dot that is filled

[6]The time needed to evaluate the decompositions that took more than a month is counted in the 1.54 years of CPU time needed for our experiments. Still, more than 99% of the decompositions required less than 1 day to evaluate. The time used examining just the decompositions that took less than 1 day to evaluate added up to over 11 months of CPU time.

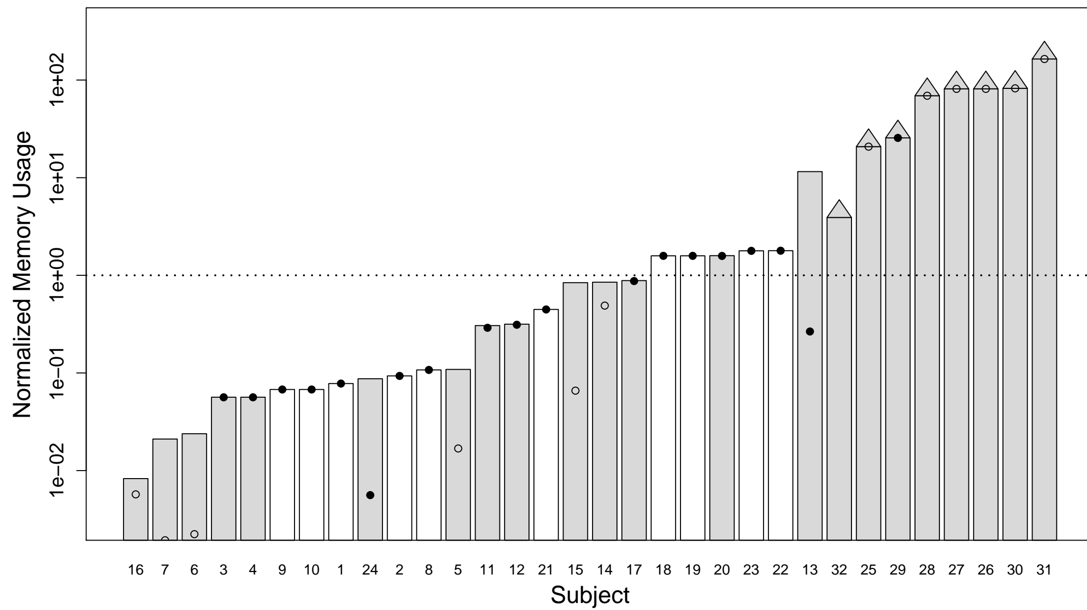

Figure 5.7: Memory used by the generalized decomposition compared to the best decomposition for FLAVERS

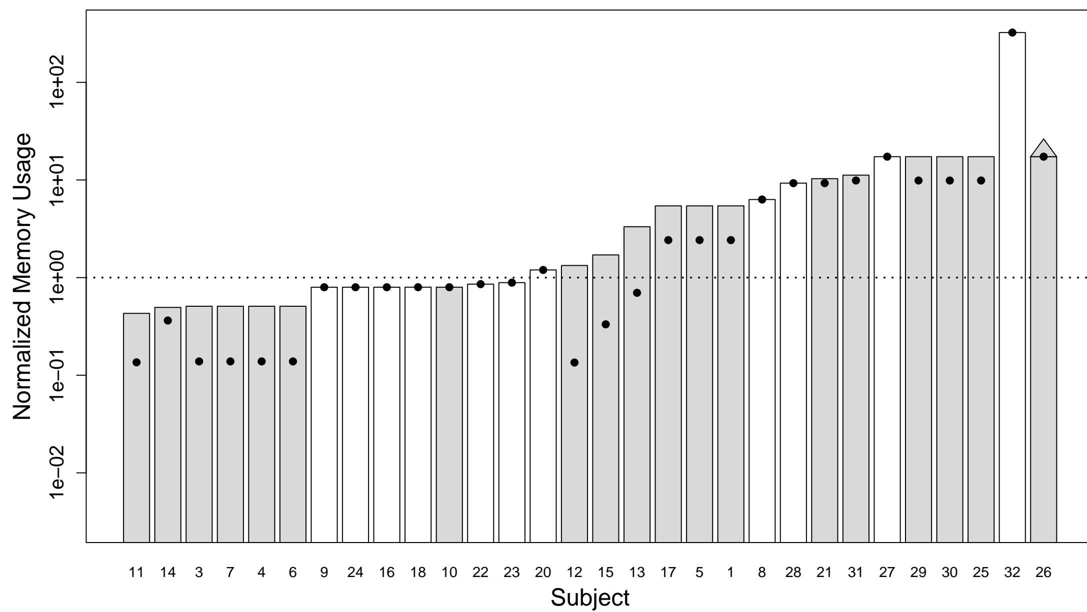

Figure 5.8: Memory used by the generalized decomposition compared to the best decomposition for LTSA

in represents a subject for which we know the best decomposition. A dot that is not filled in represents a subject for which we do not know the best decomposition, meaning there might be better decompositions than the best one of which we know. A bar with a white background represents a subject where the generalized decomposition is the same as the best decomposition, meaning the height of the bar and the dot are the same. A bar with a gray background represents a subject where the generalized decomposition is not the same as the best decomposition, meaning the height of the bar is greater than the height of the dot. Bars that are topped with triangles represent subjects where the generalized decompositions ran out of memory on systems with size 3. The height of these bars shows the lower bound on the amount of memory used by the generalized decompositions. Each subject has been labeled with the same number that was assigned to that subject in Figure 5.1. For reference, a line at 1.0 has been drawn.

As stated previously, because of the cost involved we did not always obtain data about the best decomposition at the largest system size on which we were able to use the generalized decomposition. This is why, for example, Table 5.3 states that there are only 2 subjects with LTSA where the generalized decomposition is better than monolithic verification but Figure 5.8 has 13 subjects where the height of the bar is below the reference line at 1.0.

These figures show that using the generalized decompositions is not always optimal with respect to memory usage. With FLAVERS, the generalized decomposition is the best decomposition on 10 of 32 subjects (31%). With LTSA, this is true on 11 of 30 subjects (37%). For some subjects, the difference in memory usage between the generalized decomposition and the best known decomposition is significant, while in other cases, there is almost no difference. For very few subjects, though, is the generalized decomposition worse than monolithic verification while the best decomposition is better than monolithic. This happened on only one subject with FLAVERS and three subjects with LTSA.

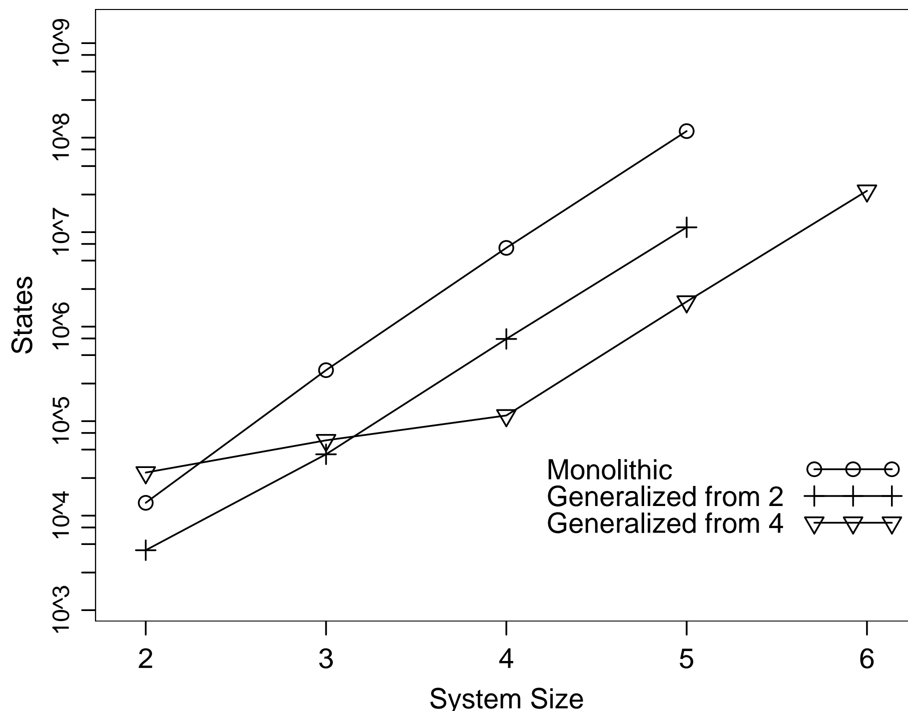

Figure 5.9: States explored on property 1 of Gas Station with FLAVERS

5.4.2 Generalizing Decompositions from the Best Known Decomposition at Larger System Sizes

We were interested in determining if, on the subjects where the generalized decomposition is not the best decomposition, generalizing decomposition based on the best known decomposition from a system size larger than 2 could be used to verify larger systems than can be verified monolithically. Thus, when we found a decomposition for size n $(n > 2)$ that was better than the generalized decomposition from size 2, we generalized the best known decomposition for size n so that is could be used on systems larger than size n. In all cases, the generalized decomposition from size n is better than the generalized decomposition from size 2. We also tried taking the best known decompositions for size n and simplifying them so they could be used on systems of size 2, similar to the process shown in Figure 5.3, but in reverse. The decompositions for size n, when simplified to size 2, are worse than monolithic verification in all but one case.

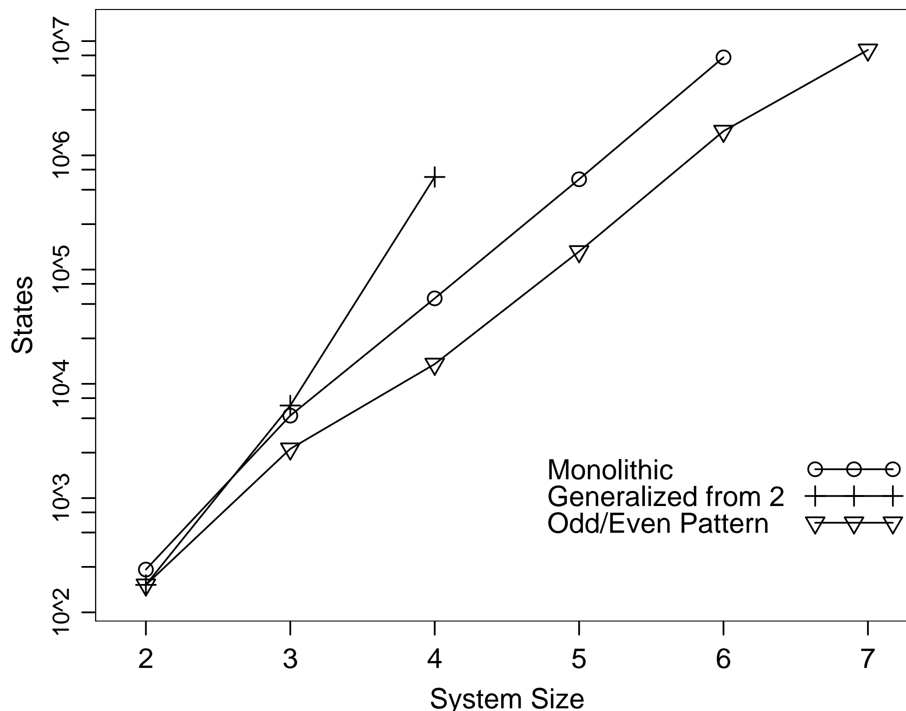

Figure 5.10: States explored on property 1 of Relay with FLAVERS

In addition we found 3 subjects where there are decompositions that can be used to verify that subject on a larger system size than either monolithic verification and the generalized decompositions from size 2. One of these subjects is property 1 of the Gas Station system with FLAVERS. Figure 5.9 compares the number of states explored using two different generalizations (from size 2 and from size 4) to the number of states explored using monolithic verification. On this subject, the generalized decomposition from size 2 is the best decomposition for size 3. When the system size is 4, however, the generalized decomposition from size 2 is not the best decomposition. We do not know what the best decomposition for size 4 is because it requires too much time to find. We do know that if we generalize the best known decomposition for size 4, we can verify this subject on systems with size 6, one size greater than monolithic verification and the generalized decomposition from size 2.

On one of the other subjects, property 1 of the Relay system with FLAVERS, the generalized decompositions from size 2 are worse than monolithic. We were able to find a

decomposition that could allow us to verify this subject on the system with size 7, one size larger than monolithic verification. However, this decomposition was not easy to find. When looking at the best decompositions for size 2, 3, 4, and 5, we noticed that there is a pattern to the best decompositions that depends on whether or not the size of the system is odd or even. For this subject, Figure 5.10 compares the number of states using the generalized decomposition from size 2, the decompositions based on the odd/even pattern, and the monolithic verification.

The third subject, property 2 of the Chiron multiple dispatcher system for FLAVERS. Monolithic verification and assume-guarantee reasoning using the generalized decomposition from size 2 can verify this subject on a system of size 5, but not 6. Using the generalized decomposition from size 3 allows assume-guarantee reasoning to verify this subject on a system of size 6.

