



This is to certify that the

dissertation entitled


INTEGRATIVE ANALYSIS OF STATE-BASED REQUIREMENTS FOR
COMPLETENESS AND CONSISTENCY

presented by

Barbara Jean Czerny

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science


Major professor

Date 5/9/98

INTEGRATIVE ANALYSIS OF STATE-BASED
REQUIREMENTS FOR COMPLETENESS AND CONSISTENCY

By

Barbara Jean Czerny

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Department of Computer Science

May 1998

ABSTRACT

INTEGRATIVE ANALYSIS OF STATE-BASED REQUIREMENTS FOR
COMPLETENESS AND CONSISTENCY

By

Barbara Jean Czerny

Statically analyzing requirements specifications to assure that they possess desirable properties is an important activity in any rigorous software development project. All other stages of development depend upon the requirements specification. In addition, errors in the requirements that go undetected and propagate to later stages of development (the design and implementation stages) are the most costly to correct. Therefore, it is important to ensure that the requirements document satisfies certain desired properties before proceeding to later stages of the development process.

However, static analysis is performed on a formal model of the requirements that is an abstraction of the original requirements specification. Some degree of abstraction is necessary or the analysis becomes intractable. The output from the analysis is a report of the desirable properties that the requirements specification fails to satisfy. In many cases, abstractions in the analysis model lead to spurious errors in the analysis output. Spurious errors are conditions that are reported as errors, but information that was abstracted out of the analysis model precludes the reported conditions from being satisfied. A high ratio of

spurious errors to true errors in the analysis output makes it difficult, error-prone, and time consuming to find and correct the true errors in the specification.

Two desirable properties that certain requirements documents should satisfy (for example, the requirements for critical systems) are completeness (a behavior is specified for every possible input) and consistency (no conflicting behaviors are specified). Analyzing for completeness and consistency in state-based requirements generalizes to analyzing complex logical expressions for satisfiability and mutual exclusion. Two methods for analyzing logical expressions for satisfiability and mutual exclusion are symbolic methods such as those that rely on Binary Decision Diagrams (BDDs), and reasoning methods such as theorem proving. Symbolic methods are fast and fully automated, but generate output that may contain many spurious errors since the analysis model contains many abstractions. Reasoning methods tend to be slower and require more manual intervention, but generate more accurate output since the analysis model contains fewer abstractions.

The objective of this research is to develop a technique for analyzing logical expressions for satisfiability and mutual exclusion that is fast enough to be used on a day-to-day basis, automated, and that generates analysis output with a small ratio of spurious errors to true errors. The results of the research are: (1) an iterative technique that integrates the strengths of a symbolic and a reasoning component to analyze logical expressions for satisfiability and mutual exclusion and circumvents the weaknesses of the components, and (2) a simple technique that uses a symbolic representation of logical expressions to help identify abstractions in a model that are causing spurious errors in the analysis output.

© Copyright May 1998 by Barbara Jean Czerny
All Rights Reserved

To my Lord and Savior Yeshua (Jesus) the Messiah;
Thank you for giving your life so that I could have eternal life;
Thank you for paying the penalty for my sins that I could not pay;
And thank you for taking what was only a dream, and making it a reality.

ACKNOWLEDGMENTS

Thus says the LORD: 'Let not the wise man glory in his wisdom, let not the mighty man glory in his might, let not the rich man glory in his riches; but let him who glories glory in this, that he understands and knows me, that I am the LORD who practice steadfast love, justice, and righteousness in the earth; for in these things I delight, says the LORD.' (Jeremiah 10:23-24)

Unless the LORD builds the house, those who build it labor in vain. Unless the LORD watches over the city, the watchman stays awake in vain. (Psalm 127:1)

First and foremost, I must thank my Father in Heaven *Elohim* (The Creator), *El Elyon* (The God Most High). Thank you for revealing yourself to me (and for continuing to reveal yourself to me). You are *El Shaddai* (The All-Sufficient One). You are *Adonai* (The LORD). You are *Jehovah-jireh* (The LORD Will Provide). You are *Jehova-nissi* (The LORD My Banner). You are *Jehovah-raah* (The LORD My Shepherd). You are more than I or any of us can possibly comprehend. You are *Jehova* (The Self Existent One). Thank you for your gentle and loving guidance. Thank you for your love, patience and grace. Thank you for your wisdom, knowledge, insight, and understanding. Thank you for helping me to grow. Thank you for being my refuge and strength, a well proved help in times of trouble (Psalm 46:1). And Father, thank you for blessing me with my thesis advisor, Dr. Mats P.E. Heimdahl, without whom this dissertation would not have been possible.

Words cannot express the deep gratitude I have for my thesis advisor Dr. Mats P.E. Heimdahl. Thank you Mats for your wisdom, guidance, patience, and grace. Thank you for your constant encouragement and for your undying belief in me even when I didn't believe in myself. Thank you for letting me be honest and open about my feelings. Thank you for helping me to grow in knowledge and maturity.

I would also like to express my deep gratitude to Dr. Betty H.C. Cheng and Dr. Abdol H. Esfahanian. Thank you for serving on my committee. Thank you for believing in me and supporting

me throughout my graduate studies at Michigan State University. Thank you for not giving up on me. Thank you Dr. Cheng for your encouragement, wisdom, and guidance, and thank you for your listening ear and wise advice when things didn't always go the way I expected or hoped they would; thank you for being there for me when I had given up.

I express my thanks also to Dr. Laurie K. Dillon and Dr. Carrie J. Heeter for serving on my committee. Thank you for your help and support.

I express my sincere thanks also to Dr. Richard J. Reid for his support and encouragement throughout my years as a Master's student. Thank you also Dr. Reid for expressing confidence in me and for encouraging me to stay on as a Doctoral student. And thank you for helping me to get into the Doctoral program and for helping me get started as a Doctoral student.

Thank you to my sisters and brothers in the LORD Dr. Sally Jean Howden (and her mom), (Dr.) Ralph and Dr. Utami DiCosty, Dr. Barbara D. Birchler (and her mom), Gretel V. Coombs, Karissa Miller, Dr. Tom and Edna Manetsch, Alice M. Ribby, and Daniel Olujimi Josiah-Akintonde for your constant prayers and support. A special thank you to Dr. Tom and Edna Manetsch for the guidance, understanding, acceptance, help, and wise counsel you provided me in addition to your constant prayers and support. A special thanks also to Alice M. Ribby. Thank you Alice for everything you have done to help me grow and mature and start becoming the woman God knows I can become. Thank you also to (Dr.) William E. McUumber, and Dr. Yile Enoch Wang for your encouragement and support and for advice and tips to help me get Linux up and running on my computer. Thanks Bill for also giving me tips to get my zip drive up and running so I could back up my dissertation and other important items. Thanks to the department secretaries Linda L. Moore, Donna Landon, and Beverly Wallace for your prayers, support, and for helping me keep up with all the paperwork required; thanks for everything you did when the "crises" arose.

Thanks (Dr.) Stephen K. Wagner (C++ Guru and manager) for your help with some tough

problems I had with C++. Thanks to all the managers for their help with system problems and questions. Thanks also to Cathy Davison and the other department secretaries for all of your help.

Last but not least, I would like to thank my father and mother for helping me out with some high cost items such as my computer. Thank you also for always stressing the importance of education and saving from early on in my life. Thank you for making me think early in my life that it was normal for everyone to go to college after graduating high school. And thanks Dad for encouraging me to choose computer science as my field of study in my first year as an undergraduate. Thanks also to my big brother John M. Czerny who I have always admired and respected. Thank you John for letting your little sister tag along with you and your friends when I was young, and for teaching me how to play sports and expand my horizons with some of the risky things we did. Thanks also John, for sticking with me during my years in graduate school.

TABLE OF CONTENTS

LIST OF FIGURES	xii
1 Introduction and Motivation	1
1.1 Problem Statement	4
1.2 Contributions	6
1.3 Organization of Dissertation and Guide to Reading	7
2 Analysis Techniques	9
2.1 Dynamic Analysis	11
2.2 Static Analysis Techniques	11
2.2.1 Reachability Analysis	12
2.2.2 Model Checking	14
2.2.3 Theorem Proving	18
2.2.3.1 Characteristics of Proof Systems	19
2.2.3.2 Basic Proof Systems	20
2.2.3.2.1 Sequent Calculus Systems	21
2.2.3.2.2 Resolution Systems	22
2.2.3.3 Search Strategies	23
2.2.3.3.1 Resolution Strategies	23
2.2.3.3.2 Natural Strategies	25
2.2.3.4 Specific Theorem Provers	26
2.2.3.4.1 Prototype Verification System	26
2.2.3.4.2 PROLOG	31
2.2.3.4.3 Larch Prover	36
2.2.3.4.4 Knuth-Bendix	38
2.3 Summary	39
3 Analysis of Software Requirements	41
3.1 Analysis of State-Based Requirements for Completeness and Consistency	43
3.1.1 SCR	46
3.1.2 RSML	51
3.1.3 Common Problem Between Methods: Spurious Errors	55
3.1.3.1 Binary Decision Diagrams	56
3.1.3.2 Problems With Symbolic Representation of Predicates	59
3.1.4 Summary	64
3.2 Model Checking Software Requirements	66
3.3 Spurious Errors Revisited	70
3.4 Summary	71

4 Analysis Method	73
4.1 General Analysis Process	76
4.1.1 Applying the Analysis Approach to Analysis of Software Requirements for Completeness and Consistency	80
4.2 Tools	85
4.2.1 Control Overview	86
4.2.2 BDD Library	88
4.2.2.1 Variable Reordering	89
4.2.3 BDD Translator	90
4.2.4 AND/OR Table Translator	90
4.2.5 PVS Translator	93
4.2.6 BDD Analyzer	98
4.2.6.1 Consistency Analysis	98
4.2.6.2 Completeness Analysis	99
4.2.7 PVS Analysis	100
4.3 Adding Augmenting Information to the Analysis Process	101
4.3.1 Adding Augmenting Information to the BDD Analysis	101
4.3.2 Adding Augmenting Information to the PVS Analysis	103
4.4 Iteration Options and Analysis of Output	104
4.5 Summary	107
5 Domain Axioms and Domain Axiom Identification	109
5.1 Domain Axioms	110
5.2 Identifying Domain Axioms	113
5.2.1 Indicator Nodes	114
5.2.2 Sampling	124
5.3 Summary	128
6 Application of Method and Experimental Results	130
6.1 Consistency Analysis	136
6.1.1 Transitions in State Intruder-Status - TCAS II Version 6.04A	137
6.1.1.1 Transitions out of State Proximate-Traffic	138
6.1.1.1.1 Proximate-Traffic to Threat and Proximate-Traffic to Other-Traffic	138
6.1.1.1.2 Proximate-Traffic to Threat and Proximate-Traffic to Potential-Threat	151
6.1.1.1.3 Proximate-Traffic to Other-Traffic and Proximate-Traffic to Potential-Threat	157
6.1.1.2 Transitions out of State Other-Traffic	168
6.1.1.2.1 Other-Traffic to Threat and Other-Traffic to Potential-Threat	168
6.1.1.2.2 Other-Traffic to Threat and Other-Traffic to Proximate-Traffic	172
6.1.1.2.3 Other-Traffic to Proximate-Traffic and Other-Traffic to Potential-Threat	174
6.1.1.3 Transitions out of State Potential-Threat	174
6.1.1.3.1 Potential-Threat to Threat and Potential-Threat to Proximate-Traffic	176
6.1.1.3.2 Potential-Threat to Threat and Potential-Threat to Other-Traffic	179
6.1.1.3.3 Potential-Threat to Proximate-Traffic and Potential-Threat to Other-Traffic	185
6.1.1.4 Transitions out of State Threat	189
6.1.1.4.1 Summary of Results	192
6.1.2 Transitions in State Intruder-Status - TCAS II Version 7	193
6.2 Completeness Analysis	194
6.2.1 Transitions in State Auto-SL	194

6.2.2	Transitions in State Effective-SL	205
6.3	Summary and Heuristics	208
7	Conclusions and Future Investigations	211
7.1	Potential Future Work	213
7.2	Suggestions for Future Investigations	214
	APPENDICES	215
A	Macro Definitions	215
B	PVS Commands and Strategies	223
B.1	Built-in PVS Commands and Strategies	223
B.2	Defining Strategies	224
B.3	Our PVS Defined Strategies For Checking For Consistency and Completeness	225

LIST OF FIGURES

2.1	Model checker inputs and outputs.	15
2.2	Reachability analysis using a model checker.	16
2.3	Model checking's counter example feature.	17
2.4	PVS sequent formula.	27
2.5	PROLOG solution scenarios.	33
2.6	Infinite search example in PROLOG.	34
2.7	Finite search example in PROLOG.	35
2.8	Assertions and rewrite rules in Larch Prover.	37
2.9	An example proof using rewrite rules in LP.	38
3.1	Incomplete and inconsistent finite state machine model.	44
3.2	Completeness and consistency checks within rows.	47
3.3	FSM annotated with conditions and output variable assignments.	48
3.4	Table relating states and conditions to an output variable.	48
3.5	A state transition table abstraction.	49
3.6	FSM annotated with events and state transitions.	50
3.7	Table relating states and events to new states.	51
3.8	Transitions annotated with conditions and output variable assignments.	52
3.9	Tabular representations of guarding conditions.	53
3.10	Guarding condition for transition from Proximate-Traffic to Other-Traffic.	53
3.11	Completeness and consistency checks between tables.	54
3.12	Simple BDD example.	57
3.13	Example of variable orderings and their effect on graph size.	58
3.14	BDD representation and analysis of enumerated types.	60
3.15	BDD representation and analysis of invalid enumerated type assignments.	61
3.16	BDD example of tautology.	62
3.17	Symbolic representation of non-trivial arithmetic expressions and mathematical functions, and spurious error reports.	63
3.18	Spurious error reports generated for the state Inhibited.	70
4.1	High-level view of analysis to check for mutual exclusion.	77
4.2	High-level view of analysis to check for satisfiability.	78
4.3	General analysis process and integration of symbolic and reasoning components.	79
4.4	General integrative analysis process applied to analysis of RSML requirements for completeness and consistency.	81
4.5	Overview of analysis tools and data flow between the tools.	85
4.6	Example state diagram to demonstrate collection of triggers and associated transitions during analysis process set-up.	87
4.7	Example BDD node profile portion.	88
4.8	Examples of enumerated types and enumerated type predicates.	92

4.9	The transitions out of the state Inhibited.	95
4.10	A PVS theory for the transition from Inhibited to Not-Inhibited.	96
4.11	Conjecture theory for the state inhibited.	97
4.12	PVS unprovable subgoal example.	97
4.13	AND/OR table representation of PVS subgoal.	98
4.14	Specifying axioms in the PVS specification language.	103
4.15	Adding augmenting information to the PVS proof process and completing the PVS proof.	104
4.16	Analysis of output and iteration options of the analysis.	105
5.1	Domain Axioms associated with spurious errors of the state Inhibited.	111
5.2	Overview of data flows for identifying and using domain axioms.	114
5.3	BDD structure and node profile example.	115
5.4	Example of a deep BDD with a repeating sub-branch. The repeating sub-branch denoted by the open circle, repeats six times in the BDD.	117
5.5	Example of a deep BDD with links to a repeating sub-branch. The repeating sub-branch is denoted by a 1 with a circle around it.	118
5.6	Example graph showing indicator nodes.	119
5.7	Examples of indicator nodes near the top of the BDD.	120
5.8	Node profiles of indicator nodes; a dash represents a don't care.	121
5.9	Partial node profile for consistency analysis of two transitions from a large avionics specification.	122
5.10	Outline of a BDD showing the sampling algorithms and locations of the samples in the BDD.	125
5.11	Partial samples of a result BDD.	126
5.12	Partial graph and samples associated with the first three levels in the graph.	127
6.1	RSML specification of TCAS-Controller.	131
6.2	RSML partial specification of the state Other-Aircraft.	132
6.3	RSML partial specification of the state Tracked.	132
6.4	RSML specification of the state Intruder-Status.	133
6.5	RSML partial specification of the state Own-Aircraft.	135
6.6	Transitions from Other-Traffic, Proximate-Traffic, and Potential-Threat to Threat.	139
6.7	Transition from Proximate-Traffic to Other-Traffic.	139
6.8	The Threat-Condition Macro.	140
6.9	Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Other- Traffic without variable reordering.	141
6.10	Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Other- Traffic with variable reordering.	143
6.11	Selected samples from symbolic analysis of Proximate-Traffic to Other-Traffic and to Threat with variable reordering.	144
6.12	RSML specification of state Other-Air-Status.	145
6.13	Guarding condition for transition from Other-Air-Status state Airborne to Other-Air- Status state On-Ground.	145
6.14	Guarding condition for transition from Other-Air-Status state On-Ground to Other-Air- Status state Airborne.	145
6.15	RSML specification of state Alt-Reporting.	146
6.16	Guarding condition for transition from Alt-Reporting state Yes to Alt-Reporting state Lost.	147

6.17	Guarding condition for transition from Alt-Reporting states Lost and No to Alt-Reporting state Yes.	147
6.18	Guarding condition for transition from Alt-Reporting state Lost to Alt-Reporting state No.	147
6.19	Guarding condition for transition from Alt-Reporting state C to Alt-Reporting state Yes.	148
6.20	Guarding condition for transition from Alt-Reporting state C to Alt-Reporting state No.	148
6.21	Domain axiom in tabular form for Other-Alt-Reporting, Alt-Reporting assertion.	149
6.22	General PVS strategy to prove two guarding conditions are consistent.	150
6.23	Specific PVS commands to include a domain axiom into the analysis process and to prove two guarding conditions consistent.	151
6.24	Transitions from Other-Traffic and Proximate-Traffic to Potential-Threat.	151
6.25	Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Potential-Threat without variable reordering.	152
6.26	Selected samples from symbolic analysis of Proximate-Traffic to Threat and to Potential-Threat without variable reordering.	154
6.27	Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Potential-Threat with variable reordering.	155
6.28	Selected samples from symbolic analysis of Proximate-Traffic to Threat and to Potential-Threat with variable reordering.	156
6.29	Partial BDD node profile for conjunction of Proximate-Traffic to Other-Traffic and to Potential-Threat without variable reordering.	159
6.30	Portion of samples from symbolic analysis of Proximate-Traffic to Other-Traffic and to Potential-Threat without variable reordering.	160
6.31	Partial BDD node profile for conjunction of Proximate-Traffic to Other-Traffic and to Potential-Threat with variable reordering.	161
6.32	Selected samples from symbolic analysis of Proximate-Traffic to Other-Traffic and to Potential-Threat with variable reordering.	163
6.33	Partial BDD node profile for conjunction of Proximate-Traffic to Other-Traffic and to Potential-Threat with variable reordering and domain axioms.	165
6.34	Domain axiom for Other-Alt-Reporting, Other-Air-Status assertion	166
6.35	Selected unprovable subgoals from PVS analysis of Proximate-Traffic to Other-Traffic and to Potential-Threat.	167
6.36	The PVS proof commands used to include the domain axioms into the analysis process to attempt to prove two guarding conditions consistent.	168
6.37	Partial BDD node profile for conjunction of Other-Traffic to Threat and to Potential-Threat with variable reordering.	169
6.38	Selected samples from symbolic analysis of Other-Traffic to Threat and to Potential-Threat with variable reordering.	171
6.39	Transition from Other-Traffic to Proximate-Traffic.	172
6.40	Partial BDD node profile for conjunction of guarding conditions from Other-Traffic to Threat and to Proximate-Traffic with variable reordering.	173
6.41	Portion of the samples obtained from symbolic analysis for consistency of Other-Traffic to Threat and to Proximate-Traffic with variable reordering.	175
6.42	Transition from Potential-Threat to Proximate-Traffic.	176
6.43	Partial BDD node profile for conjunction of Potential-Threat to Threat and to Proximate-Traffic with variable reordering.	177
6.44	Selected samples from symbolic analysis of Potential-Threat to Threat and to Proximate-Traffic with variable reordering.	180

6.45	Selected samples for Potential-Threat to Threat and to Proximate-Traffic of predicates involved in the domain axiom.	181
6.46	Transition from Potential-Threat to Other-Traffic.	181
6.47	Partial BDD node profile for conjunction of Potential-Threat to Threat and to Other-Traffic with variable reordering.	182
6.48	Selected samples from symbolic analysis of Potential-Threat to Threat and to Other-Traffic with variable reordering.	184
6.49	Partial BDD node profile for conjunction of Potential-Threat to Proximate-Traffic and to Other-Traffic with variable reordering.	186
6.50	Selected samples from symbolic analysis of Potential-Threat to Proximate-Traffic and to Other-Traffic with variable reordering.	188
6.51	Specific PVS commands to include a domain axiom into the analysis process and to prove two guarding conditions consistent.	189
6.52	Transition from Threat to Other-Traffic.	189
6.53	Transition from Threat to Potential-Threat.	190
6.54	Transition from Failed to Potential-Threat.	190
6.55	Transition from Failed to Other-Traffic.	190
6.56	Transition from Failed to Passed.	191
6.57	Transition from Passed to Failed.	191
6.58	Guarding condition for transition from any Auto-SL state to Auto-SL state 1.	195
6.59	Guarding condition for transition from any Auto-SL state to Auto-SL state 2.	195
6.60	Guarding condition for transition from any Auto-SL state to Auto-SL state 3.	196
6.61	Guarding condition for transition from any Auto-SL state to Auto-SL state 4.	196
6.62	Guarding condition for transition from any Auto-SL state to Auto-SL state 5.	197
6.63	Guarding condition for transition from any Auto-SL state to Auto-SL state 6.	197
6.64	Guarding condition for transition from any Auto-SL state to Auto-SL state 7.	198
6.65	Mutually exclusive and all-inclusive relational expressions involving the variable Own-Alt-Radio (OAR).	200
6.66	PVS strategy for proving guarding conditions complete.	201
6.67	The Radar-Bad-For-RADARLOST-cycles macro.	201
6.68	The Radarout-EQ-0-macro.	201
6.69	The Climb-Desc.-Inhibit macro.	202
6.70	The Standby-Since macro.	202
6.71	Unprovable subgoals from PVS analysis.	203
6.72	Reduced unprovable subgoals from PVS analysis.	204
6.73	Condition to add to specification of guarding conditions to make the specification complete.	204
6.74	Guarding condition for transition from any Effective-SL state to Effective-SL state 1.	205
6.75	Guarding condition for transition from any Effective-SL state to Effective-SL state 2.	205
6.76	Guarding condition for transition from any Effective-SL state to Effective-SL state 3.	206
6.77	Guarding condition for transition from any Effective-SL state to Effective-SL state 4.	206
6.78	Guarding condition for transition from any Effective-SL state to Effective-SL state 5.	207
6.79	Guarding condition for transition from any Effective-SL state to Effective-SL state 6.	207
6.80	Guarding condition for transition from any Effective-SL state to Effective-SL state 7.	208
A.1	Alt-Separation-Test Macro	216
A.2	Low-Firmness-Separation-Test Macro	217
A.3	Noncrossing-Biased-Climb Macro	217

A.4	Noncrossing-Biased-Descend Macro	218
A.5	No-Vertical-Intent Macro	218
A.6	RA-Mode-Canceled Macro	218
A.7	RA-Inhibit Macro	219
A.8	Reply-Invalid-Test Macro	219
A.9	TCAS-TCAS-Crossing-Test Macro	220
A.10	Threat-Alt-Test Macro	221
A.11	Threat-Range-Test Macro	222
A.12	Two-Of-Three Macro	222

Chapter 1

Introduction and Motivation

Statically analyzing requirements specifications to assure that they possess desirable properties is an important activity in any rigorous software development project. Without the thorough understanding of the software specification that can be gained through static analysis, the end product, that is, the working software system, will most likely disappoint the user and bring grief to the developer [49].

The requirements specification is the foundation of any software development process. All other stages of development depend upon the requirements specification. In addition, errors in the requirements that go undetected and propagate to later stages of development (the design and implementation stages) are the most costly to correct [32, 38, 41]. Therefore, it is important to ensure that the requirements document satisfies certain desired properties before proceeding to later stages of the development process. One application area for the research described in this dissertation is in the static analysis of software requirements. Analysis of state-based requirements is used in the examples throughout this document.

There are two classes of properties we can check the requirements for: application specific properties and general, or application independent, properties. For example, a desirable property

for flight management software in an aircraft might be, if the aircraft is landing, the landing gear must be down. This constraint is applicable to most aircraft, but is not applicable to, for example, a train system or an automotive system. Thus, application specific properties are relevant only in a specific domain or in a specific product.

General properties are properties that all requirements should satisfy regardless of the application domain. For example, two desirable properties for requirements specifications to satisfy are completeness and consistency. Completeness means the requirements have a behavior specified for every input and input sequence, and consistency means there are no conflicting requirements [24]. In the context of state-based requirements, completeness and consistency analysis becomes the problem of showing that disjunctive and conjunctive logical expressions are tautologies or contradictions [24].

Techniques for automatically analyzing and manipulating the requirements facilitate the analysis process and increase the level of confidence that a system will operate as desired. Machine analysis and manipulation of requirements presumes that the requirements are specified in a language with a formal foundation [59]. The research described in this document presumes that the requirements to be analyzed can be specified in a language suitable for machine analysis and manipulation.

There are two general methods that can be used to analyze the requirements to determine if they satisfy desired properties: *dynamic analysis* techniques and *static analysis* techniques. Dynamic analysis techniques include testing and simulation. Testing and simulation rely on providing input to a system and observing the output generated by the system. Since the input space is often infinite (or at the very least, extremely large), dynamic analysis techniques cannot provide the necessary levels of confidence required for certain classes of systems, for example, so called *critical systems*, and only a small portion of the input space can be tested [7, 25, 41, 52, 53]. Critical systems are often *real-time reactive systems* in which a failure could result in harm to life, property, or the

environment. Very high levels of confidence that such systems will operate in a safe manner (i.e., in such a way so as not to cause harm to life, property, and the environment, or in such a way so as to minimize the amount of harm that results in the event of a failure) are required.

Static analysis techniques are based on statically studying a model and attempting to formally prove that certain properties hold. Static analysis techniques include techniques such as reachability analysis, model checking, and theorem proving. Static techniques rely on abstraction techniques to simplify the model since including all details may make the models too complex to analyze; i.e., the analysis process will become intractable. The abstractions, however, may remove details needed for the analysis, and thus, the analysis may report spurious errors along with the true errors. Spurious errors are conditions in the model that are reported as errors in need of correction, but information that was abstracted out of the model precludes the reported conditions from being satisfied. For example, assume a system has a control variable that is used to determine the flow of control in such a way that the system will enter a dangerous configuration only if the control variable has a specific value, but the static technique being applied has abstracted away the variables that determine control flow. Based on the analysis model, the static technique may report that the system will enter a dangerous configuration since the control flow information is missing. Manual inspection of the requirements, however, reveals that the control variable that has been abstracted out of the model actually prohibits the system from entering the dangerous configuration; thus, the error reported by the analysis technique is a spurious error.

The end product of any analysis effort, whether static or dynamic, is a report of the analysis results. This report should provide the analysts with the information they need to find and correct errors in the artifact being analyzed. The number of spurious errors reported by static analysis techniques in the analysis results can be excessive. When analysis reports contain too many spurious errors, it is difficult, error-prone, and time consuming for the analyst to identify and correct the true

errors. The necessity of providing the analyst with useful output from the static analysis is the motivation for this research and is discussed more thoroughly in the next section.

1.1 Problem Statement

The goal of this research was to develop an analysis method to check state-based requirements for completeness and consistency in a way that is automated, generates error reports with an acceptable level of accuracy, is fast enough to be used on a day-to-day basis, and that is scalable and generalizable (i.e., applicable to real-world problems and not limited to state-based requirements). Since analyzing for completeness and consistency in the context of this research generalizes to the problem of showing that disjunctive and conjunctive logical expressions are tautologies or contradictions, our goal was to be able to analyze logical expressions to check for tautologies and contradictions in a way that generates error reports with an acceptable level of accuracy, and is fast and automated enough to be used on a day-to-day basis by practicing engineers.

No one individual static technique for performing the analysis is sufficient to satisfy the desired goals [48, 65, 67]. There are trade-offs between the amount of abstraction an analysis method relies on, the degree of automation, the speed with which the analysis completes, and the level of accuracy in the analysis output. Each analysis method has its own strengths and weaknesses in relation to the trade-offs. For example, static symbolic methods are fast and fully automated, but may generate many spurious error reports since with symbolic methods many functions are not interpreted; i.e., the semantics of the functions are abstracted away. Reasoning techniques such as theorem proving may generate more accurate error reports, but are less automated and tend to be slower. When more details are included in the analysis process, the analysis often requires more user intervention and more time may be required to complete the analysis.

We know that some information must be abstracted from the model or the analysis becomes

intractable. Furthermore, not all information needs to be included in the analysis model, only that information that is relevant to the properties of interest to the analysts. Even the most powerful static techniques will generate spurious errors if information relevant to the analysis has been abstracted away. Therefore, if the spurious errors are the result of information missing from the analysis model, it is important to determine what information has been abstracted away and that is leading to the spurious errors. Determining the abstractions leading to spurious errors can be difficult, since generally, the analysis output does not give the analyst any indication of where to look for the missing information. Indeed, without error-prone and time consuming manual inspection of the analysis output, the analyst does not know which reported errors represent true errors in the specification, and which reported errors represent spurious errors.

Since no one individual analysis technique can satisfy all of our goals, the problem is then, how can we combine different static analysis methods to take advantage of the strengths of each method and circumvent their weaknesses, and how can we identify information that is relevant to the accuracy of the analysis, but that has been abstracted out of the model being analyzed. This dissertation addresses these problems; it describes an iterative and integrative method that capitalizes on the strengths of the more detailed analysis methods and the strengths of the less detailed analysis methods while circumventing their weaknesses. In addition, this dissertation describes an approach to help the analyst identify relevant information that has been abstracted away from the analysis model.

To illustrate the power of the research described in this dissertation, we applied our technique to the requirements specification for a large real-world avionics system specified in the requirements language called Requirements State Machine Language (RSML) [41]. The elements of RSML that are relevant to this research are described in Chapter 3, Section 3.1.2.

1.2 Contributions

The contributions of this research are the following:

- We developed an iterative and integrative analysis method to analyze state-based requirements for completeness and consistency, in a way that generates analysis reports with an acceptable level of accuracy, is fast and automated enough to be used on a day-to-day basis by practicing engineers, and is scalable and generalizable. The analysis process combines a symbolic component with a reasoning component. We can generalize this contribution to: an iterative and integrative analysis method to check disjunctive and conjunctive logical expressions for tautologies and contradictions.
- We developed a general and simple method that uses a symbolic representation of logical expressions to help identify abstractions causing spurious incompletenesses and spurious inconsistencies in the analysis output.
- We identified four classes of spurious errors and associated the spurious errors with the abstractions that lead to the spurious errors, when analyzing state-based requirements for completeness and consistency.
- We developed various analysis tools to automate the analysis process. These tools include:
 - a translator from Requirements State Machine Language (RSML) to Prototype Verification System (PVS)¹ specifications,
 - a translator from RSML AND/OR tables² to Binary Decision Diagrams (BDDs)³,
 - a BDD analyzer to check logical expressions represented as BDDs for tautologies and contradictions,
 - a translator from BDDs to RSML AND/OR tables to present the analysis results to the analyst, and
 - proof strategies for PVS to help automate the proof process and allow the analyst to use a single command in most cases to perform PVS analysis.

¹PVS is a specification and verification system and is described in Chapter 2.

²AND/OR tables are disjunctive normal form tables used to represent logical expressions and are described in Chapter 3.

³BDDs are data structures used to represent and manipulate logical expressions and are described in Chapter 3.

1.3 Organization of Dissertation and Guide to Reading

Chapter 2 discusses different analysis techniques and describes the strengths and weaknesses of each technique. Dynamic analysis techniques are discussed only briefly at the beginning of the chapter. The bulk of the chapter is devoted to static analysis techniques including reachability analysis, model checking, and theorem proving. Persons with a good understanding of static analysis techniques may skip the majority of this chapter. However, we recommend reading at least the introduction and summary to maintain a link with the rest of the dissertation.

In Chapter 3 we discuss related work in the area of analysis of software requirements. We describe the strengths and weaknesses of each technique, and describe a problem common to all techniques. The common problem is that abstractions in the model may yield spurious errors in the analysis report.

The next two chapters describe the results of the research discussed in this dissertation. In Chapter 4 we present our solution to the problem detailed in Chapter 3, specifically, our iterative and integrative analysis method. In Chapter 4, however, we leave out one important detail; namely, how we identify the information that is abstracted away from the model and that leads to the spurious errors.

In Chapter 5 we discuss our method for identifying the missing information that leads to the spurious errors, and describe our overall method to analyze logical expressions to check for tautologies and contradictions in a way that is timely, automated, and generates analysis reports with an acceptable level of accuracy.

To demonstrate the usefulness of our approach, we applied our method to a large real-world avionics specification. The results of this application are reported in Chapter 6. In the conclusion of Chapter 6 we provide some heuristics to help guide analysts in using our technique efficiently and effectively; we learned the heuristics from the application of our method. In Chapter 7 we present

our conclusions and discuss future enhancements that could be made to our analysis technique, and we discuss future investigations that follow from this research.

Appendix A contains definitions of macros that are discussed in Chapter 6, but not included in the chapter itself. Appendix B contains a description of the PVS prover commands we used, and descriptions of the PVS proof strategies we developed during the course of this research.

Chapter 2

Analysis Techniques

In Chapter 1 we introduced the motivation for this research and the problem we set out to solve. In this chapter we describe several different analysis techniques and investigate the strengths and weaknesses of each technique. Our investigation helped us identify analysis methods and tools that we could potentially use to achieve the goals of the research described in this dissertation. Our investigation also helped us eliminate certain analysis methods and tools from consideration.

Analysis techniques are conventionally classified as *dynamic analysis* and *static analysis* techniques, according to their operational characteristics [62, 65]. Dynamic analysis techniques require actual program execution, whereas static analysis techniques do not require the actual program to be executed. In the context of this chapter, a program refers to any executable artifact. For example, an executable state-based specification or a traditional program. The emphasis of this chapter is on static analysis, but dynamic analysis is discussed briefly in Section 2.1. Several static analysis methods are discussed in detail in Section 2.2. The bulk of the static analysis discussion is spent on theorem proving systems; we knew early on that we needed some type of reasoning component to achieve our goals. In Section 2.3 we provide a summary of the analysis methods discussed, and review some of their strengths and weaknesses.

Analysis techniques may be applied to requirements documents, design documents, or source code. Each analysis method works on some model (requirements, design, code) of the system being analyzed. Each analysis technique incorporates some compromise between accuracy and completeness, and tractability (or computational cost) [65]. For example, exhaustive testing (i.e., testing all possible input cases) is guaranteed to be accurate since all true errors will be found and no spurious errors will be reported, but exhaustive testing is intractable for all but the simplest of programs. The compromise between accuracy and computational cost results because the question “Does program P obey specification S ” is undecidable for arbitrary programs and specifications [65].

Most analysis techniques rely on constructing a representation of possible executions of the executable artifact, and comparing this representation to a specification of intended behavior [65]. The *state space* of an executable artifact is the set of states that the artifact (referred to as program from here on) can reach in all possible executions. Many analysis methods (both static and dynamic) rely on analysis of the state space, since the actual behavior of a program is represented as program states and state transitions, or as an abstraction of program states and state transitions. Analysis techniques that explicitly construct some representation of program states are referred to as *state space analysis* techniques [65].

There are two main ways to reduce the computational complexity of state space analysis: *folding* and *sampling* [65]. With folding, the state space shrinks because some of the details of program execution are abstracted away (i.e., some details are ignored or removed). Abstracting away details, typically results in a smaller state space, since each state in the model state space may represent several states in the normal program execution. Sampling, on the other hand, does not alter the size of the state space, but rather, examines only a portion of the entire state space. In testing, for example, the program is executed only on selected input data, since testing all possible inputs is intractable. Most dynamic analysis techniques rely on sampling of the state space, whereas most

static analysis methods rely on folding of the state space.

2.1 Dynamic Analysis

Dynamic analysis methods such as testing, involve selective exploration of the entire state space of program execution [62]. Testing, as used here, refers to testing a system or portion of a system by providing a set of inputs to a system or portion of a system, and observing the output generated by the system. The output is then compared with the predetermined expected results. Testing is successful when an error is found. Not finding any errors does not ensure that the program has no errors. Thus, testing can reveal the presence of errors but not the absence of errors.

No dynamic analysis technique can check all possible input sequences of a program, since the input space for even a moderately sized program can be exceptionally large [49]. If a good sampling of the input space is made, then testing will be successful to some degree. However, if a bad sampling is made, many errors will be missed. Unfortunately, there is no method for determining a good sampling of the input space. Thus, dynamic analysis techniques can only demonstrate the presence of errors in a program, they cannot demonstrate that the program is error-free [49, 56], and so dynamic analysis techniques cannot provide the levels of confidence desired to ensure that a critical system will operate as expected.

2.2 Static Analysis Techniques

Static analysis techniques create a model of the system to be analyzed, and check the model to make sure it satisfies some desired properties. There are many different ways to model system behavior: axiomatic methods [37], state-based methods, such as mode table models [4, 30, 32], Statecharts [20, 21, 41], and finite state machines [38], and a plethora of other methods. There are

also many different ways of analyzing the models.

Static methods for analyzing models include *reachability analysis*, *model checking*, and *theorem proving*. Section 2.2.1 discusses reachability analysis. Section 2.2.2 discusses model checking, and Section 2.2.3 discusses theorem proving methods. The discussion of each method includes an analysis of the strengths and weaknesses of the method.

2.2.1 Reachability Analysis

Reachability analysis is a group of analysis techniques that involves the systematic enumeration of all reachable states in a finite-state model; that is, a state transition model of larger modules or of a complete system, is constructed from models of individual processes [48, 62, 67]. The resulting composite state-transition model is generally referred to as a *reachability graph*. Once the reachability graph is constructed, it can be analyzed for certain general properties. Typically, reachability models abstract away all details of execution except for synchronization structure. Thus, reachability analysis is primarily used to verify properties related to the synchronization structure of software, such as freedom from deadlock, livelock (absence of starvation), race conditions, and mutual exclusion [62, 67]. Reachability analysis is attractive because it is relatively straightforward to automate and conceptually simple.

Reachability analysis is attractive and widely used, however it suffers from several drawbacks. The biggest drawback to practical application of reachability analysis to real systems is the well known *state space explosion problem*. In the worst case, as the number of processes in a concurrent system increases, the total number of reachable states in the system grows as the product of the numbers of states of all the processes [62]. This exponential growth in the number of states from the composition of processes limits the analysis of synchronization structure to small collections of tasks and can easily make it impractical to analyze even a simple system. According to Young [67],

the task limitation is on the order of a dozen. Yeh [62, 63] introduced a divide and conquer compositional approach that may help reduce the limitations of reachability analysis that result from the state space explosion problem [67]. However, reachability analysis suffers from other limitations as well.

Since reachability analysis is primarily used to check the synchronization structure of a program, it is not as powerful (comprehensive) as theorem proving or testing [62, 67]. Theorem proving and testing can be used to verify a wider variety of properties, such as functional correctness. In general, the number of states grows exponentially in the number of independent items one models. The more details that are modeled, the greater the number of states that are needed to represent the model. This is why for analysis of synchronization structure, for example, all details except synchronization structure are abstracted away from the model. The abstractions make the analysis tractable, but they also may limit the accuracy of the analysis, and abstractions may limit the types of properties that can be analyzed. For example, to avoid the state space explosion problem, details related to flow of control may be abstracted away by not completely modeling the values of the variables that determine control flow. As a result, spurious errors may be reported; that is, a state representing an error condition may be reachable in the reachability graph model of a system and will be reported as an error, when in actuality, certain variable values that were not modeled completely would preclude that state from being reached in the actual system. Thus, the abstractions made can make the reachability analysis conservative (i.e., all true errors will be reported, but spurious errors may also be reported.). It is then left up to the analyst to determine which reported errors are true errors and which reported errors are spurious errors.

2.2.2 Model Checking

The focus of model checking is on checking a model of a system to determine if certain application dependent properties are satisfied in the model. These properties are all temporal properties and thus, involve, in some way, the sequencing or ordering of events in time. Traditionally, model checking has been applied to hardware verification. More recently, Atlee and Gannon [3, 4], Wing [60], and Bharadwaj and Heitmeyer [6] have applied model checking to software requirements. Their work will be discussed in Chapter 3, Section 3.2.

Early work by Clarke and his colleagues involved the use of explicit model checking in the area of hardware verification and concurrent program verification [7, 13, 14]. Explicit model checking requires two inputs: a behavioral model of the system, and a formal specification of system behavior [9, 13]. The first input to a model checker is in the form of a state transition graph. The second input, the formal specification of system behavior, is a logical formula expressing certain properties that must hold true in the system. Some properties of interest are *safety* and *liveness* properties [3, 4, 7, 13, 14, 22]. Safety properties generally state what should not happen and are usually required to remain invariant throughout the lifetime of the entire system [3, 4]. Liveness properties state that some current state leads to an expected future state (for example, forward progress is made) [3].

Specifying requirements about safety and liveness properties requires a logic language capable of describing the relative ordering of events in time (a temporal logic language) [22]. There are many different variations of temporal logic languages that can be used. One of the more popular temporal logic languages, and the one Clarke and his colleagues use [7, 13, 14, 42], is *Computation Tree Logic* (CTL), a *branching time temporal logic*; viewing time as linear suggests that at each moment there is only one possible future, whereas viewing time as branching suggests that at each moment, time may split into alternate courses representing different possible futures [17]. The details of CTL

are not important for the purposes of this research and will not be discussed. See [13, 14] for further details.