To summarize, although finding best decomposition at a small system size and generalizing it so that it is applicable for larger system sizes was not too costly a process, it does not often produce the best decomposition at those larger sizes. There are subjects where using decompositions other than the generalized decomposition from size 2 allowed us to verify larger systems than when we used those generalized decompositions. Because we were unable to find heuristics that enabled us to find good decompositions, finding decompositions that perform better than the generalized decomposition from size 2 necessitated trying all two-way decompositions for larger system sizes, a process that is probably too costly to be useful in practice.

5.4.3 Discussion

While we tried to find the best decomposition for all subjects at some larger systems sizes, because of the costs involved we did not always attempt this at the largest size for each subject in our study. At the largest size where we could compare the generalized and the best decomposition, they were the same on 10 of 32 subjects with FLAVERS and on

11 of 30 subjects with LTSA. With FLAVERS, for 15 of the 32 subjects we were not able to find the best decomposition at the largest system size we attempted, but, in all of these cases, we were able to find a decomposition that uses less memory than the generalized decompositions from size 2.

As stated previously, our use of generalized decompositions from size 2 did not often allow us to verify a larger system than monolithic verification and, when it did, it only allowed us to verify a system 1 or 2 sizes larger. Had we been able to find the best decomposition for every subject at every system size, we do not think that our results would be significantly different. There might be subjects, like the three discussed in Section 5.4.2, where decompositions other than ones we examined would have allowed us to verify a larger system than monolithic verification. Since there are few subjects on which the best known decomposition is significantly better than the generalized decomposition, as shown in Figures 5.7 and 5.8, we do not believe that there exist other decompositions that would have allowed us to verify a significantly larger system than monolithic verification.

5.5 Does Assume-Guarantee Reasoning Save Time?

While assume-guarantee reasoning was not successfully in increasing the size of the systems that could be verified, if assume-guarantee reasoning could reduce the time needed for verification, it might still be worth using. When we compared the time used by assume-guarantee reasoning to the time used by monolithic verification, we found that assume-guarantee reasoning with FLAVERS uses less time on 11 of the 32 subjects (34.4%) and that assume-guarantee reasoning with LTSA uses less time on 7 of the 30 subjects (23.3%)

Figures 5.11, 5.12, and 5.13 show the amount of time used by assume-guarantee reasoning with generalized decompositions (from size 2) normalized by dividing by the amount of time used by monolithic verification as the size of the systems is increased. Each solid line represents a single subject. A dotted line at 1.0 has been provided for reference. Figures 5.11 and 5.12 show data for FLAVERS. The former shows the data for system sizes

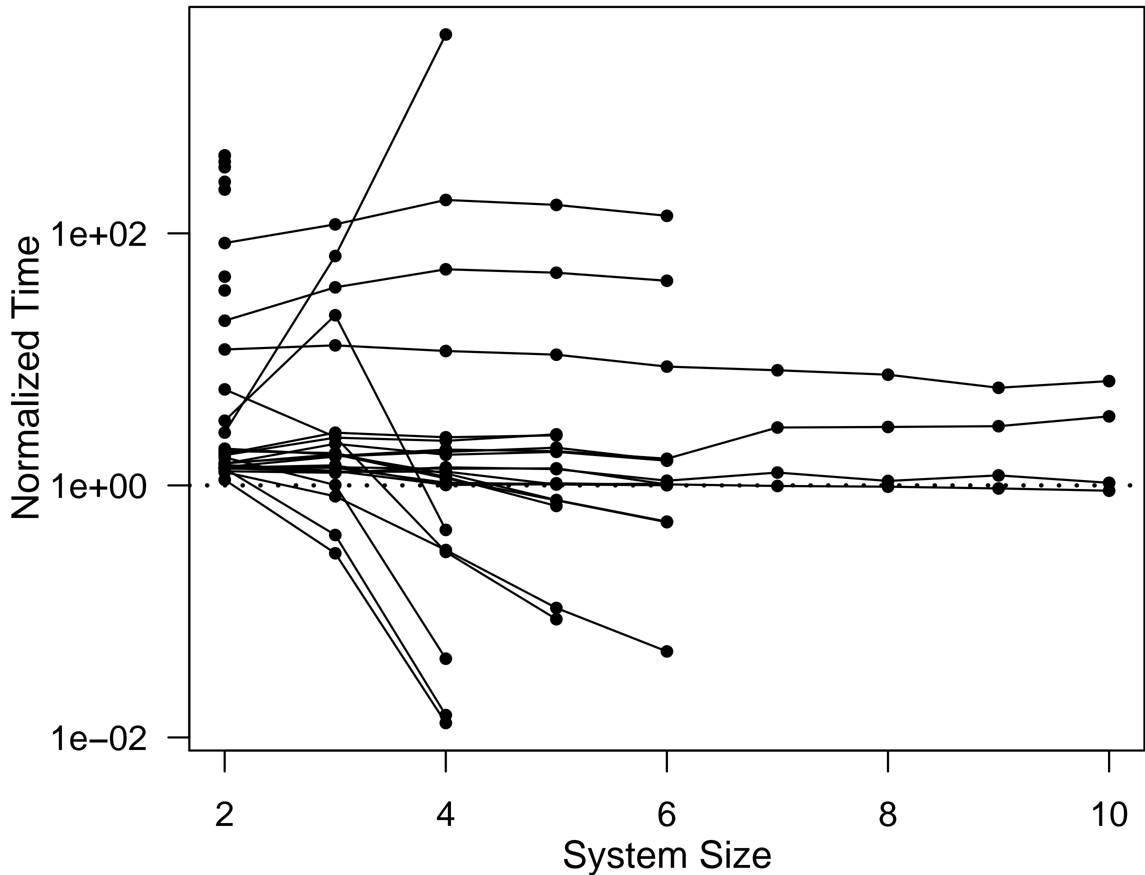

Figure 5.11: Time used by the generalized decompositions for FLAVERS up to size 10

less than or equal to 10 while the latter shows the data for only those subjects that could scale above size 10. Note that each line in Figure 5.12 corresponds to a line that is shown in Figure 5.11. Figure 5.13 shows all of the data for LTSA. As before, for 8 subjects with FLAVERS and 3 subjects with LTSA there are single points at size 2. On these subjects, the generalized decompositions runs out of memory at size 3.

Two subjects in particular stands out in these figures. In Figure 5.12 there are two lines where the normalized time at the largest size where a comparison could be made is significantly lower than at the next smallest size and the lines take a sudden turn downward. These subjects are property 4 of the Gas Station system and property 8 of the Smokers system. For both of these subjects, monolithic verification runs out of memory at the next

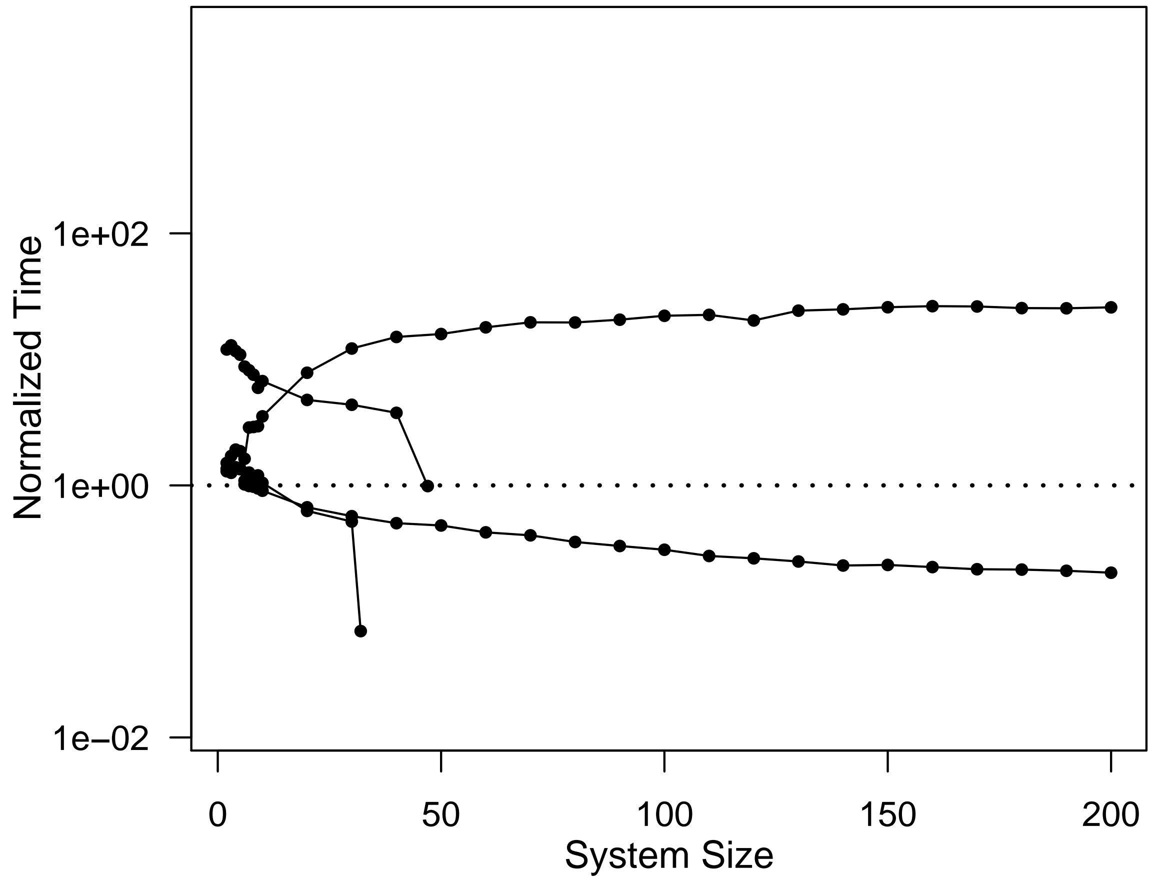

Figure 5.12: Time used by the generalized decompositions for FLAVERS for just those subjects where the largest system size that could be verified is greater than 10

larger system size. We believe that these subjects came very close to using the maximum amount of memory, resulting in the garbage collector being invoked more frequently, causing monolithic verification to use more time than would otherwise have been expected.

To summarize, on the small number of subjects for which assume-guarantee reasoning uses less time than monolithic verification, it sometimes uses significantly less time. When assume-guarantee reasoning uses more time than monolithic verification, it often uses significantly more time, particularly for LTSA. A large portion of this time cost is due to the learning algorithm we used, and this is discussed more in the next section

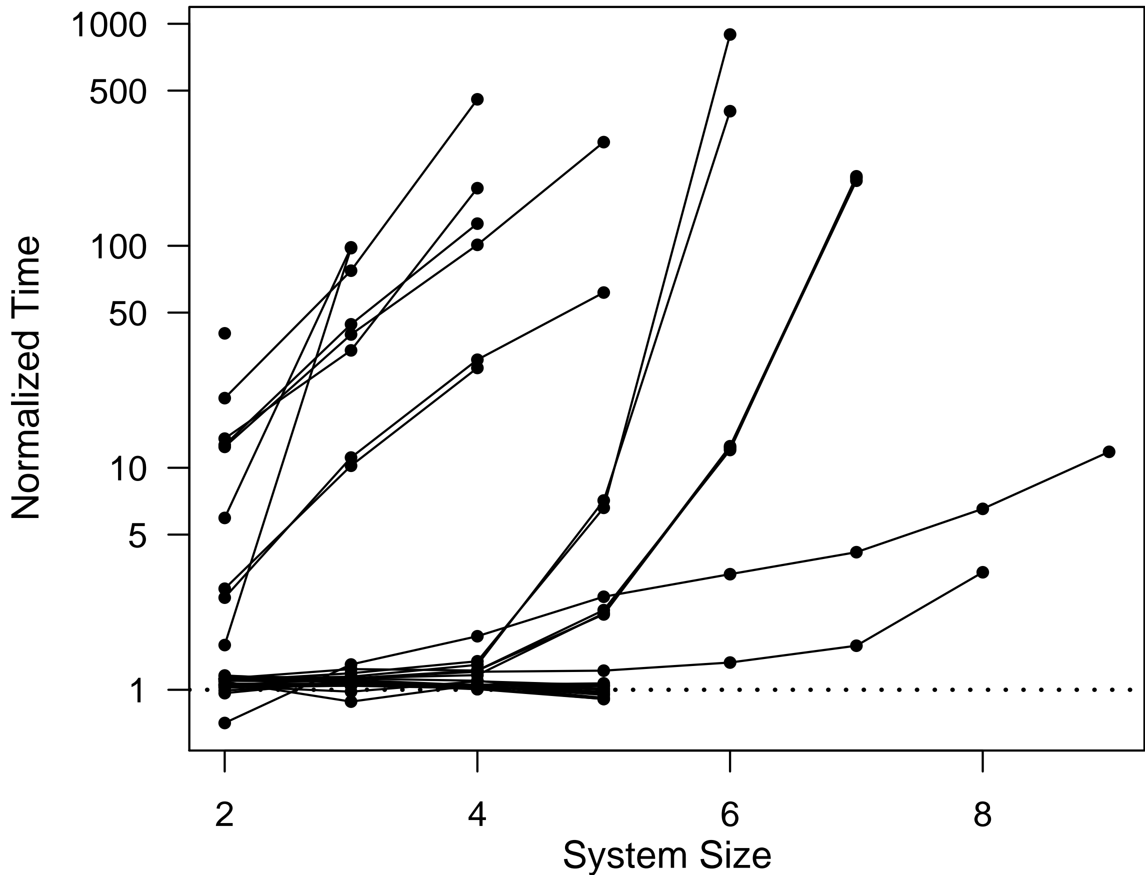

Figure 5.13: Time used by the generalized decompositions for LTSA

5.6 What is the Cost of Using the L* Algorithm?

We also tried to investigate whether or not using the L* algorithm to learn assumptions increased the cost of assume-guarantee reasoning. To do this, we first determined, for each subject, the cost of assume-guarantee reasoning when the L* algorithm is used to learn an assumption. At the end of each verification done with assume-guarantee reasoning, we saved the assumption that was used to complete the assume-guarantee proof. These assumptions were then used to evaluate the cost of assume-guarantee reasoning when assumptions are not learned. For each subject with property P, this cost was determined by checking $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$, letting A be the assumption that was previously learned when P was verified using assume-guarantee reasoning with the L* algorithm. For

each subject, we compared the time and memory costs for the largest sized system on which that subject could be verified using automated assume-guarantee reasoning with the decompositions generalized from size 2.