Model checking is concerned with determining if a specific property, expressed as a formula in temporal logic, holds in a system. Figure 2.1 shows a graphical depiction of the inputs and outputs of a model checker [3]. The notation $M, s_0 \models f$ means that formula f holds at state s_0 in M [13]. To check if temporal logic formula f holds in M , both the formula f and a specification of the system are input to the model checker (Figure 2.1). The model checking algorithm operates in stages. In the

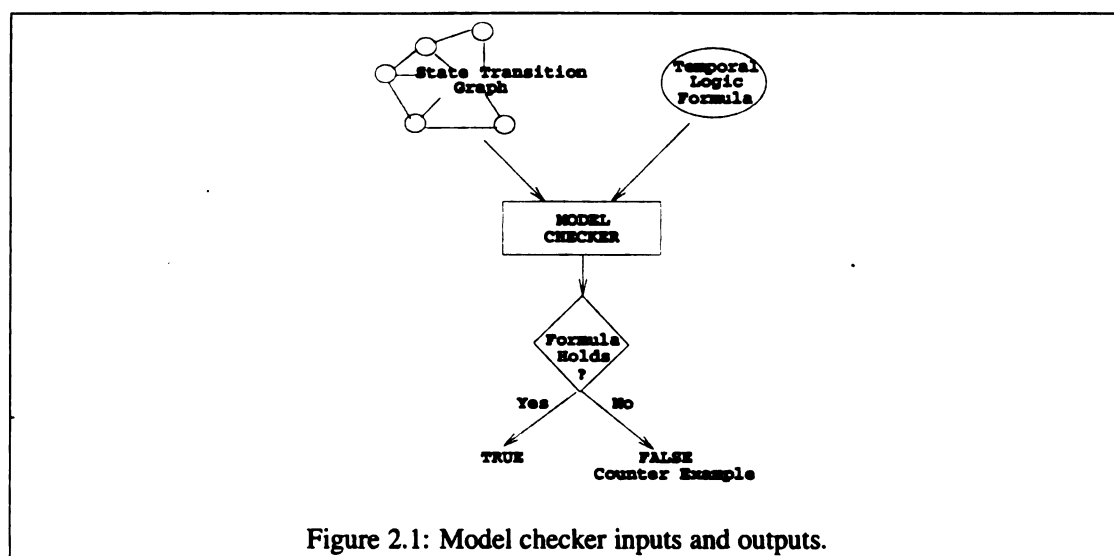


Figure 2.1: Model checker inputs and outputs.

first stage, each state of the finite state transition model M is labeled with the formulas that hold in that state. In the second stage, the model checker examines the model to determine if the temporal logic formula f holds in the model. If the system specification was translated correctly into a state transition model, then a formula determined true by the model checker must also hold true for the corresponding specification [7]. For example, consider Figure 2.2. Let the specification require that at some time in the future, relative to state s , there is some state in which the formula $f = (a \wedge b)$ holds (assume that the model checker has already completed the first step of labeling the states with the formulas that hold in them; not all formulas that hold in the states are shown). The CTL formula

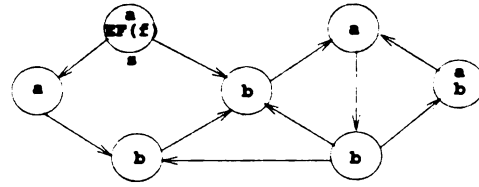


Figure 2.2: Reachability analysis using a model checker.

$EF(f)$ means that there is some path from s that leads to a state at which f holds. Then, given the state s and the CTL formula $EF(f)$, the model checker will check the label of state s to see if $EF(f)$ is in the label set of s . If the formula is found, then $s \models f$.

A model checker is also equipped with the capability to generate a counter example in certain situations. If the model checker determines that a formula is false, it will, if possible, provide a counter example that shows that the negation of the formula is true. For example, consider Figure 2.3. Let the specification require that formula f holds in every state on every path from s_0 ; this is represented by the CTL formula $AG(f)$. The model checker will check the label at s_0 to see if the CTL formula $AG(f)$ is in the set of labels. Since the formula does not hold globally in the model shown, the model checker will attempt to find a counter example; that is, a path to a state in which $\neg f$ holds; in this case, it will report either the path from s_0 to s_i , or the path from s_0 to s_j . A counter-example will not be generated for example, when the model checker is checking to see if a specific property eventually holds; if the model checker finds that the property does not eventually hold, there is no counter-example to generate to show this.

The major disadvantage to model checking is the explicit enumeration of the state space. As the complexity of a model increases, the size of the state space can in certain situations, increase exponentially. Since the model checker searches a model by explicitly enumerating the state space, the running time scales linearly in the number of states. This means that the model checker's running time increases exponentially as the complexity of the system increases. Thus, model checking

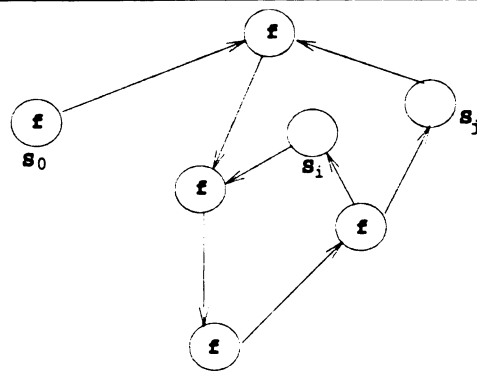


Figure 2.3: Model checking's counter example feature.

suffers from the state space explosion problem.

The state space explosion problem can be alleviated by representing the state space symbolically rather than explicitly [9, 10]. This can be done by representing states, and transitions between states as Boolean formulas [35]. The details of symbolic model checking are not important and are not discussed in this dissertation. For details, see [10, 42].

To summarize, model checking has several advantageous features:

- it can automatically verify that required properties hold in a finite state model,
- it can generate a counter example for some properties if it determines a specific property does not hold in the model, and
- in practice, model checkers can run quickly [7, 13, 14, 35].

However, as with all analysis methods, model checking also suffers from several problems:

- it can be difficult or impossible to specify some properties in the restricted temporal logic [2, 4] used in model checkers, thus, no verification can be performed for such properties,
- in some cases, the complexity of temporal logic formulas makes it difficult to know what has actually been specified and proven, and
- the explicit generation of the state space results in the state space explosion problem (although this can be somewhat alleviated by using symbolic model checking).

We have investigated both reachability analysis and model checking which are closely related static techniques. We showed that both reachability analysis and model checking have certain strengths and weaknesses, and we described these strengths and weaknesses. We now look at a static method that is not closely related to the previous two techniques, namely, theorem proving.

2.2.3 Theorem Proving

Theorem proving is used extensively in the area of hardware verification and there are a plethora of different types of theorem proving systems in use; HOL [36, 58], Boyer-Moore [16, 43], PVS [15, 55], Knuth-Bendix [16], and the Larch Prover [19] to name a few. Theorem proving is also used in the software arena, mainly to verify that implementations satisfy some specification [52, 53, 60]. More recently, theorem proving is being considered for use in the analysis of requirements for completeness and consistency [26, 27, 28, 30, 32].

To solve the problem described in this dissertation, we evaluated different theorem provers to determine which theorem prover fit best with our goal of developing a method for ANDing and ORing complex logical expressions together to check for contradictions and tautologies in an automated, timely, and accurate way. The information in Sections 2.2.3.1 through 2.2.3.3 is intended to introduce some concepts and techniques about proof systems which are further developed in Section 2.2.3.4 when specific theorem provers are described and discussed. Section 2.2.3.1 provides a discussion of the general characteristics of proof systems. Section 2.2.3.2 discusses two different types of proof systems we investigated: sequent calculus systems and resolution systems, and Section 2.2.3.3 describes some of the various search strategies that can be used to enhance the efficiency of proof methods. Four specific theorem provers are described and discussed in Section 2.2.3.4: PVS, PROLOG, Larch Prover, and Knuth-Bendix. The strengths and weaknesses of the various methods are summarized at the end of this chapter.

2.2.3.1 Characteristics of Proof Systems

The majority of the information contained in Sections 2.2.3.1, 2.2.3.2, and 2.2.3.3, is derived from [16]. All proof systems have four components in common:

1. A set of logical or formal axioms which are universal truths (formulas that are *valid*; i.e., true in all interpretations).
2. A set of inference rules which are transformation rules to map valid formulas to valid formulas.
3. A proof development method.
4. A proof strategy.

The proof development method is the style of the proof employed. This methodology may be top-down (analytic) or bottom-up (synthetic). A top-down approach starts with the conjecture that is to be proved, and applies the rules in a backwards manner reducing the conjecture to sub-goals until axioms are derived. The less practical bottom-up approach starts with the axioms and applies the rules until the conjecture to be proved is deduced. In the bottom-up approach it is possible for the prover to continue synthesizing theorems that lead further and further away from the goal. As a result, the goal may never be reached. A combination of top-down and bottom-up can also be used.

There are basically three different facts that can be proven in a proof system. A proof system can be designed to prove that

1. A formula may be deduced from a given set of axioms.
2. A single formula of the form $(a_1 \wedge a_2 \wedge \dots \wedge a_n) \rightarrow c$ is valid, where the a_i 's are logical axioms and c is the conjecture to be proved.
3. A set of axioms taken together with the negation of the goal conjecture is unsatisfiable.

Systems of the first type are known as *deduction systems* and provide the most natural approach of the three. In the second approach, the correctness of the result depends on the fact that a formula α is a logical consequence of a finite set of formulas $\beta_1, \beta_2, \dots, \beta_n$ if and only if the conjunction of the finite set of formulas, $(\beta_1 \wedge \beta_2 \wedge \dots \wedge \beta_n)$, implies α is valid (i.e., is a tautology); systems of this type are known as *affirmation systems*. More simply, the above formula means that if all of

the formulas $\beta_1, \beta_2, \dots, \beta_n$ are valid, then α is valid, and it will never be the case that if all of the β_i 's are valid α is not valid. The third approach corresponds to a *proof by contradiction*; the goal is assumed false and the system shows that this leads to a contradiction. The correctness of this third style of proof depends on the assumption that if the axioms together with the formula $\neg\alpha$ is inconsistent (i.e., unsatisfiable), then α must be a logical consequence of the axioms. *Resolution systems* use this last style of proof.

A proof strategy is imposed on the development method in order to facilitate proof development. There are many different strategies that may be used. Essentially, a strategy may include using derived rules of inference, specifying directions on how to apply a given set of rules, and using special procedures for dealing with subtasks. Derived rules of inference are rules that combine several inferences in the original system into one step. Derived rules do not increase the power of the system in a technical sense, but merely save time in practice. Any application of a derived rule could be translated into the sequence of applications of the rules from which it was derived [18]. The directional component determines which rules should be applied and to which formulas, and it effects both the speed with which a proof is derived and the naturalness of the proof. The special procedures for dealing with subtasks include the use of decision procedures and procedures for dealing with difficult tasks such as using induction to prove theorems. Search strategies are described in more detail in Section 2.2.3.3.

2.2.3.2 Basic Proof Systems

There are several different types of theorem proving systems including semantic tableaux [18], natural deduction [16], sequent calculus [16], and resolution based systems [16]. Only sequent calculus systems and resolution based systems will be considered further here. Natural deduction systems will not be considered because such systems are not designed for the purpose of developing

proofs in an efficient manner, but rather, they are designed for displaying a proof in a “natural” form once the proof has been derived [16]. Consequently, other approaches to proof development are more efficient, particularly the sequent calculus approach. Resolution and semantic tableaux systems are similar: both systems are refutation systems and they represent information in the form of clauses [18]. Semantic tableaux systems use clauses in disjunctive normal form, while resolution systems use clauses in conjunctive normal form. Both resolution and semantic tableaux systems are amenable to automation.

2.2.3.2.1 Sequent Calculus Systems

A sequent is an expression that has the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sets (possibly empty) of formulas. The meaning of the sequent formula $\Gamma \vdash \Delta$ is: $(\gamma_1 \wedge \gamma_2 \cdots \wedge \gamma_i) \rightarrow (\delta_1 \vee \delta_2 \vee \cdots \vee \delta_j)$, which says that if all of the formulas on the left of the arrow are true, then at least one of the formulas on the right of the arrow is true. Sequent calculus systems have few, if any, axioms and many inference rules [16]. The reason there are few, if any, axioms and many inference rules is because sequent calculi systems are based on the work of Gerhard Gentzen who developed a system of “natural deduction” intended to allow proofs to be performed in a manner corresponding to human reasoning. The major principle that characterizes “natural deduction” systems is that for each logical symbol of the chosen logic, there should be separate rules that allow its introduction and its removal [16]. Thus, in Gentzen systems, at least two rules of inference are needed for each logical symbol, and so there are many rules of inference and only few, if any, axioms. In sequent calculus systems, the inference rules are of the form of *antecedent* and *consequent* rules, where the antecedent is taken as the left side of the sequent formula and the consequent is taken as the right side of the sequent formula (Γ and Δ respectively). Sequent calculus systems are affirmation systems that use backward reasoning (top-down approach to proof development); they start with

the goal conjecture and reduce the goal to subgoals by applying the rules backwards until axiom sequents are derived.

In sequent systems, proofs are expressed as a tree of sequents built from axiom sequents such that if node N is labeled with $\Gamma \vdash \Delta$ then: if N is a leaf node, $\Gamma \vdash \Delta$ must be an axiom; and if N has children, their labels must be the premisses from which $\Gamma \vdash \Delta$ follows by one of the sequent calculus rules [18]. The label on the root node is the sequent to be proved.

The backward reasoning approach results in a more direct style of proof than the forward reasoning approach, so sequent systems can be efficiently implemented. The Prototype Verification System (PVS) (Section 2.2.3.4.1) is based on sequent calculus.

2.2.3.2.2 Resolution Systems

Resolution systems are refutation systems; i.e., they do proof by contradiction. They assume the conjecture is false and then show that this leads to a contradiction. The conjecture is represented in clausal form (conjunctive normal form). A clause is a disjunction of a finite set of literals with no literal appearing twice. A formula is a conjunction of a finite set of clauses.

As an example of the resolution of two clauses, consider the two clauses $C = (x \vee C')$ and $D = (\neg x \vee D')$, where C' is the rest of clause C and D' is the rest of clause D [46]. The two clauses contain two opposing literals, x and $\neg x$. The clause $(C' \vee D')$, containing all literals of the two clauses except for the two opposing ones, is called the *resolvent* of C and D . The *resolution* of two clauses is the deduction of a resolvent from the two clauses (if one exists). For example, the result of applying resolution to the following two clauses

$$C_1 \vee L \vee C_2 \tag{2.1}$$

$$D_1 \vee \neg L \vee D_2 \tag{2.2}$$

is the resolvent $C_1 \vee C_2 \vee D_1 \vee D_2$. Resolution systems generate resolvents from clauses until a contradiction is reached.

The main problem with resolution systems is to extract from a proof (by resolution) the steps of the proof that are truly necessary [16]. This problem is partly solved by the imposition of search strategies, but even with these strategies there is the problem of the vast search spaces inherent in the proof of complex theorems via strategies that must be complete (i.e., assured always to prove any theorem) for all of first-order logic. PROLOG is a resolution based theorem prover. Resolution will be discussed more thoroughly in relation to PROLOG (Section 2.2.3.4.2).

2.2.3.3 Search Strategies

It is necessary to impose search strategies on proof systems in order to avoid the redundancy involved in simply carrying out all possible deduction sequences. There are two major types of search strategies: *resolution strategies* (strategies that were created specifically for resolution systems, but that can be applied to other theorem proving techniques), and *natural strategies* [16]. These strategies can also be classified as *complete strategies* or *heuristic strategies* respectively. Not all strategies will be considered since there are many of them, and only a brief overview of the types will be presented.

2.2.3.3.1 Resolution Strategies

Resolution strategies are complete in the sense that they are assured to always prove any theorem that follows from the axioms [16, 46]. There are three major categories of resolution strategies: *simplification*, *refinement*, and *ordering*.

Simplification strategies are strategies for removing redundant clauses. Simplification strategies include *subsumption* of clauses and *demodulation*. A clause C subsumes a clause D if a substitution θ exists such that every literal in $C\theta$ appears in D [16]. For example, $P(a, x) \vee P(y, b)$ subsumes $P(a, b)$ where a and b are constants and x and y are variables; here, θ is $x = b$ or $y = a$. There are several different versions of demodulation, but the basic idea is that equations are applied as

destructive rewrite rules to replace certain terms by others. The rewrite rules are destructive in the sense that the replaced rules disappear (are removed).

Refinement strategies are those strategies that determine those clauses to which resolution should be applied and those to which it should not. Examples of refinement strategies are the *set-of-support strategy* and *hyper-resolution*. In the set-of-support strategy, a subset of the set of clauses is taken as the “goal” and resolvents are derived from these clauses and then from the resolvents themselves until the empty clause results. When the set-of-support strategy is used, a reasoning program is not allowed to apply an inference rule (resolution, for example) unless at least one of the potential clauses to which the inference rule is being applied to yield a resolvent has been deduced from some specified subset of the input clauses, or is a member of the specified subset of input clauses [61]. For example, let S be a set of clauses. Choose a nonempty subset, T , of S . The subset T is called the set-of-support, and the clauses in T are said to be supported or to have support. An inference rule cannot be applied unless at least one of the clauses to which it is being applied to yield a resolvent, is a member of T or has been deduced from T .

Hyper-resolution is an inference rule that combines several resolution steps into one. Hyper-resolution is applied to a set of clauses to produce a positive clause (a clause containing no negative literals) [61]. The set of clauses to which hyper-resolution is applied must contain one negative clause (a clause with no positive literals) or one mixed clause (a clause with positive and negative literals), and the rest must be positive clauses. The number of positive clauses in the set must be equal to the number of negative literals in the negative or mixed clause. For example, consider the following set of clauses:

$$\neg Path(x, y) \vee \neg Path(y, z) \vee Accessible(z, x) \quad (2.3)$$

$$Path(townA, townB) \vee Inaccessible(townC, townA) \quad (2.4)$$

$$Path(townB, townC) \quad (2.5)$$

Clause 2.3 is a mixed clause containing two negative literals and one positive literal. Clause 2.4

is a positive clause containing two positive literals, and clause 2.5 is a positive clause containing one positive literal. The number of positive clauses is equal to the number of negative literals in the mixed clause. Applying hyper-resolution to these clauses yields the clause, $Accessible(townC, townA) \vee Inaccessible(townC, townA)$, where the clauses have the following interpretations: if there is a path from x to y and from y to z , then z is accessible from x ; if there is no path from $townA$ to $townB$ then $townC$ is inaccessible from $townA$; there is a path from $townB$ to $townC$; and, if $townC$ is not accessible from $townA$ then $townC$ is inaccessible from $townA$. Normally, the above result would require several steps to determine.

The third type of resolution strategies considered here are *ordering strategies*. Ordering strategies determine the order in which a set of clauses should have resolution applied to them [16]. Examples of ordering strategies include *depth-first* and *breadth-first search*, and the *unit preference strategy*. The unit preference strategy requires that resolution be applied to pairs of clauses such that one of them is a unit clause (a clause containing a single literal), before it can be applied to more general clauses.

2.2.3.3.2 Natural Strategies

Natural strategies are heuristic in the sense that they are not assured to prove any theorem but may be more efficient and natural in the cases they do succeed [16]. Natural strategies are usually associated with natural deduction systems or sequent calculus systems. They are intended to capture some aspect of human reasoning (thus, the term natural). There are many types of natural strategies. A few of these are: *reduction*, *unification*, and the use of *decision procedures*. A reduction is a *rewrite rule*. A rewrite rule is a rule used for replacing any instance of one expression by the corresponding instance of another expression. In terms of natural systems, unification is the process of finding appropriate terms to substitute for variables in the application of quantifier rules. Decision

procedures are built-in theories and models. The built-in theories are special inference rules or procedures that take the place of sets of axioms for certain theories.

2.2.3.4 Specific Theorem Provers

We discuss and describe four specific theorem provers in this section: PVS, PROLOG, Larch Prover, and Knuth-Bendix. The theorem prover incorporated into PVS is based on the sequent calculus. PROLOG is a resolution refutation system, and Larch Prover and Knuth-Bendix are rewrite systems. The strengths and weaknesses of these four systems are presented. The ability of each of these systems to solve the problem described in this dissertation is also discussed.

2.2.3.4.1 Prototype Verification System

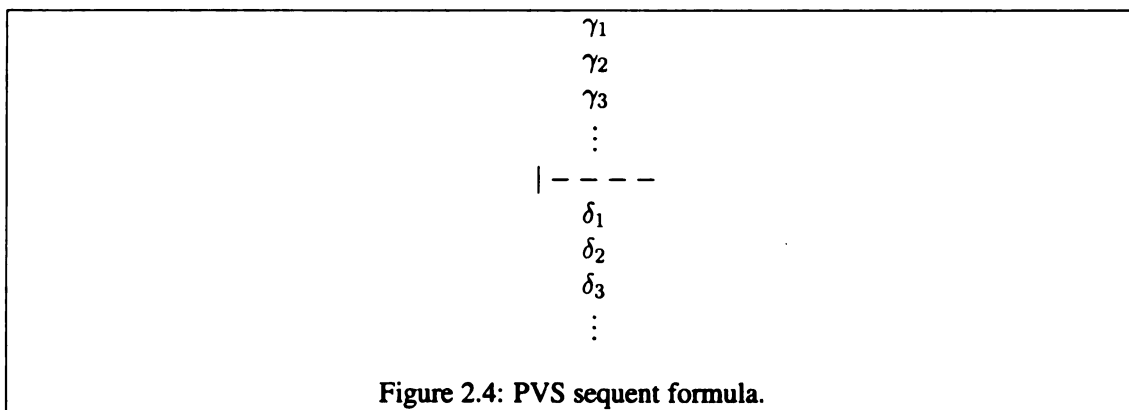
The *Prototype Verification System* (PVS) is a verification system that provides an interactive environment for the development and analysis of formal specifications [15, 44, 45]. PVS consists of a specification language, a parser, a typechecker, a proof checker, specification libraries, and browsing tools.

PVS is designed to help in the detection of errors and in the confirmation of “*correctness*” of hardware and software systems during early life-cycle applications of formal methods. Formal support for conceptualization and debugging is provided during the early design stages in which both requirements and design are expressed in abstract terms. A rich type-system and correspondingly rigorous typechecking is one way in which PVS supports early error detection. For example, an invariant to be maintained by a state machine can be embedded in PVS types by expressing it as a type constraint. In addition, typechecking can generate proof obligations that amount to a very strong consistency check on certain aspects of the specification.

A theorem prover is provided in PVS as another way that PVS supports error detection. An effective way to understand the content of a specification and to identify errors in the specification is to

attempt to prove properties about the specification. There are two ways in which proving properties about a specification can be done: *incidentally*, while attempting to prove a “real” theorem, such as, “does a particular algorithm achieve its purpose”, or *deliberately*, by challenging a specification as part of a validation process. A challenge is a test case posed as a putative (supposed) theorem and is of the form: “if this specification is correct, then the following ought to follow”.

The PVS proof checker is an interactive theorem prover based on sequent calculus (described in Section 2.2.3.2.1). To reiterate, a sequent is an expression that has the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sets (possibly empty) of formulas. The meaning of the sequent formula is: $(\gamma_1 \wedge \gamma_2 \cdots \wedge \gamma_i) \rightarrow (\delta_1 \vee \delta_2 \vee \cdots \vee \delta_j)$, which says that if all of the formulas on the left of the arrow are true, then at least one of the formulas on the right of the arrow is true. A proof goal in PVS is a sequent and has the form shown in Figure 2.4. The sequent formulas γ_i and δ_j are PVS formulas; the γ_i being the antecedents and the δ_j the consequents. Restating the meaning of the sequent formula in terms of antecedents and consequents, the conjunction of the antecedents of a sequent implies the disjunction of the consequents of the sequent. PVS is an affirmation system



that uses backward reasoning (top-down approach to proof development); it starts with the goal conjecture and reduces the goal to subgoals by applying the inference rules backwards until axiom sequents are derived. The inference rules in PVS are in the form of antecedent and consequent

rules. The reason for this is because formulas that are candidates for reduction can appear in either the antecedent or consequent of a sequent and different results need to be generated in each case.

For example, the antecedent and consequent rules for conjunction in PVS are:

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \wedge \vdash \quad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \vdash \wedge$$

The interpretations respectively are, if a conjunction appears in the antecedent of a sequent subgoal, then applying the antecedent conjunction rule to the subgoal yields $A, B, \Gamma \vdash \Delta$, and if a conjunction appears in the consequent of a sequent subgoal, applying the consequent conjunction rule to the subgoal yields two subgoals: $\Gamma \vdash A, \Delta$ and $\Gamma \vdash B, \Delta$. The symbols ' $\wedge \vdash$ ', and ' $\vdash \wedge$ ' signify that the conjunction to which the rule is applied appears in the antecedent or consequent respectively. As an example of the utilization of the antecedent and consequent rules in PVS, assume we have the following goal sequent:

$$\begin{array}{c} A \wedge B \\ C \vee D \\ | \text{-----} \\ B \wedge D \\ A \vee C. \end{array}$$

Applying the antecedent conjunction rule to this sequent yields:

$$\begin{array}{c} A \\ B \\ C \vee D \\ | \text{-----} \\ B \wedge D \\ A \vee C. \end{array}$$

Applying the consequent conjunction rule to this new sequent yields two new subgoals:

$$\begin{array}{cc} \begin{array}{c} A \\ B \\ C \vee D \\ | \text{-----} \\ B \\ A \vee C \end{array} & \begin{array}{c} A \\ B \\ C \vee D \\ | \text{-----} \\ D \\ A \vee C. \end{array} \end{array}$$

The first of these new subgoals reduces to true by another rule called the propositional axiom rule which is automatically applied to every sequent that is ever generated in a proof. Application of

a rule for disjunctions, analogous to the rule for conjunctions, to the second subgoal, followed by application of the propositional axiom rule reduces this second subgoal to true. In actuality, PVS will automatically reduce the original subgoal to true by automatic recursive application of a single rule, followed by automatic application of the propositional axiom rule.

There are three kinds of proof commands in PVS: *primitive rules*, *defined rules*, and *proof strategies*. The primitive rules define the underlying logic of PVS and include structural rules, propositional rules, quantifier rules, and equality rules as well as some others. Defined rules are strategies that are applied in a single atomic step so that only the final effect of the strategy is visible; intermediate steps are hidden from the user. Proof strategies are intended to capture patterns of inference steps and are formed by combining proof commands in various ways.

A *proof tree* is maintained by the prover and it is the goal of the user to construct a proof tree that is complete [55]. A complete proof tree is one in which all of the leaves are recognized as true. Each node in the proof tree is a proof goal (a subgoal: a goal that is yet to be discharged; i.e., proved).

PVS contains a large prelude file that contains many built-in theories that can be used during the proof process. Some of the built-in theories included in the prelude file are Boolean properties, quantifier properties, equality properties, functions, relations, orders, sequences, well-founded induction, measure inductions, sets and set lemmas, reals and real properties, rationals and rational properties, exponentiation, and so on. In addition, PVS contains decision procedures for equality and linear inequality that are complete for linear arithmetic (multiplication by literal constants; expressions of the form $2x + 3y \leq 4z$) [51, 55]. These decision procedures are the workhorses of almost any nontrivial PVS proof [15] and are used by PVS to prove trivial theorems, to simplify complex expressions (particularly definitions), and to perform pattern matching. Linear arithmetic reasoning is performed over the natural numbers and reals. The decision procedures deal solely

with ground formulas (that is, those formulas that contain no quantifiers). Tedious equality and arithmetic reasoning is simplified by the procedures so that the number of trivial subgoals can be minimized and the sequent formulas kept simple.

Efficient data structures are maintained by the decision procedures. Assumptions that are true in the current context are recorded in these data structures. A rule called `record` is used to add more assumptions to the data structures. The assumptions that are recorded are antecedent formulas and negations of consequent formulas. Recall that a proof in sequent calculus systems is expressed as a tree of sequents where the root is labeled with the sequent to be proved. New assertions that get added to the data structures are valid for any descendent proof node of the current sequent and are automatically employed whenever the decision procedures are invoked at the lower nodes. An example of a rule that both adds assumptions to the data structures and attempts simplification using the decision procedures, is the `assert` rule.

Rules using the decision procedures and adding assumptions to the data structures used by the decision procedures, can be sensitive to the order in which they are invoked, since each application of these rules adds assumptions to the data structures used by the decision procedures. As a result, it may be necessary to apply the rule more than once to achieve the desired effect. For example, let a formula A be asserted before a formula B , where the formula B is used to simplify A . The `assert` rule needs to be re-applied to effect the simplification of A , since B is used to simplify A .

Although the decision procedures mainly deal with linear arithmetic, they can also do some simple nonlinear reasoning, but the decision procedures are not complete for this case [51]. There are some modest extensions to the decision procedures for dealing with expressions involving nonlinear sub-terms using simplifications such as $(x + y) * (x - y) = (x * x) - (y * y)$ and simplifications involving division, such as, $x * \frac{y}{x} = y$. In general, however, to prove nonlinear facts, formulas from the prelude must be cited.

PVS is a powerful system that combines a variety of methods to reason about expressions and arrive at solutions, however, it too has some drawbacks. If a subgoal contains all atomic expressions and the user cannot complete the proof for this subgoal, then the subgoal may or may not be unprovable – the burden is on the user to come up with a proof if one exists. Since the subgoal is in a reduced format, it may be possible for the user to examine the subgoal and detect obvious problems with the original conjecture. If no obvious problems can be seen, the user does not know if the goal is unprovable or if the knowledge required to prove the goal is lacking. In addition, since the burden of constructing a proof is placed on the user, PVS may not be amenable to automation if a common strategy cannot be found to arrive at a solution to the problem posed.

2.2.3.4.2 PROLOG

PROLOG is based upon the principles of unification and resolution (Section 2.2.3.2.2) [16]. It uses a backward reasoning, proof by contradiction methodology. PROLOG uses incomplete search strategies (i.e., strategies that are not guaranteed to prove any given theorem, but in the cases they succeed, they may be more efficient and natural) (Section 2.2.3.3).

A PROLOG program consists of a set of facts and rules expressed as *Horn clauses*. A clause is a disjunction of a finite set of literals with no literal appearing twice (clauses were discussed in Section 2.2.3.2.2). In general, *Horn clauses* are clauses that contain at most one positive literal. In PROLOG, the clauses contain exactly one positive literal. A *query* in PROLOG is a Horn clause with no positive literals; it is the negation of an existence theorem converted to PROLOG clausal form (derived below). For example, the clause

$$A \vee \neg B_1 \vee \neg B_2 \vee \cdots \vee \neg B_n$$

is logically equivalent to

$$A \leftarrow (B_1 \wedge B_2 \wedge \cdots \wedge B_n),$$

which, in PROLOG, is written

$$A :- B_1, B_2, \cdots, B_n,$$

where the head of the clause is A and the tail of the clause is the sequence, B_1, B_2, \dots, B_n . A query has the form

$$\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_m,$$

which is equivalent to

$$\neg(A_1 \wedge A_2 \wedge \dots \wedge A_m).$$

In PROLOG the query has the form

$$?- A_1, A_2 \dots A_m.$$

Thus, respectively, the symbols “:-”, “;”, and “?-” in PROLOG have the same meaning as the symbols “ \leftarrow ”, “ \wedge ”, and “ \neg ”. For example, the clause

$$\text{Greater}(X, Z) \vee \neg \text{GreaterBase}(X, Y) \vee \neg \text{GreaterBase}(Y, Z)$$

is logically equivalent to

$$\text{Greater}(X, Z) \leftarrow \text{GreaterBase}(X, Y) \wedge \text{GreaterBase}(Y, Z),$$

and in PROLOG is

$$\text{Greater}(X, Z) :- \text{GreaterBase}(X, Y), \text{GreaterBase}(Y, Z).$$

The interpretation of this clause is: if $(x > y) \wedge (y > z)$ then $(x > z)$. Assuming that the facts $\text{GreaterBase}(5, 3)$ and $\text{GreaterBase}(3, 1)$

are in the rule base along with the above clause, an example of a query relating to the above clause is:

$$\neg \text{GreaterBase}(X, 3) \vee \neg \text{GreaterBase}(3, Z) \vee \neg \text{Greater}(X, Z)$$

which is equivalent to

$$\neg(\text{GreaterBase}(X, 3) \wedge \text{GreaterBase}(3, Z) \wedge \text{Greater}(X, Z)),$$

which in PROLOG is

$$?- \text{GreaterBase}(X, 3), \text{GreaterBase}(3, Z), \text{Greater}(X, Z).$$

The result from the query would be $X = 5, Z = 1$.

In addition, all program clauses and queries are prefixed implicitly by universal quantifiers. For

example, let x_1, \dots, x_j be the variables appearing in a query. Then the query has the form:

$$\forall x_1 \dots \forall x_j (\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_m),$$

which is equivalent to

$$\forall x_1 \dots \forall x_j \neg(A_1 \wedge A_2 \wedge \dots \wedge A_m),$$

which in turn is equivalent to

$$\neg(\exists x_1 \dots \exists x_j (A_1 \wedge A_2 \wedge \dots \wedge A_m)).$$

Thus, as previously mentioned, a query is the negation of an existence theorem converted to

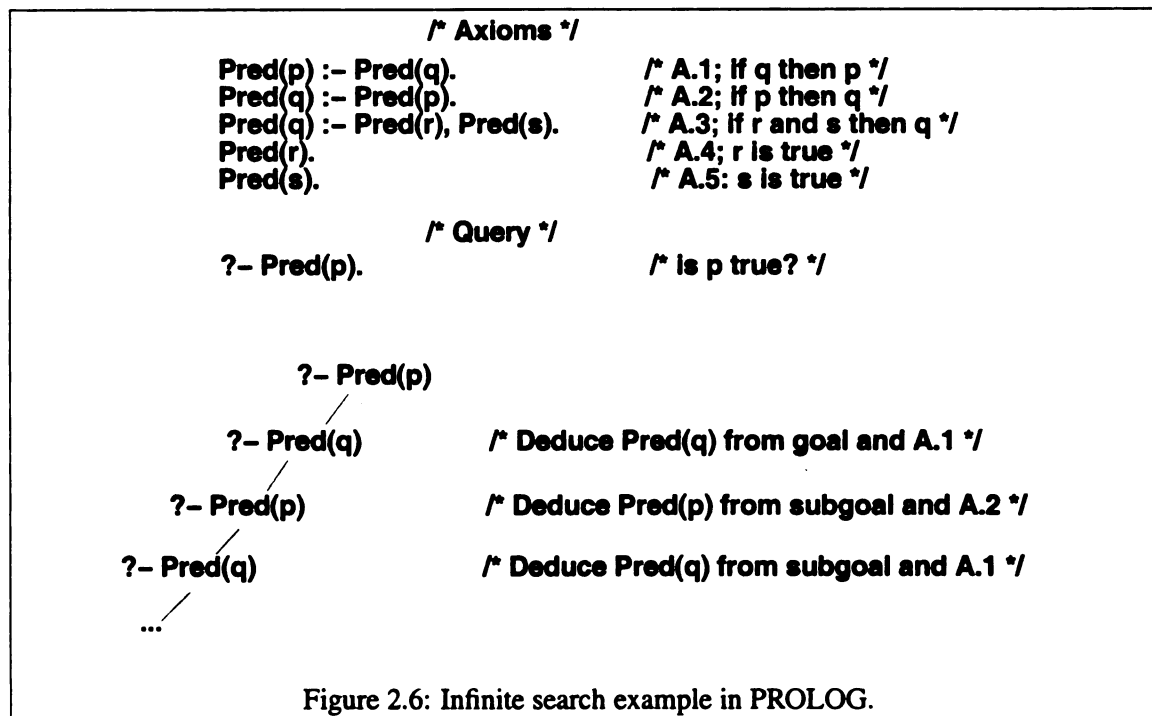
PROLOG clausal form.

The computation strategy used in PROLOG (based on unification and resolution) starts with the goal clause, and unifies it with the head of a program clause to produce a resolvent. This resolvent becomes the new goal. The leftmost subgoal of each goal is selected and a depth-first search strategy is used. A built-in ordering rule is used that selects the “top-most” program clause in the unification with the selected goal. In essence, PROLOG searches through all possible sequences of deductions from the initial set of clauses until a solution is found. A solution to a query may be a yes/no answer, or a specific instantiation for the variable(s) in the query. Figure 2.5 shows two possible solution scenarios; the first query demonstrates a yes/no solution, while the second query demonstrates finding an instantiation for the variables X and Y in the query. Note that there are other possible solutions to the latter query that could have been found. The search strategy and ordering rule determine which solutions will be found first. The user could tell PROLOG to find another solution (and another) if desired; PROLOG will uninstantiate the variable (or variables) and backtrack to seek another solution from there.

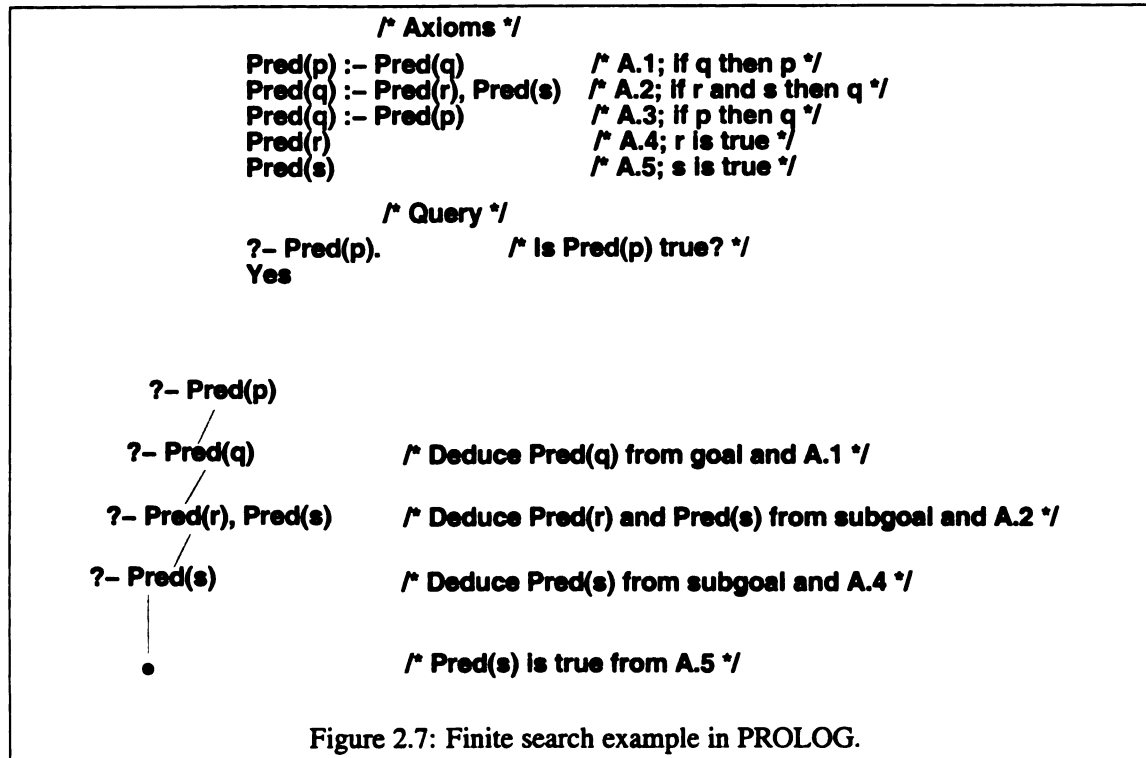
/* Axioms */	
Plus(2,3,5).	/* 2 + 3 = 5 */
Plus(3,4,7).	/* 3 + 4 = 7 */
Plus(2,5,7).	/* 2 + 5 = 7 */
Plus(1,6,7).	/* 1 + 6 = 7 */
/* Queries */	
?- Plus(2,3,5).	/* 2 + 3 = 5 ? */
Yes	
?- Plus(X,Y,7).	/* X + Y = 7 */
X = 3,	
Y = 4	

Figure 2.5: PROLOG solution scenarios.

One problem with the search mechanism in PROLOG is that the depth-first search strategy is an incomplete search strategy in the sense that solutions to some problems may never be found. An example of a set of axioms and a query that result in an infinite search is shown in Figure 2.6; the corresponding search tree is also shown. The speed with which other problems are solved however, offsets the problem of an incomplete search strategy. In addition, there are mechanisms available to alter the control sequence; these include *cut* and *fail*. The *cut*, written '!' in PROLOG, is a special goal that when activated succeeds immediately but only once [50]. If processing returns to the cut due to backtracking, the cut will fail and cause the failure of its *parent goal* (i.e., the goal that called the rule in which the cut appeared) as well. The *fail* predicate is a predicate built-in to PROLOG that when activated as a goal, immediately fails and thus always causes backtracking. Thus, the *cut* and *fail* can be used to alter the normal control sequence used in PROLOG.



The example shown in Figure 2.6 illustrates another drawback of PROLOG; the order of the rules in the rule base may effect whether or not a solution is found. Figure 2.7 shows the same



example as shown in Figure 2.6, except that the order of axioms A. 2 and A. 3 has been reversed. PROLOG can now find a solution to the query $\text{Pred}(p)$. The finite search tree showing how the solution is obtained when the order of the two axioms is reversed is also shown in the figure. The order of rules in the rule base may also effect the speed with which a solution is found.

Other important attributes of PROLOG that require consideration include the following:

1. PROLOG is based on a *closed world* assumption in that it assumes that the given program contains all true assertions (if a fact is not in the described world it is not true).
2. Most PROLOG implementations do only integer arithmetic.
3. A term to be evaluated as an arithmetical expression must contain no uninstantiated variables and must be a valid expression.
4. PROLOG is not suitable for number crunching.

The closed world assumption is detrimental for complex domains since it means every true fact related to a particular domain must be included. For the domain of research discussed in this dissertation, since PROLOG has no built-in theories for real numbers, all relevant theories for real

numbers in our domain of application must be added; if some relevant theories are missed, certain queries may result in no solution being found when there is a solution, or certain queries may result in an infinite loop when there is a solution. Items 2, 3, and 4 are particularly detrimental in the domain of this research since the problem described in this dissertation requires arithmetic expressions involving real numbers and uninstantiated variables. PROLOG has other attributes also, but those discussed in this section are the most important to consider in regards to the work described in this dissertation.

2.2.3.4.3 Larch Prover

The Larch Prover (LP) is an interactive theorem proving system for multi-typed first-order logic that is intended to assist users in finding and correcting flaws in conjectures; the predominant activity in the early stages of the design process. LP is designed to treat equations as rewrite rules. It can also carry out other inferences such as induction and proof by cases. The Larch Prover has been used to reason about designs for circuits, concurrent algorithms, hardware, and software [19, 43].

The Larch Prover assists in the proof process by carrying out routine steps in a proof automatically, but leaves it up to the user to build a complete proof. The underlying methodology of the Larch Prover supports both the forward (bottom-up) reasoning approach and the backward (top-down) reasoning approach. One problem with the forward approach (starting with axioms and trying to synthesize the goal conjecture) is that the system may continue to synthesize theorems that lead further and further away from the goal. As a result, the goal may never be reached. During the course of a proof, the user chooses commands to apply that use either the forward or the backward reasoning method. In addition, there are some built-in axioms that can be incorporated into the proof specification (the specification created by the user to initiate the proof process) and used during proof development. These axioms are originally specified in the Larch Shared Language (LSL;

not discussed in this document) and need to be converted to axioms usable by the Larch Prover.