5.6.1 What is the Memory Cost of Using the L* Algorithm?

For a subject with property P, we consider the amount of memory used during assume-guarantee reasoning with the L* algorithm to be the maximum number of states explored when the teacher answers a query or conjecture of the L* algorithm. We consider the amount of memory used during assume-guarantee reasoning without the L* algorithm to be the maximum number of states explored when verifying $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$, where A is the assumption that was previously learned by the L* algorithm.

For FLAVERS, the amount of memory used by the two approaches is the same on 29 of the 32 subjects. On the remaining three subjects, assume-guarantee reasoning without the L* algorithm uses 78.6%, 86.6%, and 96.7% of the memory used by assume-guarantee reasoning with the L* algorithm. On two of these three subjects, the amount of memory used by assume-guarantee reasoning without the L* algorithm uses more memory than monolithic verification. On the third subject, assume-guarantee reasoning without the L* algorithm uses 0.54% of the memory of monolithic verification (compared to 0.69%, with the L* algorithm). Since this difference is so small, on this third subject we do not believe that learning an assumption affected the results of whether or not assume-guarantee could verify a larger system than monolithic verification. Thus, we do not believe our use of learning significantly impacted the results of our experiments with FLAVERS.

For LTSA, the amount of memory used by the two approaches is the same on 29 of the 30 subjects. On the remaining subject, assume-guarantee reasoning without the L* algorithm uses 14.5% of the memory used by assume-guarantee reasoning with the L* algorithm. On this subject, assume-guarantee reasoning with learning uses 491.1% of memory that monolithic verification uses. Assume-guarantee reasoning without learning uses

Percentage of Time Spent Learning

0 20 40 60 80 100

0 50 100 150 200 250

Assumption Size

18 of 32 subjects have assumption size 1

Figure 5.14: Percentage of time spent learning for FLAVERS

71.4% of memory that monolithic verification uses. Thus, on this subject, our use of learning could have affected whether or not assume-guarantee could verify a larger system than monolithic verification. When we looked at the assumption generated for this system at size 2 and size 3, we could not see a way to generalize the assumption to make it applicable on larger system sizes, thus, our use of learning was necessary to apply assume-guarantee reasoning to verify this subject.

To summarize, in very few subjects does verification with assume-guarantee reasoning using a learned assumption use more memory than verifying that subject with assume-guarantee reasoning using a supplied assumption. Even on the small number of subjects where there is a memory overhead as a result of our use of the L* algorithm to learn an assumption, we do not believe that this overhead had a significant impact on the results of whether or not assume-guarantee reasoning could verify a larger system than monolithic verification.

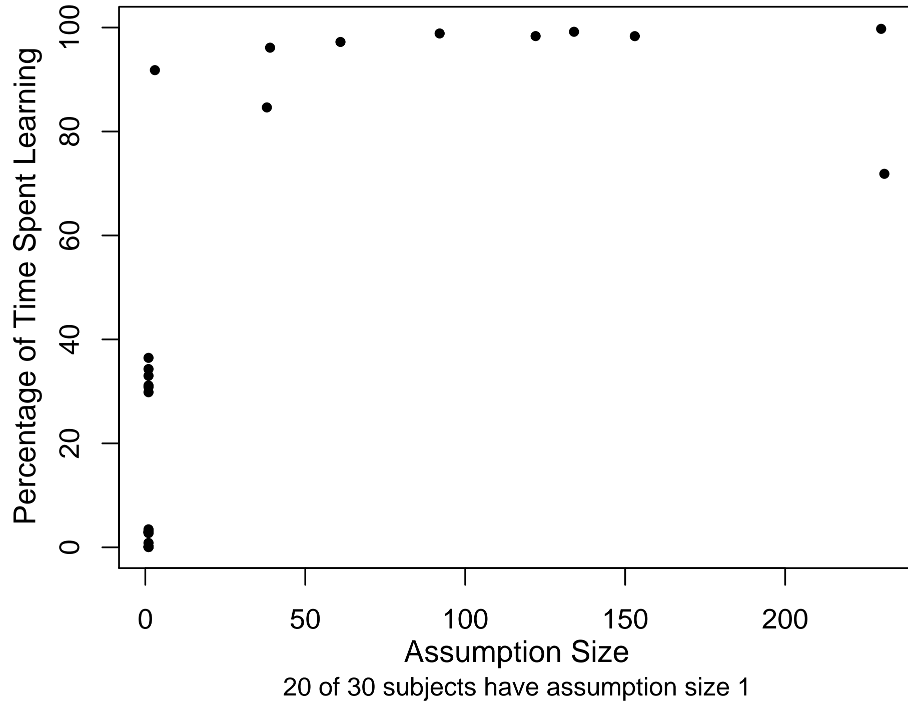

Figure 5.15: Percentage of time spent learning for LTSA

5.6.2 What is the Time Cost of Using the L* Algorithm?

For a subject with property P, we consider the amount of time used during assume-guarantee reasoning with the L* algorithm to be the amount of time needed to verify that subject, including the time to build the artifacts (i.e, the TFGs, LTSs, etc.) and run the L* algorithm. We consider the amount of time used during composition analysis without the L* algorithm to be the amount of time needed to verify $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$, including the time needed to build the artifacts, where A is the assumption that was previous learned by the L* algorithm.

Figures 5.14 and 5.15 show the percentage of time that is spent learning an assumption compared to the size of the assumption that was learned. When the size of the learned assumption is small, fewer than 10 states, less than 50% of the total verification time is spent learning the assumptions. When the size of the learned assumption is larger than 10 states, however, in most cases over 90% of the verification time is spent learning the

assumptions, a substantial overhead. Thus, our use of learned assumptions had a significant impact on the time cost of assume-guarantee reasoning for many subjects.

5.6.3 Reducing the Cost of Using the L* Algorithm

Because of this time overhead, we looked at two ways to reduce the cost of automatically learning an assumption. First, since we were able to use generalized decompositions to apply assume-guarantee reasoning to larger-sized systems, we tried generalizing the assumptions in a similar fashion. When the learned assumption was large, however, it was difficult to understand what behavior the assumption is trying to capture. Without such an understanding, it is not possible to determine what the assumption should be for a larger-sized system. As a result, we were unable to use generalized assumptions to reduce the cost of automated assume-guarantee reasoning.

Second, Groce et al. developed a technique for initializing some of the L* algorithm's data structures when given an automaton that recognizes a language close to the one being learned [57]. By doing this, they reduced the amount of time needed to run Angluin's version of the L* algorithm. To determine if this technique would reduce the cost of using learning, for each subject, we used the L* algorithm to learn an assumption capable of completing an assume-guarantee proof. This assumption was then used to initialize some of the data structures of Angluin's version of the L* algorithm. Performing this initialization reduced the number of queries made by the L* algorithm (and consequently the running time) when compared to not initializing these data structures. In our experiments, initializing these data structures in Angluin's version of the L* algorithm did not offer any performance benefits over using Rivest and Schapire's version of the L* algorithm, which has better worst-case bounds. We have been unable to find a similar technique to initialize the data structures of Rivest and Schapire's version of the L* algorithm because of the constraints this version places on its data structures.

To summarize, using the L* algorithm to learn an assumption often increased the time needed and rarely increased the memory needed to complete an assume-guarantee proof compared to the cost of completing a proof using a supplied assumption. The overhead for using an automated assumption generation technique, however, is probably unavoidable on several of our subjects. Some of the learned assumptions are very large: in fact, one has over 250 states. For such systems, we suspect that small assumptions do not exist that can be used to complete the assume-guarantee proof and analysts cannot be expected to develop these assumptions manually. Thus, some automated support is needed to make assume-guarantee reasoning practical on these systems.

5.7 Threats to Validity

Although our experiments examined several systems in detail, they are still limited in scope: we used two finite-state verifiers, one assume-guarantee reasoning technique, and a small number of systems. Even in this limited context, our experiments were expensive to perform; we examined over 43,500 two-way decompositions and used over 1.54 years of CPU time.

Although we used only two verifiers, we expect that using the L* algorithm to learn assumptions with other verifiers will produce similar results. This conjecture is consistent with the results of Alur et al. for NuSMV in which they found some subjects where assume-guarantee reasoning verifies a larger system than monolithic verification and other subjects where assume-guarantee reasoning uses more memory than monolithic verification [5].

We looked only at one assumption generation technique, which influenced the assumptions used in completing the assume-guarantee proofs. While other assumptions could be used to complete assume-guarantee proofs in our examples, automated support to help find assumptions is necessary to make assume-guarantee reasoning useful in practice. Additionally, we expect that other assumption generation techniques based on two-way decompositions (e.g., [5, 14, 23, 52]) would produce assumptions similar to the ones generated

by the algorithm we used. Since discharging the premises of the assume-guarantee rules tended to be the most expensive part of the analysis with respect to memory, we suspect that using these other techniques will not produce better results. While automated techniques based on assume-guarantee rules that allow for more than two-way decompositions (e.g., [65, 73, 98]) might perform better with respect to memory, there has not yet been enough empirical evaluation of these techniques to draw any conclusions.

Additionally, we looked only at a small number of systems that are mostly based on a client-server architecture and where scaling is achieved by replicating the number of clients. This allowed us to easily increase the size of the system to look at the effects of scaling on assume-guarantee reasoning. Looking at just one kind of scaling, however, might limit the generality of our results. Alur et al., however, looked at systems with different architectures and had results that were similar to ours [5].

We also used one generalization approach when looking at larger system sizes, shown in Figure 5.3, which could have affected our results. In particular, in the case where none of the repeatable tasks are treated in a different way in the property, we put one task into S_1 and the rest into S_2 (or vice versa). For this case, we also looked at splitting the tasks evenly between S_1 and S_2, either by putting the first half of the repeatable tasks in S_1 and the rest in S_2 or by putting the repeatable tasks with odd IDs in S_1 and the rest in S_2 (or vice versa, in both cases). For most of the subjects, these alternative generalization approaches made little difference in the amount of memory used. For one subject with FLAVERS, these alternate generalizations used between three and four times the memory as the one generated using the algorithm shown in Figure 5.3. For another subject with FLAVERS, the alternative generalizations used between one third and one half the memory as the one generated using the algorithm shown in Figure 5.3. Because these alternative generalization approaches did not often produce significantly different results, we do not believe other generalization approaches will be more successful than the one we used.

In their work on compositional analysis, Cheng et al. also looked at the Chiron system and were able to verify properties at larger system size than using their approach than we were with our approach [27]. Cheng et al., however, changed the Chiron system to make it have a smaller state space by replacing some arrays by bitsets. This change was done because without this change the "refactored structure has little hope to scale well compositionally" [27]. In our study, we used the original implementation based on arrays in which the state space grows more quickly. While their results are better on the Chiron system than ours, they did not look at exactly the same system we did.

Despite these threats, our experiments are a careful study of one automated assume-guarantee reasoning technique and raise doubts about the usefulness of assume-guarantee reasoning as an effective compositional analysis technique.

CHAPTER 6

ASSUME-GUARANTEE REASONING OF SOFTWARE USING DESIGN-LEVEL ASSUMPTIONS

While we have been using assume-guarantee reasoning to verify properties of software systems, our analysis of these systems has only looked at the software and not at any other artifacts that may have been written during earlier phases in the development of the software. If a software system has been implemented based on a previously developed design, then the design can be leveraged to aid in finding assumptions for the software, as shown in Figure 6.1.