Equations play a major role in the Larch Prover (LP). Some of LP's inference rules work directly on equations, but most require that equations be oriented into rewrite rules [19]. Rewrite rules and equations have the same logical meaning, but operationally they behave differently. A *rewrite rule* is an ordered pair $\langle l, r \rangle$ of terms (a term consists of either a variable or an operator and a sequence of terms known as its arguments), usually written $l \rightarrow r$, where l is not a variable and every variable that occurs in r also occurs in l , and the position of l and r and the direction of the arrow determine the *orientation* of the rewrite rule; $l \rightarrow r$ means whenever l appears in an equation, it can be rewritten by replacing it with r . The restriction placed on l and r is necessary so the rewrite rule will be terminating (i.e., there is no infinite sequence of rewrites). A set of rewrite rules comprises a *term-rewriting system*, or a *rewriting system* for short.

Equations are oriented into rewrite rules by the Larch Prover and these rules are then used to reduce terms to normal forms. For example, the assertions shown on the left in Figure 2.8 will be oriented into the rewrite rules shown on the right in the figure. With these rewrite rules (and some declarations which are not shown in the figure), the conjecture $1 < 1 + 1$ can be proved. The proof is shown in Figure 2.9.

Assertions:	Rewrite Rules:	
$i + 0 == i$	$0 + i \rightarrow i$	R.1
$i + s(j) == s(i + j)$	$s(i) + j \rightarrow s(i + j)$	R.2
$\text{not}(i < 0)$	$i < 0 \rightarrow \text{false}$	R.3
$0 < s(i)$	$0 < s(i) \rightarrow \text{true}$	R.4
$s(i) < s(j) == i < j$	$s(i) < s(j) \rightarrow i < j$	R.5
$1 == s(0)$	$1 \rightarrow s(0)$	R.6

Figure 2.8: Assertions and rewrite rules in Larch Prover.

$1 < 1 + 1$	Conjecture
$s(0) < s(0) + s(0)$	Apply R.6 three times
$s(0) < s(0 + s(0))$	Apply R.2
$s(0) < s(s(0))$	Apply R.1
$0 < s(0)$	Apply R.5
true	Apply R.4

Figure 2.9: An example proof using rewrite rules in LP.

In the sense that it is an interactive theorem proving system and used in the early stages of the development process it is similar to PVS. However, it appears that PVS has both a more structured environment and a more user friendly environment for specification and proof development, and a more structured underlying proof methodology. The specification language and interactive theorem prover for PVS are highly integrated. This is not the case for the Larch Prover; the specification environment (not discussed in this document) and prover are not integrated at all. In addition, PVS has a large prelude file that contains many predefined theorems that can be easily incorporated into PVS proofs. This feature is also lacking in the Larch Prover. In addition, term-rewriting systems are not amenable to semantic analysis of equations involving arithmetic expressions, and the Larch Prover has no built-in theories for real numbers and operations on real numbers. Both of these capabilities are required to solve the problems described in this dissertation.

2.2.3.4.4 Knuth-Bendix

The Knuth-Bendix theorem prover is concerned almost entirely with the equality relation and deciding the equality of terms with respect to sets of equations [16]. The Knuth-Bendix system is a term rewriting system in the sense that an equation is treated as a rewrite rule; i.e., an equation may be oriented into a rule and used to rewrite a term into another term; the rewrite rule can only be applied in one direction, and can not be applied backwards, thus, the orientation. For example,

the equation $a + s(b) = s(a + b)$ may be oriented into the rule $a + s(b) \rightarrow s(a + b)$ and the term $(a + 0) + s(a + b)$ may be rewritten, using the rule, into $s((a + 0) + (a + b))$. In some cases, this procedure makes it possible to prove that a certain equation is a consequence of other equations. The major application of the Knuth-Bendix procedure (i.e., the area for which it was originally designed) is proof by induction.

Consider a specification with the following axioms for addition on the Natural numbers [16]:
 $x + 0 = x$ and $x + s(y) = s(x + y)$, where $s()$ is the successor function. To prove the associativity of '+' according to the principle of induction, we have to prove the base case, $(u + v) + 0 = u + (v + 0)$, and the induction case, $(u + v) + s(w) = u + (v + s(w))$, based on the assumption that the hypothesis, $(u + v) + w = u + (v + w)$, is true; i.e., there exists a proof for it. From the first axiom, the base case may be reduced to $u + v = u + v$ by substituting $u + v$ for x and rewriting the left and right sides of the equation with the consequent of the axiom; the result is trivially true. In a similar way, using the second axiom above, the induction case may be reduced to the subgoal $s((u + v) + w) = s(u + (v + w))$. Since by assumption, $(u + v) + w = u + (v + w)$ is true, the conjecture is proven.

Other applications of Knuth-Bendix are in the areas of program synthesis and refutation theorem proving.

2.3 Summary

In this chapter, we discussed several analysis techniques and presented the strengths and weaknesses of each method. No single technique, whether dynamic or static, is capable of addressing all analysis concerns [48, 65, 67]. Reachability analysis entails enumeration of all reachable system states and can lead to state space explosion for even moderately sized systems. Early model checking techniques suffered from the same limitations since they also relied on enumeration of all reachable

system states. Symbolic model checking allows larger systems to be analyzed since the states can be represented symbolically and therefore, a set of states in the original specification can map to a single state in the model.

We discussed three types of theorem proving systems in this chapter: sequent calculus systems, resolution based systems, and rewrite systems. PVS is based on the sequent calculus, PROLOG is a resolution-based system, and the Larch Prover and Knuth-Bendix are both types of rewrite systems, though they use different types of rules and strategies.

Our investigation of the strengths and weaknesses of the theorem provers discussed in this chapter, led us to choose PVS as one of the analysis components for the research described in this dissertation. The decision procedures in PVS give PVS a major advantage over other theorem provers that do not have decision procedures; in particular, they are advantageous for applying to one aspect of the problem discussed in this dissertation. Rushby notes that “it is enormously tedious to perform verifications involving even modest quantities of arithmetic in systems such as HOL that lack decision procedures” [52]. In addition, PVS allows automation, is freely available, and is well documented and supported.

Chapter 3

Analysis of Software Requirements

In Chapter 2 we discussed analysis techniques such as model checking and theorem proving, that have traditionally been applied in the area of hardware verification [39, 54] to determine if certain application specific properties are satisfied, or to verify that an implementation satisfies its specification [52, 53, 60]. We discussed the strengths and weaknesses of each technique. We used the information about the strengths and weaknesses of the analysis techniques to help us determine which techniques and tools we might use to achieve the goals of this research.

The motivation for the research discussed in this dissertation was to find a way to check large disjunctive and conjunctive expressions to see if they form tautologies and contradictions and subject to the following criteria: the analysis must be automated, the analysis must generate analysis reports in a reasonable amount of time, the analysis must generate analysis reports with an acceptable level of accuracy, and the analysis must be scalable and generalizable (i.e., applicable to real-world problems and not limited to state-based requirements). The first intended application of our technique was for analyzing software specifications; specifically, state-based specifications. In this chapter we discuss several specific techniques for analyzing software requirements, and describe the shortcomings of each approach.

Both Heimdahl and Heitmeyer are working on analysis of state-based requirements. Section 3.1 discusses analyzing state-based requirements for completeness and consistency. We discuss Heitmeyer's work in detail in Section 3.1.1. Heimdahl's doctoral work provided the foundation for this research and is discussed in detail in Section 3.1.2. In Section 3.1.3, we identify a problem common to both approaches, namely, *spurious errors*, and discuss in detail the difficulties that occur when checking logical expressions for tautologies and contradictions. Section 3.1.4 summarizes some of the similarities and differences between the two methods, and summarizes the strengths and weaknesses of both approaches.

Recently, more work is being done in the area of model checking software requirements. Section 3.2 discusses the work in this area, including work by Atlee and Gannon [4], Sreemani and Atlee [57], Wing and Vaziri-Farahani [60], Anderson et al. [1], and most recently, Bharadwaj and Heitmeyer [6]. At the end of Section 3.2 we note that spurious errors may also be a problem in the area of model checking software requirements and we note why spurious errors tend to be a common problem among analysis methods. In Section 3.3 we list four classes of spurious errors that we identified during the course of this research and we identify the undetected contradictions that lead to them.

Section 3.4 provides a summary of the work in analysis of software requirements, and discusses the shortcomings of the current analysis methods and why an enhanced analysis technique is required.

3.1 Analysis of State-Based Requirements for Completeness and Consistency

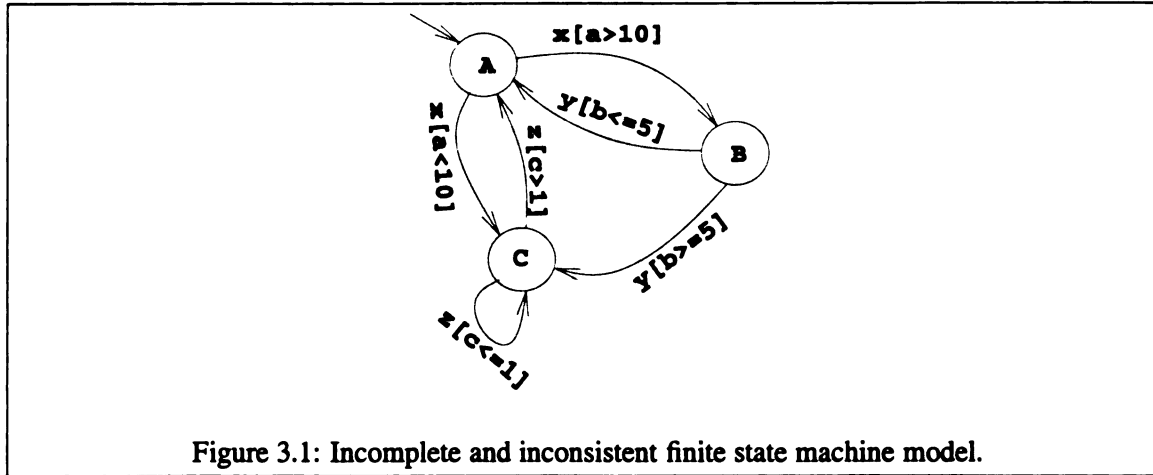
Jaffe, Leveson, Heimdahl, and Melhart have described a set of criteria that all requirements documents should satisfy [38]. The criteria include *precision* or *lack of ambiguity* in the requirements, and *robustness*. *Lack of ambiguity* in the requirements means they are free from conflicting requirements and undesired nondeterminism [26]. Critical systems are often reactive systems. The behavior of reactive systems is defined with respect to assumptions about the environment within which the systems operate [38]. A *robust* system will detect violations of these assumptions (such as unexpected inputs) and respond appropriately to these violations. Since the system is built from the specification, its robustness depends on the completeness of the specification of the environmental assumptions. In terms of a state-based requirements model, robustness implies the following [26]:

- a behavior (transition) must be defined for every possible input in every state,
- the disjunction of conditions on every transition out of any state must form a tautology, and
- a behavior (transition) must be defined in every state in case there is no input for a given period of time (a timeout).

If a specification is free from ambiguity and is robust, then it is *consistent* and *complete*. Both completeness and consistency are essential criteria that should be satisfied by all requirements specifications. These terms will now be defined in terms of a finite state machine model.

Consider the finite state machine shown in Figure 3.1. This model consists of three states, *A*, *B*, and *C*, and three inputs, *x*, *y*, and *z*. The inputs potentially trigger a transition from one state to another when they are received in a current state. A guarding condition is associated with each transition; the guarding conditions are enclosed in brackets and follow the associated transition input. If an input is received and the guarding condition is satisfied, then a transition occurs. The *a*, *b*, and *c* in the guarding conditions represent variables in the system. If a specification is complete, then there will be a transition specified for every possible input and input sequence; this implies that

the disjunction of the guarding conditions out of a state for a given input must be a tautology. If a specification is consistent, it is deterministic; i.e., there will not be more than one possible transition out of a state under the same conditions. This implies that the pairwise conjunction of the guarding conditions out of a given state for a given input, must be unsatisfiable (i.e., must be a contradiction).



The finite state model shown in Figure 3.1 is both incomplete and inconsistent. The transitions on input x out of state A are incomplete; if $a = 10$ on input x , no action is specified. In addition, the transitions specified on input y out of state B are inconsistent; if $b = 5$, a transition will either occur from B to A or from B to C . The disjunction of the transitions out of state C with respect to input z is a tautology so the transitions out of state C are complete with respect to z . In addition, the conjunction of the transitions out of state C with respect to z is a contradiction, therefore, the transitions out of state C are also consistent with respect to z . The incompletenesses and inconsistencies in the specification of transitions out of states A and B must both be detected and corrected. If the requirements are incomplete or inconsistent, the design and implementation might be incomplete and inconsistent. For critical systems, this can mean that responses to certain environmental stimuli are not considered, or that conflicting choices exist in certain situations that can lead to a system state that causes harm to life, property, or the environment.

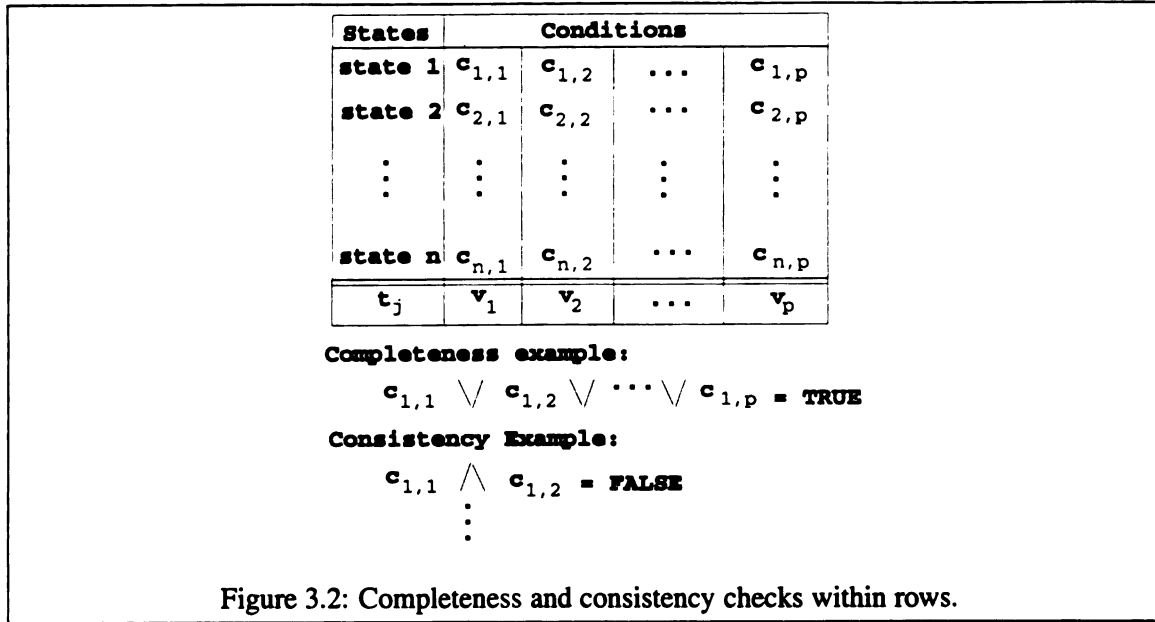
Heitmeyer, Jeffords, and Labaw [30, 31, 32] and Heimdahl [25, 26, 27, 28] are doing research related to analysis of requirements for completeness and consistency. Heitmeyer and her colleagues use the requirements language Software Cost Reduction (SCR). Heimdahl uses the requirements language Requirements State Machine Language (RSML). The underlying model used in both languages is a finite state machine model; Heitmeyer and her colleagues use a *Moore* machine, whereas Heimdahl uses a *Mealy* machine (in a Moore machine, outputs are associated with states; in a Mealy machine, outputs are associated with transitions). Both requirements languages attempt to abstract away implementation details and describe only the externally visible behavior required of the system. Both languages specify conditions (guarding conditions) that must be satisfied before a transition can occur; these guarding conditions are specified as logical expressions (*predicates*). Both models use a tabular format to represent different aspects related to states, events, and transitions, since it is easier to comprehend guarding conditions expressed in a tabular format than the equivalent predicate logic expression [41].

SCR uses several different tabular representations to show the relations between different aspects of the system behavior, whereas RSML uses only one type of table. The similarities and differences in the two requirements languages are not important here; essentially, different requirements languages have been chosen to model the same types of systems: real-time reactive systems that fall into the category of critical systems. The important aspect of their work is regarding the analysis of the requirements for completeness and consistency. Both Heitmeyer and Heimdahl rely on ORing and ANDing together information represented in their tables to check the requirements for completeness and consistency. Section 3.1.1 describes the analysis techniques Heitmeyer and her colleagues use to check for completeness and consistency with respect to SCR. Section 3.1.2 describes the analysis techniques Heimdahl uses to check for completeness and consistency with respect to RSML. Section 3.1.3 describes and discusses a problem common to both approaches.

3.1.1 SCR

One tabular representation used in SCR relates states and conditions to output variables. Figure 3.2 shows the format of this type of SCR table (note that this table is not an exact representation of an actual table in SCR – SCR uses the term mode rather than state.). For this type of table, pairs of conditions (or conditional events) are ANDed within a row to check for mutual exclusion (consistency), and all conditions within a row are ORed together to check for completeness; Figure 3.2 provides a graphical depiction of this method. The $c_{i,j}$'s represent conditions (logical expressions), t_j represents a state variable, and the v_i 's represent values that t_j can be assigned. The semantics of the table shown in Figure 3.2 are: in state i if condition $c_{i,1}$ is true, then $t_j = v_1$; if condition $c_{i,2}$ is true, then $t_j = v_2$, and so on to condition $c_{i,p}$. The table is deterministic (consistent) if none of the $c_{i,j}$'s in a row overlap. To check the table for consistency, a pairwise ANDing of the conditions in a row must be performed, and the result of each conjunction should be false; $\forall j, k, k'; k \neq k' : c_{j,k} \wedge c_{j,k'} = \text{FALSE}$. The table is complete if the disjunction of all conditions in a row is true; $\forall j, \bigvee_{k=1}^p c_{j,k} = \text{TRUE}$. Completeness means that at least one condition can be satisfied and consistency ensures that only one condition can be satisfied. Their tool applies a tableaux-based decision procedure to determine whether the logical expressions representing consistency and completeness given above are tautologies [31].

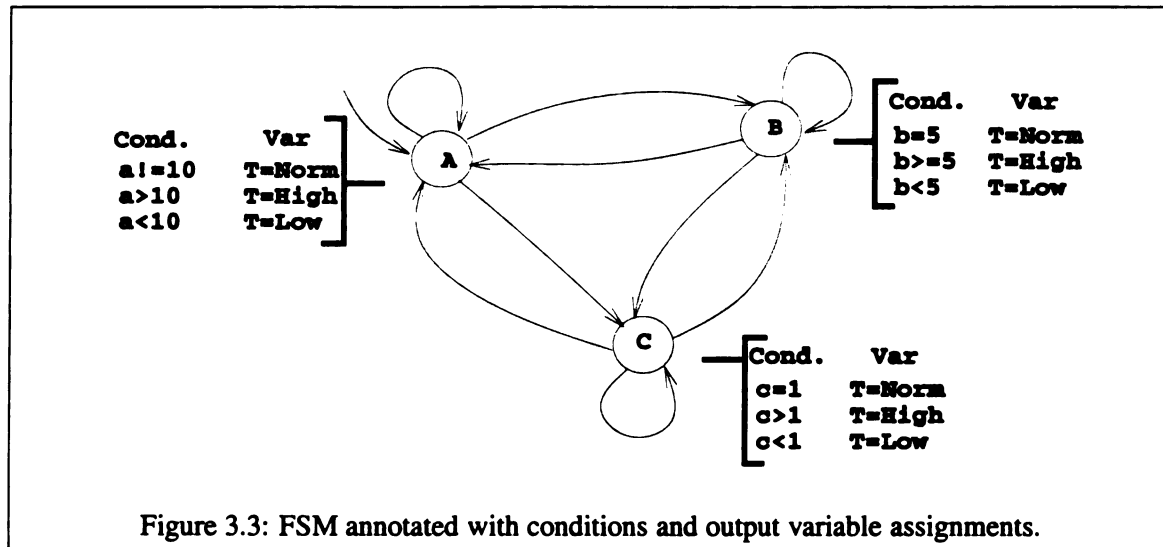
Consider Figure 3.3 which shows a state transition model with three states, A , B , and C , and nine transitions. Each state is annotated with a set of conditions followed by an assignment to an output variable. System variables are represented by a , b , and c . T is an output variable that can take on the values *Low*, *Norm*, and *High*. If the condition $c > 1$ is true in state C , T will be set to *High*. The other condition/assignment annotations are interpreted likewise. Figure 3.4 shows the tabular representation relating states and conditions to output variables derived from the specification. To interpret the tabular representation, assume that the current state is state B and that



the condition $b = 5$ is true; the value of T will be set to *Norm*.

The following results are obtained from performing consistency analysis on the conditions shown in the table:

- The conditions out of state A are inconsistent; that is, more than one condition out of state A will be true if $a > 10$ or $a < 10$:
 - $(a < 10) \wedge (a \neq 10)$ does not form a contradiction,
 - $(a < 10) \wedge (a > 10)$ does form a contradiction, and
 - $(a \neq 10) \wedge (a > 10)$ does not form a contradiction.
- The conditions out of state B are inconsistent; that is, more than one condition out of state B will be true if $b = 5$:
 - $(b < 5) \wedge (b = 5)$ does form a contradiction,
 - $(b < 5) \wedge (b \geq 5)$ does form a contradiction, and
 - $(b = 5) \wedge (b \geq 5)$ does not form a contradiction.
- The conditions out of state C are consistent; all pairwise conjunctions form a contradiction.



States	Conditions		
A	$a < 10$	$a \neq 10$	$a > 10$
B	$b < 5$	$b = 5$	$b \geq 5$
C	$c < 1$	$c = 1$	$c > 1$
T	Low	Norm	High

Figure 3.4: Table relating states and conditions to an output variable.

The following results are obtained from performing completeness analysis on the conditions shown in the table:

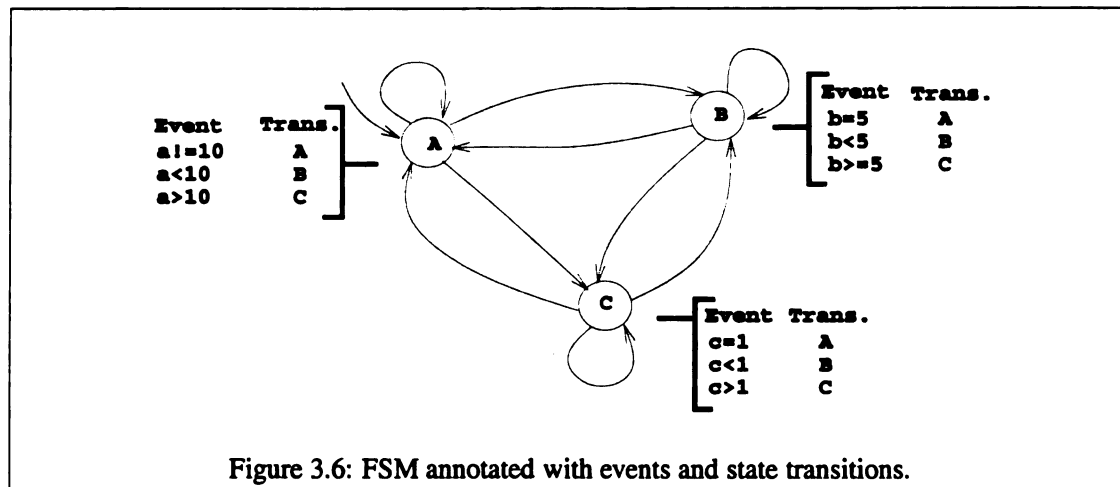
- The conditions out of state A are incompletely specified; condition $a = 10$ is not considered:
 - $(a < 10) \vee (a \neq 10) \vee (a > 10)$ is not a tautology.
- The conditions out of state B are completely specified:
 - $(b < 5) \vee (b = 5) \vee (b \geq 5)$ is a tautology.
- The conditions out of state C are completely specified:
 - $(c < 1) \vee (c = 1) \vee (c > 1)$ is a tautology.

The tabular representation in SCR that most closely relates to the tabular representation in RSML, is the state transition table (referred to as mode transition table in SCR). The general structure of a state transition table is shown in Figure 3.5 (note that this table is not an exact representation of an actual table in SCR for the same reason as stated earlier). The S_i 's represent the current state, the $E_{i,j}$'s represent events that trigger transitions out of state S_i , and the $S_{i,j}$'s represent the new state that will be entered if the event $E_{i,j}$ occurs in state S_i . For example, if the current state is S_1 and event $E_{1,2}$ occurs, then the system will enter state $S_{1,2}$. The table is consistent if the pairwise ANDing of the events results in a contradiction. These tables are not checked for completeness.

Current State	Events	New State
S_1	$E_{1,1}$ $E_{1,2}$ \dots E_{1,j_1}	$S_{1,1}$ $S_{1,2}$ \dots S_{1,j_1}
S_2	$E_{2,1}$ $E_{2,2}$ \dots E_{2,j_2}	$S_{2,1}$ $S_{2,2}$ \dots S_{2,j_2}
\dots	\dots	\dots
S_p	$E_{p,1}$ $E_{p,2}$ \dots E_{p,j_p}	$S_{p,1}$ $S_{p,2}$ \dots S_{p,j_p}

Figure 3.5: A state transition table abstraction.

Figure 3.6 shows a state transition model modeling the transitions from and to the states *A*, *B*, and *C*. The states are annotated with events and transitions. The tabular representation of this model is shown in Figure 3.7. Let the current state be *B*. The table shows that if the event $b \geq 5$



is true, then a transition will occur from state *B* to state *C* (note that if $b = 5$ the specification is nondeterministic; a transition can also occur from state *B* to state *A*). The semantics of the other table entries are the same. The specification is consistent if the pairwise conjunction of events associated with a particular state form a contradiction. The following results are obtained from performing consistency analysis on the events shown in the table:

- The conditions out of state *A* are inconsistent; that is, more than one transition out of state *A* can occur if the event $a > 10$ or $a < 10$ occurs:
 - $(a \neq 10) \wedge (a < 10)$ does not form a contradiction,
 - $(a \neq 10) \wedge (a > 10)$ does not form a contradiction, and
 - $(a < 10) \wedge (a > 10)$ does form a contradiction.
- The conditions out of state *B* are inconsistent; that is, more than one transition out of state *B* can occur if the event $b = 5$ occurs:
 - $(b < 5) \wedge (b = 5)$ does form a contradiction,
 - $(b < 5) \wedge (b \geq 5)$ does form a contradiction, and
 - $(b = 5) \wedge (b \geq 5)$ does not form a contradiction.
- The conditions out of state *C* are consistent; all pairwise conjunctions form a contradiction.

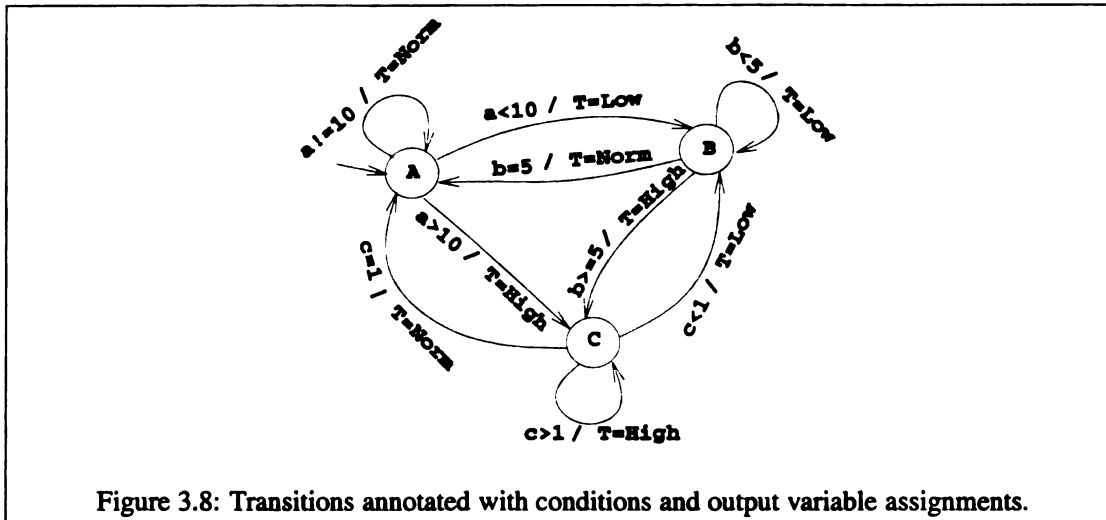
Curr. State	Events	New State
A	a != 10	A
	a < 10	B
	a > 10	C
B	b < 5	B
	b = 5	A
	b >= 5	C
C	c > 1	C
	c = 1	A
	c < 1	B

Figure 3.7: Table relating states and events to new states.

3.1.2 RSML

SCR uses multiple tabular representations to represent the behavior of a system. RSML uses a single and slightly different tabular representation. The single tabular representation used in RSML describes the conditions that must be met before a transition can take place. Consider the state transition model shown in Figure 3.8. The transitions are annotated with guarding conditions and output variable assignments. In RSML, the annotations on transitions have the following interpretation: the expressions preceding the '/' represent guarding conditions on the transitions, and the expressions following the '/' represent the output that gets generated if a transition occurs. Also associated with each transition is a trigger event (not shown in the figure). In this example, we assume that all transitions are triggered by the same event. A transition occurs if the trigger event associated with a transition occurs and the guarding condition associated with that transition is true.

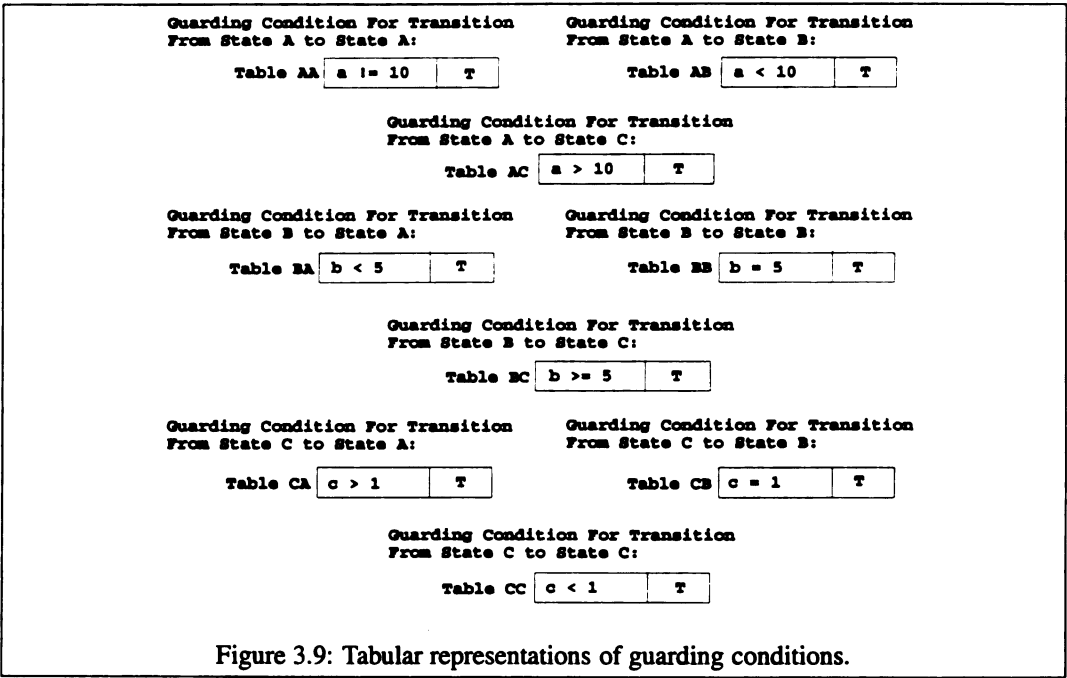
In RSML the guarding conditions are represented as disjunctive normal form (DNF) tables called AND/OR tables. Figure 3.9 shows the tabular representations of the guarding conditions on transitions out of the states in Figure 3.8 (in actuality, guarding conditions are much more complex than those shown and the tabular representations consist of multiple rows and columns; Figure 3.10 shows an example of an actual guarding condition). If the guarding conditions specified for



transitions out of a particular state are consistent, then the pairwise conjunction of the guarding conditions is a contradiction. If the guarding conditions are completely specified for transitions out of a particular state, the disjunction of the guarding conditions is a tautology. Thus, in RSML the analyses occur between tables, whereas in SCR the analyses occur within tables. That is, with RSML, tabular representations of guarding conditions are ORed and ANDed together to test for completeness and consistency; Figure 3.11 shows a high level representation of this. If the disjunction of all tables for a specific state and trigger event is a tautology, then the guarding conditions are complete as specified. If the conjunction of pairs of tables for a specific state and input (trigger event) is a contradiction, then the transition conditions represented by the tables are consistent.

Consistency analysis on the tables shown in Figure 3.9 yields the following results:

- The guarding conditions for transitions out of state *A* are inconsistent; if $a \neq 10$, any of the transitions can be taken.
 - $TableAA \wedge TableAB$ is not a contradiction,
 - $TableAA \wedge TableAC$ is not a contradiction, and
 - $TableAB \wedge TableAC$ is a contradiction.



Guarding Condition For Transition
From State B to State C:

Table BC

b >= 5

T

Guarding Condition For Transition
From State C to State A:

Table CA

c > 1

T

Guarding Condition For Transition
From State C to State B:

Table CB

c = 1

T

Guarding Condition For Transition
From State C to State C:

Table CC

c < 1

T

Transition(s):

Proximate-Traffic

 →

Other-Traffic

Location: Other-Aircraft > Intruder-Status_{s-136}

Trigger Event: Air-Status-Evaluated-Event_{e-279}

Condition:

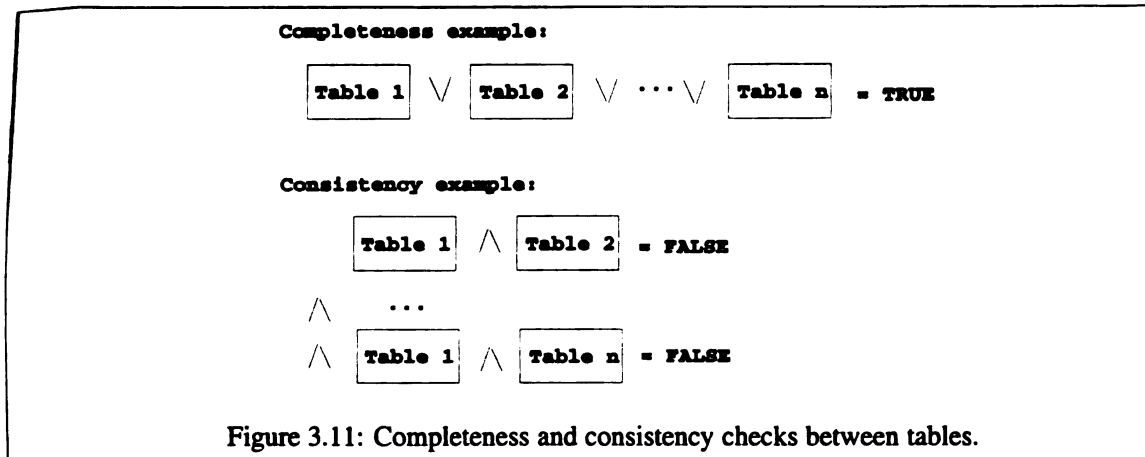
AND

Alt-Reporting _{s-101} in state Lost	T	T
RA-Mode-Canceled _{m-218}	.	.	T	T
Alt-Reporting _{s-101} in state No	.	.	T	T	T	T	.	.
Other-Bearing-Valid _{v-120} = True	F	.	F	.	F	.	.	.
Other-Range-Valid _{v-117} = True	.	F	.	F	.	F	.	.
Proximate-Traffic-Condition _{m-216}	F	.
Potential-Threat-Range-Test _{m-214}	T	T	.	.
Potential-Threat-Condition _{m-213}	F	.
Threat-Condition _{m-140}	F	.
Other-Air-Status _{s-101} in state On-Ground	T

OR

Output Action: Intruder-Status-Evaluated-Event_{e-279}

Figure 3.10: Guarding condition for transition from Proximate-Traffic to Other-Traffic.



- The guarding conditions for transitions out of state *B* are inconsistent; if $b = 5$, two transitions can be taken.
 - $\text{TableBA} \wedge \text{TableBB}$ is a contradiction,
 - $\text{TableBA} \wedge \text{TableBC}$ is a contradiction, and
 - $\text{TableBB} \wedge \text{TableBC}$ is not a contradiction.
- The guarding conditions for transitions out of state *C* are consistent:
 - $\text{TableCA} \wedge \text{TableCB}$ is a contradiction,
 - $\text{TableCA} \wedge \text{TableCC}$ is a contradiction, and
 - $\text{TableCB} \wedge \text{TableCC}$ is a contradiction.

Completeness analysis on the tables shown in the figure yields the following results:

- The guarding conditions for transitions out of state *A* are incompletely specified; the condition $a = 10$ has not been specified:
 - $\text{TableAA} \vee \text{TableAB} \vee \text{TableAC}$ is not a tautology.
- The guarding conditions for transitions out of state *B* are completely specified:
 - $\text{TableBA} \vee \text{TableBB} \vee \text{TableBC}$ is a tautology.
- The guarding conditions for transitions out of state *C* are completely specified:
 - $\text{TableCA} \vee \text{TableCB} \vee \text{TableCC}$ is a tautology.

Both analysis approaches discussed above report results that are quite promising [26, 27, 28, 30, 31, 32]. However, both approaches suffer from a similar problem; the analysis methods being applied often generate spurious errors. These spurious errors can be traced to several sources. The

next section details the sources leading to spurious errors. Since Heimdahl's work is the foundation for the research described in this dissertation, the bulk of the discussion in Section 3.1.3 focuses on his work.

3.1.3 Common Problem Between Methods: Spurious Errors

In Heimdahl's work, one source of spurious errors was the lack of a type system in the RSML notation [27]. For example, consider the variable T defined earlier. T is an enumerated type that can take on the values *Low*, *Norm*, and *High*. Enumerated types are mutually exclusive and all-inclusive, but the analysis method used does not know the semantics of enumerated types so it will report conditions as errors that cannot occur. For example, in analyzing transitions for completeness that involve enumerated types, the analysis output would report that no transition out of a certain state is satisfied when:

- $(T \neq Low) \wedge (T \neq Norm) \wedge (T \neq High)$,
- $(T = Low) \wedge (T = Norm) \wedge (T \neq High)$,
- $(T = Low) \wedge (T \neq Norm) \wedge (T = High)$,
- $(T \neq Low) \wedge (T = Norm) \wedge (T = High)$, and when
- $(T = Low) \wedge (T = Norm) \wedge (T = High)$,

Clearly, these situations are impossible for enumerated types and should not be reported as errors. In addition, the analysis method Heimdahl used is unable to reason about the relationships between the predicates making up the conditions (recall that the predicates in the conditions are the objects ultimately being ANDed and ORed together). Heimdahl used a data structure known as a binary decision diagram (BDD) to represent and manipulate the guarding conditions. BDDs represent expressions symbolically, thus, the semantics of the expressions are ultimately lost. To clarify the problem and show how and why a symbolic representation may lead to spurious errors, we now explain BDDs in detail.

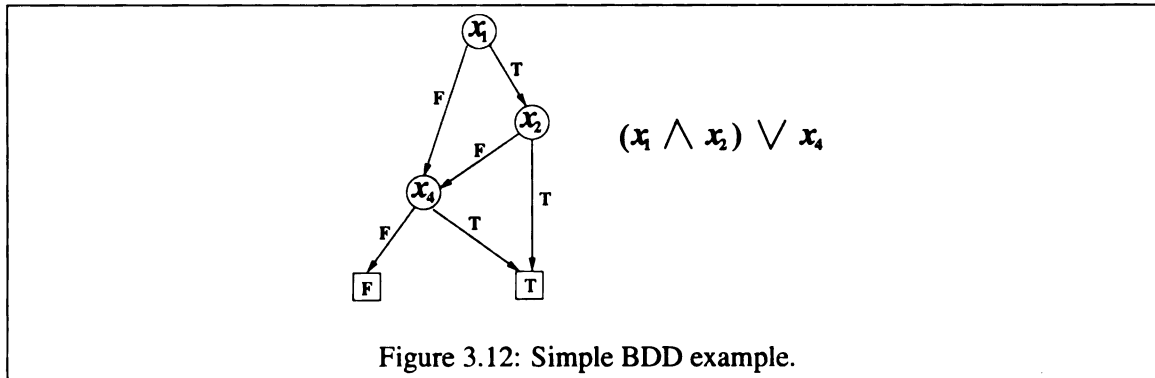
3.1.3.1 Binary Decision Diagrams

Bryant [8] proposed a method that provides a *canonical* (unique) representation for Boolean functions. The representation uses *binary decision diagrams* (BDDs). BDDs are directed acyclic graphs (DAGs). In addition to providing a canonical representation for Boolean functions, algorithms exist for efficiently manipulating BDDs [8, 42]; for example, testing two BDDs for equality, ANDing and ORing BDDs, and negating a BDD.

Representing Boolean formulas as reduced directed acyclic graphs with restrictions on the ordering of decision variables in the vertices results in a canonical form (every formula has a unique representation). Testing for equivalence then becomes the task of simply testing to see if two graphs match exactly, and testing for satisfiability requires a simple comparison of the graph with the constant function F (FALSE); if the graph reduces to FALSE it is unsatisfiable. Similarly, to check for a tautology, merely requires a simple comparison of the graph with the constant function T (TRUE); if the graph reduces to TRUE the formula it represents is a tautology.

Semi-formally, the set of vertices in a Binary Decision Diagram consists of non-terminal and terminal vertices. The non-terminal vertices are labeled with the variables of a Boolean formula $X = \{x_1, x_2, \dots, x_n\}$. Terminal vertices are labeled T or F ; true or false respectively. Each variable can be assigned the truth value T or F . The edges out of each vertex are labeled T or F representing the possible truth values that can be assigned to the variable labeling the vertex. If a truth assignment sets variable x_i to T , then at the vertex labeled x_i the edge labeled T is taken, otherwise the edge labeled F is taken. The value of a formula for a given truth assignment equals the value of the terminal node that is reached. For example, consider the binary decision diagram in Figure 3.12. The Boolean formula represented by this BDD is $(x_1 \wedge x_2) \vee x_4$.

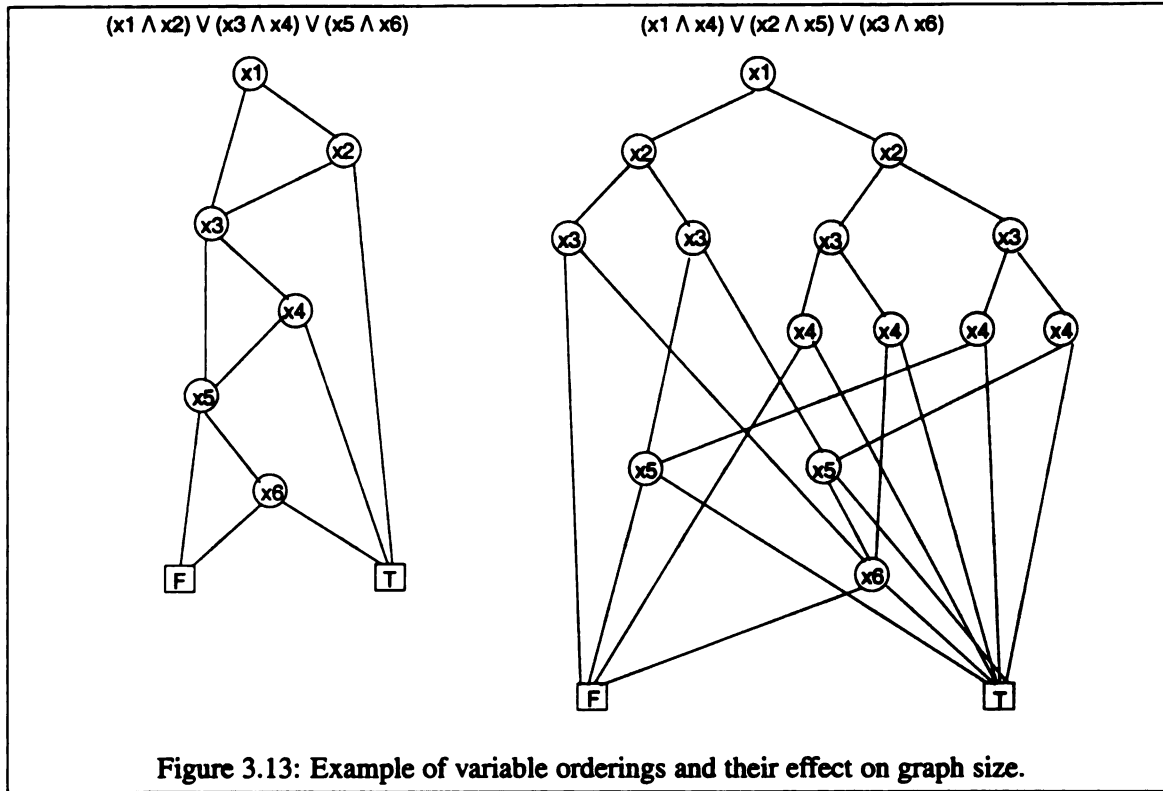
A canonical representation is ensured in an ordered BDD (OBDD) by imposing a strict total order on the occurrence of the variables as the graph is traversed from the root to the leaves. On



a path from the root to the leaves, for any two variables x_i and x_j , if $i < j$ then x_i must occur before x_j along any path. An ordering of variables in a system must be chosen and maintained for all functions that are to be represented in the system. The ordering of the variables can affect the number of nodes required to denote a given function [8]. For example, consider Figure 3.13; the permutation of arguments for the graph on the left results in only 8 nodes, while the same arguments permuted as on the right of the figure result in a graph with 16 nodes.

There are many different algorithms for attempting to find a variable ordering that yields the fewest number of total nodes [5, 8]. The reordering algorithms are based on heuristics. Two such reordering algorithms are the sift reordering method and the window reordering method [5]. Sift reordering moves each variable into different positions in the BDD in an attempt to find the encoding of variables that yields the fewest number of nodes. The window reordering method moves a group (or window) of variables around, maintaining the same order of the variables within the window. The goal of window reordering is to find a position for the groups of variables that yields the fewest number of total nodes. There are algorithms that will find the optimal ordering. Unfortunately, they all run in exponential time [8].

The major difficulty Heimdahl encountered in using BDDs is the inaccuracy of the output generated by ANDing and ORing BDDs together to test logical expressions for contradictions (mutual



exclusion) and tautologies. The difficulty arises for two reasons: the predicates are represented symbolically and a symbolic representation loses the semantics of the predicates contained in the logical expression; and information related to the structure of the state machines is not considered in the analysis process; i.e., the analyses are local rather than global, so information about the global state machine is absent from the analysis process. Incorporating all structural information into the analysis process would be difficult and time consuming and may result in state explosion, thus, rendering the analysis intractable. The latter problem and its solution will be discussed in more detail later in this document. The problem with a symbolic representation of complex predicates is discussed next.

3.1.3.2 Problems With Symbolic Representation of Predicates

A symbolic representation of complex predicates means that the semantic information about expressions involving the predicates is lost. No semantic consideration of the expressions precludes reasoning about the relationships between predicates. For example, Figure 3.14 shows a set of expressions assigned to the variables x_1 and x_2 , where S is an enumerated type that can be one of a or b (an enumerated type variable cannot be both values at the same time and the variable must have one of the values). To demonstrate how BDDs may report inaccurate results because the semantics of the expressions are lost, let x_1 and x_2 represent the expressions $S = a$ and $S = b$ respectively. Two possible truth assignments for x_1 and x_2 are $(x_1 = T) \wedge (x_2 = F)$ and $(x_1 = F) \wedge (x_2 = T)$. Let g be a function of x_1 and x_2 such that $g(x_1, x_2) = x_1 \wedge \neg x_2$. Let h be a function of x_1 and x_2 such that $h(x_1, x_2) = \neg x_1 \wedge x_2$. The functions g and h represent the two possible valid truth assignments for x_1 and x_2 described above, and they completely specify the enumerated type S . The BDD representation of each of these functions is shown in the figure. Let f be a function of g and h such that $f(g, h) = g(x_1, x_2) \vee h(x_1, x_2)$. The BDD representation of $f(g, h)$ is also shown in the figure.

If the functions g and h completely specify the truth assignments x_1 and x_2 can have with respect to S , then the function $f(g, h)$ should be equivalent to TRUE, signifying that the Boolean expression represented by f forms a tautology (i.e., is all-inclusive). Since S can only have one of the values a or b and it must have exactly one of those values, the functions g and h do completely specify the truth assignments x_1 and x_2 can have with respect to S , and $f(g, h)$ should reduce to TRUE. However, as the graphical depiction of the function reveals, $f(g, h)$ does not reduce to TRUE and the two paths leading to F (FALSE) signify the truth assignments for x_1 and x_2 that need to be added to completely specify the possible truth assignments for x_1 and x_2 with respect to S ; i.e., these truth assignments represent spurious errors that would get reported as missing conditions.

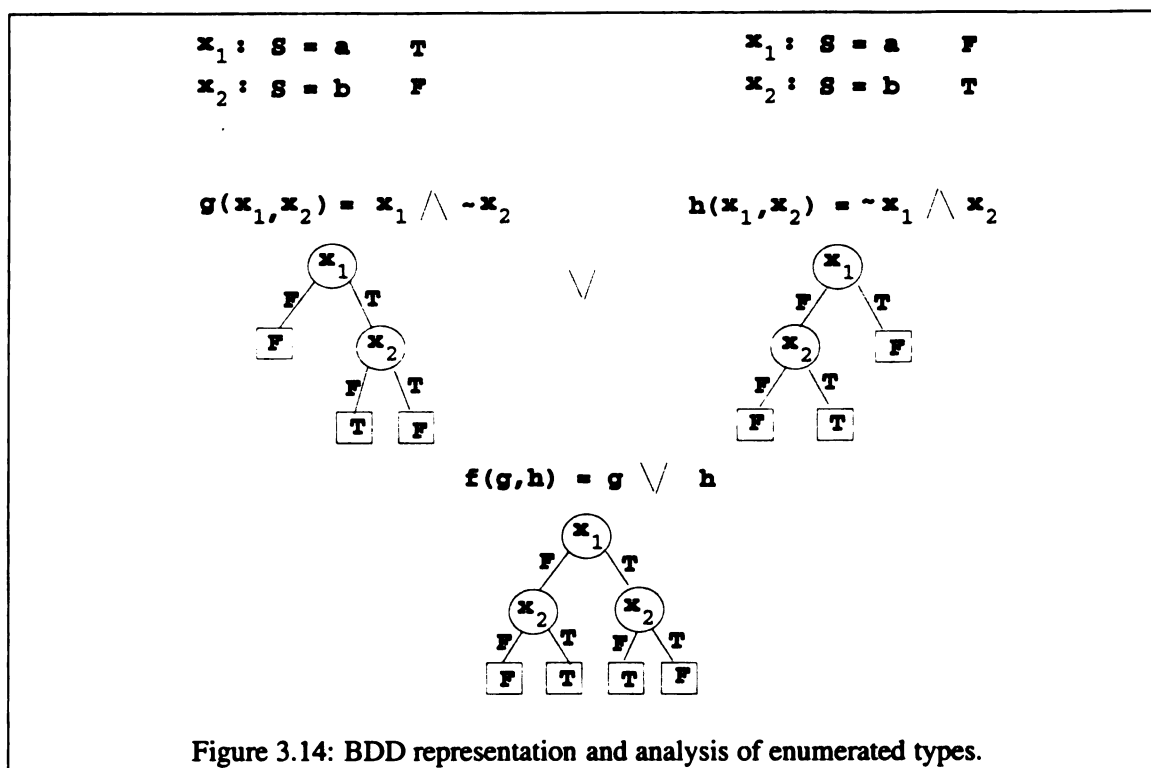


Figure 3.14: BDD representation and analysis of enumerated types.

To see how specifying the invalid truth assignments along with the valid truth assignments results in a BDD that reduces to T (TRUE), consider Figures 3.15 and 3.16. Figure 3.15 shows the Boolean expressions and BDDs representing the paths leading to F in the BDD of $f(g, h)$ in Figure 3.14. The function $i(x_1, x_2) = \neg x_1 \wedge \neg x_2$ (Figure 3.15) represents the leftmost path leading to F in the BDD representing $f(g, h)$. The function $j(x_1, x_2) = x_1 \wedge x_2$ (Figure 3.15) represents the rightmost path leading to F in the BDD representing $f(g, h)$. The BDD representing the function $f(i, j) = i(x_1, x_2) \vee j(x_1, x_2)$ shows the result of ORing the two invalid truth assignments for x_1 and x_2 with respect to S . The function $f'(g, h, i, j) = f(g, h) \vee f(i, j)$ shown in Figure 3.16 is the disjunction of all combinations of assignments that the enumerated type S can have, including the invalid assignments where S equals both a and b at the same time, and where S has no value at all (neither a nor b). As expected, the BDD representation of f' reduces to TRUE signifying that the Boolean expression represented by f' is a tautology (i.e., all combinations of

assignments to the enumerated variable S have been included). But the truth assignments given to x_1 and x_2 in the functions i and j are not valid truth assignments for the enumerated type S ; enumerated types are all-inclusive and mutually exclusive. Knowledge about the all-inclusive and mutually exclusive nature of enumerated types and post-processing of the results gleaned from the BDD representing $f(g, h)$ can be used to eliminate these falsely reported missing truth assignments that would otherwise be reported.

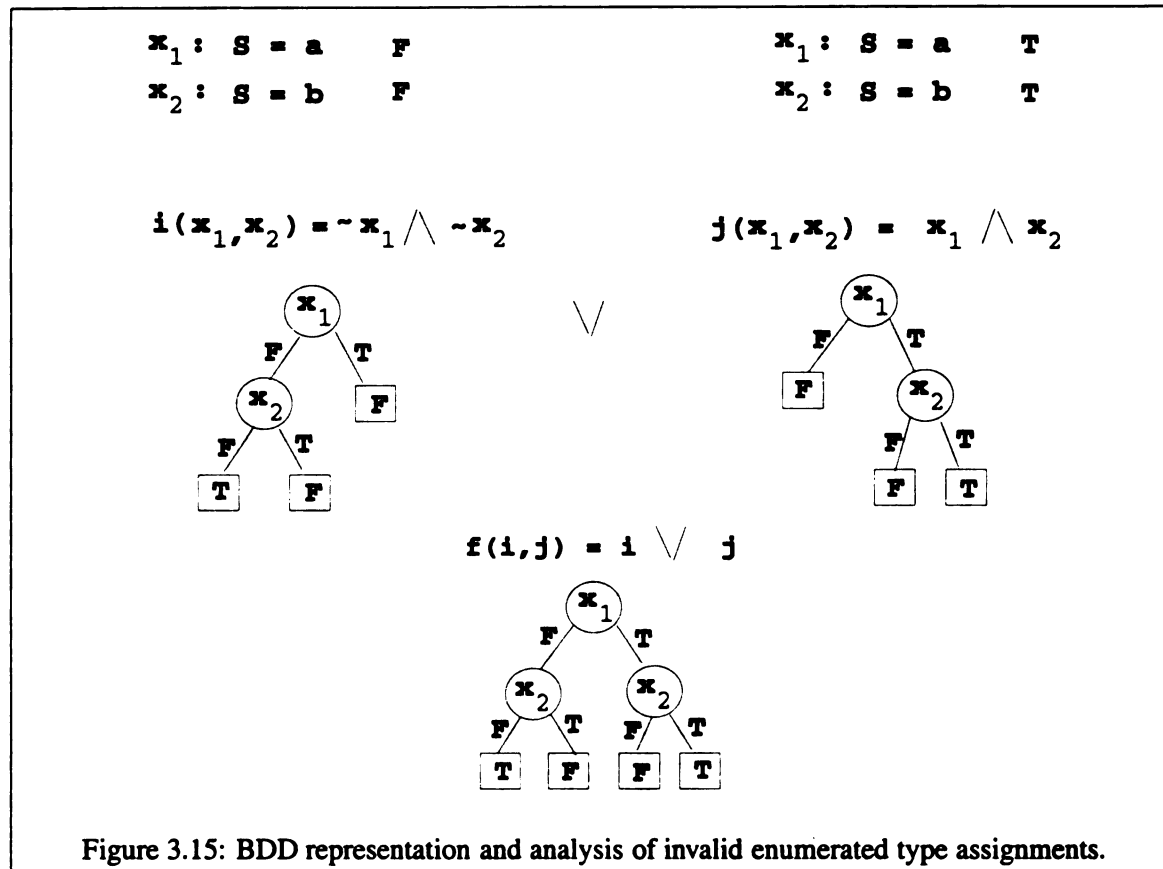
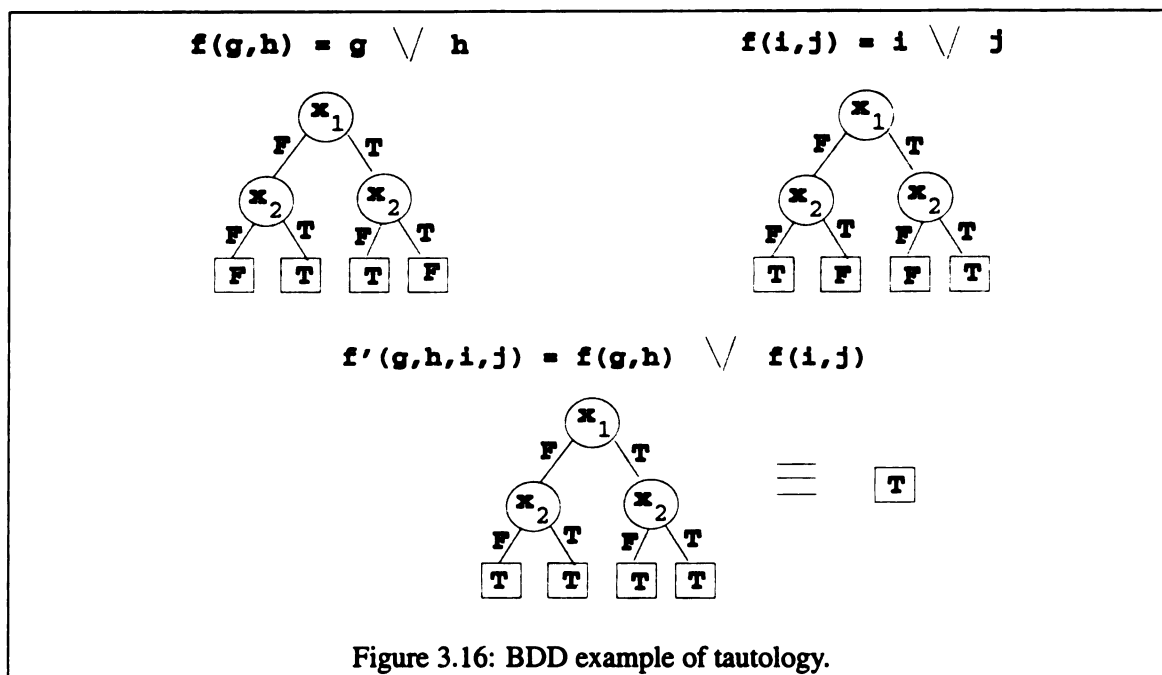


Figure 3.15: BDD representation and analysis of invalid enumerated type assignments.

Heimdahl relied on post-processing the results from the analysis program before presenting them to the user, to alleviate the spurious error reports related to enumerated types. Heimdahl augmented his BDDs with pointers to the actual constituent components of the Boolean expressions. This allowed him to perform post-processing on the BDDs generated from the conjunction and disjunction of expressions. Paths containing invalid truth assignments for enumerated types were



not included in the analysis reports. The post-processing method involves examining the actual expressions at each node as the augmented BDD is traversed, and keeping track of the values being assigned to the enumerated types. If on a single path during the traversal, two elements of an enumerated type are TRUE, or if all values of an enumerated type are FALSE, that path is not checked further and the conditions represented by it and the sub paths generated from it are ignored. This same general form of post-processing can be used to eliminate paths containing simple equality and inequality contradictions such as $(x > 10) \wedge (x < 10)$ or $(x = 10) \wedge (x = 5)$.

There are other types of expressions that will lead to spurious error reports when they are represented symbolically but for which the post-processing method discussed above is not useful. Thus, the inability of the analysis method to reason about the underlying relationships between predicates is a more troublesome matter. *Complex predicates* containing arithmetic expressions and mathematical functions are especially troublesome to deal with. Consider the following three predicates:

$x_1 : a > 10$	T	$x_4 : f(y) > z$	T
$x_2 : b > 0.55$	T	$x_5 : g(y) < z$	T
$x_3 : \text{abs}(a*b) \leq 0.00278$	T	$x_6 : g(y) > f(y)$	T
(a)		(b)	

Figure 3.17: Symbolic representation of non-trivial arithmetic expressions and mathematical functions, and spurious error reports.

$$a > 10 \quad (3.1)$$

$$b > 0.55 \quad (3.2)$$

$$\text{abs}(a * b) \leq 0.00278 \quad (3.3)$$

These predicates are represented symbolically in Figure 3.17(a) with the Boolean variables x_1 , x_2 , and x_3 . In analyzing transitions out of a state for completeness where the guarding conditions on those transitions involve these complex predicates, the following spurious error is reported as a condition that was not specified in the requirements specification and that needs to be specified:

$$(a > 10) \wedge (b > 0.55) \wedge [\text{abs}(a * b) \leq 0.00278] \quad (3.4)$$

When the semantics of the expressions represented symbolically by x_1 , x_2 , and x_3 are considered, it is clear that all three terms cannot be true at the same time; the lower bound on $\text{abs}(a * b)$ is 5.5 which is never less than or equal to 0.00278. This obvious contradiction cannot be detected with Heimdahl's augmentation and is reported as a missing condition; i.e., a condition that has not been considered and that needs to be added as a condition in one of the tables.

As another example, assume that the following predicates appear in the guarding conditions of transitions out of a particular state:

$$f(y) > z \quad (3.5)$$

$$g(y) < z \quad (3.6)$$

$$g(y) > f(y) \quad (3.7)$$

These predicates are represented symbolically in Figure 3.17(b) with the Boolean variables x_4 , x_5 ,

and x_6 . The current analysis method would report that there is no transition out of the state when

$$(f(y) > z) \wedge (g(y) < z) \wedge (g(y) > f(y)) \quad (3.8)$$

Again, when the semantics of x_4 , x_5 , and x_6 are considered, it is obvious that the conjunction of these predicates is a contradiction and was not included in any guarding conditions since it cannot occur in the first place; the function $g(y)$ can never be greater than the function $f(y)$. But analysis using BDDs precludes reasoning about mathematical relationships between predicates, and therefore, precludes detection of such contradictions. Unfortunately, for non-trivial arithmetic expressions and mathematical functions, post-processing using the method discussed above will not work.

Non-trivial mathematical expressions in the predicates pose one challenge to developing an analysis procedure that will generate as few spurious errors as possible, operate in an efficient manner, and be amenable to automation. Nonlinear multiplication, such as in the example above (Equation 3.4), is especially difficult to manage since arithmetic with nonlinear multiplication is undecidable [51] and so is generally impossible for automated procedures to manage.

3.1.4 Summary

Both Heitmeyer's and Heimdahl's analyses are purported to be efficient in terms of speed. In terms of accuracy, both methods suffer from spurious error reports. *Efficiency* and *accuracy* in the context of this dissertation are defined in terms of speed and in terms of the number of true errors and the number of spurious errors reported. An analysis process is efficient if it is fast enough to be used on a day-to-day basis; i.e., the analysis completes in seconds or minutes, rather than hours or days. An analysis process is accurate if it reports all true errors and as few spurious errors as possible. The smaller the ratio of spurious errors to true errors, the more accurate the analysis report is. Obviously, some tradeoffs are required between efficiency and accuracy. One difficulty in achieving more accurate analysis output depends on the complexity of the predicates used to

represent the transition conditions.

Examination of Heitmeyer's work [32] and the software requirements for the A-7E aircraft [33], and personal communication with a colleague of Heitmeyer, suggests that the conditions being analyzed in Heitmeyer's work are less complex than those being analyzed in Heimdahl's work. The reduced complexity is because requirements in SCR are specified at a higher level of abstraction than requirements in RSML. Heitmeyer et al. reported in [30, 32] that of 44 variables used in the condition tables, 33 were of type Boolean or were converted to Boolean manually; those that were converted to Boolean were either modes (states) or enumerated types with two values; the remaining 11 variables were either continuous or enumerated with more than two values and were used in relational expressions; the relational expressions were manually converted to Boolean variables also. The conditions Heimdahl analyzed, on the other hand, can be quite complex, consisting of arithmetic expressions and mathematical functions that cannot be converted to Boolean without losing important details. We know that arithmetic expressions and mathematical functions pose a significant challenge to the analysis process. In some system domains, such as process control systems, mathematical relationships are unavoidable and must be expressed in the requirements if the requirements are to provide a true representation of the system under development. Therefore, it is important to be able to both model and analyze these relationships.

Another source for spurious errors in both analysis methods is that the analysis methods used may abstract away many details related to the structure of the system as a whole; i.e., the analysis is local rather than global. This abstraction precludes spurious errors related to system structure from being eliminated. For example, an error may be reported stating that an inconsistency exists between two transitions, when examining the system reveals that in fact, both transitions cannot be satisfied at the same time (i.e., an expression in the first guarding condition may require that one subsystem be in a particular state, while the second guarding condition may require that another

subsystem be in a particular state, when the structure of the system is such that both conditions cannot be satisfied at the same time). Including more detail into the analysis process could help alleviate some of the problems with spurious errors. Model checking, discussed in the next section, does this to some extent.

3.2 Model Checking Software Requirements

Model checking has traditionally been applied to hardware verification. Recently, several groups have investigated applying model checking to software systems.

Atlee and Gannon have applied model checking to the requirements of a cruise control system and a water-level monitoring system and showed how model checking could be used to verify safety properties for event-driven systems [4]. In their work, they formalized SCR-style requirements [30, 32] and transformed the resulting formal specification into a state-based structure that a CTL model checker can analyze. Safety assertions are expressed as CTL formulas, and the model checker is used to determine if the formula holds in the model. If the formula held in the model, they could conclude that the safety property represented by the formula also held in the system requirements.

More recently, Sreemani and Atlee presented a case study of model checking the non-trivial A-7E requirements document for specific properties that the system requirements should satisfy [57]. The case study is intended to demonstrate the scalability of model checking software requirements. They implemented a program that translates an SCR requirements specification into an equivalent SMV (Symbolic Model Verifier) [42] specification, and use SMV to verify the required properties. The model is an abstraction of the original specification. The authors state that “the SCR methodology requires the requirements writer to create abstractions of the (potentially infinite) environmental state space by determining which predicates on values of environmental variables affect mode transitions”. The authors further state that the SCR methodology requirements that must be satisfied,

ultimately provide an easy method for obtaining abstractions; i.e., since the SCR methodology already requires abstractions, it is easy to determine additional abstractions for the SMV model. Most of the conditions are represented as Boolean variables, but if two or more conditions are related such that only one and exactly one condition can be true at all times, the conditions are represented as enumerated-type variables [57]. To limit the state space explosion problem, SMV uses Binary Decision Diagrams to represent Boolean functions, thus, SMV can check to see if a property holds in a set of states rather than enumerating the entire state space and checking to see if the property holds in every state. Recall (Section 3.1.3.1) that the size of a BDD is sensitive to the order of variables in the BDD. In Sreemani and Atlee's approach, the BDD is constructed so that variables in the BDD occur in the order in which they are declared. In one case, they were not able to reach a solution after executing for two weeks, but when they reordered the variables and reran the analysis, a solution was generated in about 15 seconds. They experimented with several different variable orderings. They manually change the order of the variables by rearranging the order of the variable declarations.

Wing and Vaziri-Farahani [60] have also investigated ways in which model checking can be applied to software systems. They apply a variety of abstraction techniques to a software system to create a finite state machine model of the system. The model and a specification expressed in CTL are input to a model checker, SMV, and the model checker determines if the specification is satisfied in the model. The abstractions are manually determined based on the formula being verified or by exploiting domain-specific or application-specific knowledge [57, 60]. Determining the abstractions manually, requires much human thinking and case-specific knowledge.

Anderson et al. have investigated the feasibility of model checking large software specifications [1]. They translated a portion of the TCAS II¹ (Traffic Alert and Collision Avoidance System

¹TCAS II is an airborne, collision-avoidance system required on all commercial aircraft capable of carrying 30 or more passengers through U.S. airspace.

II [40]) requirements specification into a form acceptable to a model checker and used the model checker to analyze for a number of dynamic properties of the system. Since completeness and consistency properties can be expressed as CTL [13] formulas, model checking can be used to check the translated specification for these properties. Model checking for transition completeness and consistency is not as efficient (in terms of speed) as the analysis method Heimdahl used, but the results may contain fewer spurious errors since their approach is able to handle addition and subtraction, whereas Heimdahl's symbolic approach is not.

Most recently, Bharadwaj and Heitmeyer [6] have integrated the Spin model checker [34] into the SCR toolset so users can establish logical properties of an SCR specification. To limit the state explosion, they verify abstractions of the original requirements. They describe two types of reductions (abstractions) that are derived using the formula to be verified and special attributes of SCR specifications. The first reduction involves eliminating irrelevant entities in the original model; irrelevant entities are entities that are not needed in the analysis since they do not appear in the formula being checked. The second reduction involves using more abstract representations of monitored variables. Essentially, the abstraction involves eliminating certain monitored variables from the identified entity set that are causing state explosion. The reductions are applied manually and the abstract model produced is then analyzed automatically by the SCR toolset using Spin. The authors state that the approach they describe can be used to verify properties of a complete SCR specification. The approach allows variables to range over arbitrary (finite) domains such as enumerated values and integer subranges. They present two theorems and a corollary that guarantee that an invariant that holds for the abstract model also holds for the full specification. Currently, the second reduction principle they apply is incomplete, since there are "extra" transitions in the abstract state machine that generate states that are not reachable in the original specification.

Since all of the methods described in this section rely on abstraction to generate a system model

that can be analyzed in a computationally tractable manner (i.e., avoid the state explosion problem), spurious errors may be reported that would be eliminated if certain abstractions were not made. Currently, it is left up to the analyst to determine which error reports represent true errors and which error reports represent spurious errors.

The problem of spurious errors is common to all the analysis methods described in this chapter. To summarize, spurious errors in the analysis output are conditions that are reported to the analyst as errors, but the reported conditions cannot actually be satisfied in the system being analyzed. In other words, there is some augmenting information that the analysis procedures need in order to determine that certain conditions or expressions are not true errors. As we showed in Section 3.1.3.2, if we augment the analysis process with the appropriate information, previously reported spurious errors can be eliminated. The latter statement is true for any analysis method that is reporting spurious errors, since spurious errors are reported because some relevant details have been abstracted out of the system model. If one augments the analysis process with the relevant information, the spurious errors are no longer reported.

In essence, in each spurious error report, there is some undetected contradiction that exists between the constituent components of the reported expression that actually precludes the reported error from being a true error. During the course of the research described in this dissertation, we identified several classes of undetected contradictions that lead to spurious errors, and classified the spurious errors according to the undetected contradictions that lead to them. The next section discusses the different types of spurious errors that can appear in the analysis output, the undetected contradictions that lead to the spurious errors, and the analysis techniques, if any, that can eliminate particular classes of spurious errors from the analysis output.

3.3 Spurious Errors Revisited

After numerous case studies and experiments, we identified four classes of spurious errors. We classified the spurious errors according to the undetected contradictions that cause them.

1. Spurious errors involving simple and obvious contradictions between two predicates such as enumerated type predicates and predicates involving simple arithmetic expressions such as the one shown in Figure 3.18.
2. Spurious errors involving three or more predicates containing related linear arithmetic expressions.
3. Spurious errors involving non-linear expressions.
4. Spurious errors related to the structure of the state machine, or spurious errors related to information about the environment in which the system will operate, that is not part of the specification.

```
Inhibited --> Inhibited conflicts with
Inhibited --> Not-Inhibited if

(Own_Tracked_Alt() - Ground_Level()) > 1200 : T ;
Own_Tracked_Alt() <= (1200 + Ground_Level()) : T ;

No transition out of Inhibited is satisfied if :

(Own_Tracked_Alt() - Ground_Level()) > 1200 : F ;
Own_Tracked_Alt() <= (1200 + Ground_Level()) : F ;
```

Figure 3.18: Spurious error reports generated for the state Inhibited.

Different analysis techniques are able to eliminate different classes of spurious errors, but no single analysis technique can eliminate all spurious errors. Spurious errors of the types involved in the first two classes listed above, can be eliminated by augmenting the analysis process with decision procedures. As we will show in Chapter 4, Section 4.2.4, the symbolic analysis component of our technique augmented with decision procedures for enumerated type predicates, can easily eliminate spurious errors involving enumerated type predicates.

Tools using reasoning components augmented with decision procedures, such as PVS, can easily eliminate all types of class 1 and class 2 spurious errors. As we mentioned in Chapter 2,

Section 2.2.3.4.1, PVS can also eliminate some spurious errors involving simple non-linear inequalities.

In general, no analysis technique can eliminate class 4 spurious errors, since this class of spurious errors results because information that is required by the analysis process has been abstracted from the model, or is related to some environmental constraints that were not modeled. To eliminate spurious errors of class 4 requires one to identify the relevant information that is missing from the analysis process and to augment the analysis process with this information.

3.4 Summary

In this chapter we discussed analysis of state-based requirements for completeness and consistency, and model checking software requirements for application dependent properties. Work in analyzing requirements for the application independent properties of completeness and consistency forms the foundation for the research described in this dissertation.

All of the analysis methods described in this chapter suffer from the same general problem – spurious errors resulting from a model that is an abstraction of the original requirements specification. Requirements specifications describe all acceptable system implementations. Existing verification methods do not scale well for such requirements, since the requirements are too detailed [6]. To avoid the state space explosion problem and make the desired analysis tractable, an abstraction of the original specification must be modeled for the analysis procedures to work with. Details contained in the original specification are abstracted away. Different analysis methods rely on different types of abstractions, but the lost details may include: (1) the semantics of individual predicates comprising the transition conditions, and the semantics of the relationships between the individual predicates comprising the transition conditions, and (2) information related to the structure of the original state machine used to represent a model of the system, or information related to the

environment that the system will operate in. These abstractions (and the underlying methods used for the analysis) make the analysis approach pessimistic (conservative); all true errors are reported, but spurious errors may be reported as well. The number of spurious errors reported can make it difficult for an analyst to find and correct the true errors in the specification.

We identified four classes of spurious errors during the course of our investigations. Different analysis techniques are able to eliminate different classes of spurious errors. For example, there are analysis methods, such as theorem proving (discussed in Chapter 2), that can be used in place of abstracting away the details described in (1) above. Analysis methods such as theorem proving can also be used in place of abstracting the details described in (2) above. However, in general, including all information contained in the original requirements specification in the analysis process will render the analysis intractable. In truth, many of the details are not required for the analysis to be performed with satisfactory results (i.e., so that the analyst is able to easily find and correct the true errors that show up in the final analysis report). Our investigations revealed that there are generally only a few details that have been abstracted away that are causing all or most of the spurious errors to be reported in the analysis output. The analysis methods described in this chapter and in the previous chapter, do not help the user identify the relevant information to include in the analysis process.

The next two chapters describe an iterative and integrative approach that does not lose the details related to (1) above, and that assists the analyst in identifying the missing details related to the structure of the state machine or its environment that are causing the majority of spurious errors being reported. Upon completion of the iterative and integrative analysis process, the analyst is presented with an error report that is easily managed in terms of allowing the analyst to locate and correct the true errors in the original specification.

Chapter 4

Analysis Method

Static analysis methods suffer from a common problem: spurious errors in the analysis output. Spurious errors are caused by a lack of information in the underlying analysis. The loss of information occurs when we create a model of the original specification that is suitable for analysis. An abstraction of the original specification must be created, since analysis most often becomes intractable if all details about a system and its environment are included in a model. Not all analysis methods are the same. Different levels of abstraction can be applied. We identified four classes of spurious errors and found that different analysis methods could deal effectively with different classes of spurious errors. We also found that some classes of spurious errors are difficult for any analysis method to deal with efficiently and automatically. In general, an analysis method that can check a more detailed model will generate more accurate analysis reports.

In the previous two chapters, we showed that both BDD analysis and PVS analysis work well for certain problems, but it is clear that neither method alone can always provide the analyst with accurate analysis output in a timely and automated manner. We define accurate analysis output as output in which the ratio of spurious errors to true errors is small. We want to eliminate as many spurious errors as possible from the analysis report, so that the analyst can easily find and correct

the true errors in a specification. Furthermore, we want the analysis process to be fast enough to be used on a day-to-day basis, and we want the process to be automated. In addition, we want the analysis process to be applicable to real-world problems and not limited to state-based requirements; in other words, the underlying method of checking state-based requirements for completeness and consistency can be generalized to checking any large logical expressions for tautologies and contradictions.

We saw that there is a trade-off between the speed with which the analysis completes and the accuracy of the analysis output. Generally, the faster the analysis, the less accurate the output. This is directly related to the level of abstraction used to represent the model. The less detail included in the model, the faster the analysis can be, but the less accurate. For example, symbolic analysis methods, such as those that rely on BDDs, are fast, but may generate many spurious errors. The accuracy of the analysis report also depends on the abstractions made in relation to the properties of interest to the analyzer. If the abstractions do not adversely effect the analyzers ability to identify and eliminate the contradictions leading to spurious errors, then the analysis output in relation to the properties of interest will be accurate. Of course, checking to see if the model satisfies additional properties may lead to spurious error reports.

In addition, there is a tradeoff between efficiency and accuracy, and automation. Generally, the less detail an analysis method requires, the faster it will complete, the less accurate the output will be, and the more automated the analysis method will be. The more detail an analysis method requires and/or uses, the slower it will complete, and the more manual intervention will be required; i.e., the less automated the method will be. Since no single analysis method used alone will allow us to satisfy all of our goals, we developed an iterative and integrative approach that combines the strengths of the individual analysis methods (graph-based symbolic analysis using BDDs, and theorem proving using PVS) and circumvents their weaknesses.

Note that the integrative approach in itself is not unique. There are several others who have also developed integrative analysis techniques. Most of these techniques have been applied in the area of hardware verification, or analysis of concurrent programs. Joyce and Seger [39] developed an integrated approach to formal hardware verification that combines BDD-based symbolic simulation techniques with interactive theorem proving. Young et al. [64, 66, 67], have integrated static concurrency analysis with symbolic execution to detect anomalous synchronization patterns in concurrent Ada programs. Havelund and Shankar [23] describe a series of protocol verification experiments that combine theorem proving and model checking. Chan et al. [11], discuss integrating a constraint solver with a BDD manipulator to handle nonlinear constraints. It is clear that an integrative approach to analysis is essential, since all individual analysis techniques suffer from limitations of one form or another. The integrative approaches represent attempts to capitalize on the strengths of the individual techniques and circumvent their weaknesses.

Our integrative analysis approach is based on trying the simple, straightforward methods first and, if these methods fail, applying the more complex and computationally expensive methods. The iterative analysis is based on identifying the information that was abstracted out of the model that is leading to spurious errors, feeding the information back into the model, and re-running the analysis. Since the spurious errors are the result of missing information (abstraction) in the model, identifying the missing information and adding it back into the analysis process should make the spurious errors disappear. In other words, if all of the missing information is identified, and the original guarding conditions are consistent (complete), then augmenting the analysis process with the missing information will yield the correct output of FALSE (TRUE), showing that the guarding conditions are consistent (complete).

Traditionally, the analyst must manually inspect the analysis report to locate any true errors that may exist. If the analysis output contains many spurious errors, the analysts task is difficult,

error-prone, and time consuming. Our method is unique in that it helps an analyst identify the missing information so that the spurious errors can be eliminated and the true errors can be more readily identified. In addition, the technique we use to identify the missing information is unique. In this chapter, we refer to the missing information as *augmenting information* since its inclusion into the analysis process augments the accuracy of the analysis output by eliminating spurious errors.

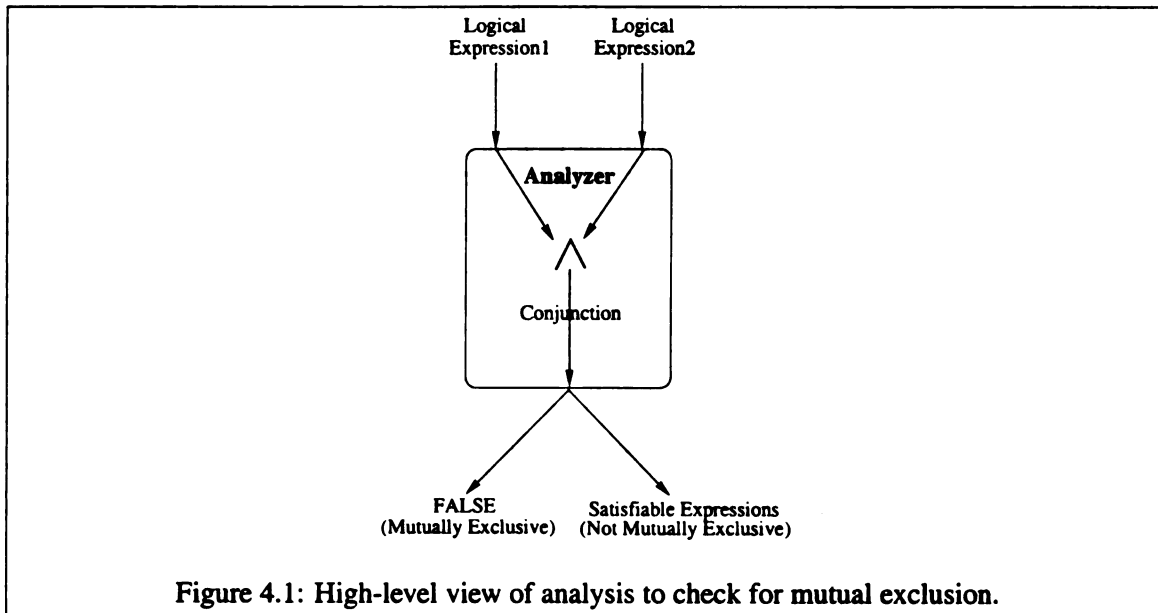
In this chapter we describe the method we developed to analyze disjunctive and conjunctive expressions to see if they are satisfiable or mutually exclusive. The method integrates symbolic manipulation techniques and theorem proving techniques: Binary Decision Diagrams (BDDs) and the Prototype Verification System (PVS). An application of the process developed in this research is in the analysis of state-based requirements for completeness and consistency. To demonstrate the scalability of our analysis process to real-world problems, we applied the technique to a large real-world avionics specification, specified in RSML, to check parts of the specification for completeness and consistency. The results of this application are reported in Chapter 6.

Chapter 4 is organized as follows: Section 4.1 provides a high-level description of the analysis process and describes the process using some typical scenarios we encountered in the analysis of state-based requirements for completeness and consistency. Section 4.2 describes the specific tools our technique uses, and Section 4.3 describes how to add augmenting information to the analysis process. An analyst ultimately initiates and guides the overall analysis process, therefore, Section 4.4 discusses the iteration and integration options available to the analyst during the analysis process.

4.1 General Analysis Process

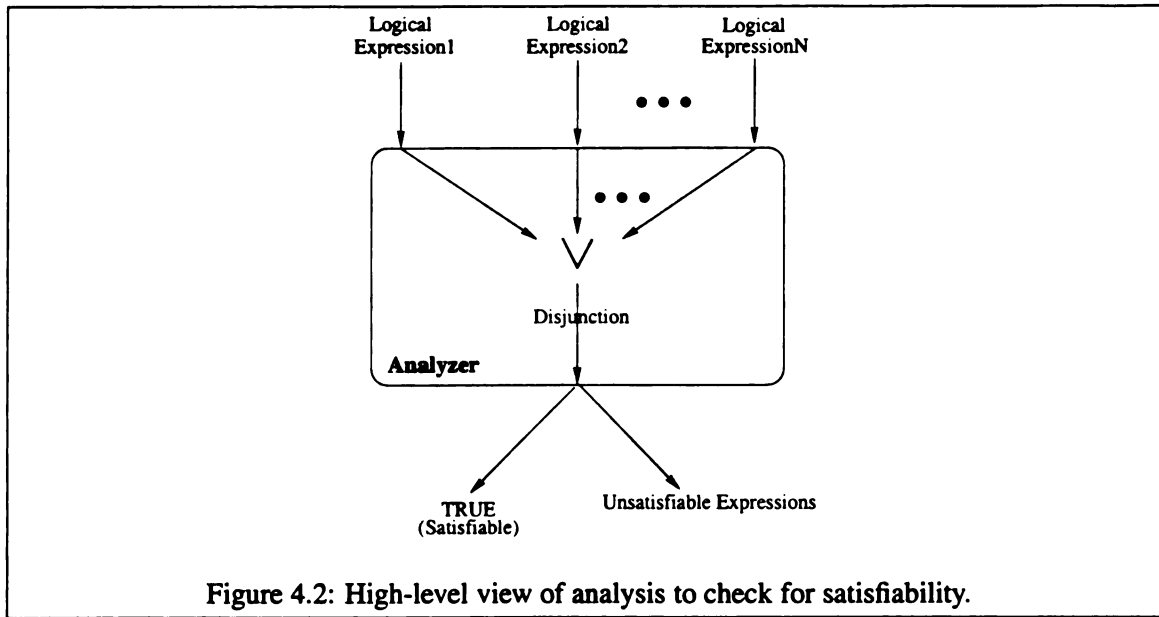
Figures 4.1 and 4.2 show a high-level view of the general analysis process. The inputs to the analysis process for mutual exclusion are two logical expressions. The analysis process forms the

conjunction of the two expressions. If the expressions are mutually exclusive the conjunction will reduce to FALSE. If the expressions are not mutually exclusive, i.e., the conjunction of the expressions does not reduce to FALSE, the analysis will report the logical expressions to the analyst that are satisfiable by both of the disjuncts at the same time (Figure 4.1). The analyst is then left with the task of determining how to modify the two original logical expressions so they are mutually exclusive.