Consider a system with two subsystems with designs M_1 and M_2 with corresponding software implementations S_1 and S_2. Suppose that the system should obey a property P and it can be shown for some assumption A that both $\langle A \rangle\ M_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A \rangle$. Then, if the system closely matches its design, it should be possible to reuse the assumption A to show that both $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$ are true, potenially reducing the cost of verification.

Alternatively, the verification of the software can be performed by showing that the design obeys the property P (either monolithically or compositionally) and that each component at the software level refines its corresponding design model, meaning $\mathscr{L}(S_i) \subseteq \mathscr{L}(M_i)$. We would expect that if $|A| < |M_i|$ then the cost of using assume-guarantee reasoning of the software would be less expensive than checking refinement. Our experience, as described previously, is that small assumptions do not necessarily result in assume-guarantee reasoning that is inexpensive. Still, future work in assumption generation might find algorithms that produce better assumptions than the learning algorithm we used. As a result, reducing the cost of assume-guarantee reasoning by using artifacts from different phases of software developing is an approach that should be considered.

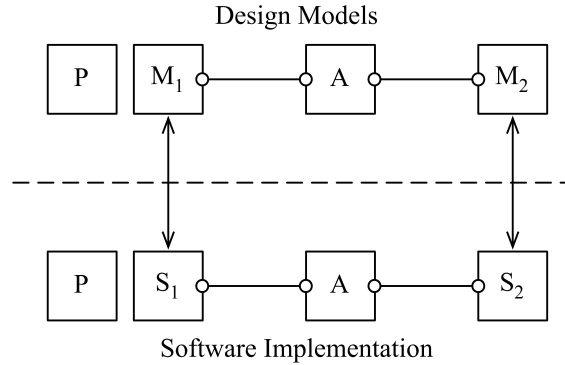

Figure 6.1: Verification at the design level and software level

In [53], we described how assume-guarantee reasoning could be used to verify the executive subsystem of the K9 Mars Rover that was developed at the NASA Ames Research Center. For this example, the design was verified using LTSA and the software was verified using JPF. In Chapter 4, we described how the L* algorithm can be used to learn assumptions for LTSA. In this chapter, we will describe how assume-guarantee triples can be checked using JPF and then describe the case study that uses assume-guarantee reasoning to verify the executive subsystem of the K9 Mars Rover.

6.1 Checking Assume-Guarantee Triples Using JPF

6.1.1 Background

JPF is a reachability-based finite-state verification tool that checks properties of Java software. By default, JPF can check for assertion violations, deadlocks, and uncaught exceptions. While recent versions of JPF allow the user to write extensions that allow JPF to verify a wider range of properties, the version available when these experiments were performed could not be extended in this way. As a result, properties to be verified had to be written so that a violation of the property resulted in an assertion violation in the software. In addition to supporting most of the standard language features of Java, JPF provides a special class called `Verify` that allows analysts to annotate their software to supply directives to JPF. The methods provided by the `Verify` class:

1. allow non-deterministic choices to be made using the methods `random(n)`, which returns a random integer greater than or equal to 0 and less than `n`, and `randomBool()`, which returns a random Boolean,

2. allow a search to be truncated using the method `ignoreIf(condition)`, which causes JPF to backtrack in its search if and only if the given condition is true, and

3. allow regions of code to be marked as atomic using the methods `beginAtomic()` and `endAtomic()`.

6.1.2 Instrumenting the Java Software

In our framework, the assumptions and properties come from LTSA and, as a result, are specified as LTSs. JPF, however, does not provide support for checking properties specified as LTSs or support for restricting the search space based on assumptions specified as LTSs. Before either the assumptions and properties can be adapted for use in JPF, a mapping must be made by the analyst between the events on the transitions of the LTSs and the locations in the software that correspond to the occurrences of those events. This is necessary to allow JPF to notice when events occur that would require updating the state of the assumptions or the properties. For simplicity, we will assume that events in the properties and assumptions correspond to either method calls or to the locking and unlocking of objects in the Java software.

After a mapping has been developed, the Java software then needs to be instrumented so that the state of the assumptions and properties can be tracked as JPF explores the reachable system states. Currently, this instrumentation is done by hand, but tools such as JPax [62] could help in this process. At each point in the software that corresponds to an event in the assumptions or properties, a call to the method `AG_Monitor.event()` is added. This method traps each event and updates the state of the properties and the assumptions appropriately. This method, shown in Figure 6.2, uses Java reflection to determine the name of the thread making the method call (line 2), the method being called (lines 3 to 5), and the

```
    public static void event() {
1)     Verify.beginAtomic()
2)     String threadName = Thread.currentThread().getName();
3)     Throwable throwable = new Throwable();
4)     StackTraceElement st = (throwable.getStackTrace())[1];
5)     String methodName = st.getMethodName();
6)     String className = st.getClassName();
7)     int eventID = getEvent(className, methodName, threadName);
8)     AG_Assumption.event(eventID);
9)     AG_Property.event(eventID);
10)    Verify.endAtomic();
    }
```

Figure 6.2: Method `event` of class `AG_Monitor`

class that contains the method (line 6). These three pieces of information are used as a key to look up the corresponding event from the design (line 7). Then, this event is passed on to the assumption (line 8) and to the property (line 9). It is important that the event is passed to the assumption first. If the same event causes both the assumption and the property to be violated, then this should not result in a property violation. Thus, the assumption must be passed the event first so it can violate before the property has a chance to violate. The entire block is enclosed by JPF directives (lines 1 and 10) which instruct JPF to treat the method body as an atomic step and to interleave no other threads with the execution of this method.

If more information is needed to determine the mapping between the Java software and the events from the design-level model, then the `event` method can be modified to take parameters that encode this extra information. This was necessary in our case study to obtain information about the parameters being passed into method calls, the parameters being returned from method calls, and to trap locks and unlocks of objects.

Assumptions and properties are implemented by the classes `AG_Assumption` and `AG_Property`, respectively. An except of the `AG_Assumption` class is shown in Figure 6.3. This class has a static integer field that records the current state of the assumption (line 1) and an array that encodes the transition function for the assumption (line 2). A

```
   public class AG_Assumption {
1)   private static int state = 0;
2)   private static int[][] trans = ...;

     public static void event(int e) {
3)     state = trans[state][e];
4)     Verify.ignoreIf(state < 0);
     }
   }
```

Figure 6.3: Class `AG_Assumption` (excerpt)

transition that causes the assumption to be violated is represented by having the assumption transition into a state with ID less than zero. The method `event`, which is invoked on line 8 of Figure 6.2, advances the assumption by looking up the next state in the transition table (line 3). If the state is less than zero, this represents that the current event has caused the assumption to be violated. Since we are only interested in finding property violations that occur when the supplied assumption holds, the current path should not be explored any more. The `ignoreIf` statement provided by JPF is used to cause JPF to backtrack in its search when the assumption is violated (line 4).

The `AG_Property` class is similar, except a state with an ID that is less than zero represents a property violation. As a result, line 4 is replaced by `Verify.assert(state >= 0)` allowing JPF to detect property violations.

In this way, we can cause JPF to restrict its search to executions that satisfy assumptions specified as an LTSs and to detect violations of properties specified as an LTSs.

6.1.3 Modeling Environments

JPF analyzes executable Java software and, as a result, expects complete systems as input. To analyze a subsystem using assume-guarantee reasoning with JPF, an environment must be provided for that subsystem, similar to the environment required by FLAVERS as discussed in Section 4.3.1. For FLAVERS, the environment of a subsystem has to provide

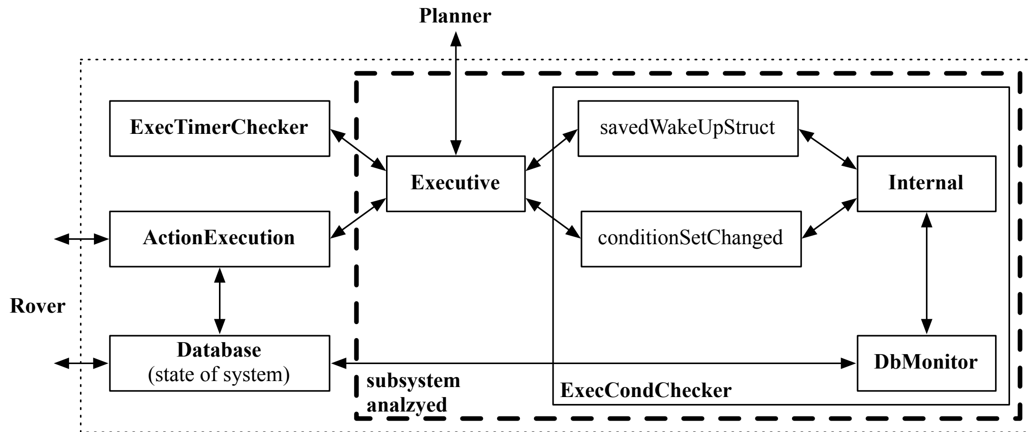

Figure 6.4: The executive of the K9 Mars Rover

call rendezvous and accept rendezvous. For JPF, the environment of a sybsystem has to invoke methods in that subsystem's interface and provide stubs for methods invoked by that subsystem. The environment has to be able to invoke zero or more of the interface methods in any order. Currently, environments are built by hand, although tools such as those described in [108, 109] may help automate this process.

6.2 Case Study

In this section, we describe our case study which verified the executive subsystem of the K9 Mars Rover developed at the NASA Ames Research Center.

6.2.1 Description of the K9 Mars Rover

The executive subsystem of the K9 Mars Rover is designed to execute plans that control the Rover as it explores the surface of Mars. A plan is a hierarchical structure that defines the actions that the Rover must perform. The Rover needs to be autonomous, so plans contain support for branching based on state or temporal conditions as well as flexibility in delaying the starting time of an action.

The executive subsystem needs to monitor the state of the Rover and the state of its environment (e.g., conditions on Mars) to correctly execute its supplied plan. The executive

subsystem has been implemented as a multithreaded system, shown in Figure 6.4. This subsystem is made up of a main coordinating component named Executive, a component that monitors state conditions named ExecCondChecker, a component that monitors temporal conditions named ExecTimerChecker, and a component that is responsible for issuing commands to the Rover named ActionExecution.

The executive is implemented in C++ and contains about 25,000 lines of code, 10,000 of which is the main control code. The remainder defines the data structures that are need for the communication with the actual Rover. The software uses the POSIX thread library and synchronization between threads is accomplished using mutexes and condition variables.

6.2.2 Analysis of the Design of the Rover

The Rover was designed using an ad-hoc flowchart-style notation that describes the synchronization between its components. These are, in essence, extended control-flow graphs and focus on such things as method calls, the locking and unlocking of mutexes, and the signaling of and waiting on of condition variables. Since these resembled LTSs, they were translated manually into about 700 lines of FSP. In this case study, we focused on one property formulated by the developer of the Rover which states:

> For the variable savedWakeUpStruct of the ExecCondChecker thread that is shared with the Executive thread (see Figure 6.4), the property states that: "if the Executive thread reads the value of the variable, then the ExecCondChecker thread should not read this value until the Executive thread clears it first."

Since the savedWakeUpStruct variable is only accessed by two threads, Executive and ExecCondChecker, our analysis was performed on these threads together with the mutexes they use for synchronization. We applied assume-guarantee reasoning using assumptions learned by the L* algorithm with LTSA where $M_1 =$ ExecCondChecker and $M_2 =$ Executive.

The assumption learned has 6 states and expresses the fact that "whenever the Executive thread reads the savedWakeUpStruct variable after acquiring mutex exec, it should not release the mutex until after it clears the savedWakeUpStruct variable." This assumption could not, however, be discharged on the Executive thread. This was because the design of the Executive thread had a bug that then was fixed, allowing the property to be verified.

6.2.3 Analysis of the Implementation of the Rover

The Rover was originally written in C++, but was translated into Java for another case-study [17]. This was a selective translation and focused on the core functionality of the executive subsystem, about 10,000 lines of C++ code. The translated Java version is approximately 7,200 lines of code.

As with the LTSA version, we focused our analysis of the software on the Executive and ExecCondChecker components. We did not start any other threads in the system and supplied the executive subsystem with a simple plan that consisted of one node and no time conditions (the latter are not relevant for the analysis of these components). JPF was able to analyze this configuration without running out of memory. Any other configuration, meaning one where more threads were started or where a more complex plan was used, caused JPF to run out of memory.