The input to the analysis process for satisfiability may be one or more logical expressions (Figure 4.2). The analysis process forms the disjunction of the logical expressions. If the disjunction of the expressions is satisfiable, then the disjunction reduces to TRUE. If the disjunction of the expressions is not satisfiable, i.e., the disjunction does not reduce to TRUE, the analysis will report the logical expressions to the analyst that are not satisfiable by any of the original logical expressions. The analyst can then determine the logical expressions that need to be added to make the disjunction of the expressions a tautology.

The symbolic component of our method is fast and fully automated, but may generate many



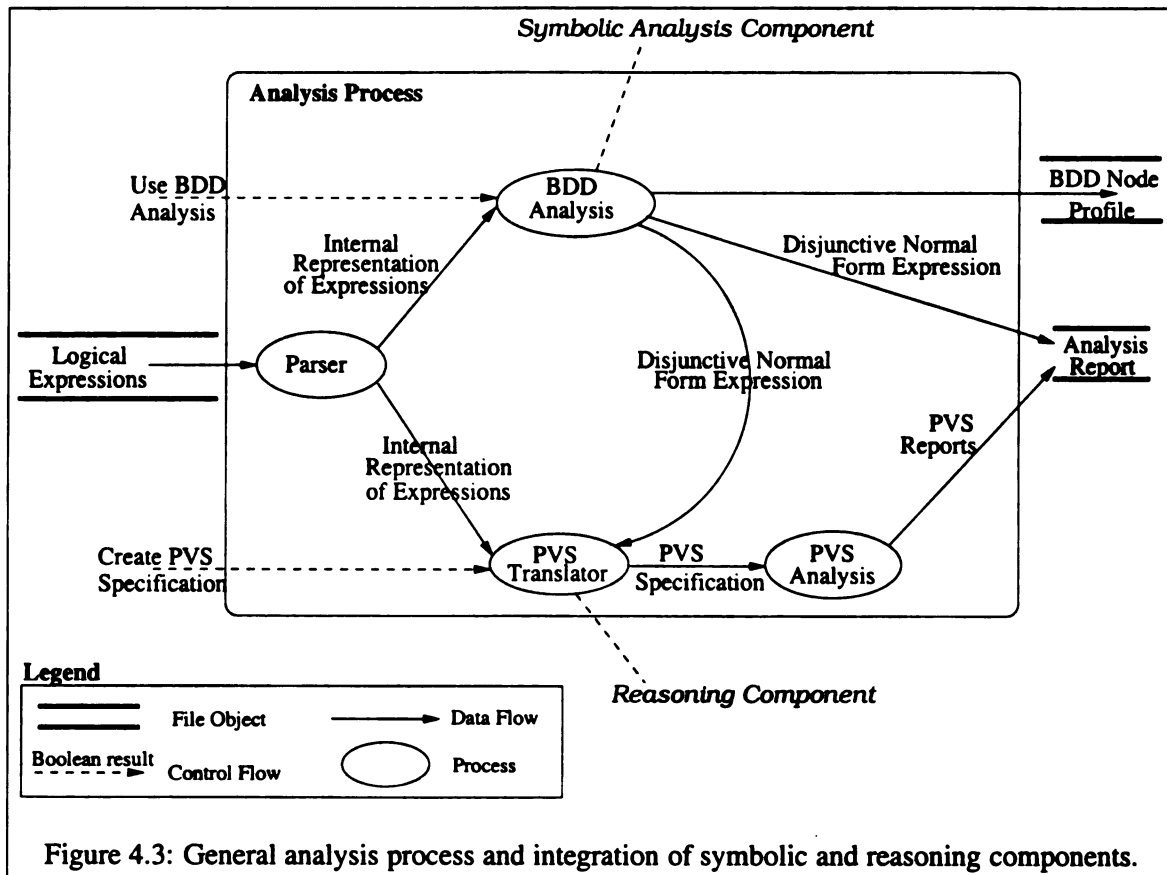
spurious errors. The reasoning component is slower and may require costly and time consuming manual intervention, but generates more accurate analysis reports. Our overall analysis process takes advantage of the strengths of the symbolic and reasoning components and circumvents their weaknesses.

The strengths of the symbolic component are its speed and automation. The symbolic component is simple and straightforward to use, and can deal effectively with some logical expressions. In particular, symbolic manipulation can deal with logical expressions where contradictions exist between structurally equivalent predicates, or enumerated types. The major weakness of the symbolic component is that it cannot reason about the relationships between predicates since the semantics of the predicates are lost. The symbolic components' inability to reason about relationships between predicates may lead to spurious errors (if there are many interdependencies between the expressions).

The strength of the reasoning component is its ability to reason about relationships between the predicates. But this strength means that more details are modeled. Modeling more details and

reasoning about the details tends to slow down the analysis process, and in some cases, the analysis may fail to halt (depending on the size and complexity of the expressions being analyzed). In addition, a reasoning technique generally requires more costly and time consuming user intervention than simpler techniques, thus, the reasoning component is less automated.

Figure 4.3 shows the general analysis process and the integration of the symbolic and reasoning components. The logical expressions to be analyzed are stored in some machine readable form. The



analysis process begins by first parsing the logical expressions and converting them into an internal representation. If the analyst chooses to perform symbolic analysis, then the tool performs symbolic analysis using BDDs and generates two outputs: a BDD node profile (described in Section 4.2.2), and an analysis report in disjunctive normal form in a tabular format. The latter output of the symbolic analysis component may be samples of the resulting disjunctive normal form expression;

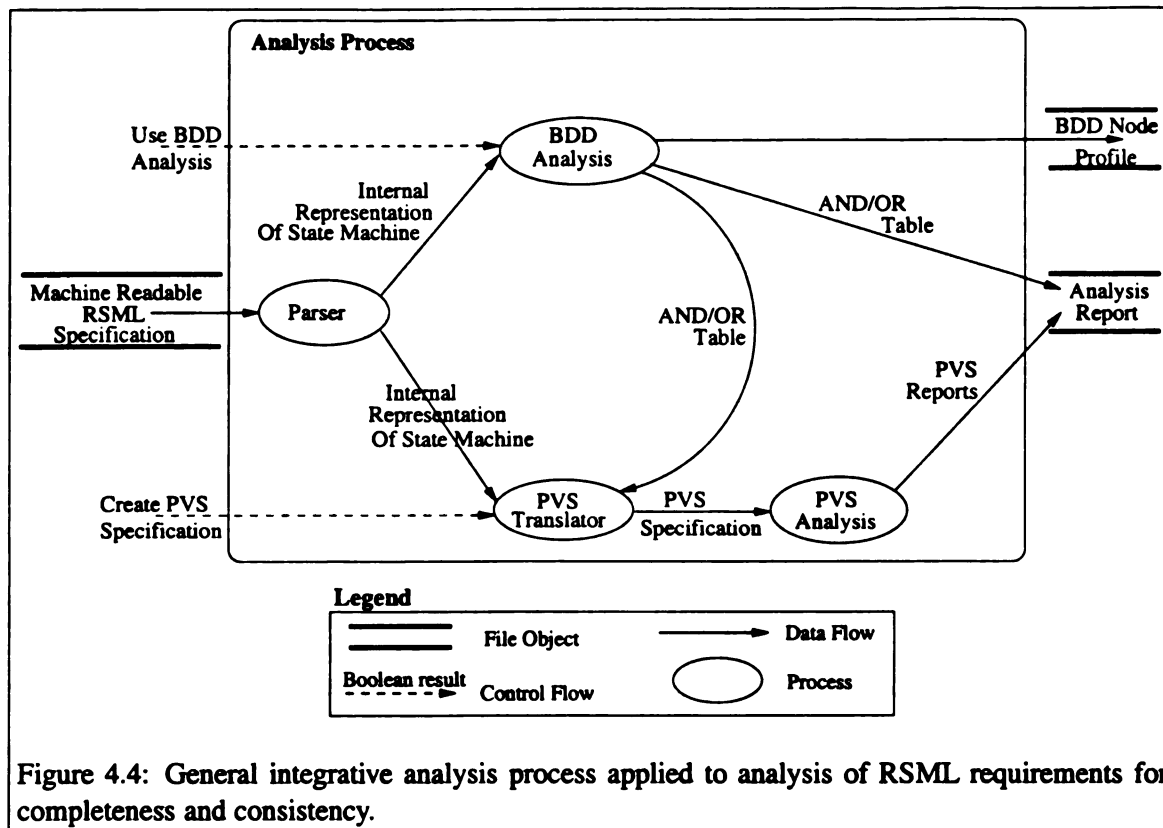
the analyst may specify samples if the resulting disjunctive expression is too large to report in its entirety.

If the analyst chooses to create a PVS specification, the tool converts the internal representation of the logical expressions into a PVS specification so the analyst can use PVS to analyze the expression. The output of the PVS analysis is either a finished proof (no unprovable subgoals), or a report of unprovable subgoals. The analyst can also choose to create a PVS specification for the disjunctive normal form expression output from the symbolic analysis component (this option is shown in the figure by the data flow from the *BDD analysis* process to the *PVS translator* process). The next section describes the general analysis process as applied to the analysis of software requirements for completeness and consistency.

4.1.1 Applying the Analysis Approach to Analysis of Software Requirements for Completeness and Consistency

The outcome of any analysis is an error report of some form that is presented to the analyst. Error reports from our analysis process are presented to the analyst as Boolean expressions in disjunctive normal form. In terms of analyzing requirements for completeness and consistency, each report represents either (1) an incompleteness, that is, a condition the requirements do not handle, or (2) an inconsistency, that is, a condition where two or more responses are specified. Spurious errors are manifested as conditions where contradictions between conjuncts in the disjunctive normal form expression comprising the error report, are not detected by the analysis tool; if the contradictions were detected, then the conditions would not be reported as errors in the analysis output.

Figure 4.4 shows the same general integrative analysis process as shown previously in Figure 4.3, but for the specific application of analyzing RSML specifications for completeness and consistency. A machine readable RSML specification is parsed and an internal representation of the



state machine is generated. For the purposes of this discussion we assume symbolic analysis is applied first. If the symbolic analysis reports that the analyzed expressions are complete (consistent), then we are done. If the symbolic analysis reports a reasonable number of errors (i.e., a number of errors that an analyst can manually inspect without too much trouble), the analyst manually inspects the report to look for the true errors. If the number of errors reported is unmanageable, we can try the reasoning component on the original expressions. If the reasoning component reports that the expressions are complete (consistent), then we are done. If the reasoning component generates a manageable number of error reports, the analyst can review them manually. If the reasoning component generates an unmanageable number of error reports, or fails to halt, we assume that some important details needed during the analysis process were abstracted away and that a majority of the reported errors are spurious. We can then apply the method described in Chapter 5 to identify the augmenting information. In general, the tools can be applied in any order the analyst determines

best fit the particular problem at hand.

To give the reader an idea of how some typical analyses may proceed, and some of the decisions that may need to be made during the analyses, we present several possible usage scenarios. These scenarios represent typical situations we encountered during our experiments.

1. The ideal case is when the analyst applies symbolic analysis to a logical expression and the result shows the logical expression complete or consistent. In this case, the analyst is done, and the analysis process was fully automated and fast (generally, on the order of seconds or minutes). No spurious errors were generated.
2. The analyst applies symbolic analysis to a logical expression and the results show under 75 errors. It took only seconds to generate the analysis report, and the process was fully automated. From here, there are two choices:
 - (a) The analyst manually inspects the analysis report and finds five true errors with the specification. The other 70 errors are spurious errors. It may take the analyst a few hours to identify the true errors mixed in with the spurious errors; every error reported must be manually inspected, since all 75 may be true errors.
 - (b) The analyst automatically generates a specification of the symbolic analysis output suitable for analysis by the reasoning component, and the 75 error reports are analyzed by the reasoning component. The reasoning component yields five error reports. The analyst inspects the error reports and finds they represent true errors in the specification. It takes the analyst a few minutes to automatically convert the symbolic analysis output to a specification that the reasoning component can analyze. It takes the reasoning component a few seconds to identify and report the five true errors. The analyst takes several minutes to determine that the five reported errors are true errors.
3. The analyst applies symbolic analysis to a logical expression and the results show over 1000 errors. There are too many error reports for the analyst to manually inspect, but not too many for PVS to handle. The analyst automatically generates a specification of the symbolic analysis output in a form that the reasoning component can analyze, and the error reports are analyzed by the reasoning component. The reasoning component runs for under 30 seconds and reports that the logical expressions are complete or consistent. The symbolic analysis took under a minute to complete, and it took only a few minutes for the analyst to automatically convert the symbolic output to pass on to the reasoning component.
4. The analyst applies symbolic analysis to a logical expression and the results show tens of thousands of errors. The symbolic component takes a couple of minutes to complete the analysis and report the results. The analyst automatically generates a specification of the original

logical expression that the reasoning component can analyze, and the reasoning component is invoked to analyze the logical expression. Two potential results are:

- (a) The reasoning component runs for a few minutes and reports that the logical expressions are complete or consistent.
 - (b) The reasoning component runs for a few minutes and reports thirty errors. The analyst manually inspects the errors and finds they all contain the same contradiction; this contradiction is one that the reasoning component could not detect because the information about the contradiction was missing from the reasoning component's specification. The analyst required only an hour to identify the contradiction and show that all reported errors are spurious.
5. The analyst applies symbolic analysis to a logical expression and the results show millions of errors (on the order of 10's or 100's of millions). The analysis takes several minutes to complete. The analyst automatically creates a specification of the original expression that the reasoning component can analyze and invokes the reasoning component on the specification. The reasoning component runs for hours without reporting any results and the analyst aborts the analysis. The analyst assumes that most of the reported errors are spurious errors that result because some information is missing from the analysis process. The analyst uses the method described in Chapter 5 to identify the information that is missing from the analysis process and that is leading to the spurious errors. Three possible scenarios are:
- (a) The analyst manually adds the identified information to the specification and reruns the symbolic analysis. The analysis runs for a few minutes and reports that the logical expressions are complete or consistent. The analysis is fully automated aside from manually adding the information to the specification.
 - (b) The analyst manually adds the identified information to the reasoning components specification and analyzes the logical expressions with the reasoning component. The analyst manually adds the information from the specification into the reasoning analysis (i.e., the proof process). The analysis finishes in under a minute and shows that the guarding conditions are complete or consistent.
 - (c) The analyst manually adds the identified information to the specification and reruns the symbolic analysis. The analysis runs for a few minutes and reports some errors. The analyst uses one of the above scenarios (1-4) to determine that the errors are spurious.
6. The analyst applies symbolic analysis to a logical expression and 10's or 100's of millions of errors are reported. The analyst automatically generates a specification for the reasoning component and analyzes the specification with the reasoning component. The reasoning component runs for hours and the analyst aborts the analysis. The analyst assumes, that most of the reported errors are spurious errors that result because some information is missing from

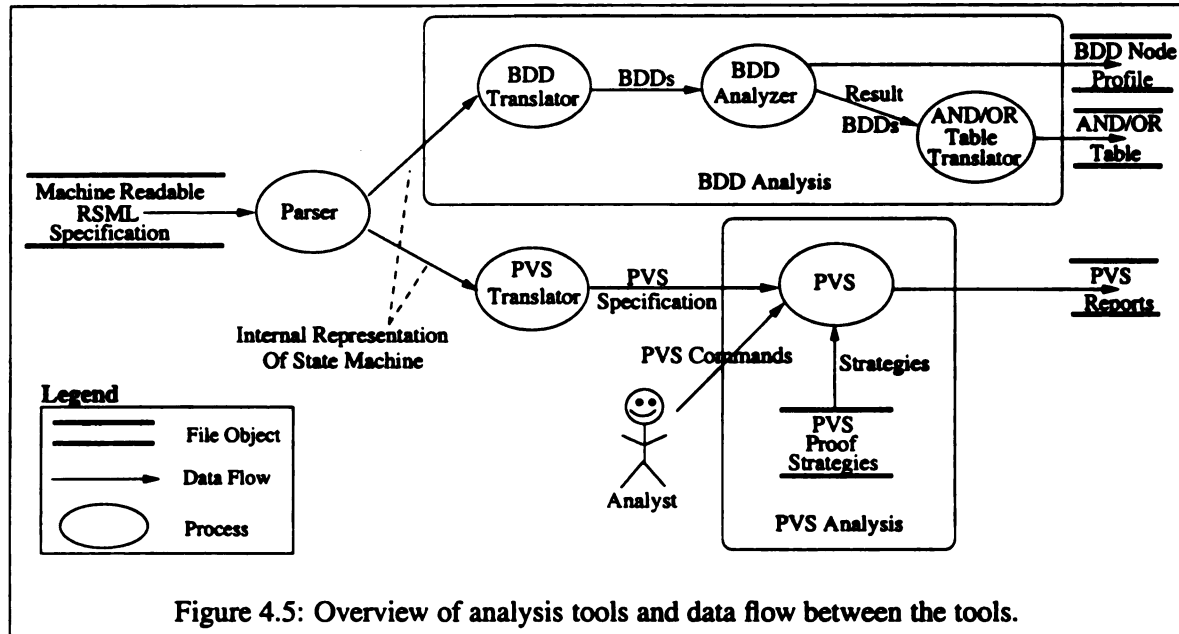
the analysis process. The analyst uses the method described in Chapter 5 to identify the information that is missing from the analysis process and that is leading to the spurious errors. The analyst adds the missing information to the symbolic analysis and reruns the analysis. The analysis reports under ten million errors. Two possible scenarios from this point are:

- (a) The analyst looks for additional information that is missing from the analysis, identifies the missing information, adds it to the reasoning component specification, reruns the reasoning analysis, and the analysis reports the expressions are complete or consistent.
 - (b) The number of spurious errors is now less than the number of true errors and the true errors occur with many permutations of sub-expressions. Symbolic analysis reports all of the permutations. The analyst adds the missing information to the reasoning component specification, reruns the reasoning analysis, and the analysis reports 500 errors. The analyst cannot find any additional missing information, so manually inspects the reported errors, making the assumption that they are true errors. The analyst spends several hours and finds that the errors are true errors and that the number of them is large due to the possible permutations of several groups of sub-expressions.
7. The analyst applies symbolic analysis to a logical expression and millions of errors are reported. The analyst assumes, that most of the reported errors are spurious errors that result because some information is missing from the analysis process. The analyst uses the method described in Chapter 5 to identify the information that is missing from the analysis process and that is leading to the spurious errors. The analyst adds the missing information to the symbolic analysis and reruns the analysis. The analysis reports that the logical expressions are complete or consistent.

The above scenarios represent typical situations we encountered during our experiments but do not cover all of the possible ways an analyst can apply our analysis process. They do, however, provide a good overview of how our analysis process may be applied and how the process may work. Our analysis process is driven by the analyst, and thus, application of our analysis process is highly flexible and can be customized by the analyst to deal with a wide variety of situations. Note that in the above scenario descriptions, we did not include specific details about the tools our process uses. In the next section we describe in detail the symbolic and reasoning components of our analysis process.

4.2 Tools

The specific tools we use in our analysis process and the flow of data between the tools are shown in Figure 4.5. The tools described in this section are specific to the application of our analysis process to the analysis of state-based requirements (namely, RSML requirements) for completeness and consistency.



The symbolic analysis component sits on top of a BDD library created by Long [5] at Carnegie Mellon University. The symbolic analysis component consists of three sub-processes: a *BDD translator* that translates the RSML AND/OR table representation of guarding conditions to Binary Decision Diagrams, a *BDD analyzer* that manipulates the BDD representation of the guarding conditions to check for tautologies and contradictions, and an *AND/OR table translator* that converts the result BDDs output from the BDD analyzer process to AND/OR table format to present to the analyst.

The *PVS translator* converts the internal representation of the state machine to a PVS specification. The analyst then initiates PVS to analyze the PVS specification. We developed a set of proof strategies that, in most cases, allow the analyst to perform proofs of completeness and consistency

with a single command.

In Section 4.2.1 we present an overview of the control flow and set-up required before analysis can begin. Section 4.2.2 discusses Long's BDD library and some of the features of the library. A brief description of the BDD translator is given in Section 4.2.3. We do not discuss the BDD translator in detail since it is fairly straightforward and the details are not important. In Section 4.2.4 we describe the AND/OR table translator. We discuss the PVS translator in Section 4.2.5. Chapter 2 contained a detailed discussion of PVS, so we will not discuss PVS further in this chapter. Section 4.2.6 describes the BDD analyzer, and in Section 4.2.7 we describe the PVS analysis process.

4.2.1 Control Overview

The tool provides several command line options that allow the analyst to determine the direction the analysis should take. When the analyst initiates the analysis process, the specified command line options guide the flow of the analysis. Regardless of the options specified, the tool does some initial set-up to prepare for the requested analysis.

When the analysis program is initiated, the following set-up occurs: for each atomic state¹ in the RSML specification, the transitions out of the atomic state are grouped together according to the events that trigger the transitions (we call this group a *trigger-transitions set*). The transitions associated with each event include all transitions (if any) out of the super-states of the atomic state triggered by the same event. For example, Figure 4.6 shows an example of a state diagram with super-states $S1$, $S1A$, $S1B$, $S2$, and $S3$, atomic states $A1$, $A2$, and so on to $A7$, and transition triggers x , y , and z . Only the triggers on transitions out of the atomic states $A1$, $A6$, and $A7$ are shown in the figure. The trigger transitions set that would be constructed from the information in the state diagram is: for atomic state $A1$ and trigger z , transitions $A1$ to $A3$, and $A1$ to $S1$;

¹An atomic state is a state with no sub-states.

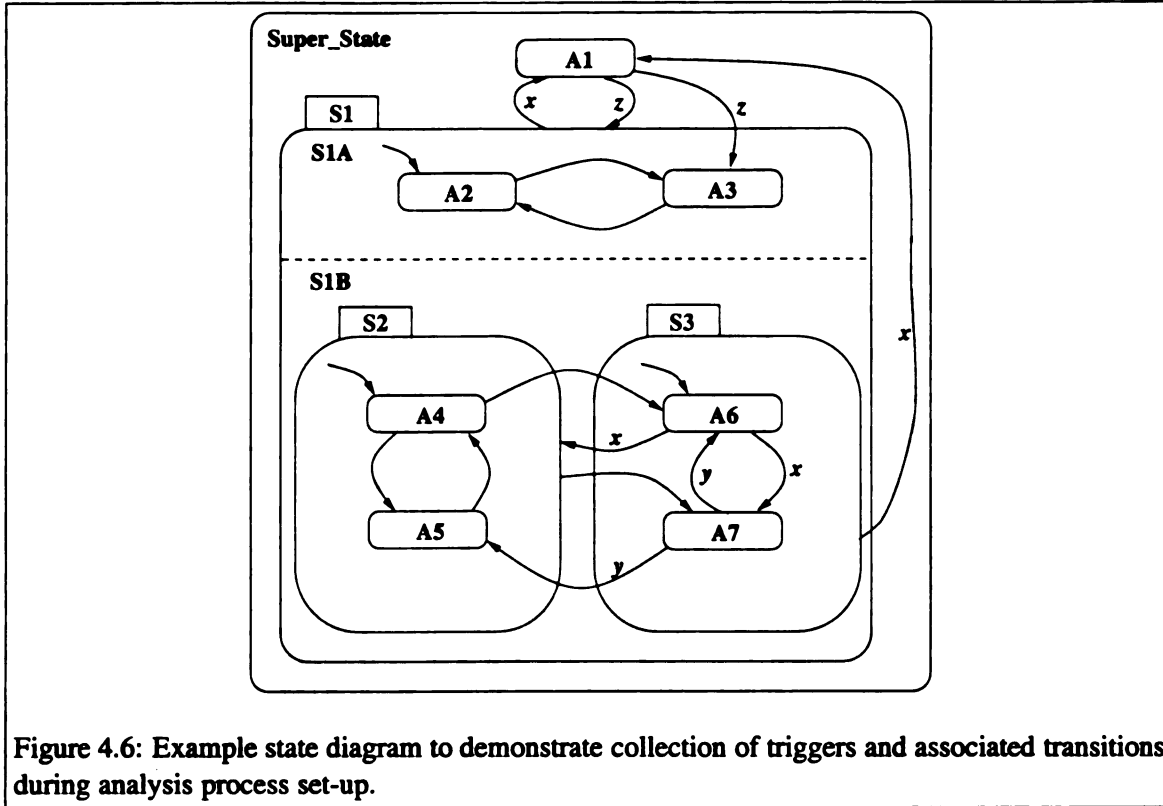


Figure 4.6: Example state diagram to demonstrate collection of triggers and associated transitions during analysis process set-up.

for atomic state *A6* and trigger *x*, transitions *A6* to *A7*, *A6* to *S2*, *S1* to *A1*, and *S3* to *A1*; for atomic state *A7* and trigger *x*, transitions *S1* to *A1*, and *S3* to *A1*; for atomic state *A7* and trigger *y*, transitions *A7* to *A5*, and *A7* to *A6*. Each trigger-transitions set is marked with the number of transitions that are included in the set. If a particular event out of an atomic state (and its super-states) has multiple transitions associated with it, we can perform completeness and consistency analysis on the associated transitions. If a particular event out of an atomic state (or its super-states) has only one transition associated with it, no consistency checks are required, but we can perform completeness checks on the single transition. Section 4.2.6 discusses the details of the completeness and consistency analysis.

Alt_Reporting IN_STATE Yes:	1
Other_Alt_Reporting = cTrue:	1
Other_Air_Status IN_STATE State_Airborne:	1
Auto_SL IN_STATE ASL_2:	1
RA_Inhibit_From_Ground = cTrue:	1
Mode_Selector = TA_Only:	1
Own_Tracked_Alt() > Other_Tracked_Alt():	1
Own_Tracked_Alt() < Other_Tracked_Alt():	2 #
Inhibit_Biased_Climb() > Down_Separation():	2 #
(Own_Tracked_Alt() - Other_Tracked_Alt()) >= MINSEP:	2 #
Up_Separation() >= ALIM():	2 #
(Own_Tracked_Alt() - Other_Tracked_Alt()) <= n_MINSEP:	2 #
Down_Separation() >= ALIM():	2 #
Current_Vertical_Separation() > LOWFIRMZ:	2 #
Current_Vertical_Separation() < ZTHRTA():	4 ##
ADOT() >= n_ZDTHRTA:	4 ##
-((Current_Vertical_Separation() / ADOT())) < TVTHRTATBL():	4 ##
Modified_Tau_Capped() < TFRTHR():	8 ####
Level_Wait IN_STATE S3:	4 ##
Up_Separation() <= SENSEFIRM():	4 ##
Down_Separation() <= SENSEFIRM():	4 ##
TCAS_TCAS_VMD() >= Zero:	4 ##
Other_Tracked_Relative_Alt() < n_MINSEP:	4 ##
TCAS_TCAS_VMD() <= Zero:	4 ##
Other_Tracked_Relative_Alt() > MINSEP:	4 ##
Other_Tracked_Range_Rate() > Zero:	10 #####
Threat_Alt_VMD() < ZT():	10 #####
leaf:	1
Total: 390	

Figure 4.7: Example BDD node profile portion.

4.2.2 BDD Library

Long's BDD package [5] includes a set of routines for creating, manipulating, printing, and destroying BDDs. There are also routines for printing status information about BDDs and for printing a histogram showing the number of nodes at each level. The histogram is called a BDD node profile; the node profile's importance to the analysis process is described in Chapter 5. Figure 4.7 shows an example of a BDD node profile. The # signs after the nodes represent a histogram signifying the number of nodes at each level in relation to the other nodes in the BDD. Long's BDD library also includes provisions for dynamically reordering the variables in a graph; dynamic reordering is discussed in Section 4.2.2.1, and its importance to the analysis process is described in Chapter 5.

4.2.2.1 Variable Reordering

Here, we provide only a high-level description of dynamic variable reordering methods. All of the material contained in this section comes from [5] and from personal communication with David Long. Long's BDD package includes two variable reordering algorithms: `bdd_reorder_stable_window3` and `bdd_reorder_sift`.

In general, the stable window methods take small groups of consecutive variables and try all permutations of the variables within each group. The process is repeated throughout the order until no more reduction is possible; i.e., the nodes within each group are permuted until no more reduction in the total number of nodes occurs. The stable window 3 method permutes the variables within windows of three adjacent variables to minimize the overall BDD size [5]. The method is not guaranteed to find an optimal ordering.

The sifting method picks up each variable in turn, moves it throughout the ordering to see what the total BDD size is for each possible position, and then moves the variable back to the spot that gives the lowest total number of nodes. Sifting is not guaranteed to find an optimal ordering of variables either. Generally, sifting achieves greater size reductions than the window-based methods, but is slower. For the stable window method to be most effective, one should know which variables should be grouped together, and these variables should be created in the appropriate order so they are adjacent to each other in the ordering within a window of three variables. It is not generally easy to automatically determine which variables are best grouped together. The analysis results described in this dissertation relied on the sift reordering method, but the analyst can specify the stable window 3 method if they desire.

4.2.3 BDD Translator

The guarding conditions are represented in disjunctive normal form in the RSML specification.

When the analyst invokes symbolic analysis using BDDs, the following actions are performed:

1. Initialize the requested command line options.
2. For each predicate in the RSML specification, create a BDD variable and create a link between the predicate and the BDD variable.
3. If dynamic variable reordering is requested, set the dynamic variable reordering routine to the requested reordering routine (stable window 3 or sifting).

Initially, the BDD translator creates only a BDD variable for each predicate in existence. During the BDD analysis process, the BDD analyzer works with BDD representations of the AND/OR tables. When the BDD analyzer attempts to invoke a BDD representation of an AND/OR table that does not exist, the BDD translator builds the BDD representation of the AND/OR table, stores it for later use, and returns it to the BDD analyzer. If dynamic variable reordering is enabled, the BDD translator will invoke the reordering algorithm to reorder the guarding condition BDD after building it, and prior to storing it and returning it to the BDD analyzer. When the BDD analyzer invokes an existing BDD representation of an AND/OR table, the existing BDD representation is simply returned to the analyzer.

4.2.4 AND/OR Table Translator

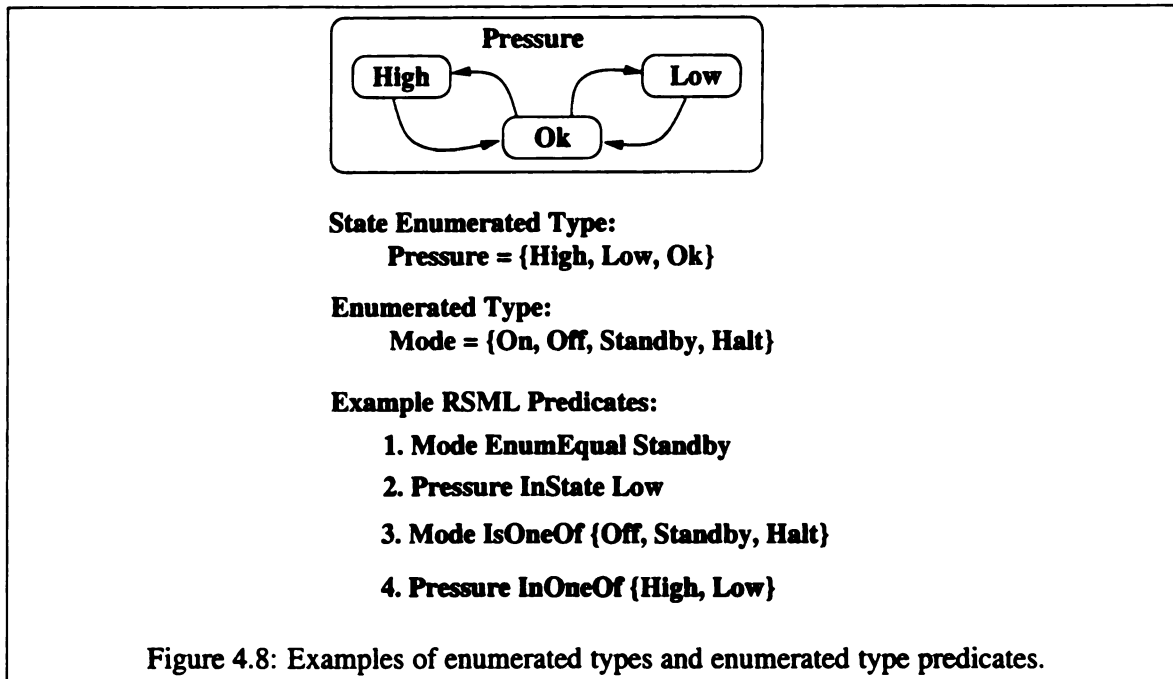
The AND/OR table translator receives the result BDD that is output from the BDD analyzer and converts the BDD into AND/OR table format to present to the analyst. The translator works by traversing the BDD and looking for satisfying paths. Each satisfying path in the result BDD represents a potential incompleteness (inconsistency) and is converted to a column in the AND/OR table. Recall that there may be some contradictions between the conjuncts comprising the satisfying paths that preclude them from being satisfiable, but the symbolic analysis without some additional capabilities cannot detect and eliminate paths containing certain contradictory predicates. Thus, each satisfiable

path containing an undetected contradiction becomes a column in the AND/OR table, and represents a spurious error reported to the analyst. Since our goal is to eliminate as many spurious error reports as possible, we augmented our AND/OR table translation process with decision procedures to detect certain contradictions so they would not be reported in the analysis report.

Decision procedures are algorithms designed to reason about, and possibly simplify expressions. Decision procedures can augment weak analysis processes and make them stronger. For example, we know that symbolic analysis using BDDs cannot reason about the components of the expressions and therefore cannot make decisions about whether or not two or more interdependent expressions contradict each other; this is because the semantics of the expressions are lost in the symbolic representation. This inability to reason about relationships between expressions may lead to many spurious error reports in the analysis output. Adding procedures to the symbolic analysis process that can perform some simple reasoning about the interdependencies between expressions will strengthen the overall analysis process and result in fewer numbers of spurious error reports.

We added some simple decision procedures to check for contradictions between enumerated types and state enumerated types (a super-state with sub-states can be thought of as an enumerated type by making the sub-states of the state the elements of the type). The term *enumerated type predicate* shall, henceforth, refer to any of four types of RSML predicates: `EnumEqual`, `InState`, `InOneOf`, and `IsOneOf` predicates. The term *enumerated type* shall, henceforth, refer to either an enumerated type variable, or a state enumerated type.

Figure 4.8 shows examples of enumerated types and what the RSML predicates for the enumerated types look like. The enumerated type for the state `Pressure` has elements corresponding to the state's child states `High`, `Low`, and `Ok`. `Mode` is an enumerated type whose elements are `On`, `Off`, `Standby`, and `Halt`. The interpretation of example RSML predicate 1 is: the value of the enumerated type `Mode` is equal to `Standby`. The interpretation of predicate 2 is: `Pressure` is



in state Low. The interpretation of predicate 3 is: Mode is in either Off, Standby, or Halt. The interpretation of predicate 4 is: Pressure is either in state High or Low.

The analyst enables the decision procedures by setting the appropriate command line option when the analysis process is initiated. The decision procedures are invoked by the AND/OR table translator during the translation of the result BDD to AND/OR table format. Recall that each satisfying path in the result BDD represents a column in the AND/OR table. Whenever one of the enumerated type predicates is encountered as each path is traversed in the result BDD, the decision procedures are called to see if the current path is valid given the truth value of the newly encountered enumerated type predicate. The decision procedures keep track of which elements have been encountered, the number of times the element may be TRUE, and the number of times a particular state or enumerated variable has been encountered. This information is kept for each individual enumerated variable and for each individual super-state. A current path is reported valid by the decision procedures if only one element for each enumerated type may be TRUE at a given time. If

two or more elements of a given enumerated type are TRUE at the same time, or if all elements are FALSE at the same time, the decision procedures return that the current path is invalid.

If the decision procedures report that a path is invalid, the AND/OR table translator stops traversing the current branch in the BDD and backs up to the next unexplored branch. Whenever the traversal algorithm backs up to a node representing an enumerated type predicate, or when the traversal algorithm backs up over an enumerated type predicate (the traversal algorithm backs up over an enumerated type predicate after both branches of the node representing the predicate have been explored) the decision procedures are notified of the incident by the AND/OR table translator. The decision procedures modify the information they are monitoring for the specific enumerated type that has changed.

The AND/OR table translator continues traversing the result BDD until all satisfying paths have been translated to columns in the AND/OR table representation of the result BDD, or until the number of satisfying paths requested by the analyst have been translated to columns in the tabular representation.

4.2.5 PVS Translator

The PVS translator tool generates a PVS theory for each transition in an RSML specification and the tool also generates the PVS conjecture theories for completeness and consistency analysis. This is done in a three stage process. First, each predicate in the AND/OR table is defined as a predicate in the PVS specification language². Second, a predicate representing the full guarding condition is built from the individual predicates defined in the first stage. For example, consider the transition from the state Inhibited to the state Not-Inhibited defined in Figure 4.9. In PVS, this transition would be defined by the theory shown in Figure 4.10. The PVS theory is named with the transition *source*

²A predicate in PVS is a function with return type Boolean.

state, to the transition *destination* state, appended with a unique id: *Inhibited_To_Not_Inhibited_ID1*. The constant *NODESHI* used in the predicate comprising the guarding condition in Figure 4.9 is declared as a constant in the PVS specification. The functions *Own_Tracked_Alt* and *Ground_Level* used in the predicate comprising the guarding condition are declared as variables of type integer in the PVS specification. The predicate comprising the guarding condition is then declared as a function with return type Boolean *Pred168_Grtr?*; each predicate in the guarding conditions is given a number and appended with the type of the predicate being specified. For the transition from Inhibited to Inhibited (Figure 4.9), the predicate comprising the guarding condition would be appended with *LessEqual* in the PVS specification. The guarding condition is defined as a function in PVS with return type Boolean (a PVS predicate), and given a name composed of the transition source and destination states appended with a unique id *Inhibited_TO_Not_Inhibited_T1?*. The definition of the PVS predicate representing the guarding condition is the disjunctive normal form expression of the previously defined predicates comprising the guarding condition. In the guarding condition in Figure 4.9 there is only one predicate and its value must be TRUE; thus, the guarding condition defined in the PVS specification consists of a single predicate, *Pred168_Grtr?*, whose value must be TRUE for the predicate to be satisfied.

In the third stage, a conjecture theory is generated that contains the proof obligations for the completeness and consistency checks. The conjecture theory generated for the state Inhibited is shown in Figure 4.11. The PVS conjecture theory imports the two theories representing the transitions from the state Inhibited, declares the necessary variables, and then sets up the conjectures to check for completeness and consistency. The PVS prover is used on the resulting specifications to attempt to prove the stated conjectures. PVS easily proves the conjectures in Figure 4.11, showing the guarding conditions are complete and consistent.

When a conjecture cannot be proved by PVS, unprovable subgoals (Figure 4.12) are reported.

Transition(s): Inhibited \rightarrow Not-Inhibited

Location: Own-Aircraft \triangleright Descend-Inhibit_{s-30}

Trigger Event: Surveillance-Complete-Event_{e-279}

Condition:

$$\boxed{\text{Own-Tracked-Alt}_{f-248} - \text{Ground-Level}_{f-237} > 1200\text{ft}_{(\text{NODESHI})}} \quad \boxed{\text{T}}$$

Output Action: Descend-Inhibit-Evaluated-Event_{e-279}

Transition(s): Inhibited \rightarrow Inhibited

Location: Own-Aircraft \triangleright Descend-Inhibit_{s-30}

Trigger Event: Surveillance-Complete-Event_{e-279}

Condition:

$$\boxed{\text{Own-Tracked-Alt}_{f-248} \leq (1200\text{ft}_{(\text{NODESHI})} + \text{Ground-Level}_{f-237})} \quad \boxed{\text{T}}$$

Output Action: Descend-Inhibit-Evaluated-Event_{e-279}

Figure 4.9: The transitions out of the state Inhibited.

Unprovable subgoals represent errors in the specification and are easily translated back to an AND/OR table for review by the analyst. For consistency analysis, the unprovable subgoal shown in Figure 4.12 is interpreted as: both transition conditions are satisfied if

```
TrafficDisplayStatus(i) IN_STATE WaitingToSend
AND (AdvisoryCode(i) = ResolutionAdvisory)
AND NOT (i = j)
AND NOT (TrafficDisplayStatus(j) IN_STATE WaitingToSend)
AND NOT (TrafficScore(OtherAircraft(i))
        >= TrafficScore(OtherAircraft(j))).
```

For completeness analysis, the same unprovable subgoal is interpreted as: there is no transition out of a state if the above condition is satisfied. Figure 4.13 shows the AND/OR table representation of the subgoal. Translation of the unprovable subgoals from PVS format to AND/OR table format is currently done manually but could be automated.

```

%-----%
% Theory for transition from state %
% Inhibited to state Not_Inhibited %
%-----%
Inhibited_To_Not_Inhibited_ID1: THEORY
BEGIN

%-----%
% Constant Declarations %
%-----%
    NODESHI: int = 1200 % Threshold for descend inhibit

%-----%
% Variable Declarations: %
%-----%
Own_Tracked_Alt, Ground_Level: VAR int

%-----%
% Definition for the first (and in this case, only) %
% predicate in the guarding condition on the %
% transition from state Inhibited to state %
% Not_Inhibited. %
%-----%
Pred168_Grtr?(Own_Tracked_Alt, Ground_Level):
    bool = Own_Tracked_Alt - Ground_Level > NODESHI

%-----%
% Guarding condition and interface to the theory for the transition %
% from state Inhibited to state Not_Inhibited. %
%-----%
Inhibited_TO_Not_Inhibited_T1?(Own_Tracked_Alt, Ground_Level):
    bool = Pred168_Grtr?(Own_Tracked_Alt, Ground_Level)

END Inhibited_To_Not_Inhibited_ID1

```

Figure 4.10: A PVS theory for the transition from Inhibited to Not-Inhibited.

```

Inhibited_Tests: THEORY
BEGIN
%-----%
% Import declarations for theories for transitions %
% - Inhibited_To_Not_Inhibited, %
% - Inhibited_To_Inhibited, %
%-----%
IMPORTING Inhibited_To_Not_Inhibited_ID1, Inhibited_To_Inhibited_ID0

%-----%
% Variable Declarations: %
%-----%
Own_Tracked_Alt, Ground_Level: VAR int

%-----%
% Conjecture: The conditions for transitions out of %
% state Inhibited are completely specified. %
% Cond1 /\ Cond2 -> True %
%-----%

InhibitedToNotInhibited_InhibitedToInhibited_Complete:
CONJECTURE
  Inhibited_To_Not_Inhibited_T1?(Own_Tracked_Alt, Ground_Level)
OR
  Inhibited_To_Inhibited_T0?(Own_Tracked_Alt, Ground_Level)

%-----%
% Conjecture: The conditions for transitions out of %
% state Inhibited are consistent; non-conflicting. %
% Cond1 /\ Cond2 -> False %
% rewritten as (not(Cond1 /\ Cond2)) %
%-----%

InhibitedToNotInhibited_InhibitedToInhibitedConsistent:
CONJECTURE
  NOT ( Inhibited_To_Not_Inhibited_T1?(Own_Tracked_Alt, Ground_Level)
    AND
    Inhibited_To_Inhibited_T0?(Own_Tracked_Alt, Ground_Level) )

END Inhibited_Tests

```

Figure 4.11: Conjecture theory for the state inhibited.

```

PVS SUBGOAL:
{-1}   WaitingToSend?(TrafficDisplayStatus!1(i!1))
{-2}   ResolutionAdvisory?(AdvisoryCode!1(i!1))
|-----
{1}    (i!1 = j!1)
{2}    WaitingToSend?(TrafficDisplayStatus!1(j!1))
{3}    TrafficScore!1(OtherAircraft!1(i!1))
      >= TrafficScore!1(OtherAircraft!1(j!1))

```

Figure 4.12: PVS unprovable subgoal example.

TrafficDisplayStatus(i) IN_STATE WaitingToSend	T
(AdvisoryCode(i) = ResolutionAdvisory)	T
(i = j)	F
(TrafficDisplayStatus(j) IN_STATE WaitingToSend)	F
(TrafficScore(OtherAircraft(i)) >= TrafficScore(OtherAircraft(j)))	F

Figure 4.13: AND/OR table representation of PVS subgoal.

4.2.6 BDD Analyzer

The BDD analyzer contains routines to check logical expressions represented as BDDs for satisfiability (completeness) and mutual exclusion (consistency). Section 4.2.6.1 describes the process used to check for consistency. Section 4.2.6.2 describes the process used to check for completeness.

4.2.6.1 Consistency Analysis

Recall that in RSML, transitions out of a state are annotated with a trigger (event) and a guarding condition. A particular transition from one state to another state can only occur if the trigger event occurs and the guarding condition associated with that event is satisfied. Further, recall that to check two guarding conditions for consistency requires computing the conjunction of the guarding conditions to see if the conjunction is a contradiction (reduces to FALSE). This section describes how the BDD analyzer checks the guarding conditions out of a state for consistency, and how the inconsistencies are reported to the analyst.

For each trigger-transitions set (described in Section 4.2.1) associated with a particular atomic state, do the following:

1. See if there are multiple transitions associated with the trigger.
2. If there are multiple transitions associated with the trigger, get the BDD representation for each transition's guarding condition from the BDD translator.
 - (a) Compute the pairwise conjunctions of the BDD representations of the guarding conditions.
 - i. For each result BDD, if dynamic variable reordering is enabled, invoke the reordering algorithm to reorder the result BDD.

- (b) If the guarding conditions are consistent (the conjunction reduces to FALSE, a contradiction), report the guarding conditions consistent.
 - (c) If the guarding conditions are not consistent (the conjunction did not reduce to FALSE), report to the analyst the conditions under which both guarding conditions can be satisfied.
 - i. For inconsistent guarding conditions, convert the result BDD to AND/OR table format (via the AND/OR table translator) and print the result table. All satisfiable paths in the result BDD represent potential inconsistencies and are reported as columns in the AND/OR table output.
3. If there are not multiple transitions associated with a trigger, skip the trigger.

4.2.6.2 Completeness Analysis

Recall that to check to see if the guarding conditions on transitions out of a particular state triggered by the same trigger are complete, it is necessary to compute the disjunction of the guarding conditions to see if the disjunction is a tautology (reduces to TRUE). This section describes how the BDD analyzer checks the guarding conditions out of a state for completeness, and how the incompletenesses are reported to the analyst.

For each trigger-transitions set (described in Section 4.2.1) associated with a particular atomic state, do the following:

1. See if there are multiple transitions associated with the trigger.
2. If there are multiple transitions associated with the trigger, get the BDD representation for each transition's guarding condition from the BDD translator.
 - (a) Compute the disjunction of the BDD representations of the guarding conditions.
 - i. For each result BDD, if dynamic variable reordering is enabled, invoke the reordering algorithm to reorder the result BDD.
 - (b) If the guarding conditions are complete (the disjunction reduces to TRUE, a tautology), report the guarding conditions complete.
 - (c) If the guarding conditions are not complete (the disjunction did not reduce to TRUE), report to the analyst the conditions under which no transition can be made out of the state.

- i. For incomplete guarding conditions, negate the result BDD so the unsatisfiable paths in the BDD (the paths leading to FALSE) become satisfiable paths (paths leading to TRUE – this is necessary because the AND/OR table translator translates only satisfiable paths to columns in an AND/OR table, and for completeness analysis, we actually want to report the unsatisfiable paths to the analyst).
 - ii. Convert the result BDD to AND/OR table format (via the AND/OR table translator) and print the result table. All satisfiable paths in the negated result BDD represent potential incompletenesses and are reported as columns in the AND/OR table output.
3. If there are not multiple transitions associated with a trigger, perform the relevant steps described in step 2 above, on the single transition.

4.2.7 PVS Analysis

We provided an overview of the translation process from an RSML specification to a PVS specification in Section 4.2.5. Some additional details related to the translation that were not discussed previously, include: all enumerated type variables in the RSML specification are mapped to abstract datatypes in the PVS specification, and all super-states in the RSML specification are mapped to abstract datatypes whose elements are the immediate child states.

Once the translation process is complete, the analyst initiates PVS and automatically parses and typechecks the theories and declarations generated from the RSML specification. During typechecking, theories for abstract datatypes, such as the disjointness property of abstract datatypes, are automatically generated by PVS. When parsing and typechecking is finished, the analyst invokes the PVS prover on the conjecture to be proved.

When the analyst invokes the prover, a set of strategies developed during this research are loaded into PVS and the strategies become available to the PVS prover. The strategies allow a sequence of commands to be carried out with the invocation of only one command (the strategy name). The strategies allow the PVS analysis to be more automated, and largely frees the analyst from having intimate knowledge of the PVS prover commands. We developed strategies for proving that the

pairwise conjunction of two logical expressions is a contradiction (i.e., the logical expressions are mutually exclusive, or consistent), and we developed strategies for proving that the disjunction of logical expressions is a tautology (i.e., the disjunction of the logical expressions is satisfiable). There are other strategies that are basically sub-strategies within the major strategies. The sub-strategies are useful when the analyst may need to apply some intermediate steps during the proof process. We developed and used the strategies during the course of this research. In the majority of cases, we only needed one command (strategy) to carry out and finish a proof. Appendix B contains definitions of the PVS strategies we applied. As a side note, several additional strategies were developed for and used successfully in a related research area [29].

4.3 Adding Augmenting Information to the Analysis Process

Recall that if the number of error reports is large, it may be that the majority of the reports are spurious and that the reason for the spurious error reports is that information needed by the analysis process to detect and eliminate the spurious errors has been abstracted out of the model. If this is the case, then we want to identify the information that has been abstracted out, augment the analysis process with the information, and rerun the analysis to achieve more accurate results. The method for identifying the missing information is described in Chapter 5. This section describes how to incorporate the augmenting information into the analysis process once it has been identified. Section 4.3.1 describes the process of adding augmenting information to the BDD analysis, and Section 4.3.2 describes how augmenting information is added to the PVS analysis.

4.3.1 Adding Augmenting Information to the BDD Analysis

It is easy to add augmenting information to the BDD consistency analysis process. To demonstrate, let E be a logical expression with sub-expressions X , Y , and Z , where X , Y , and Z are logical

expressions containing the variables $x_1, x_2, \dots, x_m, y_1, y_2, \dots, y_n$, and z_1, z_2, \dots, z_p respectively.

Assume that two variables, say x_i and z_j , cannot both be satisfied (TRUE) at the same time due to the structure of the specified system, and that this information has been abstracted out of the model.

Assume also, that every path in the BDD representation of E that eventually leads to TRUE has both x_i and z_j TRUE; i.e., there does not exist any satisfiable path in the BDD that does not pass through x_i and z_j , such that x_i and z_j are both TRUE. The analysis report shows 100's of millions of satisfying paths, implying that there are 100's of millions of inconsistencies between the logical expressions. The analyst wisely assumes that most of the reported errors are spurious errors, and uses the method described in the next chapter to locate the source of the spurious errors. She quickly locates the source of the spurious errors, adds the required augmenting information to the analysis process by including it in the above expression, and reruns the analysis. In this case, the required augmenting information is: $\neg(x_i \wedge z_j)$. If E was $X \wedge Y \wedge Z$, then the logical expression (E') with the augmenting information included is: $E' = X \wedge Y \wedge Z \wedge \neg(x_i \wedge z_j)$. Rerunning the analysis on the BDD representation of E' will show the original logical expression consistent.

The same process applies to adding augmenting information to the completeness analysis, however, the original result BDD (the BDD obtained from taking the disjunction of the guarding conditions) must be negated prior to adding the augmenting information. Completeness and consistency are duals of each other. In completeness analysis we want to report unsatisfiable paths to the analyst, while in consistency analysis we want to report satisfiable paths. Thus, by negating the BDD that results from completeness analysis, we change the unsatisfiable paths to satisfiable paths and vice versa, thereby reformulating the problem in terms of consistency analysis. We can then add the augmenting information to the results of the completeness analysis in the same way as described above for consistency analysis.

4.3.2 Adding Augmenting Information to the PVS Analysis

Augmenting information is added to the PVS analysis using the following steps:

1. Add the augmenting information to the PVS specification as an axiom.
2. Start the prover to prove the desired conjecture.
3. Use the (lemma ``axiom-name'') prover command (described in Appendix B) to bring the axiom axiom-name into the proof process.
4. Determine instantiations for the variables used in the axiom.

Figure 4.14 shows two axioms as specified in the PVS specification language. The

```

OtherAltReporting_AltReporting_Assertion :
  AXIOM
    cTrue?(Other_Alt_Reporting) IFF Yes?(Alt_Reporting)
OtherAltReporting_OtherAirStatus_Assertion :
  AXIOM
    NOT (cTrue?(Other_Alt_Reporting))
      IMPLIES State_Airborne?(Other_Air_Status)

```

Figure 4.14: Specifying axioms in the PVS specification language.

interpretation of the OtherAltReporting_AltReporting_Assertion is, the variable Other_Alt_Reporting is TRUE IFF the super-state Alt_Reporting is in state Yes. The OtherAltReporting_OtherAirStatus_Assertion axiom is interpreted as, the variable Other_Alt_Reporting NOT TRUE (FALSE), IMPLIES the super-state Other_Air_Status is in state Airborne. Another way to state this latter axiom is, IF Other_Alt_Reporting is FALSE, then Other_Air_Status is NOT in state OnGround. Figure 4.15 shows an example of how our PVS sub-strategy (skolemizeandrewrite\$) and the PVS prover commands would be applied to add the above augmenting information (axioms) to the PVS proof process, and to complete the proof. The PVS prover commands and our strategies are defined and explained in Appendix B.

```
(skolemizeandrewrite$) strategy
(lemma "OtherAltReporting_AltReporting_Assertion")
(lemma "OtherAltReporting_OtherAirStatus_Assertion")
(apply (repeat* (inst?)))
(apply (repeat* (try (bddsimp) (record) (postpone)))))
```

Figure 4.15: Adding augmenting information to the PVS proof process and completing the PVS proof.

4.4 Iteration Options and Analysis of Output

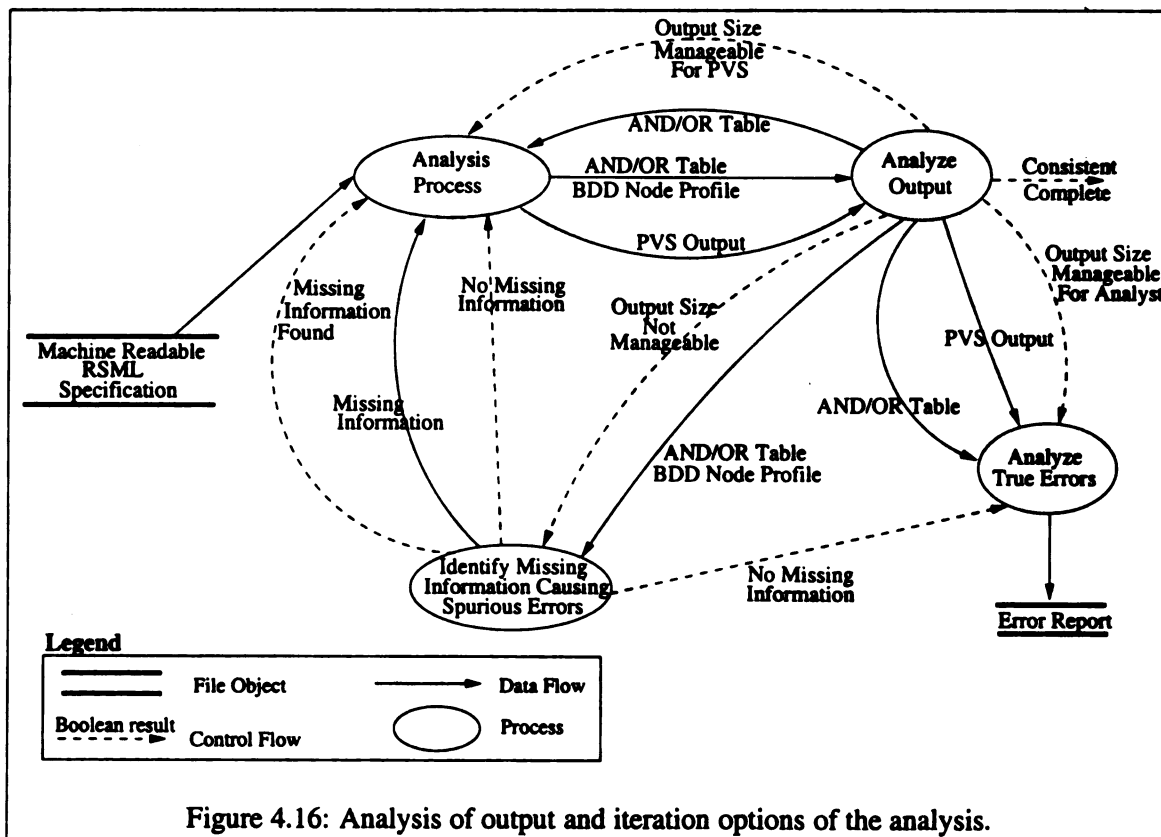
Section 4.1 provided a high level view of our analysis process, and described some typical scenarios of how an analysis might proceed when applied to RSML requirements to check the requirements for completeness and consistency. In sections 4.2 and 4.3 we described the specific tools our analysis process uses and how to add augmenting information to the analysis process to reduce or eliminate the number of spurious errors reported in the analysis output. We now describe the analysis process in more detail, specifically highlighting how the analyst directs the analysis process, the analysis of output, and the iteration options available to the analyst. Note that, as in Section 4.1.1, the description is specific to the analysis of state-based requirements (specifically, RSML requirements) for completeness and consistency.

The analyst directs the analysis process by setting specific command line options when the process is initiated. The command line options available include options to:

- Run BDD analysis on a given specification. When the analyst chooses to run BDD analysis, the following options are available:
 - Run completeness analysis.
 - Run consistency analysis.
 - Run both completeness and consistency analysis.
 - Enable dynamic variable reordering using the stable window 3 reordering algorithm.
 - Enable dynamic variable reordering using the sift reordering algorithm.
 - Enable decision procedures.
 - Count the number of satisfying paths in the result BDD.
 - Print the result BDD.
 - Print only x number of paths from the result BDD in AND/OR table format.

- Enable sampling (described in Chapter 5, Section 5.2.2) so that samples of the result BDD are taken and printed out in AND/OR table format. The analyst can choose to take any number of samples from four different parts of the result BDD. The analyst can specify a different number of samples from each of the four sample areas, including zero samples. At this time, the sample algorithms cannot be changed by the analyst.
 - Expand macros.
- Generate PVS specifications for the guarding conditions, and generate the conjecture theory to specify the completeness and consistency conjectures.
 - Generate PVS specifications for individual AND/OR tables, and generate the conjecture theory to specify the completeness and consistency conjectures to check the individual tables for completeness and consistency. This feature is used when the analyst wants to apply PVS analysis to the AND/OR table output from the BDD analysis.

Figure 4.16 shows the possible outputs from the analysis process, the possible outcomes of the output analysis process, and the options available to the analyst based on the outcome of the *analyze output* process. After the automated portion of the analysis is finished and the analysis report is generated, the analyst inspects the output to determine the next course of action to take; the *Analyze Output* process in the figure. If the report shows the guarding conditions complete or



consistent, the process is finished and no further iterations are required. If the output size from either analysis process is manageable, the analyst manually inspects the output for the true errors. As in Section 4.1.1, for this discussion we assume that symbolic analysis using BDDs is applied first. If the output size from the symbolic analysis is not manageable for an analyst, but is manageable for PVS, it is fed back into the analysis process, a PVS specification is generated, and the PVS output is reported to the analyst.

If the output size from either analysis process is not manageable, we assume that most of the error reports are spurious and exist because there is some information missing from the model, such that if the information was added to the analysis process, the spurious errors would be reduced or eliminated. The outputs from the symbolic analysis, the BDD node profile and the AND/OR table, are used to identify the information that is missing from the model being analyzed that is causing the spurious errors; the process labeled *Identify Missing Information Causing Spurious Errors* in the figure. If no missing information can be found, and both analysis processes have been tried, then the errors must be true errors, and the analyst must examine them manually. If only symbolic analysis has been tried and no missing information can be found, either the symbolic analysis output is converted to a PVS specification and PVS analysis is run, or PVS analysis is applied to the original expressions. If missing information is found, it is added to either the machine readable RSML specification or the PVS specification generated from the machine readable RSML specification, and either symbolic analysis or PVS analysis is run on the augmented specification. Adding augmenting information to the analysis process was discussed in detail in Section 4.3. The process to identify missing information causing spurious errors is described in the next chapter.

Note that it is not necessary to start the analysis process using symbolic analysis. The analyst may choose to start with PVS analysis. If the number of subgoals reported from PVS is too large, or the analysis fails to halt, the analyst can run BDD analysis on the original specification and use

the process described in Chapter 5 to look for the missing information leading to spurious errors. It may be beneficial to start the analysis using the symbolic component, however, for the following reasons:

- The symbolic analysis using BDDs is fully automated, faster, and simpler to use than PVS analysis.
- The technique used to identify the augmenting information relies on the output from the symbolic analysis to locate the augmenting information.
- If all or most of the augmenting information is related to the structure of the system, starting with PVS analysis will require an unnecessary extra iteration through the analysis process. The extra step is required since the PVS output gives no indication of where to begin to look for missing information.

4.5 Summary

In this chapter we described an analysis technique for checking disjunctions and conjunctions of logical expressions to see if they form tautologies (completeness) or contradictions (consistency). The process is iterative, and integrates graph-based symbolic analysis using BDDs, with theorem proving using PVS. There are some consistent (complete) logical expressions that a first run BDD analysis shows consistent (complete). For the purposes of the discussions in this chapter, we assumed that symbolic analysis using BDDs was applied first, and we presented several possible benefits to applying symbolic analysis first. Ultimately, however, it is up to the analyst to decide the flow the analysis process takes. In Chapter 6 we describe some heuristics based on the types of predicates comprising the guarding conditions, that the analyst may use to decide the most promising order to apply the analysis methods in.

We described several potential interaction scenarios between the analyst and our analysis technique. Some scenarios require the analyst to identify information that was abstracted away from the analysis model, and that needs to be re-introduced into the model for the analysis to finish

successfully, or to generate more accurate analysis reports. However, we did not describe how our process helps an analyst identify the augmenting information. In the next chapter we describe how the output of the symbolic analysis using BDDs can help the analyst locate the missing information leading to spurious errors.

Chapter 5

Domain Axioms and Domain Axiom Identification

In the previous chapter we described in detail our iterative and integrative analysis process. We saw that if the analysis reports are unwieldy, the analyst can assume that many of the error reports are spurious, and that some contradictions in the logical expressions are not being detected because the analysis procedures lack the information to detect these contradictions. We also saw that if the analyst can determine what missing information is leading to the spurious error reports, augment the analysis process with this information, and re-run the analysis, the analysis process will yield more accurate error reports. In the last chapter, however, we left out the details of how the analyst identifies the augmenting information.

In this chapter we present our technique to help the analyst identify the contradictions and missing information causing the spurious errors. Our technique uses the structure of the result BDD from the symbolic analysis to identify potential contradictions, and, thus, to identify the information that the analysis procedures lack. Since the augmenting information represents a truth or axiom about the system (i.e., given the structure and properties of a system, the augmenting information is

something that is always true about the system), we refer to the augmenting information describing the contradictions as *domain axioms*¹.

Section 5.1 describes the meaning of domain axioms as used in this research, and Section 5.2 describes our technique to help the analyst identify the domain axioms.

5.1 Domain Axioms

Recall that we represent the input to our analysis in disjunctive normal form. We also present the analysis output to the analyst in disjunctive normal form. If two guarding conditions are consistent (they cannot both be satisfied at the same time), their conjunction should reduce to FALSE. The spurious errors that get reported are disjuncts that contain contradictions within the conjunctive terms; the conjunction should reduce to FALSE, but the contradictions between the conjuncts are not being recognized by the analysis methods. As a result, these contradictory conjunctions are reported to the analyst as errors. If we can find the contradictory predicates leading to the spurious errors, create domain axioms representing the contradictions, and feed these domain axioms back into the analysis process, then the output resulting from this second round of analysis will correctly reduce to FALSE, showing that the guarding conditions are consistent. If the guarding conditions are not consistent, then the disjuncts containing contradictions will be eliminated and the true errors will remain and be reported to the analyst.

In Chapter 3 we identified four classes of spurious errors. The latter two classes of spurious errors are the most difficult for analysis procedures to identify. In the previous chapter, we showed the guarding conditions for transitions out of the state *Inhibited*. Symbolic analysis will generate

¹The name *domain axioms* was suggested by Dr. Laura Dillon at Michigan State University.

the following spurious error reports for the guarding conditions shown in Figure 4.9:

```
Inhibited --> Inhibited conflicts with
    Inhibited --> Not-Inhibited if

    (Own_Tracked_Alt() - Ground_Level()) > 1200    : T ;
    Own_Tracked_Alt() <= (1200 + Ground_Level()) : T ;

No transition out of Inhibited is satisfied if :

    (Own_Tracked_Alt() - Ground_Level()) > 1200    : F ;
    Own_Tracked_Alt() <= (1200 + Ground_Level()) : F ;
```

Figure 5.1 shows an example of the domain axioms that can be added to the analysis process to eliminate the spurious errors reported above. The first domain axiom, to eliminate the spurious consistency report, says that the two predicates cannot both be TRUE at the same time. The second domain axiom, to eliminate the spurious completeness report, says that the predicates cannot be both FALSE and that one of them has to be TRUE.

Recall that the goal is to eliminate as many spurious errors as possible from the analysis output so the analyst can easily identify and correct the true errors in the specification. A single undetected contradiction between two or more predicates may be responsible for causing many spurious error reports. The reason for this is that there are many satisfying permutations that exist among the predicates comprising the conjuncts. Therefore, the undetected contradiction causing the spurious error reports may occur in thousands or millions of permutations in the analysis output. For example, when we used BDD analysis on two of the guarding conditions from the TCAS II specification, the result BDD contained 4,909,890 satisfying paths (paths leading to TRUE). The predicates

<p>Domain Axiom to eliminate consistency spurious error:</p> <pre>NOT [((Own_Tracked_Alt() - Ground_Level()) > 1200) AND (Own_Tracked_Alt() <= (1200 + Ground_Level()))]</pre> <p>Domain Axiom to eliminate completeness spurious error:</p> <pre>((Own_Tracked_Alt() - Ground_Level()) > 1200) XOR (Own_Tracked_Alt() <= (1200 + Ground_Level()))</pre>
--

Figure 5.1: Domain Axioms associated with spurious errors of the state Inhibited.

represented by the first two nodes in the result BDD were always FALSE and always TRUE respectively. When we examined the TCAS II specification we saw that these two predicates could not be FALSE and TRUE at the same time. However, all 4,909,890 satisfying paths in the result BDD had the first node always FALSE and the second node always TRUE. Thus, the original guarding conditions were consistent, but the analysis reported almost five million inconsistencies, and all of the reported inconsistencies involved a single undetected contradiction between two predicates in the guarding conditions; the reported inconsistencies were all spurious error reports. We added the information about the contradictory predicates to the analysis process as a domain axiom and reran the BDD analysis (see Chapter 6, Section 6.1.1.1.2 for details). The analysis report from the second iteration with the domain axiom showed the guarding conditions consistent. Thus, finding the undetected contradictions causing the spurious errors, identifying the appropriate domain axioms associated with the undetected contradictions, and feeding these domain axioms back into the analysis process can eliminate thousands or millions of spurious errors from the analysis output at one time.

The problem is, “how can one find the undetected contradictions so that one can identify the missing information and create the relevant domain axioms to add to the analysis process?”. In particular, how can one find undetected contradictions that are artifacts of the state machine structure or that are related to the environment in which the system operates; that is, contradictions related to the last class of spurious errors described in Chapter 3 (class 4 spurious errors). Class 4 spurious errors and their associated domain axioms are more difficult to identify since identifying them often requires intimate and global knowledge of the state machine structure and intimate domain knowledge.

The brute force way to identify domain axioms is to take each error reported and examine the specification to see if the reported error represents a true inconsistency (incompleteness), or

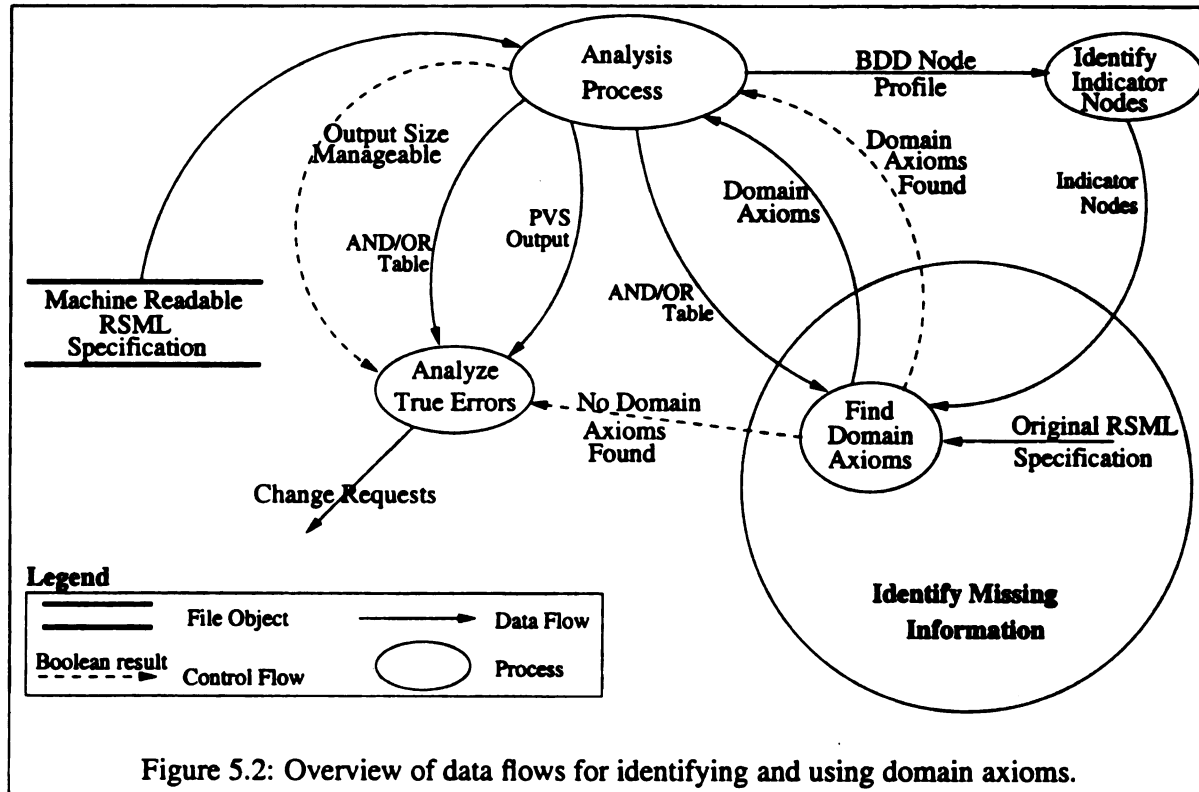
if there is some information in the original specification that precludes the reported error from occurring. This method can be difficult, error-prone, and time consuming since the error reports can be voluminous and the original specification complex. Therefore, it is beneficial to have a method that will aid in the process of identifying the necessary domain axioms. We describe such a method in the next section.

5.2 Identifying Domain Axioms

The process we developed for identifying domain axioms uses the structure of the result BDD to identify predicates that may be indicative of domain axioms in the specification. Normally, the analyst may need to check many predicates to see if they are involved in the required domain axioms. With our approach, we can generally narrow the search space of predicates down to a small number. For example, in our large study (Chapter 6) we were able to narrow the search space of predicates to fewer than seven predicates, from sixty or seventy predicates.

Figure 5.2 shows an overview of the data flows in our technique for identifying and using domain axioms. The discussion in this chapter assumes that the output from the first run analysis was too large for the analyst to easily eliminate the spurious errors by hand. First, we perform symbolic analysis using BDDs with dynamic variable reordering enabled, and generate a BDD node profile using a routine in Long's BDD library (discussed in Chapter 4, Section 4.2.2). From the BDD node profile, we locate what we call *indicator nodes*. Indicator nodes indicate predicates that may be causing spurious errors in the analysis report, and so, are used to assist in the process of finding domain axioms. This process is shown inside of the *Identify Missing Information* circle in the figure. We call the predicates associated with the indicator nodes *indicator predicates*. Once we identify the indicator predicates, we locate the places in the original RSML specification where the indicator predicates appear and look for truths about the system that are related to the indicator

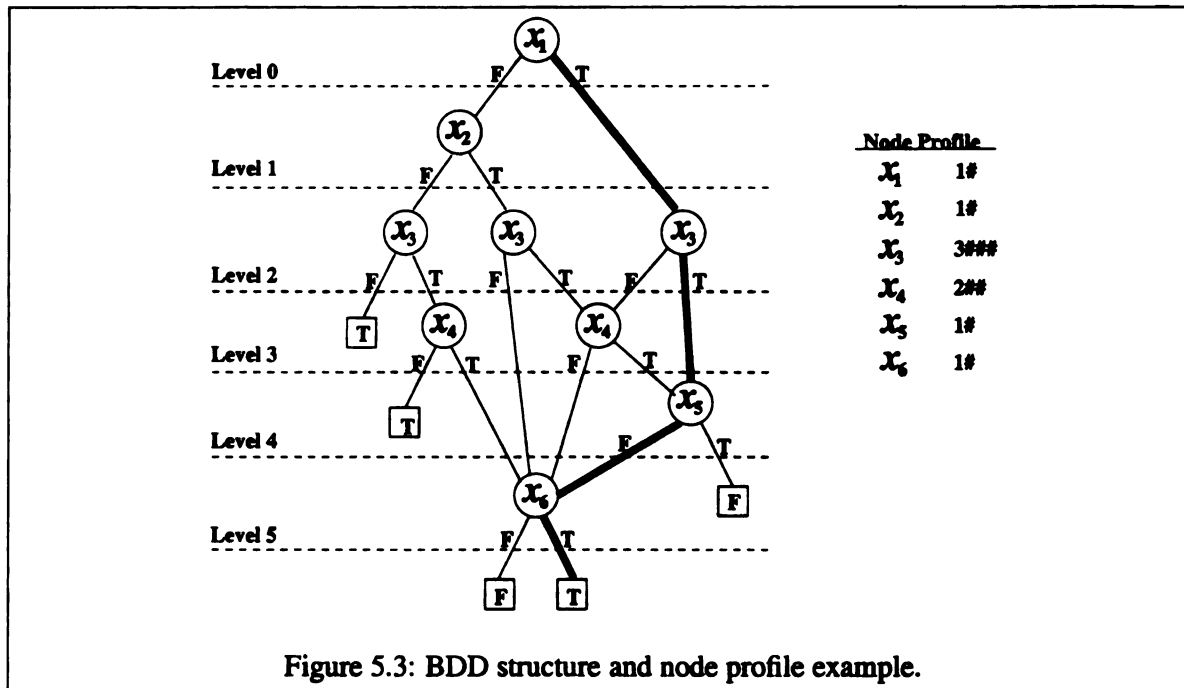
predicates. Once we find these truths (domain axioms), we can use the AND/OR table output from the symbolic analysis to verify that the domain axioms are actually violated in the analysis report (Section 5.2.2). If we find domain axioms that are violated, we feed them back into the analysis process by adding them to the machine readable RSML specification or to the PVS specification, and re-run the analysis. If the output size is manageable on the second iteration, or no domain axioms are found, then the AND/OR table output or the PVS output (whichever is considered more readable) is used to manually analyze the errors and generate the final error report. We now describe indicator nodes in more detail.



5.2.1 Indicator Nodes

Figure 5.3 shows an example of a BDD and its node profile. Recall (Chapter 3, Section 3.1.3.1) that a strict total order is imposed on the occurrence of the variables in a BDD as the graph is traversed from the root to the leaves. On a path from the root to the leaves, for any two nodes labeled x_i and

x_j , if $i < j$ then x_i must occur before x_j along any path [8]. The figure shows the nodes at each level. At each level in the BDD there may be zero or more node occurrences. A node that does



not appear along a particular path in a BDD is a *don't care* for that particular path; for example, variables x_2 , and x_4 are *don't cares* in the highlighted path shown in Figure 5.3 (their values do not matter along this particular path). The BDD shown in the figure has nine potentially satisfying paths; nine paths through the BDD lead to terminal node T.

Assume the BDD represents the conjunction of two consistent guarding conditions. Since the BDD did not reduce to FALSE, this means there are nine paths in the BDD that led to T when they should have led to F and thereby reduced the entire BDD to FALSE. Essentially, there is some contradiction between the nodes along each of the nine satisfiable paths that has not been detected. The BDD will reduce to FALSE when the information related to the undetected contradictions is included in the analysis.

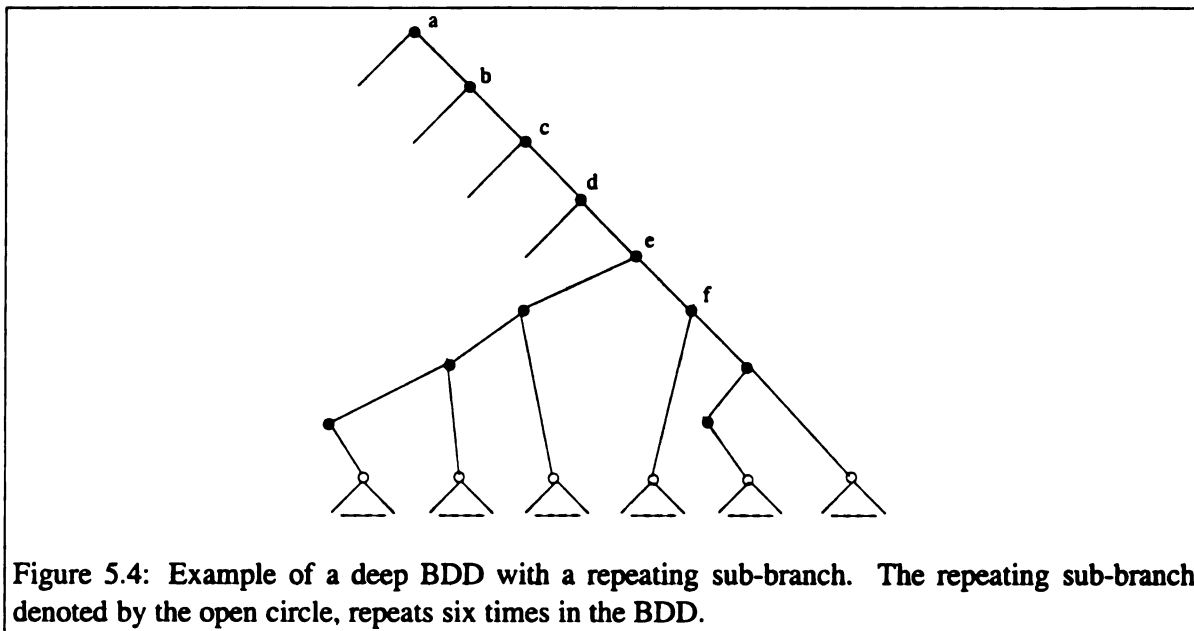
The BDDs we are working with are much larger than the simple BDD shown in the figure, and

it is not feasible to look at the BDD itself to try to identify contradictions between predicates along the satisfying paths. We cannot easily visualize what a large BDD looks like, however, we can get an idea of what the structure of the BDD is by looking at the number of nodes that occur at each level. A BDD node profile, also shown in Figure 5.3, shows the node count for each level in the BDD. For example, the node profile shows a count of three for node x_3 , the node at level two in the BDD.

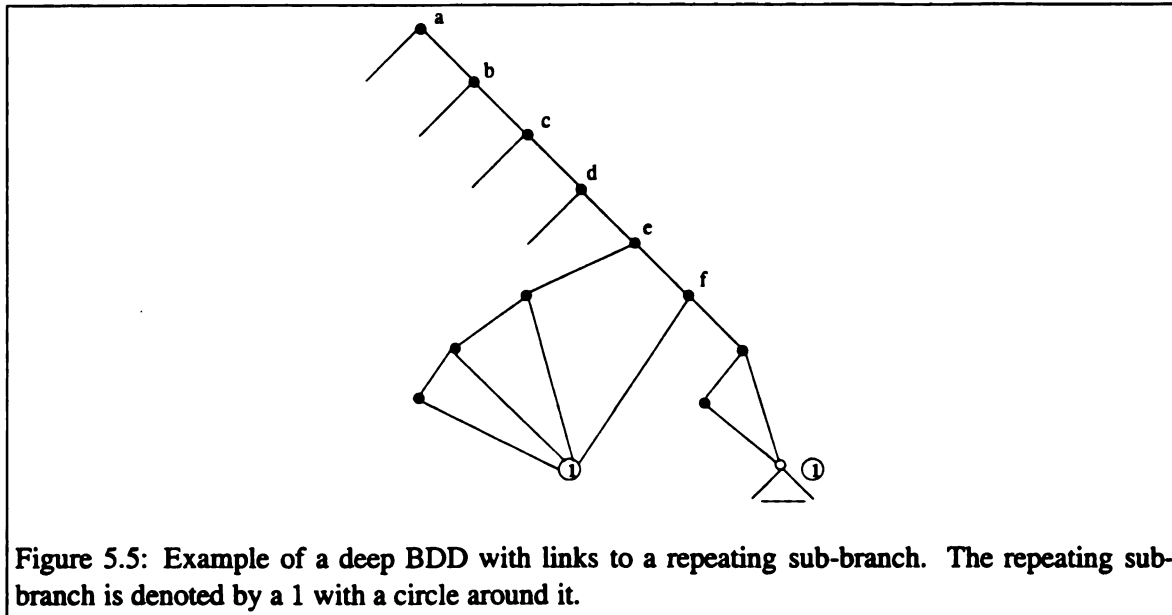
As discussed in Chapter 4, Section 4.2.2.1, Long's BDD package allows the variables to be dynamically reordered during BDD analysis, and the ordering of the variables can greatly affect the number of nodes required to denote a given function (Chapter 3, Section 3.1.3.1). The goal of dynamic variable reordering is to rearrange the BDD variable ordering in an attempt to reduce the number of nodes in the graph [5, 8]. The rearrangement attempts to move nodes to positions in the BDD so that the total number of nodes in the BDD is minimized. For example, if all paths pass through a particular node, the best position for that node would be the root, since of necessity, all paths pass through the root node. Similarly, we know that the higher up in the BDD a node occurs, the more paths are likely to pass through it. In addition, if a node near the bottom of the BDD only occurs once, it is possible that this node is a convergence point so that many paths from higher up pass through it. Node x_6 in Figure 5.3 is an example of a node that acts as a convergence point; seven paths out of the nine pass through this node.

The structure of the larger BDDs we are investigating in this work tends to be deep (30-70 variables, which means 30-70 levels in the BDD) with many repeating sub-branches lower in the BDDs (Figure 5.4). The sub-branches are not actually replicated in the BDD with Long's BDD package. Rather, one sub-branch exists and all other branches that lead directly to the sub-branch are linked to the existing sub-branch. Figure 5.5 shows how all branches that lead to the same sub-branch, point to a single instance of the repeating sub-branch; the repeating sub-branch is

denoted by a 1 with a circle around it. The significance of the figures is that they demonstrate what the general structure of the BDDs for the conditions we are working with tends to look like. In addition, they show that for BDDs with this general structure, if an undetected contradiction exists between any of the nodes from *a* to *d*, identifying the contradictory predicates and augmenting the analysis with information about the contradictions will show the original expressions mutually exclusive.