6.2.3.1 Environment Modeling

Following our methodology, we set $S_1 =$ ExecCondChecker and $S_2 =$ Executive and attempted to verify $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$, where A was the assumption learned during the verification of the design. As stated previously, to check $\langle A \rangle\ S_1\ \langle P \rangle$, an environment needed to be written to invoke any sequence of method calls for the class ExecCondChecker. This environment, shown in Figure 6.5, loops forever generating method calls (line 1). It begins by making a non-deterministic choice of whether or not to acquire the lock on the Executive object (line 2). If it acquires the lock (line 3), we then use a specialized call to the `AG_Monitor.event` method to trap the lock event (line 4). It then

```
  class Executive { ...
    public void run() { ...
1)      while(true) {
2)        if(Verify.randomBool()) {
3)          synchronized(exec) {
4)            AG_Monitor.event("Executive", "lock");
5)            while(Verify.randomBool()) {
6)              switch(Verify.random(4)) {
7)                case 0: condChecker.deleteSavedWakeup(); break;
8)                case 1: condChecker.getSavedWakeup(); break;
9)                case 2: condChecker.addConditionCheck(id,...); break;
10)               case 3: condChecker.removeConditionCheck(id,...); break;
              } // end switch
            } // end while
11)           AG_Monitor.event("Executive", "unlock");
          } // end synchronized
        } else {
12)         switch(Verify.random(4)) {
13)           case 0:  condChecker.deleteSavedWakeup(); break;
14)           case 1:  condChecker.getSavedWakeup(); break;
15)           case 2:  condChecker.addConditionCheck(id,...); break;
16)           case 3:  condChecker.removeConditionCheck(id,...); break;
          } // end switch
        } // end if-then-else
      } // end while
    } // end run
  } // end Executive
```

Figure 6.5: Environment for the ExecCondChecker

non-deterministically decides whether or not to make a method call (line 5). Depending on this choice, zero or more method calls are made (lines 6 to 10). Once the environment stops making method calls, it releases the lock and generates an event saying the lock was released (line 11). If the choice was made to not acquire the lock on line 2, then the environment invokes a single method (lines 12 to 16).

To maintain a finite list of elements in the list of conditions, we added an annotation forcing JPF to backtrack if more than one call to `addConditionCheck` is made (on either line 9 or 15). This is a reasonable restriction since the configuration uses an input plan with only one node and only one condition can ever be added to it.

To check $\langle true \rangle\ S_2\ \langle A \rangle$, we built stubs that implemented methods that are invoked in the ExecCondChecker thread by the Executive. Some care needed to be taken when doing this

since, for example, the `getSavedWakeup` method can either return null or an object. To simulate this, the method nondeterministically chooses which to return.

6.2.3.2 Results

Our experiments were run on an Intel Xeon 2.2 GHz machine with 4 Gb of memory (although a single process could only access 2 Gb of memory). This system was running RedHat Linux version 8.0 with Sun's JDK version 1.4.2-01. We used JPF version 2.4 using the -no-verify-print, -no-deadlocks, and -verbose flags.

We used JPF to monolithically verify the property described earlier. We also used the assumption A that was learned during the analysis of the design to verify the property using assume-guarantee reasoning. In both cases, we discovered the same error in the Executive that was discovered during the analysis of the design.

After we corrected the error, we reran the verification and discovered that the property could be verified using monolithic analysis but not using assume-guarantee reasoning because Premise 1 did not hold. We reexamined the design and noticed that there was an assumption that was encoded into the model for M_1 but not for S_1. This assumption stated that all accesses to savedWakeupStruct by the Executive would be protected by the lock exec. This assumption was encoded explicitly based on instructions from the developer who gave us the original models and was subsequently discharged on M_2. Using this new assumption, A', we checked that the property holds by showing that $\langle A \wedge A' \rangle\ S_1\ \langle P \rangle$ is true and then discharging both assumptions on S_2.

Table 6.1 gives the results of the experiment. The System column describes the system being analyzed. The States and Transitions columns report the number of states and transitions explored by JPF. The Memory and Time columns report the amount of memory needed and the time taken to perform the analysis.

Table 6.1: Results of verifying the Mars K9 Rover

System	States	Transitions	Memory (Mb)	Time
Whole System	183,132	425,641	952.85	12m, 24s
Whole System (bug)	255	338	23.07	10s
Premise 1, A' as automaton	60,830	134,177	315.98	6m, 55s
Premise 1, A' encoded	53,215	117,756	255.96	4m, 49s
Premise 2, Assumption A	13,884	20,601	118.97	1m, 16s
Premise 2, Assumption A (bug)	145	144	44.49	20s
Premise 2, Assumption A'	13,884	20,601	109.58	1m, 7s
Premise 2, Assumption A' (bug)	13,884	20,601	121.37	49s

The Whole System rows give the results for checking the property monolithically. The version marked bug corresponds to the original system in which the property does not hold while the other version has had the bug fixed so that the property does hold.

The Premise 1 lines report the results of verifying Premise 1. As was mentioned previously, while performing the verification, we discovered that an additional assumption, A' was needed to complete the verification. We looked at two ways of incorporating this assumption into the analysis. The first uses the universal environment shown in Figure 6.5 and uses an automaton representation of A', using the method shown in Figure 6.3. The second uses a modified universal environment that directly encodes A'. This is done by replacing lines 12 to 16 of the universal environment with code that makes a choice only between the two events on lines 15 and 16. The bug that caused a violation of the property in the monolithic analysis was in the Executive, not the ExecCondChecker, so these analyses were not affected by the presence or absence of the bug since they only analyzed S_1, the ExecCondChecker.

The Premise 2 lines report the results for checking Premise 2, in which the assumptions used in checking Premise 1 need to be discharged. We discharged the assumptions A and A' separately, on the system containing the bug and on the system in which the bug is fixed.

From Table 6.1, we can see that using assume-guarantee reasoning does reduce the number of states that JPF needs to explore and the amount of memory necessary for the

analysis in the version of the Rover that does not contain the bug. In the version that does contain the bug, monolithic verification uses less memory because JPF can stop immediately when it discovers the property violation. Since Premise 1 holds whether or not the bug is present, JPF cannot stop early and must run to completion, using more memory than the quickly terminating monolithic verification.

6.3 Discussion

In this chapter, we presented an approach that uses assume-guarantee reasoning at different phases of system development. By learning an assumption to complete an assume-guarantee reasoning proof on the design of a software system, we were able to reuse that assumption to compositionally verify the actual software system resulting in a reduction in memory usage. Still, there are some obstacles to the use of this approach.

First, a suitable assumption has to be found. As our results in Chapter 5 show, finding assumptions that allow assume-guarantee reasoning to use less memory than monolithic verification is difficult and we do not expect this problem to any easier in this context. Still, other assumption generation algorithms might work better and we would expect this approach to benefit from future work in this area.

Second, this approach requires that a design suitable for finite-state verification be developed for the software system being analyzed, which is not often done in practice. In addition, there must be a high degree of correlation between the design and the software, otherwise it will be difficult to effectively use assumptions based on the design for assume-guarantee reasoning of the software.

Finally, many of the steps in our approach have not yet been automated. Currently, the analyst must manually define the mapping between events in the design and locations in the software, annotate the software to trap events when they occur, and write the environments needed for assume-guarantee reasoning of the software. Although there has been some

research that could help in these areas, more automation would be needed to make this approach truly useful.

CHAPTER 7

CONCLUSIONS

Assume-guarantee reasoning is one approach to finite-state verification that has been proposed to address the state-explosion problem. There are several difficulties that are encountered when trying to apply assume-guarantee reasoning to a system. First, if the assume-guarantee rule cannot handle an arbitrary number of subsystems, then a decomposition of the system must be selected in which its subsystems are divided into a suitable number of pieces based on the assume-guarantee rule being used. Second, once a decomposition is selected, it may be difficult to manually find assumptions to complete the assume-guarantee proof.

Recent work in assume-guarantee reasoning has allowed assumptions to be generated automatically, thus removing one of the obstacles to its use. In our initial experiments, we could not find good heuristics on how to decompose the systems, so in the study described here we examined *all* two-way decompositions to find the best one with respect to memory use and then generalized those best decompositions to make them applicable on larger system sizes.

Unfortunately, the results of our experiments are not very encouraging. The vast majority of decompositions explores more states than monolithic verification. While this is not surprising, it is worth noting. The process of examining all two-way decompositions is too costly to be useful in practice, and we do not know a good way to predict whether or not a given decomposition will save memory over monolithic verification. Thus, even in those cases where assume-guarantee reasoning can save memory over monolithic verification, it is unclear how analysts will be able to find those decompositions without some guidance

on how to decompose the system being analyzed. If we restrict our attention to just the best decomposition at the smallest size for each property, then in only about half of the cases we examined does the assume-guarantee reasoning technique we used explore fewer states than monolithic verification.

We were interested in determining if this memory savings would be substantial enough to allow us to verify properties on larger systems than can be verified monolithically. Since it is impractical to examine all two-way decompositions for larger system sizes, we used a generalization approach. For each property, we found the best decomposition for a small system size and then generalized that best decomposition so it could be used on larger system sizes. Using this approach, we found that even when assume-guarantee reasoning can save memory over monolithic verification, there were only eight subjects for which this savings is large enough to allow verification of a larger system. Furthermore, for the cases where assume-guarantee reasoning can verify a larger system than monolithic verification, it cannot significantly increase the size of the systems that can be verified.

Of course, there are decompositions other than the generalized ones that we could have tried on larger systems. Furthermore, we have examples where decompositions other than the generalized decompositions can be used to verify larger systems than can be verified using the generalized decompositions. Unfortunately, we were unable to find such decompositions intuitively and we did not observe any pattern that could be used to select a good decomposition for a given system.

When we initiated this study, we did not expect that assume-guarantee reasoning would save memory in all cases. We were surprised, however, to discover that in about half of our subjects, assume-guarantee used more memory than monolithic verification, no matter what decomposition was selected. In many cases, this additional cost was due to the assumption learned. While the assumptions were almost always smaller than the subsystems they replaced (i.e., $|A| < |S_2|$), they often allowed more behavior than the subsystems do. Thus, checking $\langle A \rangle\ S_1\ \langle P \rangle$ was more expensive than checking $\langle true \rangle\ S_1 \parallel S_2\ \langle P \rangle$. Further-

more, the assumptions were often larger than the property (i.e., $|A| > |P|$), making the cost of checking $\langle true \rangle\ S_2\ \langle A \rangle$ greater than we expected.

Although these results are preliminary, they raise doubts about the usefulness of assume-guarantee reasoning as an effective compositional analysis technique. In our experiments, we found that assume-guarantee reasoning only rarely allowed us to verify larger systems than can be verified monolithically. While automated assume-guarantee reasoning techniques can make compositional analysis easier to use, determining how to apply these techniques most effectively is still difficult, sometimes expensive, and not guaranteed to significantly increase the sizes of the systems that can be verified.

These results, although discouraging, indicate several directions for future work. First, the learning algorithm we used converges on the weakest possible assumption [52], that is, the assumption that allows the most behavior. As stated previously, the high cost for assume-guarantee reasoning was largely due to the fact that the learned assumptions were often more permissive than the subsystems they replaced. Thus, approaches that try to learn more specific assumptions should be developed to determine whether or not they perform better than the approach we used. Second, the assume-guarantee rule we used requires that a system under analysis be divided into two subsystems. Rules that allow decomposition into an arbitrary number of subsystems might perform better. Although there is some evidence of this [21], such rules need to be better studied and evaluated. Finally, heuristics need to be developed to help analysts determine when assume-guarantee reasoning will likely save memory over monolithic verification. In the study reported here, all the systems that we analyzed use a client-server architecture. Systems with other architectures, however, might be more amenable to assume-guarantee reasoning. For instance, a research group at NASA successfully demonstrated how assume-guarantee reasoning could be applied to a system in which a visiting vehicle does autonomous rendezvous and docks with a space station [16]. This system was built from two large subsystems which communicated with each other via a small interface. In this case study, assume-guarantee reasoning was

able to verify properties that could not be verified monolithically. More experimentation with different architectural models is needed to determine if assume-guarantee reasoning is more effective when applied to certain architectures.

Although assume-guarantee reasoning has been advocated for over twenty years as a way to lessen the effects of the state-explosion problem and recent work in automated assumption generation has made assume-guarantee reasoning easier to apply, our work shows that effectively applying assume-guarantee reasoning still remains a difficult task. These results provide insight into research directions that should be pursued and highlight the importance of further experimental evaluation of compositional analysis techniques.

APPENDIX A

DESCRIPTION OF EXAMPLES

In this appendix, we describe the systems and properties we verified in the experiments that were described in Chapter 5.

A.1 Chiron

Chiron is a user-interface framework developed at the University of California at Irvine [79]. In Chiron, there are one or more artist tasks that draw on the screen. There are also one or more events, for example mouse clicks, that artists may be interested in. To listen for events, artists register with a dispatcher. The dispatcher maintains a list of artists registered for each event, and when the dispatcher receives an event, it passes it on to all of the artists that are registered for it. An alternative to a centralized dispatcher is to have dedicated dispatchers for each event [10]. This alternative "multiple dispatcher" version of Chiron was created and also considered in our evaluation.

Because the number of states explored by FLAVERS grows more quickly when the number of artists increases than when the number of events increases, we scaled the number of artists in the system and fixed the number of events at two. We checked nine properties of Chiron, given below:

1. *artist1* never registers for *event1* if it is already registered for this event.

2. If *artist1* is registered for *event1* and the dispatcher receives *event1*, then the dispatcher will not accept another event before passing *event1* to *artist1*.

3. The dispatcher does not notify any artists of *event1* until it receives *event1*.

4. Having received *event1*, the dispatcher never notifies artists of *event2*.

5. If no artists are registered for *event1*, the dispatcher does not notify any artist of *event1*.

6. The dispatcher never gives *event1* to *artist1* if *artist1* is not registered for *event1*.

7. If *artist1* registers for *event1* before *artist2* does, then when the dispatcher receives *event1* it will first notify *artist1* and then *artist2* of this event.

8. The size of the list used to store the IDs of artists registered for *event1* never exceeds the number of artists.

9. Chiron does not terminate while there is an artist that is registered for an event.

A.2 Gas Station

The Gas Station system is a simulation of a self-serve gas station [64]. The Gas Station consists of a set of pumps, an operator, and a set of customers. In the Gas Station, the operator receives prepayment from a customer to use a pump, then the operator activates the pump for that customer. Next, the customer starts pumping gas and then stops pumping gas. Once the customer stops pumping gas, the pump lets the operator know how much gas was pumped. Finally, the operator gives the customer change. Instead of an explicit queue to represent the clients that are waiting to pump gas, the customers block on a rendezvous until the pump is available.

While the number of states explored by FLAVERS grows more quickly when the number of pumps increases than when the number of customers increases, we scaled the number of customers in the system and fixed the number of pumps at two. This was necessary because the language processing toolset was unable to build models for the Gas Station system with five or more pumps. We checked four properties of the Gas Station, given below:

1. *customer1* and *customer2* cannot use *pump1* at the same time.
2. *customer1* loops, first starting pumping and then stopping pumping.
3. *pump1* loops, first letting a customer start pumping, then letting a customer stop pumping.
4. If *customer1* prepays on *pump1*, then *customer1* receives the change for *pump1*.

A.3 Peterson

Peterson's mutual exclusion protocol [96] is an algorithm that provides exclusive access to a critical region to a set of tasks. Unlike other solutions to the mutual exclusion problem, Peterson's protocol does not rely on any higher-level synchronization constructs such as locks. Instead, it achieves mutual exclusion by having the tasks read from and write to arrays stored in shared memory. We scaled the Peterson system by increasing the number of tasks attempting to access the critical region. We checked one property of Peterson's mutual exclusion protocol, given below:

1. Two tasks cannot both be in the critical region at the same time.

A.4 Relay

In the Relay system, a set of tasks get and set the value of a variable that is shared among all the tasks [104]. Unlike Peterson's mutual exclusion protocol, the shared variable is protected by rendezvous, so only one task can get or set the variable at a time. In the Relay system where n tasks are accessing the variable, each of these task is assigned a unique id i such that $0 \leq i < n$. The task with id i repeatedly gets the value of the shared variable and when the value is i, it sets the value to $((i+1) \bmod n)$. We scaled the Relay system by increasing the number of tasks attempting to access the shared variable. We checked one property of the Relay system, given below:

1. The shared variable is always set to 1 in between sets to 0.

A.5 Smokers

The Smokers system consists of a supplier task, a table task, and a set of assembler tasks. In the Smokers system with n assemblers, each assembler is trying to assemble an item which is made up of one of each of n different pieces. The supplier has an infinite supply of all pieces and assembler i has an infinite supply of piece i. The supplier beings by putting $n-1$ different pieces on the table. Next, the assembler that has the missing piece picks up all $n-1$ pieces from the table and then puts all n pieces together to form an item. We scaled the Smokers system by increasing the number of assemblers. We checked eight properties of the Smokers system, given below:

1. The correct assembler makes an item after the supplier finishes putting out pieces.
2. *assembler1* assembles an item after the supplier puts out the pieces he or she needs.
3. Only one assembler can be making an item at a time.
4. *assembler1* and *assembler2* cannot be making an item at the same time.
5. The supplier never put all of the pieces for an item on the table at the same time.
6. After the supplier put out pieces, one item will be assembled.
7. The following actions alternate, in order: *piece1* is put on the table (by the supplier), *piece1* is picked up from the table (by an assembler).
8. The locking protocol for the table is obeyed.

APPENDIX B

SUBJECT NUMBERS

Table B.1 gives the mapping from subject number to subject and Table B.2 gives the mapping from subject to subject number.

Table B.1: Mapping from subject number to subject

Subject Number	System	Property
1	Gas Station	3
2	Chiron multiple	8
3	Chiron multiple	1
4	Chiron multiple	6
5	Gas Station	1
6	Chiron multiple	9
7	Chiron multiple	5
8	Gas Station	4
9	Chiron single	1
10	Chiron single	6
11	Chiron multiple	4
12	Chiron multiple	3
13	Relay	1
14	Chiron multiple	7
15	Chiron multiple	2
16	Chiron single	9
17	Gas Station	2
18	Chiron single	2
19	Chiron single	8
20	Chiron single	7
21	Smokers	8
22	Chiron single	4
23	Chiron single	3
24	Chiron single	5
25	Smokers	6
26	Smokers	2
27	Smokers	1
28	Smokers	7
29	Smokers	3
30	Smokers	4
31	Smokers	5
32	Peterson	1

Table B.2: Mapping from subject to subject number

System	Property	Subject Number
Chiron multiple	1	3
Chiron multiple	2	15
Chiron multiple	3	12
Chiron multiple	4	11
Chiron multiple	5	7
Chiron multiple	6	4
Chiron multiple	7	14
Chiron multiple	8	2
Chiron multiple	9	6
Chiron single	1	9
Chiron single	2	18
Chiron single	3	23
Chiron single	4	22
Chiron single	5	24
Chiron single	6	10
Chiron single	7	20
Chiron single	8	19
Chiron single	9	16
Gas Station	1	5
Gas Station	2	17
Gas Station	3	1
Gas Station	4	8
Peterson	1	32
Relay	1	13
Smokers	1	27
Smokers	2	26
Smokers	3	29
Smokers	4	30
Smokers	5	31
Smokers	6	25
Smokers	7	28
Smokers	8	21

APPENDIX C

DETAILED DATA

This appendix provides detailed data for the experiments described in Chapter 5. Each table in this appendix gives information about the performance of monolithic verification, assume-guarantee reasoning using the generalized decompositions based on the best decomposition from size 2, and the assume-guarantee reasoning using best decomposition of which we know. The State columns give the number of states explored during verification. For the generalized decomposition and the best decomposition, this number is the maximum number of states explored by a verifier when answering a query or a conjecture of the L* algorithm. If the number of the states used by the best decomposition is preceded by a star, this means that the number of states is for the best decomposition of which we know and there may be a better decomposition that explores fewer states. Recall that we imposed a time bound when exploring some decompositions, as described in Section 5.4.1. In Table C.23 the number of states explored by the best decomposition at size 3 is preceded by a greater than sign. This denotes the fact that while none of the decompositions at size 3 completed within the time bound we used, we know that the best decomposition for this property explores at least 100,000 states. Entries where the number of states is “OOM” denote the fact that the verifier ran out of memory and do not have a corresponding entry in the Time column. Entries in the Best States columns that are blank denote a size for which we did not attempt to find the best decomposition. All times are given in seconds.

The majority of these experiments were run on an Intel Xeon 2.2GHz machine with 4Gb of memory (although a single process could only access 2Gb of memory). This system was running RedHat Linux version 8.0 with Sun’s JDK version 1.5.0_04-b05. The data in

Appendices C.2.2, C.2.3, C.2.5, and C.2.6 were run on an Intel Pentium 4 3.2GHz machine with 2Gb of memory. This system was running Fedora Core 2 with Sun's JDK version 1.5.0_06-b05. The difference in machines has no effect on the memory number of states explored, but could effect the timing data. Since we never compare timing data between different subjects and the timing data for each subject was collected on the same machine, this difference does not affect our data analysis.

C.1 FLAVERS Data

C.1.1 Chiron Single

Table C.1: Chiron Single property 1 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	724	0.91	325	1.32	325
3	2,294	1.36	325	1.77	325
4	5,922	2.90	401	2.93	401
5	13,774	6.83	654	6.97	
6	28,568	21.50	995	22.17	

Table C.2: Chiron Single property 2 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	2,756	1.18	3,351	1.76	3,351
3	22,938	3.12	35,513	5.50	35,513
4	202,259	26.34	320,239	29.93	320,239
5	2,014,415	501.10	3,498,885	381.14	
6	21,681,290	12,109.85	37,530,117	6,180.33	

Table C.3: Chiron Single property 3 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	877	0.93	1,316	1.38	1,316
3	2,751	1.38	4,976	2.95	4,976
4	6,751	3.04	12,028	5.32	12,028
5	14,543	7.28	27,558	13.43	
6	27,613	23.68	51,156	37.19	

Table C.4: Chiron Single property 4 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	688	0.93	1,009	1.30	1,009
3	2,048	1.40	3,709	2.37	3,709
4	4,818	2.96	8,613	5.46	8,613
5	10,190	6.68	19,429	13.30	
6	19,056	23.08	35,481	36.41	

Table C.5: Chiron Single property 5 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	3,557	0.96	5,472	1.60	5,472
3	220,779	6.04	135,706	6.08	39,340
4	23,863,255	1,423.87	2,081,070	60.04	133,815
5	OOM		45,357,966	1,574.86	

Table C.6: Chiron Single property 6 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	724	0.95	325	1.30	325
3	2,294	1.39	325	1.78	325
4	5,922	2.37	401	3.05	401
5	13,774	6.79	654	6.90	
6	28,568	21.91	995	21.98	

Table C.7: Chiron Single property 7 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	4,326	1.25	5,299	2.34	5,299
3	51,056	3.79	79,672	6.84	78,791
4	718,341	62.48	1,142,369	72.90	1,134,987
5	12,866,528	2,885.87	22,391,566	1,983.30	

Table C.8: Chiron Single property 8 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	2,756	1.12	3,351	2.19	3,351
3	22,938	3.09	35,513	5.42	35,513
4	202,259	24.09	320,239	29.97	320,239
5	2,014,415	501.00	3,498,885	383.37	
6	21,681,290	11,846.11	37,530,117	6,108.04	

Table C.9: Chiron Single property 9 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	4,910	1.00	4,016	3.26	4,016
3	336,026	9.74	44,941	218.09	44,941
4	37,526,650	3,232.49	310,553	1,433.29	*214,247
5	OOM		8,260,254	12,881.27	
6	OOM		144,145,585	51,854.39	

C.1.2 Chiron Multiple

Table C.10: Chiron Multiple property 1 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	1,521	1.04	325	1.44	325
3	3,933	1.38	325	1.99	325
4	8,663	2.79	489	2.81	488
5	18,389	5.35	762	5.56	
6	35,461	13.73	1,121	13.81	

Table C.11: Chiron Multiple property 2 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	14,797	1.76	12,050	3.06	12,050
3	151,923	7.38	132,857	17.64	46,581
4	1,531,962	90.41	1,288,246	204.04	*100,949
5	18,460,786	1,584.48	16,532,689	4,027.31	

Table C.12: Chiron Multiple property 3 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	7,586	1.09	4,371	22.09	4,371
3	29,372	2.17	11,790	80.92	11,459
4	80,344	4.19	25,387	217.23	25,051
5	207,926	10.07	51,452	490.46	
6	438,252	25.73	91,091	1,081.56	

Table C.13: Chiron Multiple property 4 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	8,560	1.08	4,727	90.43	4,727
3	32,434	2.22	12,802	261.48	12,362
4	90,816	3.93	27,754	723.20	26,363
5	230,464	10.13	56,271	1,702.69	
6	494,250	26.93	100,001	3,708.52	

Table C.14: Chiron Multiple property 5 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	13,991	1.19	5,472	1.62	5,472
3	944,001	15.35	135,706	6.20	*89,773
4	98,848,737	3,830.06	2,081,070	57.60	*191,092
5	OOM		45,357,966	1,373.83	

Table C.15: Chiron Multiple property 6 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	1,521	1.06	325	1.48	325
3	3,933	1.40	325	1.93	325
4	8,663	2.78	489	2.92	488
5	18,389	5.37	762	5.48	
6	35,461	13.66	1,121	13.90	

Table C.16: Chiron Multiple property 7 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	25,099	1.91	20,435	3.39	20,435
3	382,207	12.50	337,455	32.74	71,780
4	6,466,376	311.40	5,499,921	750.16	*3,165,393
5	146,347,904	11,440.27	132,254,739	28,516.35	