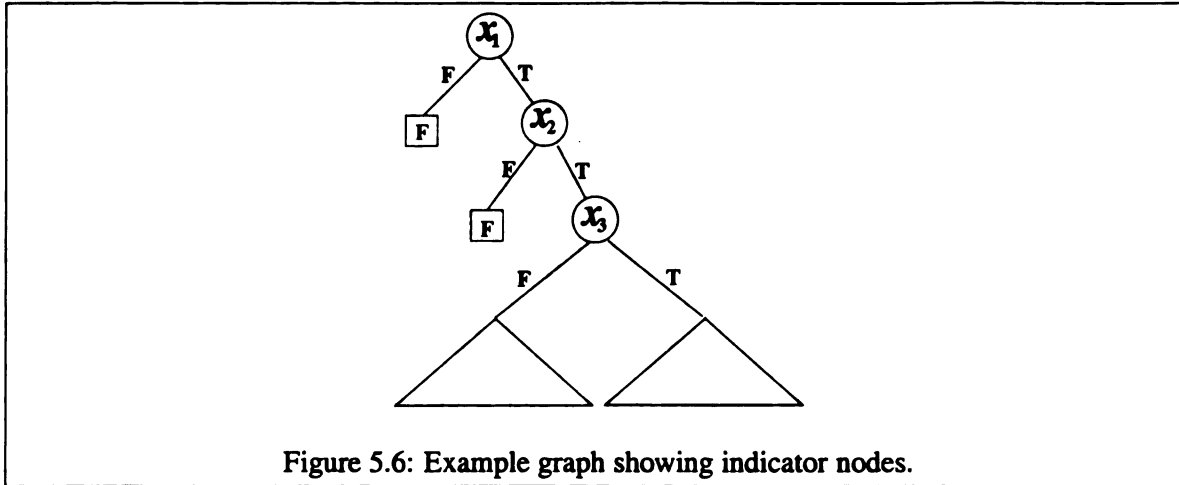


The method we developed for symbolic analysis with BDDs uses the procedures in Long's BDD package to reorder the nodes and to generate a BDD node profile for the result BDD. We used the sift reordering algorithm in all the test cases reported in Chapter 6, but the analyst can select the window method if she desires; in the majority of cases, we found that the sift reordering algorithm provided better results than the window method. We found that nodes that occur along every path, or on a majority of paths near the top or bottom of the BDD are indicative of predicates that may be involved in domain axioms in the specification. Recall that a single undetected contradiction may lead to many permutations of the spurious error reports (Section 5.1). Since these permutations



often outnumber any real errors by orders of magnitude, the contradictory predicates involved in the permutations tend to percolate up near the top of the BDD or drop down near the bottom of the BDD so the nodes representing the predicates will occur as few times as possible in the BDD, thus reducing the total number of nodes. We call these nodes that may be indicative of predicates involved in spurious error reports *indicator nodes*. The nodes are indicative in the sense that they point out (indicate) predicates that are likely to be the cause of spurious errors.

For example, consider Figure 5.6 which shows a typical graph generated during our analysis for consistency. In the figure, we see that all paths pass through the first three nodes of the graph. All three of these nodes would be considered indicator nodes; it is not important to know what the rest of the graph looks like. Assume that the guarding conditions being analyzed are consistent so that the BDD that results from taking the conjunction of the guarding conditions should be a contradiction. However, as the figure shows, the resulting graph does not reduce to FALSE. Assume the number of errors reported is on the order of millions. In this case, the analyst assumes that most or all of the errors are spurious and so examines the BDD node profile to identify indicator nodes. The analyst identifies the first three nodes as indicator nodes and identifies the indicator predicates associated



with the nodes. The analyst then examines the specification to see if the indicator predicates are involved in any domain axioms that are violated in the result BDD. In this case, assume that the predicates associated with the first two nodes cannot both be true at the same time. The analyst adds the appropriate domain axiom $\neg(x_1 \wedge x_2)$ to the model and reruns the analysis. With the appropriate domain axiom added to the analysis process, the analysis report correctly shows that the original guarding conditions are consistent.

It is trivial to pick out the indicator nodes in the BDD node profile. The node profile shows only the nodes that actually occur in the graph under consideration, it shows the nodes in the order in which they occur within the graph, and it shows the number of nodes that occur at each level in the graph. Therefore, to identify the indicator nodes, one simply examines the node profile and identifies those nodes near the top or bottom of the profile that show an occurrence count of one, or that show by their count that they occur along all or a majority of paths in the graph. Nodes that occur along every path or along a majority of paths can be determined from the node profile based on knowledge of the structure of a BDD.

Consider the BDD fragment shown in graph (1) of Figure 5.7. The BDD node profile for graph (1) is shown in Figure 5.8. BDDs are structured such that from each node, one can traverse to the left or to the right only. Therefore, there are always two ways and only two ways to leave a node; one

or both of the alternatives may lead to a terminal node or to another non-terminal node. In addition, the nodes are ordered in a BDD and must maintain that order throughout the graph. Therefore, if a node occurs twice in the graph (as indicated in the node profile) and is ordered such that it appears in the graph after the root, then it is necessarily the case that all paths from the root to the terminal nodes pass through both the root and the second node. Furthermore, if the third node from the root occurs four times while the second node occurs twice, then it is necessarily the case that the third node also occurs along every path in the BDD from the root to the terminal nodes (see Figure 5.7 graphs (1) and (2)). A similar argument exists for other patterns of indicator nodes.

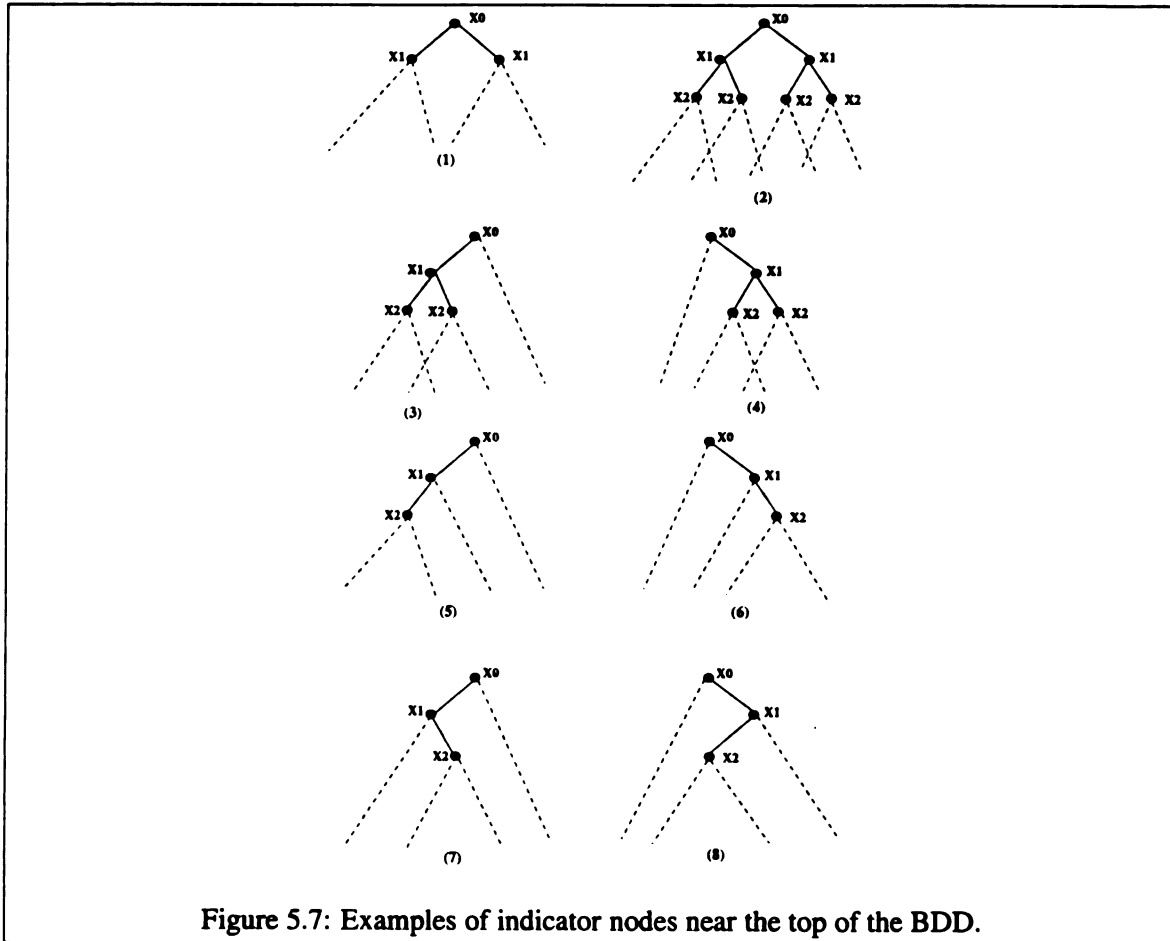


Figure 5.7: Examples of indicator nodes near the top of the BDD.

There may be as many as tens or hundreds of predicates in a single result BDD. In our case, we had between 60 and 70 predicates in some of the result BDDs. The indicator nodes help the analyst to determine those predicates that are worth investigating further. Without some indication of which

Number of Nodes Per Level		Graph Number							
	Node	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
	X0	1	1	1	1	1	1	1	1
	X1	2	2	1	1	1	1	1	1
	X2	-	4	2	2	1	1	1	1

Figure 5.8: Node profiles of indicator nodes; a dash represents a don't care.

predicates to look for to find domain axioms, the analyst, in the worst case, may need to examine all combinations of the predicates. Manually looking for domain axioms among a large number of predicates can be difficult and time consuming. Our technique can narrow the field to a small number of predicates worth examining further. For the cases we looked at with 60 or 70 predicates, we were able to reduce the number of predicates to investigate further, to under seven predicates.

In tests with the TCAS II specification, we have successfully applied this method on several pairs of large guarding conditions that were generating millions of error reports, all of which turned out to be spurious; i.e., the guarding conditions for the transitions were consistent in all cases. Figure 5.9 shows a portion of one of the node profiles that we analyzed. The predicates associated with the first four nodes are indicator predicates and were indicative of domain axioms in the specification. In this case, the first, second, third, fourth, seventh, and ninth nodes were all involved in the domain axioms whose absence resulted in over 18 million spurious errors.

In general, the fewer the domain axioms whose absences are leading to spurious errors, the better. In cases where the undetected contradictions causing the spurious errors are all related to a single domain axiom, the upper levels of the graph should contain the contradictory nodes only once, and in many cases, the truth values should be the same in all satisfying paths. This is because if a single contradiction is causing all of the spurious errors, the contradictory predicates will contradict each other with the same truth values in every case. If there are multiple undetected contradictions, then the graph tends to be wider (i.e., multiple nodes at each level) since the different undetected

Other_Air_Status IN_STATE State_On_Ground:	1
Alt_Reporting IN_STATE Lost:	2 #
Alt_Reporting IN_STATE No:	4 ##
Alt_Reporting IN_STATE Yes:	6 ###
PREV_RA_Inhibit = cTrue:	3 #
Other_Tracked_Range() < PROXR:	9 #####
Other_Alt_Reporting = cTrue:	9 #####
Current_Vertical_Separation() < PROXA:	9 #####
Other_Air_Status IN_STATE State_Airborne:	9 #####
Own_Tracked_Alt() >= ABOVNMC:	6 ###
Auto_SL IN_STATE ASL_2:	9 #####
RA_Inhibit_From_Ground = cTrue:	9 #####
Mode_Selector = TA_Only:	9 #####
Other_Bearing_Valid = cTrue:	6 ###
Own_Tracked_Alt() > Other_Tracked_Alt():	3 #
Own_Tracked_Alt() < Other_Tracked_Alt():	6 ###
Inhibit_Biased_Climb() > Down_Separation():	6 ###
(Own_Tracked_Alt() - Other_Tracked_Alt()) >= MINSEP:	6 ###
Up_Separation() >= ALIM():	6 ###
(Own_Tracked_Alt() - Other_Tracked_Alt()) <= n_MINSEP:	6 ###
Down_Separation() >= ALIM():	6 ###
Current_Vertical_Separation() > LOWFIRMZ:	6 ###
Up_Separation() <= SENSEFIRM():	6 ###
Down_Separation() <= SENSEFIRM():	6 ###
Modified_Tau_Capped() < TFRTHR():	8 #####
Other_Tracked_Range_Rate() > Zero:	11 #####
Threat_Alt_VMD() < ZT():	11 #####
Current_Vertical_Separation() < ZT():	33 #####
Time_To_Co_Alt() < True_Tau_Capped():	11 #####
Time_To_Co_Alt() < TVTHR():	11 #####
ADOT() >= n_ZDTHR():	11 #####
Current_Vertical_Separation() < ZTHRТА():	14 #####
ADOT() >= n_ZDTHRТА:	14 #####
-((Current_Vertical_Separation()/ADOT())) < TVTHRТАТBL():	14 #####
Other_Capability = TA_RA:	22 #####
Own_Tracked_Alt_Rate() <= OLEV:	12 #####
Other_Range_Valid = cTrue:	26 #####
Other_Tracked_Range() >= DMOD():	12 #####
Tau_Rising IN_STATE PLUS_3:	12 #####
Intruder_Status IN_STATE Threat:	12 #####
Other_Tracked_Range_Rate() > RDTHRТА:	13 #####
Other_Tracked_Range() <= DMODТА():	7 ###
(Other_Tracked_Range()*Other_Tracked_Range_Rate())<=H1ТА():	7 ###
Other_Tracked_Range() <= RMAX:	6 ###
Modified_Tau_Capped() < TRTHR():	6 ###
Other_Tracked_Range() > DMOD():	6 ###
(Other_Tracked_Range_Rate() * Other_Tracked_Range())>H1():	6 ###
Other_VRC = No_Intent:	3 #
Intent_Received IN_STATE IR_No:	3 #
Modified_Tau_Capped() < FRTHR():	2 #
Other_Tracked_Alt_Rate() <= OLEV:	1
Other_Tracked_Alt_Rate() >= Zero:	2 #
Current_Vertical_Separation() > MAXALTDIFF:	1
Current_Vertical_Separation() > MAXALTDIFF2:	1
PT_Timer IN_STATE PT_0:	1
Total: 480	

Figure 5.9: Partial node profile for consistency analysis of two transitions from a large avionics specification.

contradictions will require different truth values and more predicates to represent the contradictions.

Similarly, indicator nodes (and therefore, domain axioms) may be more difficult to find if true errors are occurring on a large number of satisfying paths in the BDD. The paths containing the true errors may outnumber the paths containing the spurious errors and therefore the predicates related to the true errors will tend to percolate up near the top of the BDD. In general, this does not happen, since the initial intent is to create a complete and consistent specification. However, for inconsistent and incomplete guarding conditions whose paths containing spurious errors originally outnumber the paths containing true errors by orders of magnitude, once the domain axioms are found and added to the analysis process, the number of permutations of paths containing the true errors can still be unmanageable (for example, on the order of millions). Fortunately, PVS analysis with the proper domain axioms added can generally reduce the error reports to a manageable size, since many of the satisfying path permutations may be able to be collapsed into fewer numbers of satisfying paths with the proper reasoning skills, such as the decision procedures included in PVS. For example, there may be several paths in the result BDD that PVS' decision procedures could collapse into a single path, whereas with the BDD analysis all of the paths would appear as individual columns in the AND/OR table representation of the result BDD.

Ideally, we want to apply the reordering algorithm only to the result BDD. If reordering is applied to multiple BDDs at one time, the algorithm attempts to find the best overall ordering for all of the BDDs. This may cause non-indicator nodes in the result BDD that occur in the majority of paths in the other BDDs, to take an indicator node position in the result BDD. Thus, it becomes more difficult to identify the true indicator nodes in the result BDD. Unfortunately, the current tool reorders all existing BDDs and not just the result BDD. This problem can be corrected, and will be corrected in a future version of the tool. However, we do not know for sure the effect this change will have. We will investigate the effect once the revision is made. So far, reordering all the BDDs

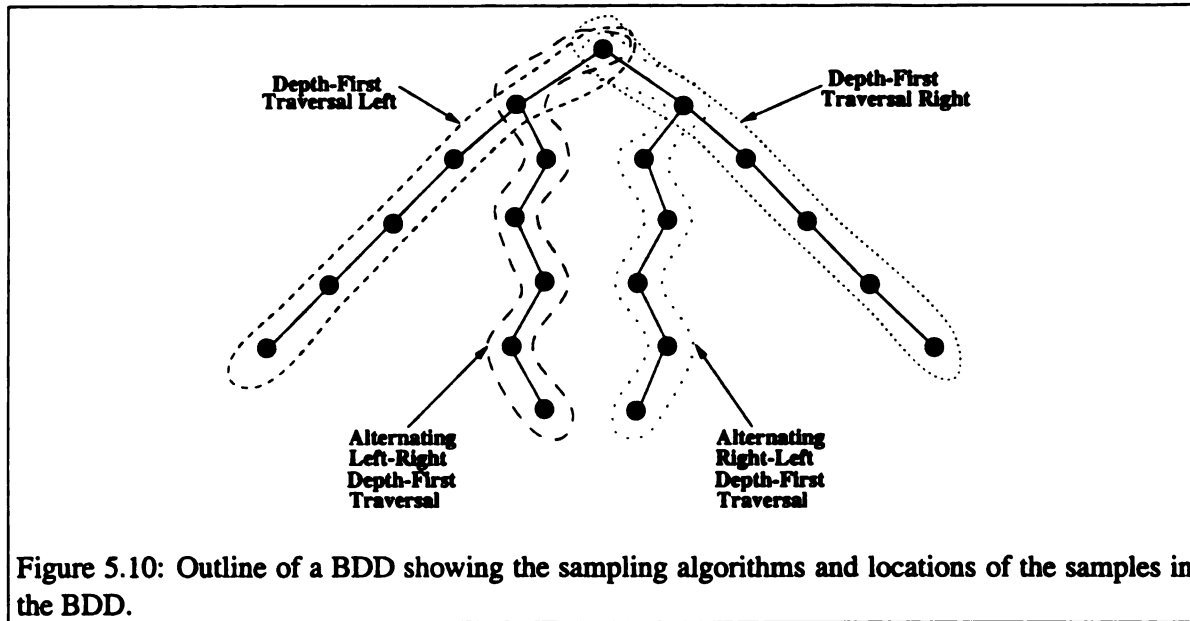
has not caused any problems in identifying the proper indicator nodes and the domain axioms that need to be added to the analysis process.

The indicator nodes method helps us to identify predicates that may be involved in contradictions, however, it does not help us determine which combination of truth assignments is making the predicates contradictory or which domain axioms are needed to eliminate the spurious errors. To help determine this information, we can take samples of the result BDD and present them to the analyst as an AND/OR table.

5.2.2 Sampling

A large BDD may contain millions of satisfying paths. To augment the information obtained from the node profile, the analyst may request samples of several paths from various portions of the BDD. The current tool allows an analyst to take four samples of any desired size (a different sample size can be chosen for each of the four samples) from the BDD. The samples represent paths leading to TRUE (satisfiable paths), and are constructed through a depth-first traversal to the right in the BDD, an alternating right-left traversal down the middle portion of the BDD, an alternating left-right traversal down the middle portion of the BDD, and a depth-first traversal to the left in the BDD (Figure 5.10). The particular samples chosen were made in an attempt to provide the analyst with a good idea of what the different parts of the BDD look like. Currently, the analyst cannot change the sampling algorithms or the locations sampled in the BDD. The combined samples are presented to the analyst as an AND/OR table.

After we identify potential domain axioms from the indicator nodes as described in the previous section, we want to make sure the domain axioms are actually violated in the result BDD. If the domain axioms are not violated, then adding them to the analysis process will only result in increasing the number of satisfying paths, since more satisfiable expressions are being added to the BDD. The



samples allow the analyst to determine if the domain axioms are violated, since the samples show the truth values of the indicator predicates. Once the analyst determines that the domain axioms are actually violated in a significant number of paths in the BDD, the domain axioms can be introduced into the analysis process and the analysis rerun.

Consider the samples shown in Figure 5.11. These samples are from the same result BDD that had the node profile shown in Figure 5.9. The columns in the figure are labeled from 1 – 38. The predicates labeled (1) – (6) were all involved in the domain axioms that needed to be added to the analysis process to eliminate the spurious errors. The original sampling contained 400 samples (100 samples from the right side of the BDD, 100 from the left side of the BDD, 100 from the right middle part of the BDD, and 100 from the left middle part of the BDD). The repeating patterns that can be seen in the samples, occurred many times. For each repeating pattern, only two samples are shown. The domain axioms identified from the indicator nodes (nodes (1) – (4) in the figure), are:

1. The Alt_Reporting states are mutually exclusive,
2. The Other_Air_Status states are mutually exclusive, and
3. The structure of the state machine is such that whenever Other_Air_Status is in state

On_Ground, Other_Alt_Reporting = cTrue cannot be FALSE.

The samples show that domain axiom (1) is violated in columns 1 – 14, 23 – 26, and 31 – 38.

Domain Axiom (2) was violated in columns 1, 2, 5, 6, 9, 10, 13 – 16, 19, 20, 25, 26, 29 – 32, 37, and

38. Domain Axiom (3) was violated in columns 3, 4, 7, 8, 11, 12, 17, 18, and 21 – 24.

Consistency Analysis Result:

Number of Satisfying Paths found: 18457091

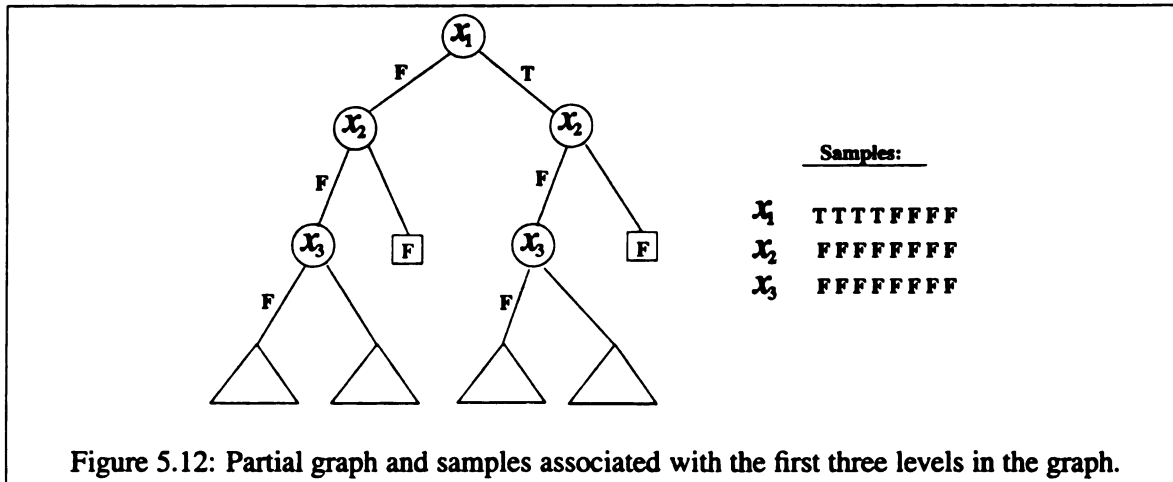
```

-----|
PREDICATES |
-----|
(1) Other_Air_Status IN_STATE State_On_Ground
(2) Alt_Reporting IN_STATE Lost
(3) Alt_Reporting IN_STATE No
(4) Alt_Reporting IN_STATE Yes
    PREV_RA_Inhibit = cTrue
    Other_Tracked_Range() < PROXR
(5) Other_Alt_Reporting = cTrue
    Current_Vertical_Separation() < PROXA
(6) Other_Air_Status IN_STATE State_Airborne
-----|
SAMPLES |
-----|
          1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8
(1) T T T T T T T T T T T T T T T T T T T T T T T T F F F F F F F F
(2) T T T T T T T T T T T T T F F F F F F F F F F F F F F F T T F F T T T
(3) T T T T T T T T T T T T F F T T T T T T T T T T T F F F F F F T T F F F F
(4) T T T T F F F F F F F F T T F F F F F F F F T T T T T T T T T T T T T
    . . . . T T T T F F F F . . T T T T F F F F . . . . . . F F . . . .
    T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T T
(5) T T F F T T F F T T F F T T T T F F T T F F F F T T F F T T T T F F F F T T
    T T . . T T . . T T . . T T T T . . T T . . . . T T . . T T T T . . . . T T
(6) T T . . T T . . T T . . T T T T . . T T . . . . T T . . T T T T . . . . T T

```

Figure 5.11: Partial samples of a result BDD.

The contradictions involving the Alt_Reporting and Other_Air_Status enumerated type predicates are trivial to identify and can be easily eliminated by enabling the decision procedures, but eliminating the contradictions involving predicates (1) and (5) requires the domain axiom information (domain axiom (3)) to be added to the specification. Domain Axiom (3) is an example of a domain axiom for class 4 spurious errors, and is not easy to find without some indication of what predicates are worth investigating. Since the patterns shown in the samples repeated in all 400 samples, all 400 samples represented spurious errors. Once the domain axiom is added into the analysis



and the decision procedures enabled, the BDD reduces from over 18 million satisfying paths to zero satisfying paths (a contradiction) on the next iteration of the symbolic analysis.

If the indicator nodes domain axiom identification technique fails, the analyst can look for repeating patterns in the AND/OR table that may be indicative of domain axioms in the specification. One repeating pattern that may be indicative of domain axioms is a pattern that shows a particular predicate maintains the same truth value a majority of times (for example, in all samples). For example, consider the partial graph shown in Figure 5.12. Assume that for some reason, the analyst is unable to identify any domain axioms using the indicator nodes method. The analyst examines the samples (also shown in the figure) to see if she can find any repeating patterns. She sees that x_2 and x_3 are always FALSE in the samples, checks the specification with regard to the predicates associated with the two nodes, and finds that the two predicates cannot both be FALSE at the same time. She adds this information to the analysis process and reruns the analysis. The results show that the original conjunction (disjunction) was mutually exclusive (satisfiable).

The example shown in Figure 5.12 is trivial, and in actuality the indicator nodes approach would work here. However, let x_2 and x_3 now be arbitrary nodes x_i and x_j in the graph, such that they are not considered indicator nodes when the BDD node profile is examined; i.e., they occur near

the middle level in the BDD (for example, levels 30 and 31 in a BDD with 65 levels). Assume that there are some domain axioms associated with a few of the nodes that do represent indicator nodes, but the analyst finds that there are still a large number of error reports after the identified domain axioms have been added to the analysis and the analysis rerun.

The analyst can then look at the samples from either the initial run or from the current run, and will find that the nodes x_i and x_j have the same truth values in all samples. The analyst will then examine the specification to see if the predicates associated with the nodes are involved in a domain axiom that needs to be added to the analysis process. When the appropriate domain axiom is found, added to the analysis, and the analysis rerun, the results show that the original conjunctive (disjunctive) expression was mutually exclusive (satisfiable). Note that three iterations are not really required here. Since the analyst is looking at the samples to see if the domain axioms related to the indicator predicates are actually violated, she can easily scan the samples for predicates that maintain the same truth value in all or a majority of the samples. Once she locates such predicates, she can check the specification to see if there are any domain axioms related to predicates identified via the samplings. If all domain axioms are found from the initial run results, only two iterations of the analysis are required. We found that in a majority of our test runs, only two iterations were required.

5.3 Summary

In this chapter we defined the term *domain axiom* as used in the context of this research and described a method to assist the analyst in identifying domain axioms. The domain axiom identification process uses the structure of the BDD that results from the initial symbolic analysis run to identify what we call *indicator nodes*. The indicator nodes are nodes that may be indicative of domain axioms in the specification. Both indicator nodes and sampling may be used to help identify

domain axioms, and sampling may be used to verify that the domain axioms are actually violated in the analysis output.

A specification can consist of large numbers of predicates and any of these predicates may have constraints placed on the truth values they can hold in relation to the other predicates. Without some indication of what predicates to look at, all possible combinations of predicates may have to be examined. Therefore, it is important to be able to narrow the search for domain axioms by pointing out predicates that are worth investigating. This is especially important for predicates involved in domain axioms related to the structure of the state machine or related to the environment in which the system will operate. When the appropriate knowledge about domain axioms related to the structure of the state machine or the operating environment is missing from the specification, class 4 spurious errors result. Class 4 spurious errors are the most difficult to identify, since locating them requires intimate knowledge of the system and the specification. The analysis process we developed and described in this chapter helps the analyst identify the domain axioms that are missing from the analysis process by pointing out predicates that may be involved in the domain axioms.

To test our analysis process and domain axiom identification technique, we ran tests on several large expressions from a real-world avionics specification. Our results were quite promising, and are reported in the next chapter.

Chapter 6

Application of Method and Experimental Results

We applied our iterative and integrative analysis and our domain axiom identification processes, to the TCAS (Traffic alert and Collision Avoidance System) II requirements specification. TCAS II is a complex avionics system that is required on all commercial aircraft carrying 30 or more passengers through US Airspace. The system monitors the airspace around the aircraft for other aircraft that may be on a collision course, and takes evasive action if a collision is imminent. Figure 6.1 shows a high-level view of the TCAS controller as specified in RSML. The entire specification for TCAS II (version 6.04A) comprises a little over 400 pages [11]. The requirements consist of two main parts, Own-Aircraft and Other-Aircraft [12]. About 30% of the document is devoted to Own-Aircraft, while the remaining 70% is devoted to Other-Aircraft. Own-Aircraft and Other-Aircraft are two of three parallel states in the state Fully-Operational, which is one of two states in the state Power-On in the TCAS-Controller (Figure 6.1).

We concentrated our efforts on several transitions in the Other-Aircraft portion of the requirements specification (Figure 6.2). We chose the transitions because of the complexity of their

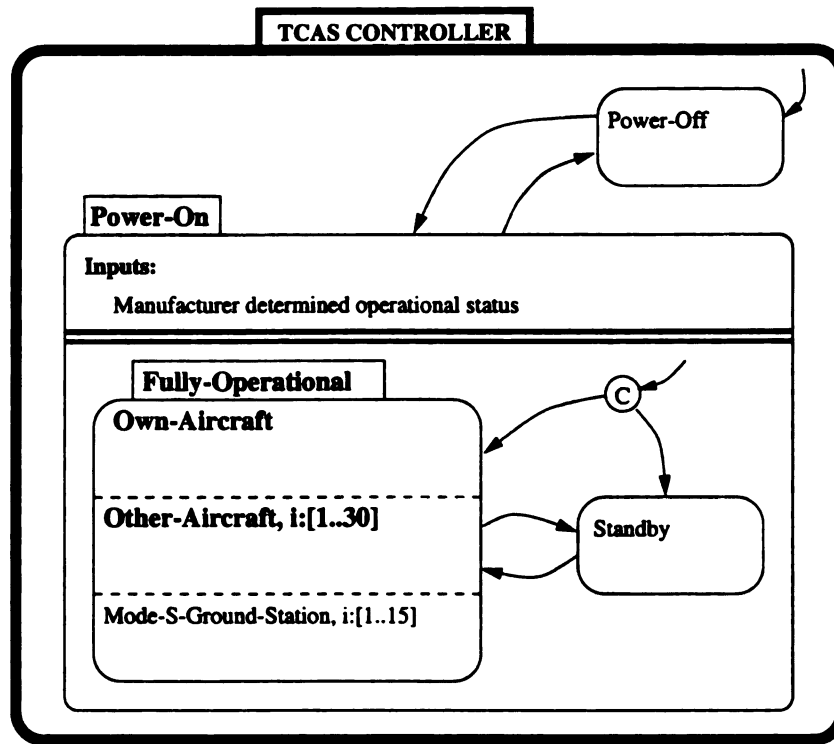


Figure 6.1: RSML specification of TCAS-Controller.

guarding conditions and because it is critical for the specification of these guarding conditions to be correct. Specifically, our analyses efforts focused on one of the most complex portions of the TCAS II requirements specification, the state *Intruder-Status*. The state *Other-Aircraft* consists of four parallel states, one of which is the state *Track-Status* (shown in bold in Figure 6.2). One of the three states comprising *Track-Status* is the state *Tracked* (also shown in bold in the figure). The state *Intruder-Status* is one of nine parallel states in the state *Tracked* (Figure 6.3).

The state *Other-Aircraft* tracks up to thirty aircraft¹ in the vicinity of the aircraft containing the TCAS controller that is monitoring the airspace around the plane. The state *Intruder-Status* (Figure 6.4) is responsible for upgrading or downgrading the threatening status of intruder aircraft (known as other aircraft) within the airspace of the

¹The variable *i* in brackets next to the state name in Figure 6.2 specifies the number of the other aircraft being monitored; thus, *i* ranges from 0 to 30.

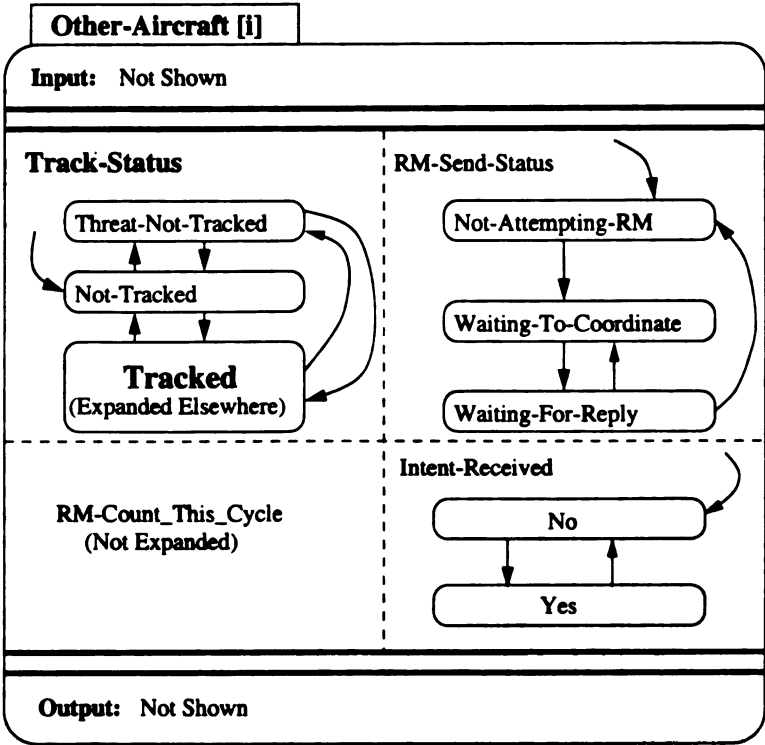


Figure 6.2: RSML partial specification of the state Other-Aircraft.

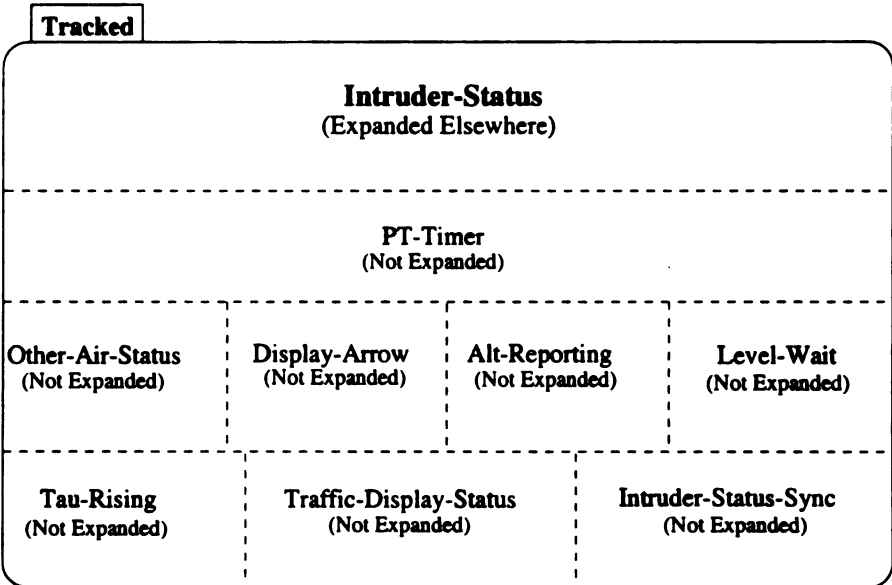


Figure 6.3: RSML partial specification of the state Tracked.

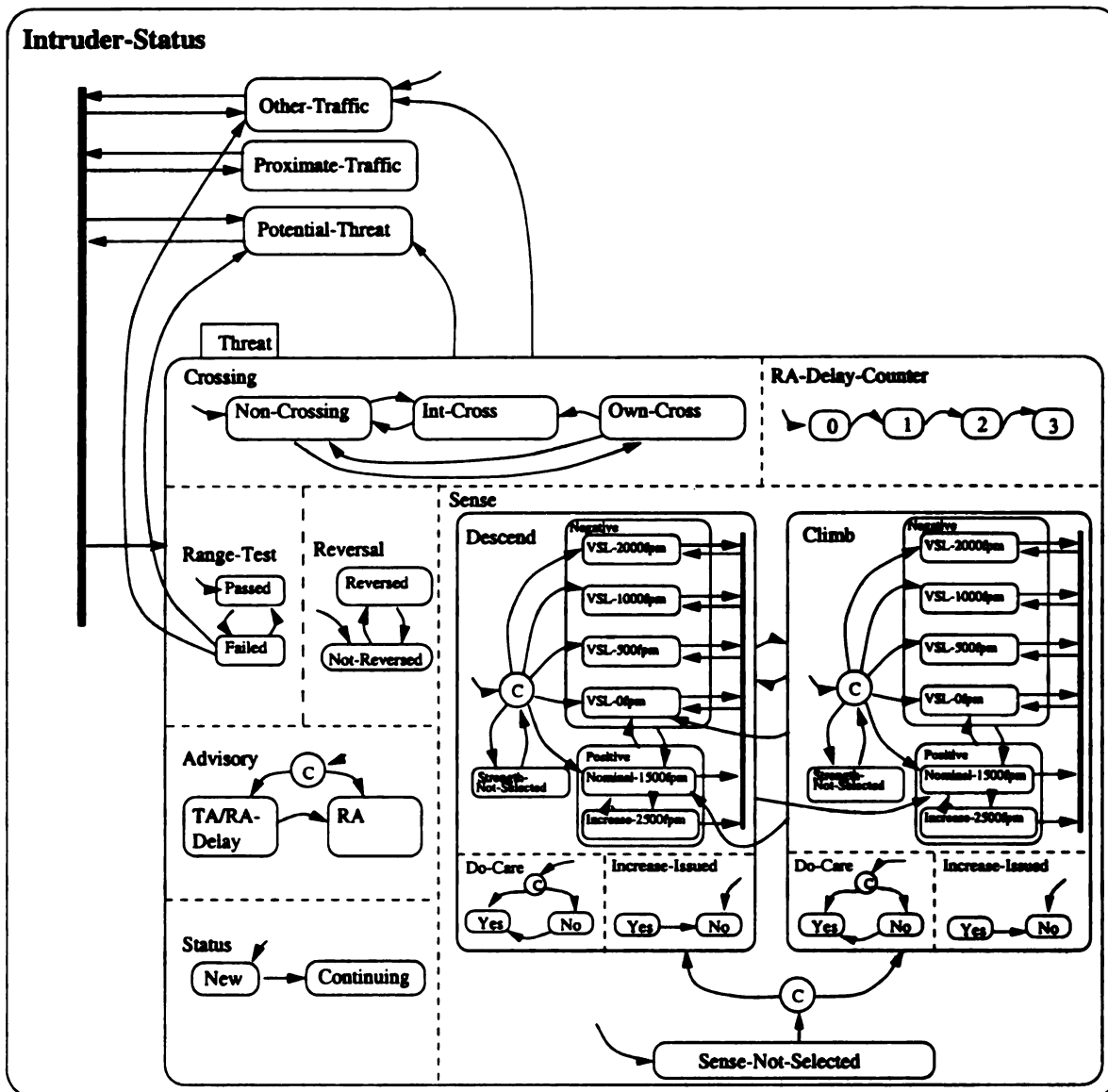


Figure 6.4: RSML specification of the state Intruder-Status.

monitoring aircraft (known as own aircraft); Intruder-Status monitors the status of an intruder aircraft in relation to the aircraft (own aircraft) that is monitoring the airspace for collisions with other traffic in the near vicinity. Intruder-Status contains three atomic states (Other-Traffic, Proximate-Traffic, and Potential-Threat), and one parallel state, Threat. A transition from Proximate-Traffic, Potential-Threat, or Threat to the state Other-Traffic means that the intruder aircraft is no longer in the airspace close to own aircraft but is still being monitored; in other words, the status of the intruder in relation to

the monitoring aircraft has been downgraded. A transition from either Potential-Threat or Threat to Proximate-Traffic signifies that the intruder aircraft is still in the near vicinity and is closer than those other aircraft placed in the Other-Traffic category. When an intruder aircraft is in state Threat, own aircraft is directed to take evasive action to avoid a potential collision. Going from state Threat to any of the other three states represents a downgrading of the status of the intruder. Likewise, going from Potential-Threat to either Other-Traffic or to Proximate-Traffic also represents a downgraded intruder.

Our consistency analyses of the state Intruder-Status included all transitions out of the three atomic states and out of the parallel state to one of the three atomic states. It is important to ensure that the guarding conditions on these transitions are consistent since we do not want to be able to go from a state of heightened alert (i.e., Threat or Potential-Threat) to a state signifying no threat (Other-Traffic or Proximate-Traffic) at the same time we could stay in or move to the highest alert state, Threat. Such an inconsistency could have serious consequences. For example, if an inconsistency exists between the guarding conditions of the transitions Threat to Potential-Threat and Threat to Other-Traffic, the own aircraft might non-deterministically enter state Other-Traffic when the intruder aircraft is still on collision course with own aircraft. Our analysis helps detect such inconsistencies so they can be eliminated. Section 6.1.1 discusses the results we obtained from applying our analysis technique to the transitions described above.