Table C.17: Chiron Multiple property 8 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	7,123	1.31	957	1.67	957
3	54,268	3.37	5,941	2.76	5,941
4	431,337	20.88	40,235	6.41	40,235
5	4,049,151	254.35	304,439	27.09	
6	41,046,169	4,157.57	2,572,615	200.24	

Table C.18: Chiron Multiple property 9 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	18,981	1.49	6,731	1.65	6,731
3	1,399,453	26.26	201,162	7.60	*111,099
4	152,928,193	7,492.12	3,646,122	97.56	*341,412
5	OOM		95,201,173	3,356.53	

C.1.3 Gas Station

Table C.19: Gas Station property 1 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	13,674	1.22	4,306	7.05	4,306
3	346,330	6.08	44,602	14.72	44,602
4	6,814,798	129.20	740,840	38.31	*114,824
5	117,309,034	3,581.18	11,244,914	310.76	

Table C.20: Gas Station property 2 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	92	0.83	84	1.25	81
3	130	0.87	110	1.49	110
4	168	0.91	146	1.75	146
5	206	1.06	182	1.98	180
6	244	1.42	218	2.31	
7	282	1.16	254	3.34	
8	320	1.30	290	3.78	
9	358	1.38	326	4.07	
10	396	1.49	362	5.26	
20	776	2.97	722	23.26	
30	1,156	5.54	1,082	67.55	
40	1,536	9.73	1,442	146.51	
50	1,916	16.57	1,802	263.18	
60	2,296	24.73	2,162	444.04	
70	2,676	35.87	2,522	705.55	
80	3,056	51.37	2,882	1,007.73	
90	3,436	72.20	3,242	1,490.61	
100	3,816	87.58	3,602	1,942.02	
110	4,196	112.89	3,962	2,545.47	
120	4,576	166.29	4,322	3,384.70	
130	4,956	178.96	4,682	4,357.70	
140	5,336	218.16	5,042	5,437.74	
150	5,716	265.35	5,402	6,878.62	
160	6,096	318.24	5,762	8,413.88	
170	6,476	378.71	6,122	9,950.98	
180	6,856	434.76	6,482	11,092.81	
190	7,236	512.63	6,842	13,037.20	
200	7,616	592.50	7,202	15,326.82	

Table C.21: Gas Station property 3 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	970	0.82	105	1.11	105
3	1,991	0.87	175	1.21	175
4	3,376	1.07	263	1.45	263
5	5,125	1.15	369	1.57	
6	7,238	1.66	493	1.69	
7	9,715	1.88	635	1.86	
8	12,556	2.12	795	2.07	
9	15,761	2.39	973	2.26	
10	19,330	2.69	1,169	2.44	
20	75,040	9.73	4,119	6.52	
30	167,150	25.59	8,869	14.58	
40	295,660	55.33	15,419	27.74	
50	460,570	103.95	23,769	49.96	
60	661,880	178.81	33,919	75.82	
70	899,590	295.96	45,869	118.80	
80	1,173,700	466.45	59,619	165.87	
90	1,484,210	692.35	75,169	228.75	
100	1,831,120	1,013.25	92,519	312.71	
110	2,214,430	1,445.31	111,669	398.00	
120	2,634,140	1,949.96	132,619	514.45	
130	3,090,250	2,560.25	155,369	637.53	
140	3,582,760	3,433.82	179,919	794.48	
150	4,111,670	4,053.01	206,269	947.55	
160	4,676,980	5,165.89	234,419	1,161.40	
170	5,278,690	6,347.57	264,369	1,371.60	
180	5,916,800	7,654.96	296,119	1,644.28	
190	6,591,310	9,145.25	329,669	1,921.71	
200	7,302,220	10,788.36	365,019	2,189.26	

Table C.22: Gas Station property 4 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	7,075	0.98	2,847	11.74	2,847
3	41,163	2.07	6,674	26.73	6,674
4	108,448	4.74	11,649	55.22	11,649
5	224,575	10.36	17,964	112.86	
6	402,720	23.30	25,631	204.12	
7	656,059	42.17	34,711	345.53	
8	997,768	71.64	45,208	541.39	
9	1,441,023	132.60	57,089	788.78	
10	1,999,000	182.07	70,354	1,223.93	
20	16,787,200	3,240.06	279,124	15,442.27	
30	57,541,200	17,204.01	626,294	75,064.47	
40	137,437,000	59,555.44	1,111,864	224,231.86	
47	223,709,179	450,262.72	1,534,111	444,746.59	
48	OOM		1,599,968	480,113.89	
50	OOM		1,735,834	539,672.30	

C.1.4 Peterson

Table C.23: Peterson property 1 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	201	0.81	3,621	28.55	3,621
3	25,557	4.97	OOM		$>$ 100,000

C.1.5 Relay

Table C.24: Relay property 1 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	237	0.80	174	2.10	174
3	5,271	1.24	6,462	81.96	2,699
4	56,022	3.67	646,453	13,880.79	14,897
5	618,069	41.66	OOM		143,978
6	7,209,228	1,322.30	OOM		

C.1.6 Smokers

Table C.25: Smokers property 1 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	515	0.85	4,571	284.49	4,424
3	12,395	1.74	OOM		*1,007,640

Table C.26: Smokers property 2 with FLAVERS

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	515	0.84	4,292	347.39	4,292
3	12,395	1.70	OOM		*1,007,640

Table C.27: Smokers property 3 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	555	0.87	4,861	193.57	4,861
3	13,087	1.83	OOM		334,463

Table C.28: Smokers property 4 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	555	0.86	5,983	318.07	5,983
3	12,639	1.74	OOM		*1,039,727

Table C.29: Smokers property 5 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	511	0.78	8,308	35.34	7,593
3	13,256	1.61	OOM		*2,178,726

Table C.30: Smokers property 6 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	563	0.82	4,560	210.18	4,560
3	13,865	1.79	OOM		*288,447

Table C.31: Smokers property 7 with FLAVERS

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	479	0.81	4,535	336.70	4,124
3	11,747	1.52	OOM		*812,111

Table C.32: Smokers property 8 with FLAVERS

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	127	0.78	173	1.01	173
3	425	0.89	281	1.12	281
4	937	0.97	419	1.35	419
5	1,757	1.36	587	1.83	
6	2,957	2.02	785	2.20	
7	4,609	2.46	1,013	3.11	
8	6,785	4.03	1,271	4.37	
9	9,557	5.83	1,559	7.00	
10	12,997	9.99	1,877	10.47	
20	99,977	321.78	6,707	201.86	
30	332,957	2,750.24	14,537	1,420.56	
32	403,410	27,833.92	16,463	1,942.04	
33	OOM		17,471	2,380.88	
35	OOM		19,577	3,122.41	

C.2 LTSA Data

C.2.1 Chiron Single

Table C.33: Chiron Single property 1 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	137	2.18	54	2.37	54
3	638	3.55	422	3.14	422
4	2,535	4.85	2,021	5.12	2,021
5	40,127	31.42	42,071	31.47	

Table C.34: Chiron Single property 2 with LTSA

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	137	2.23	54	2.35	54
3	638	2.99	423	3.17	423
4	2,535	5.02	2,022	5.05	2,022
5	40,127	31.59	42,072	32.82	

Table C.35: Chiron Single property 3 with LTSA

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	202	2.29	64	2.36	64
3	908	2.96	679	3.14	679
4	3,448	4.85	3,062	5.11	3,062
5	54,667	31.41	63,262	33.57	

Table C.36: Chiron Single property 4 with LTSA

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	195	2.27	59	2.42	59
3	820	2.95	583	3.14	583
4	3,117	4.77	2,672	4.99	2,672
5	47,523	31.62	52,504	33.27	

Table C.37: Chiron Single property 5 with LTSA

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	137	2.30	54	2.42	54
3	638	3.07	422	3.20	422
4	2,535	4.86	2,021	4.94	2,021
5	40,127	31.14	42,071	31.52	

Table C.38: Chiron Single property 6 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	137	2.24	54	2.29	54
3	638	2.96	423	3.09	422
4	2,535	5.03	2,022	5.30	2,021
5	40,127	31.53	42,072	32.74	

Table C.39: Chiron Single property 7 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	150	2.33	62	2.42	62
3	662	3.08	654	3.02	654
4	2,579	4.80	3,089	5.24	3,089
5	40,238	31.75	63,893	33.29	

Table C.40: Chiron Single property 9 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	137	2.32	54	2.30	54
3	638	2.90	422	3.14	422
4	2,535	4.97	2,021	5.11	2,021
5	40,127	32.16	42,071	31.56	

C.2.2 Chiron Multiple

Table C.41: Chiron Multiple property 1 with LTSA

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	794	1.00	176	1.16	176
3	2,974	1.13	1,122	1.23	802
4	10,920	1.44	5,559	1.49	1,513
5	126,067	3.25	129,228	3.10	

Table C.42: Chiron Multiple property 2 with LTSA

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	794	1.02	200	2.91	200
3	2,974	1.13	2,770	11.54	1,241
4	10,920	1.44	18,638	40.57	3,629
5	126,067	3.41	OOM		

Table C.43: Chiron Multiple property 3 with LTSA

Size	Monolithic		Generalized		Best
	States	Time	States	Time	States
2	1,150	1.10	265	2.86	265
3	3,984	1.13	2,736	12.57	1,096
4	13,798	1.50	18,370	46.02	1,862
5	156,746	3.55	138,537	218.60	

Table C.44: Chiron Multiple property 4 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	1,072	1.02	238	1.12	238
3	3,693	1.13	1,122	1.23	1,003
4	12,919	1.47	5,559	1.50	1,752
5	144,052	3.39	129,228	3.08	

Table C.45: Chiron Multiple property 5 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	794	1.16	176	1.12	176
3	2,974	1.12	1,122	1.23	802
4	10,920	1.51	5,559	1.52	1,513
5	126,067	3.31	129,228	3.01	

Table C.46: Chiron Multiple property 6 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	794	1.01	176	1.13	176
3	2,974	1.14	1,122	1.24	802
4	10,920	1.46	5,559	1.53	1,513
5	126,067	3.24	129,228	3.07	

Table C.47: Chiron Multiple property 7 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	874	1.02	395	1.07	395
3	3,189	1.13	1,990	1.27	1,325
4	11,359	1.43	5,625	1.57	4,131
5	127,790	3.29	63,985	3.17	

Table C.48: Chiron Multiple property 9 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	794	1.01	176	1.13	176
3	2,974	1.12	1,122	1.24	802
4	10,920	1.50	5,559	1.57	1,513
5	126,067	3.35	129,228	3.08	

C.2.3 Gas Station

Table C.49: Gas Station property 1 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	117	1.06	272	1.19	272
3	825	1.19	2,752	1.34	1,932
4	4,905	1.40	26,624	1.71	11,882
5	25,893	1.86	249,856	4.25	
6	125,469	3.47	2,293,760	41.74	
7	570,321	8.51	20,709,376	1,749.56	
8	2,467,665	37.46	OOM		

Table C.50: Gas Station property 2 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	117	1.06	272	1.22	272
3	825	1.17	2,752	1.32	1,932
4	4,905	1.40	26,624	1.63	11,882
5	25,893	1.91	249,856	4.22	
6	125,469	3.42	2,293,760	42.78	
7	570,321	8.72	20,709,376	1,759.91	
8	2,467,665	36.03	OOM		

Table C.51: Gas Station property 3 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	117	1.05	272	1.18	272
3	825	1.17	2,752	1.45	1,932
4	4,905	1.63	26,624	1.99	11,882
5	25,893	1.95	249,856	4.26	
6	125,469	3.49	2,293,760	42.92	
7	570,321	8.92	20,709,376	1,751.77	
8	2,467,665	36.90	OOM		

Table C.52: Gas Station property 4 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	117	1.07	899	1.19	899
3	825	1.19	7,082	1.34	7,082
4	4,905	1.41	30,935	1.70	30,935
5	25,893	2.05	97,004	2.50	
6	125,469	3.46	246,347	4.59	
7	570,321	8.49	540,854	13.41	
8	2,467,665	36.13	4,782,969	122.26	
9	10,267,965	294.72	OOM		

C.2.4 Peterson

Table C.53: Peterson property 1 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	52	4.42	130	7.03	130
3	2,857	3.68	924,345	359.20	924,345

C.2.5 Relay

Table C.54: Relay property 1 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	18	1.51	12	1.07	12
3	55	0.90	125	1.17	42
4	144	1.05	412	1.83	112
5	349	1.04	1,159	2.73	244
6	810	1.04	3,020	3.45	
7	1,831	1.08	7,652	4.50	
8	4,068	1.09	18,608	7.12	
9	8,929	1.20	43,856	14.16	

C.2.6 Smokers

Table C.55: Smokers property 1 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	41	0.99	115	20.39	115
3	97	1.08	1,681	83.44	1,681
4	181	1.22	55,453	557.38	
5	296	1.33	OOM		

Table C.56: Smokers property 2 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	41	1.02	80	41.14	80
3	97	1.12	OOM		1,681
4	181	1.19	OOM		

Table C.57: Smokers property 3 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	41	1.01	96	12.79	96
3	97	1.06	1,681	46.94	958
4	181	1.22	55,453	153.54	
5	296	1.32	OOM		

Table C.58: Smokers property 4 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	41	1.02	96	12.67	96
3	97	1.09	1,681	43.44	958
4	181	1.18	55,453	119.17	
5	296	1.32	2,458,887	386.87	
6	445	1.55	OOM		

Table C.59: Smokers property 5 with LTSA

	Monolithic		Generalized		Best
Size	States	Time	States	Time	States
2	41	1.10	95	6.54	95
3	97	1.10	1,087	108.23	958
4	181	1.22	OOM		

Table C.60: Smokers property 6 with LTSA

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	41	1.01	132	13.67	132
3	97	2.12	1,681	71.65	958
4	181	1.22	55,453	221.80	
5	296	1.35	OOM		

Table C.61: Smokers property 7 with LTSA

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	41	1.03	96	1.13	96
3	97	1.07	900	1.27	900
4	181	1.22	22,464	1.64	
5	296	1.33	691,488	8.78	
6	445	1.55	25,165,824	1,387.08	

Table C.62: Smokers property 8 with LTSA

Size	Monolithic States	Monolithic Time	Generalized States	Generalized Time	Best States
2	41	1.01	88	1.13	88
3	97	1.10	1,000	1.26	900
4	181	1.22	20,736	1.58	
5	296	1.31	537,824	9.33	
6	445	1.57	16,777,216	634.20	

BIBLIOGRAPHY

[1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large database. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216. June 1993.

[3] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[4] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In Hu and Vardi [72], pages 521–525.

[5] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In Etessami and Rajamani [45], pages 548–562.

[6] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, and Yichen Xie. Zing: Exploiting program structure for model checking concurrent software. In Philippa Gardner and Nobuko Yoshida, editors, *Proceedings of the Fifteenth International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. August-September 2004.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.

[8] George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[9] George S. Avrunin, James C. Corbett, and Matthew B. Dwyer. Benchmarking finite-state verifiers. *International Journal on Software Tools for Technology Transfer*, 2(4):317–320, 2000.

[10] George S. Avrunin, James C. Corbett, Matthew B. Dwyer, Corina S. Păsăreanu, and Stephen F. Siegel. Comparing finite-state verification techniques for concurrent software. TR 99-69, University of Massachusetts, Department of Computer Science, November 1999.

[11] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 201–213. June 2001.

[12] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Havelund et al. [61], pages 113–130.

[13] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Proceedings of the Eighth SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 101–122. May 2001.

[14] Howard Barringer, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proceedings of the Second Workshop on Specification and Verification of Component-Based Systems*, pages 14–21. September 2003.

[15] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[16] Guillaume Brat, Ewen Denney, Dimitra Giannakopoulou, Jeremy Frank, and Ari Jónsson. Verification of autonomous systems for space applications. In *Proceedings of the IEEE Aerospace Conference*. March 2006.

[17] Guillaume Brat, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Păsăreanu, Arnaud Venet, and Willem Visser. Experimental evaluation of V&V tools on Martian rover software. In *SEI Software Model Checking Workshop*. March 2003.

[18] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[19] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[20] Tevfik Bultan, Jeffrey Fischer, and Richard Gerber. Compositional verification by model checking for counter-examples. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 224–238. January 1996.

[21] Sagar Chaki, Edmund Clarke, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Abstraction and assume-guarantee reasoning for automated software verification. Technical Report 05.02, Research Institute for Advanced Computer Science, October 2004.

[22] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[23] Sagar Chaki, Edmund M. Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In Etessami and Rajamani [45], pages 534–547.

[24] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[25] Robert Chatley, Susan Eisenbach, and Jeff Magee. MagicBeans: a platform for deploying plugin components. In Wolfgang Emmerich and Alexander L. Wolf, editors, *Proceedings of the Second International Working Conference on Component Development*, volume 3083 of *Lecture Notes in Computer Science*, pages 97–112. May 2004.

[26] Yung-Pin Cheng. Crafting a Promela front-end with abstract data types to mitigate the sensitivity of (compositional) analysis to implementation choices. In Godefroid [54], pages 139–153.

[27] Yung-Pin Cheng, Michal Young, Che-Ling Huang, and Chia-Yi Pan. Towards scalable compositional analysis by refactoring design models. In *Proceedings of the Ninth European Software Engineering Conference held jointly with the Eleventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 247–256. September 2003.

[28] Shing-Chi Cheung, Dimitra Giannakopoulou, and Jeff Kramer. Verification of liveness properties using compositional reachability analysis. In Mehdi Jazayeri and Helmut Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference held jointly with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 227–243. September 1997.

[29] Shing-Chi Cheung and Jeff Kramer. Enhancing compositional reachability analysis with context constraints. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 115–125. December 1993.

[30] Shing-Chi Cheung and Jeff Kramer. Compositional reachability analysis of finite-state distribute systems with user-specified constraints. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–150. October 1995.

[31] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the Fourteenth International Conference on Computer-Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. July 2002.

[32] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 353–362. June 1989.

[33] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the Twelfth International Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. July 2000.

[34] Edmund M. Clarke and Robert P. Kurshan, editors. *Proceedings of the Second International Workshop on Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*. June 1990.

[35] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[36] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. April 2003.

[37] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, April 1996.

[38] James C. Corbett and George S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, January 1995.

[39] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the* 22nd *International Conference on Software Engineering*, pages 439–448. June 2000.

[40] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. January 1977.

[41] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68. June 2002.

[42] Matthew B. Dwyer. Modular flow analysis for concurrent software. In *Proceedings of the Twelfth IEEE International Conference on Automated Software Engineering*, pages 264–273. November 1997.

[43] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology*, 13(4):359–430, October 2004.

[44] Matthew B. Dwyer, John Hatcliff, Robby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Willem Visser, and Hongjun Zhen. Tool-supported program abstraction for finite-state verification. In *Proceedings of the* 23rd *International Conference on Software Engineering*, pages 177–187. May 2001.

[45] Kousha Etessami and Sriram K. Rajamani, editors. *Proceedings of the Seventeenth International Conference on Computer-Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*. July 2005.

[46] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *Proceedings of the Tenth European Software Engineering Conference held jointly with the Thirteenth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 227–236. September 2005.

[47] Cormac Flanagan, Stephen N. Freund, Shaz Qadee, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1–3):153–183, June 2005.

[48] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In Thomas Ball and Sriram K. Rajamani, editors, *Proceedings of the Tenth SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. May 2003.

[49] Cédric Fournet, Tony Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance. In Rajeev Alur and Doron Peled, editors, *Proceedings of the Sixteenth International Conference on Computer-Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. July 2004.

[50] Dimitra Giannakopoulou, Jeff Kramer, and Shing-Chi Cheung. Behaviour analysis of distributed systems using the Tracta approach. *Automated Software Engineering*, 6(1):7–35, January 1999.

[51] Dimitra Giannakopoulou and Corina S. Păsăreanu. Learning-based assume-guarantee verification. In Godefroid [54], pages 282–287.

[52] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Assumption generation for software component verification. In *Proceedings of the Seventeenth IEEE International Conference on Automated Software Engineering*, pages 3–12. September 2002.

[53] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proceedings of the* 26th *International Conference on Software Engineering*, pages 211–220. May 2004.

[54] Patrice Godefroid, editor. *Proceedings of the Twelfth SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*. August 2005.

[55] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[56] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In Clarke and Kurshan [34], pages 186–196.

[57] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 357–370. Springer-Verlag, April 2002.

[58] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[59] Jan Hajek. Automatically verified data transfer protocols. In *Proceedings of the 4th International Computer Communications Conference*, pages 749–756. September 1978.

[60] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtul-Trauring, and Mark Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[61] Klaus Havelund, John Penix, and Willem Visser, editors. *Proceedings of the Seventh SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*. August–September 2000.

[62] Klaus Havelund and Grigore Roşu. Java PathExplorer - a runtime verification tool. In *Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics, and Automation in Space*. 2001.

[63] Matthew S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library Programming Language Series. Elsevier, North-Holland, 1977.

[64] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[65] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In Warren A. Hunt, Jr. and Fabio Somenzi, editors, *Proceedings of the Fifteenth International Conference on Computer-Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. July 2003.

[66] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70. January 2002.

[67] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In Hu and Vardi [72], pages 440–451.

[68] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[69] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[70] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[71] Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In Susanne Graf and Laurent Mounier, editors, *Proceedings of the Eleventh SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 76–91. April 2004.

[72] Alan J. Hu and Moshe Y. Vardi, editors. *Proceedings of the Tenth International Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. June–July 1998.

[73] Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, July 2000.

[74] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the* 22nd *International Conference on Software Engineering*, pages 730–733. June 2000.

[75] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 14–21. August 2000.

[76] Ralph D. Jeffords and Constance L. Heitmeyer. A strategy for efficiently verifying requirements. In *Proceedings of the Ninth European Software Engineering Conference held jointly with the Eleventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 28–37. September 2003.

[77] S. C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual, AT&T Bell Laboratories*. 1978.

[78] C. B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.

[79] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218. May 1991.

[80] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.

[81] Patrick Maier. Compositional circular assume-guarantee rules cannot be sound and complete. In Andrew D. Gordon, editor, *Proceedings of the Sixth International Conference on Foundations of Software Science and Computational Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 343–357. April 2003.

[82] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[83] Kenneth L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Hu and Vardi [72], pages 110–121.

[84] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[85] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 24–34. November 1998.

[86] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the* 21st *International Conference on Software Engineering*, pages 399–410. May 1999.

[87] Gleb Naumovich, Lori A. Clarke, and Jamieson M. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 57–65. September 1999.

[88] Gleb Naumovich, Lori A. Clarke, and Leon J. Osterweil. Efficient composite data flow analysis applied to concurrent programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 51–58. June 1998.

[89] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis gereration. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[90] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[91] Leon J. Osterweil and Lloyd D. Fosdick. DAVE - a validation error detection and documentation system for Fortran programs. *Software–Practice and Experience*, 6(4):473–486, October–December 1976.

[92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. June 1992.

[93] Corina S. Păsăreanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183. Springer-Verlag, September 1999.

[94] Suhas S. Patil. Limitations and capabilities of Dijktra's semaphore primitives for coordination among processes. Computational Structures Group Memo 57, Project MAC, February 1971.

[95] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In David L. Dill, editor, *Proceedings of the Sixth International Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. June 1994.

[96] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[97] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzystof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, October 1984.

[98] Claudio de la Riva and Javier Tuya. Modular model checking of software specifications with simultaneous environment generation. In Farn Wang, editor, *Proceedings of the Second International Conference on Automated Technology for Verification and Analysis*, volume 3299 of *Lecture Notes in Computer Science*, pages 369–383. October-November 2004.

[99] Claudio de la Riva, Javier Tuya, and José R. de Diego. Modular model checking of SA/RT models using association rules. In *Proceedings of the First International Workshop on Model-based Requirements Engineering*, pages 61–68. November 2001.

[100] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, April 1993.

[101] Barbara G. Ryder. The PFORT verifier. *Software–Practice and Experience*, 4(4):359–377, October–December 1974.

[102] Krishan K. Sabnani, Aleta M. Lapone, and M. Ümit Uyar. An algorithmic procedure for checking safety properties of protocols. *IEEE Transactions on Communications*, 37(9):940–948, September 1989.

[103] Gil Shurek and Orna Grumberg. The modular framework of computer-aided verification. In Clarke and Kurshan [34], pages 214–223.

[104] Stephen F. Siegel and George S. Avrunin. Improving the precision of INCA by eliminating solutions with spurious cycles. *IEEE Transactions on Software Engineering*, 28(2):115–128, February 2002.

[105] Kuo-Chung Tai and Pramod V. Koppol. An incremental approach to reachability analysis of distributed programs. In *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 141–151. December 1993.

[106] Jianbin Tan, George S. Avrunin, and Lori A. Clarke. Managing space for finite-state verification. In *Proceedings of the* 28th *International Conference on Software Engineering*, pages 152–161. May 2006.

[107] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13. November 1988.

[108] Oksana Tkachuk and Matthew B. Dwyer. Automated environment generation for software model checking. In *Proceedings of the Ninth European Software Engineering Conference held jointly with the Eleventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 188–197. September 2003.

[109] Oksana Tkachuk, Matthew B. Dwyer, and Corina Păsăreanu. Automated environment generation for software model checking. In *Proceedings of the Eighteenth IEEE International Conference on Automated Software Engineering*. 2003.

[110] Antti Valmari. A stubborn attack on state explosion. In Clarke and Kurshan [34], pages 166–175.

[111] Willem Visser, Klaus Havelund, Guillaume P. Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[112] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the 1991 Symposium on Testing, Analysis, and Verification*, pages 49–59. October 1991.