In addition, we examined transitions within the states Auto-SL and Effective-SL. Auto-SL and Effective-SL are parallel states within the state Own-Aircraft (Figure 6.5). Both of these states are responsible for determining the sensitivity level that own aircraft uses to notify a pilot of a potential collision. The higher the sensitivity level, the earlier an alert will be sent to the pilot if an intruder aircraft is on a potential collision course. Therefore, it is important to

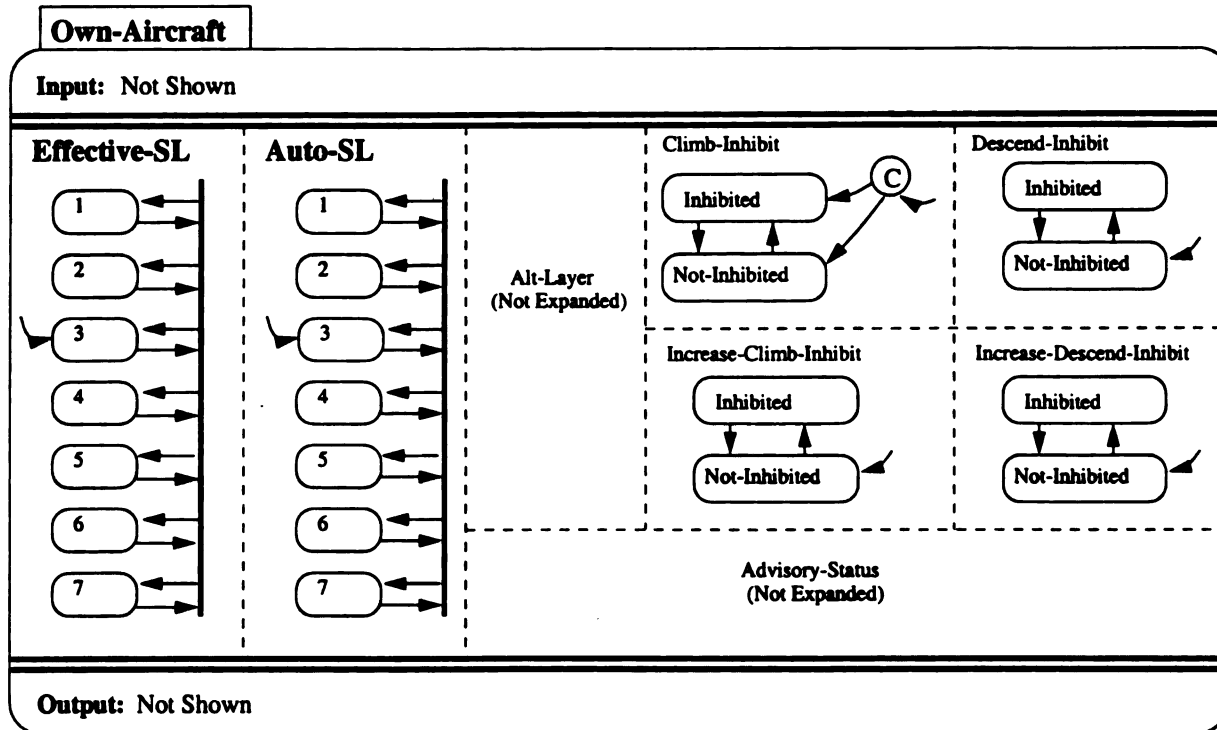


Figure 6.5: RSML partial specification of the state Own-Aircraft.

ensure that a transition cannot non-deterministically occur from a lower to higher sensitivity level at the same time it can occur from a lower to an even lower sensitivity level. For example, if the sensitivity level is currently set to 4, it should not be possible to transition from level 4 to level 2 and to level 7 at the same time. Such a non-deterministic transition could be hazardous if there is another aircraft approaching on collision course.

We applied both our consistency and completeness analysis methods to the specifications of the guarding conditions for the transitions within the states Auto-SL and Effective-SL. We do not describe the consistency analysis results in this document, since the results were comparable, but on a smaller scale, to the results we obtained for the more complex guarding conditions related to the state Intruder-Status. We describe in detail (Section 6.2) the completeness analysis of the specification of the guarding conditions for transitions in the state Auto-SL, and summarize the completeness analysis results we obtained when we checked the Effective-SL guarding

conditions for completeness.

Whenever we applied PVS analysis in our test cases, we used PVS version 2.1 (patch level 2.399).

Each of the test cases we selected demonstrated a particular scenario of using our analysis method and demonstrated the effectiveness of our method when applied to real-world specifications. In Section 6.1 we focus on consistency analysis, and in Section 6.2 we focus on completeness analysis. In Section 6.3 we provide a summary of what each of our test cases demonstrated about our analysis technique and provide some heuristics we learned from our experiments. The heuristics help guide the analyst in using our technique efficiently and effectively.

6.1 Consistency Analysis

In this section, we describe how we used our analysis technique to check pairs of guarding conditions for mutual exclusion.

Section 6.1.1 describes in detail the application of our method and the results obtained when we applied our method to some of the most complex guarding conditions in the TCAS II version 6.04A requirements specification; transitions within the state `Intruder-Status`. In Section 6.1.2 we summarize the results we obtained when we applied our method to the same transitions described in Section 6.1.1, but for the latest version of the TCAS II requirements specification; version 7.0.

In all cases reported in this section, we applied symbolic analysis using BDDs with macro expansion, sampling, and path count enabled. For the symbolic analyses with dynamic variable reordering enabled, we used the sift reordering procedure from Long's BDD library (Chapter 4, Section 4.2.2). In most cases, we reported 400 samples for each consistency check; 100 samples from each of the four sampling algorithms we discussed earlier (Chapter 5, Section 5.2.2).

6.1.1 Transitions in State Intruder-Status - TCAS II Version 6.04A

We did not apply completeness analysis to the guarding conditions associated with the transitions described in this section. This is because the original TCAS II specification was not developed to be complete, and since these guarding conditions are some of the most complex in the specification, there would be too many true incompletenesses to demonstrate anything useful about our technique. Note that the guarding conditions we analyzed in this section could not be checked in the past for consistency because analyzing them resulted in too many spurious errors in the analysis reports; i.e., the analysis reports were too large to be useful to the analyst in identifying any true inconsistencies that might exist in the specification. We verify and demonstrate the problem with spurious errors in our discussions of the results we obtained from a first run analysis of the guarding conditions.

For each of the results reported in this section (with the exception of Section 6.1.1.4), we include the number of satisfying paths found in the result BDDs, the BDD node profile of the result BDDs, the indicator nodes identified, selected samples from the result BDDs, the actions taken as a result of examining the analysis outputs, and the results achieved as a consequence of these actions.

When the pairwise conjunction of two guarding conditions did not reduce to a contradiction, we used our domain axiom identification process to identify information about the system that was missing from the analysis process and causing spurious errors in the analysis output. Once we identified the relevant domain axioms we added the domain axioms back into the analysis process and reran the symbolic analysis using BDDs. At the first use of a domain axiom, we list and discuss the domain axiom, and describe how we identified the domain axiom with our domain axiom identification technique.

If the results of the second run symbolic analysis showed the guarding conditions consistent, then we were finished. If the results of the second run symbolic analysis did not reduce to FALSE and the output was small enough to feed into PVS, we converted the output from the symbolic

analysis into a PVS specification and applied our proof strategies to the reduced output so that PVS' reasoning capabilities and decision procedures could make further reductions that the symbolic analysis component was not capable of making. If the results from the second run analysis using BDDs resulted in a report that was still too unwieldy to be useful to the analyst, and too large to convert to a PVS specification, we added the relevant domain axioms to the PVS specification of the original guarding conditions and invoked our proof strategies on the PVS specifications of the original guarding conditions, augmented with the appropriate domain axioms. In actuality, we applied both symbolic and PVS analysis techniques in all cases, so we could compare, verify, and draw appropriate conclusions from the results about our iterative and integrative analysis method and about our domain axiom identification process.

6.1.1.1 Transitions out of State Proximate-Traffic

Figure 6.4 showed that there are three transitions out of the state *Proximate-Traffic*:

1. *Proximate-Traffic* to *Other-Traffic*,
2. *Proximate-Traffic* to *Potential-Threat* and
3. *Proximate-Traffic* to *Threat*.

We ran our analysis on each pair of guarding conditions individually; note that there are three conjunctive pairs that need to be checked for mutual exclusion. We describe in detail, both the results of our analysis for each of the three pairs of guarding conditions, and the process we used to achieve those results. Two of the transitions guarding conditions proved to be consistent, while the third pair proved to be inconsistent.

6.1.1.1.1 Proximate-Traffic to Threat and Proximate-Traffic to Other-Traffic

Figures 6.6 and 6.7 show the guarding conditions for these two transitions. The predicates subscripted with an *m* represent macro predicates. The *Threat-Condition* macro is shown in

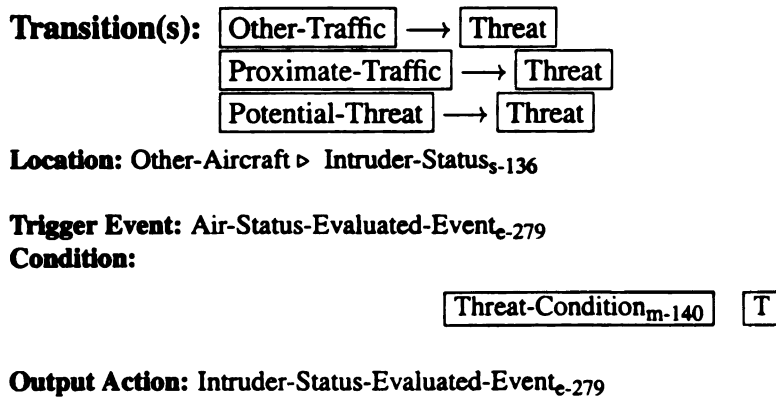
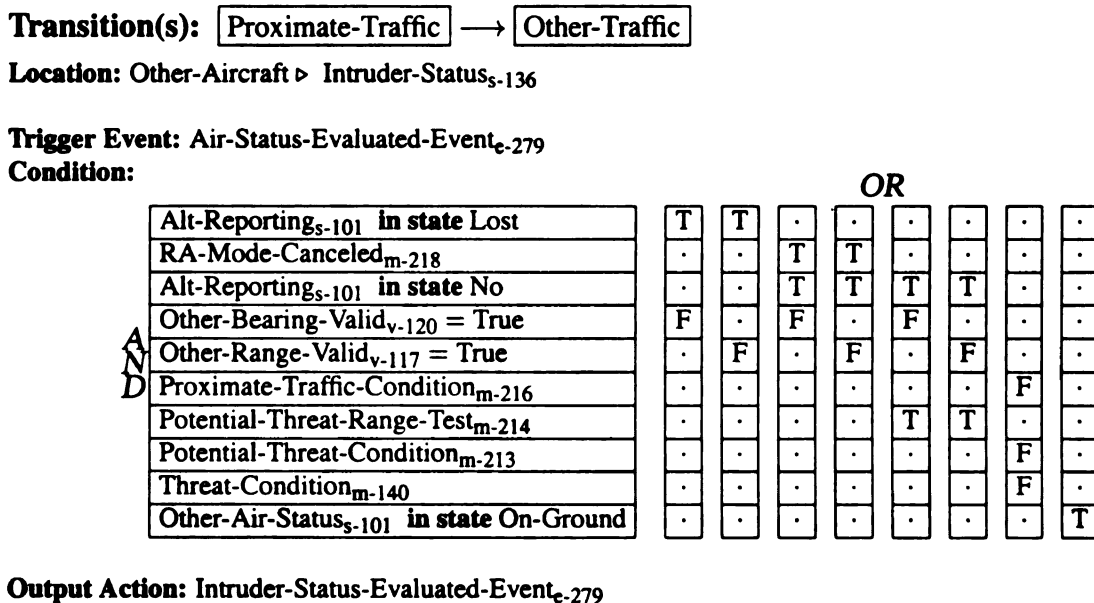


Figure 6.6: Transitions from Other-Traffic, Proximate-Traffic, and Potential-Threat to Threat.

Figure 6.8. As the Threat-Condition macro shows, a macro may contain other macros. Thus, there may be several levels of indirection within the guarding conditions. The Threat macro when fully expanded, is one of the most complex macros in the TCAS II requirements specification. The macros included in the Threat-Condition macro range in size from the smallest, a one column two row table, to the largest, a six column ten row table; the expansion of these macro predicates is included in Appendix A.



Output Action: Intruder-Status-Evaluated-Event_{e-279}

Figure 6.7: Transition from Proximate-Traffic to Other-Traffic.

Macro: Threat-Condition**Definition:**

AND	RA-Inhibit _{m-219}	OR	
	Other-Air-Status _{s-101} in state Airborne		
	Threat-Range-Test _{m-222}		
	Threat-Alt-Test _{m-221}		
	Reply-Invalid-Test _{m-219}		
	TCAS-TCAS-Crossing-Test _{m-220}		
	Level-Wait _{s-101} in state 3		
	Alt-Separation-Test _{m-216}		
	Low-Firmness-Separation-Test _{m-217}		
		F	F
		T	T
		T	T
		T	T
		F	F
		F	F
		F	·
		F	T
		F	F
		F	F

Figure 6.8: The Threat-Condition Macro.

To demonstrate the usefulness of our technique, first consider the results shown in Figure 6.9 that we obtained from a first run symbolic analysis using BDDs without variable reordering enabled. The results show the number of satisfying paths in the result BDD, the BDD node profile which shows the number of nodes at each level in the result BDD, and the total number of nodes in the result BDD. The results in the figure reflect that there were 3,986,640 satisfiable paths in the BDD and 1218 total nodes in the BDD. The 3,986,640 satisfying paths in the BDD imply that there are 3,986,640 inconsistencies in the specification of the guarding conditions. Since the specification was originally designed to be consistent, the analyst can safely conclude that most of the reported inconsistencies are spurious inconsistencies and that there is some information missing from the analysis process that is leading to the spurious error reports. Note that for this particular case there were a total of 56 predicates in the BDD that resulted from conjoining the guarding conditions on the transitions². Any combination of these 56 predicates may be involved in the undetected contradictions that are leading to the spurious error reports.

Consider now the results shown in Figure 6.10 for the same guarding conditions as described

²The total number of predicates was obtained by simply counting the number of levels (predicates) that appear in the node profile of the full result BDD; the full result BDD is not included here due to space limitations.

Number of Satisfying Paths found: 3986640

```

Inhibit_Biased_Climb() > Down_Separation(): 1
( Own_Tracked_Alt() - Other_Tracked_Alt() ) >= MINSEP: 1
Up_Separation() >= ALIM(): 1
( Own_Tracked_Alt() - Other_Tracked_Alt() ) <= n_MINSEP: 1
Down_Separation() >= ALIM(): 1
Other_Air_Status IN_STATE State_Airborne: 4 #
Other_Alt_Reporting = cTrue: 4 #
(Other_Tracked_Range()*Other_Tracked_Range_Rate())<=H1TA(): 4 #
Other_Tracked_Range_Rate() > RDTHRTA: 8 ##
Other_Tracked_Range() <= DMODTA(): 4 #
TAURTA() < TRTHRTA(): 4 #
PREV1_Other_Range_Valid = cTrue: 16 ####
PREV2_Other_Range_Valid = cTrue: 16 ####
Other_Range_Valid = cTrue: 32 #####
Other_Capability = TA_RA: 48 #####
Modified_Tau_Capped() < TFRTHR(): 48 #####
Down_Separation() <= SENSEFIRM(): 48 #####
Up_Separation() <= SENSEFIRM(): 48 #####
Own_Tracked_Alt() < Other_Tracked_Alt(): 48 #####
Own_Tracked_Alt() > Other_Tracked_Alt(): 64 #####
Current_Vertical_Separation() > LOWFIRMZ: 24 #####
Auto_SL IN_STATE ASL_2: 36 #####
Mode_Selector = TA_Only: 36 #####
RA_Inhibit_From_Ground = cTrue: 36 #####
Other_Tracked_Range_Rate() > Zero: 72 #####
Threat_Alt_VMD() < ZT(): 72 #####
ADOT() >= n_ZDTHR(): 72 #####
Time_To_Co_Alt() < TVTHR(): 72 #####
Time_To_Co_Alt() < True_Tau_Capped(): 36 #####
Tau_Rising IN_STATE PLUS_3: 36 #####
Intruder_Status IN_STATE Threat: 36 #####
Other_Tracked_Range() >= DMOD(): 36 #####
Other_Tracked_Range() > DMOD(): 36 #####
Modified_Tau_Capped() < TRTHR(): 36 #####
Other_Tracked_Range() <= RMAX: 36 #####
(Other_Tracked_Range_Rate() * Other_Tracked_Range())>H1(): 18 ####
Other_VRC = No_Intent: 10 ##
Intent_Received IN_STATE IR_No: 10 ##
Own_Tracked_Alt_Rate() <= OLEV: 8 ##
Other_Tracked_Relative_Alt() > MINSEP: 4 #
Other_Tracked_Relative_Alt() < n_MINSEP: 4 #
Own_Tracked_Alt_Rate() >= Zero: 4 #
Other_Tracked_Alt_Rate() >= Zero: 8 ##
Other_Tracked_Alt_Rate() <= OLEV: 4 #
Current_Vertical_Separation() > MAXALTDIFF: 6 #
Current_Vertical_Separation() > MAXALTDIFF2: 4 #
Modified_Tau_Capped() < FRTHR(): 6 #
Level_Wait IN_STATE S3: 2
Alt_Reporting IN_STATE Lost: 4 #
Alt_Reporting IN_STATE No: 2
Other_Bearing_Valid = cTrue: 1
Other_Air_Status IN_STATE State_On_Ground: 1
Total: 1218

```

Figure 6.9: Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Other-Traffic without variable reordering.

above but with variable reordering enabled during the symbolic analysis. From the BDD node profile, the first six nodes and the last node were deemed potential indicator nodes. The BDD node profile and the samples reveal that all satisfying paths in the BDD pass through the first six nodes, and the first five nodes maintain the same truth value on every path (this was determined by examining the truth values of the 400 samples taken). Selected samples are shown in Figure 6.11.

We analyzed the indicator nodes and samples and found several contradictions between the in-state predicate involving `Other-Air-Status`. Further analysis of the results from this run revealed that all of the contradictions involved the first two nodes in the result BDD and the last node in the result BDD. The contradiction related to the first node `Other-Alt-Reporting` was not detected because some relevant structural information about the system was missing from the analysis process; thus, the spurious error reports resulting from the above undetected contradiction were class four spurious errors. We used our indicator nodes technique to identify the following domain axiom related to the first node, and used the samples to verify that the identified domain axiom was actually violated in a majority of samples of the result BDD:

`Other-Alt-Reporting = True iff Alt-Reporting IN-STATE Yes.`

So, how did we find this domain axiom using our indicator nodes method? In the BDD node profile in Figure 6.10, where variable reordering was enabled, the first indicator predicate is `Other-Alt-Reporting = True`. This provides a place for the analyst to start looking for contradictions involving this predicate and some other predicate(s) that may be causing spurious errors in the analysis report. In this case, we would look in the index of the specification for all references to the variable `Other-Alt-Reporting`. In version 6.04A of the specification there are fifteen references to this variable. Several of them can be easily eliminated from consideration for various reasons: (1) they may occur in guarding conditions for transitions not involved in the current analysis, (2) they may be references to the definition of the variable, (3) they may be input

Number of Satisfying Paths found: 4254600

Other_Alt_Reporting = cTrue:	1
Other_Air_Status IN_STATE State_Airborne:	1
Auto_SL IN_STATE ASL_2:	1
Mode_Selector = TA_Only:	1
RA_Inhibit_From_Ground = cTrue:	1
Own_Tracked_Alt() > Other_Tracked_Alt():	1
Own_Tracked_Alt() < Other_Tracked_Alt():	2
Inhibit_Biased_Climb() > Down_Separation():	2
(Own_Tracked_Alt() - Other_Tracked_Alt()) >= MINSEP:	2
Up_Separation() >= ALIM():	2
(Own_Tracked_Alt() - Other_Tracked_Alt()) <= n_MINSEP:	2
Down_Separation() >= ALIM():	2
Current_Vertical_Separation() > LOWFIRMZ:	2
Up_Separation() <= SENSEFIRM():	2
Down_Separation() <= SENSEFIRM():	2
Other_Range_Valid = cTrue:	4 #
Other_Bearing_Valid = cTrue:	4 #
Alt_Reporting IN_STATE Lost:	8 ##
Modified_Tau_Capped() < TFRTHR():	16 ####
Level_Wait IN_STATE S3:	6 #
Other_Tracked_Relative_Alt() < n_MINSEP:	6 #
Other_Tracked_Relative_Alt() > MINSEP:	6 #
Other_Capability = TA_RA:	19 ####
Alt_Reporting IN_STATE No:	7 #
Other_Tracked_Range_Rate() > Zero:	21 #####
Threat_Alt_VMD() < ZT():	21 #####
Current_Vertical_Separation() < ZT():	63 #####
Time_To_Co_Alt() < True_Tau_Capped():	21 #####
Time_To_Co_Alt() < TVTHR():	21 #####
ADOT() >= n_ZDTHR():	21 #####
PREV1_Other_Range_Valid = cTrue:	12 ###
PREV2_Other_Range_Valid = cTrue:	12 ###
Other_Tracked_Range() >= DMOD():	18 #####
Tau_Rising IN_STATE PLUS_3:	18 #####
Intruder_Status IN_STATE Threat:	18 #####
Other_Tracked_Range_Rate() > RDTHRTA:	18 #####
(Other_Tracked_Range()*Other_Tracked_Range_Rate())<=H1TA():	6 #
Other_Tracked_Range() <= DMODTA():	6 #
Other_Tracked_Range() <= RMAX:	12 ###
Modified_Tau_Capped() < TRTHR():	12 ###
Other_Tracked_Range() > DMOD():	12 ###
(Other_Tracked_Range_Rate()*Other_Tracked_Range()) > H1():	12 ###
Other_Tracked_Alt_Rate() > OLEV:	4 #
Own_Tracked_Alt_Rate() <= OLEV:	8 ##
Other_VRC = No_Intent:	6 #
Intent_Received IN_STATE IR_No:	6 #
Other_Tracked_Alt_Rate() <= OLEV:	2
Own_Tracked_Alt_Rate() >= Zero:	2
Other_Tracked_Alt_Rate() >= Zero:	4 #
Current_Vertical_Separation() > MAXALTDIFF:	2
Current_Vertical_Separation() > MAXALTDIFF2:	2
Other_Air_Status IN_STATE State_On_Ground:	1
Total:	484

Figure 6.10: Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Other-Traffic with variable reordering.

This domain axiom can easily be identified by examining the two guarding conditions related to the transitions out of the state `Other-Air-Status` and with a basic knowledge of the semantics of the specification language. The samples revealed that this particular domain axiom was not violated in any of the 400 samples, so one could conclude that this domain axiom is not relevant for the particular guarding conditions under consideration. The remaining three references that required further examination involved the state `Alt-Reporting`.

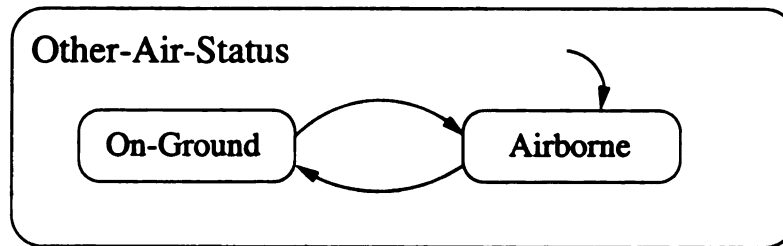


Figure 6.12: RSML specification of state `Other-Air-Status`.

Transition(s): `Airborne` → `On-Ground`

Location: `Other-Air-Statuss-101`

Trigger Event: `Effective-SL-Evaluated-Evente-279`

Condition:

$(\text{Other-Tracked-Alt}_{f-243} - \text{Ground-Level}_{f-237}) \leq 180 \text{ ft}_{(\text{NEARGROL})}$	T
<code>Other-Alt-Reporting_{v-113} = True</code>	T

Output Action: `Air-Status-Evaluated-Evente-279`

Figure 6.13: Guarding condition for transition from `Other-Air-Status` state `Airborne` to `Other-Air-Status` state `On-Ground`.

Transition(s): `On-Ground` → `Airborne`

Location: `Other-Air-Statuss-101`

Trigger Event: `Effective-SL-Evaluated-Evente-279`

Condition:

$(\text{Other-Tracked-Alt}_{f-243} - \text{Ground-Level}_{f-237}) \geq 200 \text{ ft}_{(\text{NEARGROH})}$	OR	
<code>Other-Alt-Reporting_{v-113} = True</code>	T	·
	T	F

Output Action: `Air-Status-Evaluated-Evente-279`

Figure 6.14: Guarding condition for transition from `Other-Air-Status` state `On-Ground` to `Other-Air-Status` state `Airborne`.

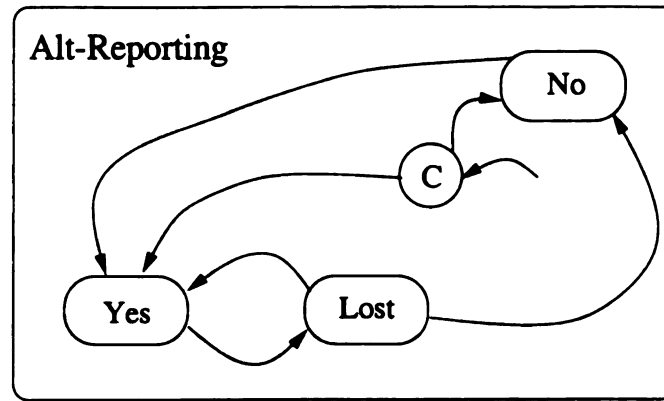


Figure 6.15: RSML specification of state Alt-Reporting.

Figure 6.15 shows the state Alt-Reporting and its child states Yes, Lost, and No. The guarding conditions associated with the transitions in the state Alt-Reporting are shown in Figures 6.16 through 6.20. It is clear from examining the guarding conditions that whenever Other-Alt-Reporting is TRUE Alt-Reporting is in state Yes, and whenever Other-Alt-Reporting is FALSE Alt-Reporting is not in state Yes. Therefore, the domain axiom

Other-Alt-Reporting = True *iff* Alt-Reporting is in state Yes

is identified. This time the samples showed that the identified domain axiom was violated in a significant number of samples; 240 of the 400 samples violated the domain axiom. It should be pointed out here that without intimate knowledge of the specification, it is not obvious that the Other-Alt-Reporting predicate and the Alt-Reporting in-state predicates are related or involved in any domain axioms. Thus, unless an analyst has intimate knowledge of the system specification, it is not obvious where to look for problems without some method to highlight areas to investigate. The complete specification for TCAS II version 6.04A is over 400 pages. With our method, the analyst only had to consider a few pages of the specification before finding the missing information leading to the spurious errors.

Identifying the above domain axiom could eliminate many spurious errors, but the indicator

Transition(s): Yes → Lost

Location: Alt-Reporting_{s-101}

Trigger Event: Effective-SL-Evaluated-Event_{e-279}

Condition:

Other-Alt-Reporting_{v-113} = True F

Output Action: None

Figure 6.16: Guarding condition for transition from Alt-Reporting state Yes to Alt-Reporting state Lost.

Transition(s): Lost → Yes
No → Yes

Location: Alt-Reporting_{s-101}

Trigger Event: Effective-SL-Evaluated-Event_{e-279}

Condition:

Other-Alt-Reporting_{v-113} = True T

Output Action: None

Figure 6.17: Guarding condition for transition from Alt-Reporting states Lost and No to Alt-Reporting state Yes.

Transition(s): Lost → No

Location: Alt-Reporting_{s-101}

Trigger Event: Effective-SL-Evaluated-Event_{e-279}

Condition:

Other-Alt-Reporting_{v-113} = True F

Output Action: None

Figure 6.18: Guarding condition for transition from Alt-Reporting state Lost to Alt-Reporting state No.

Transition(s): © → Yes

Location: Alt-Reporting_{s-101}

Trigger Event: N/A

Condition:

Other-Alt-Reporting_{v-113} = True T

Output Action: None

Figure 6.19: Guarding condition for transition from Alt-Reporting state C to Alt-Reporting state Yes.

Transition(s): © → No

Location: Alt-Reporting_{s-101}

Trigger Event: N/A

Condition:

Other-Alt-Reporting_{v-113} = True F

Output Action: None

Figure 6.20: Guarding condition for transition from Alt-Reporting state C to Alt-Reporting state No.

nodes also indicated that the `Other-Air-Status` in-state predicates also required further investigation. The potential problems with these predicates were more obvious. Samples revealed that the remaining 160 (out of the 400) samples violated the mutually exclusive nature of the `Other-Air-Status` in-state predicates. The contradictions identified in the remaining 160 samples between the second node and the last node, were also not detected because information was missing from the analysis process, but to eliminate these contradictions simply entailed enabling the decision procedures during symbolic analysis; in other words, the analysis process was augmented with information about the mutually exclusive and all-inclusive nature of enumerated types. Identifying the problem related to the second and last node was trivial and did not require any examination of the specification.

Using the information we obtained from the initial symbolic analysis to determine our next course of action, led us to enable the decision procedures in the symbolic analysis to eliminate the contradictions related to the in-state predicates, and to incorporate the identified domain axiom as a macro into the RSML specification (Figure 6.21). The second iteration of the symbolic analysis with

Macro: Other-Alt-Reporting-Alt-Reporting-Assertion

Definition:

		OR		
AND	Other-Alt-Reporting _{v-113} = True	T	F	F
	Alt-Reporting _{s-101} in state Yes	T	F	F
	Alt-Reporting _{s-101} in state Lost	F	T	F
	Alt-Reporting _{s-101} in state No	F	F	T

Figure 6.21: Domain axiom in tabular form for Other-Alt-Reporting, Alt-Reporting assertion.

decision procedures enabled and with the domain axiom included resulted in zero satisfying paths, showing that the initial guarding conditions were consistent. The fact that zero satisfying paths were reported rather than the explicit report that the guarding conditions were consistent shows that the result BDD did not reduce to the constant zero during the symbolic analysis. Rather, while

traversing the BDD to find satisfying paths to convert to AND/OR table format to present to the analyst for review, the traversal routine, using the decision procedures, eliminated all paths in the result BDD.

The run with the decision procedures enabled took a significantly long time (on the order of an hour or so). Without the decision procedures enabled however, we were unable to show the guarding conditions to be consistent with symbolic analysis. When we applied the symbolic analysis with the identified domain axiom added but without the decision procedures enabled, the analysis reported 435,720 satisfying paths in the result BDD.

We attempted PVS analysis on the original guarding conditions without adding any augmenting information; i.e., we did not add the domain axiom that we identified earlier using the process described above. We ran the analysis on a SPARCserver 1000 with 256 MB main memory and four 85MHz CPUs, using the proof strategy shown in Figure 6.22 (Appendix B provides a description of the PVS commands and strategies we used in our PVS analyses). The PVS analysis of the original guarding conditions ran for over a day and we ultimately aborted the process. We then added the domain axiom that we identified using our indicator nodes technique to the PVS specification and reran the PVS analysis on a SPARCstation 20 Model 70, with 64 MB main memory and one 75MHz CPU. With the appropriate information added to the specification, PVS proved the guarding conditions consistent in 44.87 seconds using the PVS proof commands shown in Figure 6.23.

```
(apply (then (rewrite-msg-off)
              (skolem!)
              (auto-rewrite-defs$)
              (do-rewrite$)
              (repeat* (try (bddsimp)
                            (record)
                            (postpone) ) ) ) )
```

Figure 6.22: General PVS strategy to prove two guarding conditions are consistent.

```
(skolemizeandrewrite$) strategy
(lemma "OtherAltReporting_AltReporting_Assertion")
(apply (repeat* (inst?)))
(apply (repeat* (try (bddsimp) (record) (postpone))))
```

Figure 6.23: Specific PVS commands to include a domain axiom into the analysis process and to prove two guarding conditions consistent.

This case shows that both symbolic analysis using BDDs and PVS analysis verified that this pair of guarding conditions was mutually exclusive, but only when the appropriate information was included in the analysis process.

6.1.1.1.2 Proximate-Traffic to Threat and Proximate-Traffic to Potential-Threat

Figures 6.6 and 6.24 show the guarding conditions for the transitions from Proximate-Traffic to Threat and Proximate-Traffic to Potential-Threat. Figure 6.25 shows the results obtained when we applied symbolic analysis without variable



Location: Other-Aircraft▷ Potential-Threat_{s-136}

Trigger Event: Air-Status-Evaluated-Event_{e-279}

Condition:

		OR			
AND	Alt-Reporting _{s-101} in state Yes	F	T	F	T
	Other-Bearing-Valid _{v-120} = True	T	.	T	.
	Other-Range-Valid _{v-117} = True	T	.	T	.
	Potential-Threat-Condition _{m-213}	T	T	.	.
	Threat-Condition _{m-140}	.	F	.	F
	PT-Timer _{s-136} in state 0	.	.	F	F
	Potential-Threat-Range-Test _{m-214}	.	.	F	.

Output Action: Intruder-Status-Evaluated-Event_{e-279}

Figure 6.24: Transitions from Other-Traffic and Proximate-Traffic to Potential-Threat.

reordering. Figure 6.26 shows some of the 400 samples taken from four different portions of the result BDD. The numbers in parentheses preceding each row of truth values in the samples correspond to the parenthesized numbers preceding the predicates in the node profile. Normally,

Number of Satisfying Paths found: 3722880

(1) InhibitBiasedClimb() > DownSeparation():	1
(2) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP:	1
(3) UpSeparation() >= ALIM():	1
(4) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP:	1
(5) DownSeparation() >= ALIM():	1
(6) CurrentVerticalSeparation() < ZTHRTA():	4 #
(7) ADOT() >= nZDTHRTA:	4 #
(8) -((CurrentVerticalSeparation()/ADOT())<TVTHRTATBL():	4 #
(9) OtherAirStatus INSTATE StateAirborne:	8 ##
(10) OtherAltReporting = cTrue:	8 ##
(11) (OtherTrackedRange()*OtherTrackedRangeRate())<=HlTA():	8 ##
(12) OtherTrackedRangeRate() > RDTHRTA:	16 ####
(13) OtherTrackedRange() <= DMODTA():	8 ##
(14) TAURTA() < TRTHRTA():	8 ##
(15) PREV1OtherRangeValid = cTrue:	24 #####
(16) PREV2OtherRangeValid = cTrue:	24 #####
(17) OtherRangeValid = cTrue:	40 #####
(18) OtherCapability = TARA:	48 #####
(19) OwnTrackedAlt() < OtherTrackedAlt():	48 #####
(20) OwnTrackedAlt() > OtherTrackedAlt():	66 #####
(21) CurrentVerticalSeparation() > LOWFIRMZR:	24 #####
(22) AutoSL INSTATE ASL2:	36 #####
(23) ModeSelector = TAOnly:	36 #####
(24) RAIInhibitFromGround = cTrue:	36 #####
(25) CurrentVerticalSeparation() < ZT():	36 #####
(26) OtherTrackedRangeRate() > Zero:	72 #####
(27) TimeToCoAlt() < TVTHR():	72 #####
(28) TimeToCoAlt() < TrueTauCapped():	36 #####
(29) TauRising INSTATE PLUS3:	36 #####
(30) IntruderStatus INSTATE Threat:	36 #####
(31) OtherTrackedRange() >= DMOD():	36 #####
(32) OtherTrackedRange() > DMOD():	36 #####
(33) ModifiedTauCapped() < TRTHR():	36 #####
(34) OtherTrackedRange() <= RMAX:	36 #####
(35) (OtherTrackedRangeRate()*OtherTrackedRange())>H1():	18 ####
(36) OtherVRC = NoIntent:	12 ###
(37) IntentReceived INSTATE IRNo:	12 ###
(38) OwnTrackedAltRate() <= OLEV:	9 ##
(39) OtherTrackedRelativeAlt() < nMINSEP:	6 #
(40) TCASTCASVMD() >= Zero:	6 #
(41) OwnTrackedAltRate() >= Zero:	3
(42) OtherTrackedAltRate() >= Zero:	6 #
(43) OtherTrackedAltRate() <= OLEV:	3
(44) CurrentVerticalSeparation() > MAXALTDIFF:	6 #
(45) CurrentVerticalSeparation() > MAXALTDIFF2:	3
(46) ModifiedTauCapped() < FRTHR():	6 #
(47) LevelWait INSTATE S3:	3
(48) AltReporting INSTATE Yes:	3
(49) OtherBearingValid = cTrue:	2
(50) PTTimer INSTATE PT0:	1
Total: 1293	

Figure 6.25: Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Potential-Threat without variable reordering.

the predicates would appear along with their corresponding samples, but due to space limitations in this document, we used the parenthesized numbers to identify the predicates instead. We now examine the results obtained from symbolic analysis when variable reordering was used, and compare and contrast these results with those obtained without variable reordering.

Figure 6.27 shows the results obtained from symbolic analysis with variable reordering. The result BDD contained 4,909,890 satisfying paths and a total of 390 nodes. From the BDD node profile, we identify the first seven nodes and the last node as potential indicator nodes. The 400 samples taken for the result BDD show that all satisfying paths in the BDD pass through the first seven nodes and that the first six nodes maintain the same truth value on every path. A portion of the 400 samples is shown in Figure 6.28.

We used the same process described earlier to examine the indicator nodes and samples to locate domain axioms in the specification that did not hold in the analysis output. We found that the missing domain axiom involved the first two indicator nodes

- `Alt-Reporting IN-STATE Yes` and
- `Other-Alt-Reporting = True`.

In every one of the 400 samples,

`Alt-Reporting IN-STATE Yes`, predicate (1), was always FALSE and
`Other-Alt-Reporting = True`, predicate (2), was always TRUE;

according to the specification, whenever

`Alt-Reporting IN-STATE Yes` is FALSE,
`Other-Alt-Reporting = True` is FALSE.

We added the identified domain axiom (Figure 6.21) to the machine readable specification as a macro and reran the symbolic analysis using BDDs. The output of the symbolic analysis with the domain axiom added to the analysis process reported that the two guarding conditions were consistent.

In the symbolic analysis run without variable reordering, the two predicates involved in the missing domain axiom were at levels 9 and 55 in the full result BDD (not shown due to space

```
(1) TTTT'TTTTTTTTTTTTTTTTTTTTTTFF'FFFFFFFFFFFFFFFFFFFFF
(2) .....TTT'TTTTTTTTFF'FFFFFFFFF
(3) .....FFF'FFFFFFF.....
(4) TTT'TTTTTTTTFF'FFFFFFFFFFFFF.....
(5) TTT'TTTTTTT.....
(6) TTT'TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFF'FFFFFFFFFFFFF
(7) .....FFF'FFFFFFF
(8) .....FFF'FFFFFFF
(9) TTT'TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(10) TTT'TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(11) TTT'TTTTTTTTFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(12) TTT'TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFF'FFFFFFFFFFFFF
(13) TTT'TTTTTTTT.....
(14) .....FFF'FFFFFFF
(15) TTT'TTTTTTTTFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(16) .....TTT'TTTTTTTTTTTTTTTTTTTTTTTFF'FFFFFFFFFFFFF
(17) TTT'TTTTTTTTFF'FFFFFFFFFFFFFFFFFFFFF.....
(18) TTT'TTTTTTTTTTTTTTFF'FFFFFFFTTTTFF'FFFFFFFFFFFFFFFFFFFFF
(19) .....TTT'TTTTTT.....
(20) TTT'TTTTTTTT.....TTT'TTFF'FFT'TTTT
(21) .....FFF'FFFFFF....FFF'F....FFF'F
(22) FFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(23) FFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(24) FFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(25) TTT'TTTTTTTTFF'FFFFFFFFFFFFFTTTTFF'FFFFFFFFFFFFFFFFFFFFF
(26) TTT'TTTTTTTTFF'FFT'TTTTFF'FFT'TTTTFF'FFFFFFFFFFFFF
(27) .....TTT'TTTTTTTTTT....TTT'TTTTTTTTTTTTTT
(28) .....TTT'TTTTTT....TTT'TTTTTTTTTTTTTT
(29) TTT'TTTTTTTTF'FTTTTTTTTTTTTTTTTF'FTTTTTTTTTTTFF'FTFF'FF
(30) TTT'TTTTTTTT..TF'FFFFFFFFF.TF'FTFF'FF...F....
(31) .....FFF'FFFFFF..FF.FFFF...F....
(32) FFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFTFF'FFFFF
(33) .....FTT'TFF'FFFFF
(34) .....FTT.....
(35) FFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF..FFF'FF
(36) TTT'TTTTTTTTFF'FFF.....FFF'F.....
(37) .....FFF'F.....FFF'F.....
(38) TTT'TTTTTTTT....FFF'FFFF....FFF'F....FFF'F
(39) ..TTT'TFFT.....
(40) ..TTFF..TT.....
(41) .....TF'FFT'TT....TTT'TT....FFF'F
(42) .....TF'FTT'FF....FFF'FT....FF'TT
(43) .....TTT'TT....TTFF.....FF'T
(44) TF'TFTFTFTF....FT'FTT'FF....FT..T....FT..F
(45) .....FT.....FT..
(46) F.F.F.F.F....F..FF.....F.FF.....F.F.
(47) TTTT....TT.....
(48) FFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(49) TTT'TTTTTTT.....
(50) .....FFF'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Figure 6.26: Selected samples from symbolic analysis of Proximate-Traffic to Threat and to Potential-Threat without variable reordering.

Number of Satisfying Paths found: 4909890

(1) AltReporting INSTATE Yes:	1
(2) OtherAltReporting = cTrue:	1
(3) OtherAirStatus INSTATE StateAirborne:	1
(4) AutoSL INSTATE ASL2:	1
(5) RAIInhibitFromGround = cTrue:	1
(6) ModeSelector = TAOnly:	1
(7) OwnTrackedAlt() > OtherTrackedAlt():	1
(8) OwnTrackedAlt() < OtherTrackedAlt():	2 #
(9) InhibitBiasedClimb() > DownSeparation():	2 #
(10) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP:	2 #
(11) UpSeparation() >= ALIM():	2 #
(12) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP:	2 #
(13) DownSeparation() >= ALIM():	2 #
(14) CurrentVerticalSeparation() > LOWFIRMZ:	2 #
(15) CurrentVerticalSeparation() < ZTHRTA():	4 ##
(16) ADOT() >= nZDTHRTA:	4 ##
(17) -((CurrentVerticalSeparation()/ADOT()))<TVTHRTATBL():	4 ##
(18) ModifiedTauCapped() < TFRTHR():	8 #####
(19) LevelWait INSTATE S3:	4 ##
(20) OtherTrackedRelativeAlt() < nMINSEP:	4 ##
(21) TCASTCASVMD() <= Zero:	4 ##
(22) OtherTrackedRelativeAlt() > MINSEP:	4 ##
(23) OtherTrackedRangeRate() > Zero:	10 #####
(24) ThreatAltVMD() < ZT():	10 #####
(25) CurrentVerticalSeparation() < ZT():	30 #####
(26) TimeToCoAlt() < TrueTauCapped():	10 #####
(27) TimeToCoAlt() < TVTHR():	10 #####
(28) ADOT() >= nZDTHR():	10 #####
(29) OtherCapability = TARA:	10 #####
(30) PREV1OtherRangeValid = cTrue:	8 ####
(31) PREV2OtherRangeValid = cTrue:	8 ####
(32) OtherRangeValid = cTrue:	14 #####
(33) OtherTrackedRange() >= DMOD():	15 #####
(34) TauRising INSTATE PLUS3:	15 #####
(35) IntruderStatus INSTATE Threat:	15 #####
(36) OtherBearingValid = cTrue:	12 #####
(37) OtherTrackedRangeRate() > RDTHRTA:	18 #####
(38) OtherTrackedRange() <= DMODTA():	12 #####
(39) (OtherTrackedRange()*OtherTrackedRangeRate())<=H1TA():	12 #####
(40) OtherTrackedRange() > DMOD():	12 #####
(41) (OtherTrackedRangeRate()*OtherTrackedRange())>H1():	12 #####
(42) OtherTrackedAltRate() > OLEV:	4 ##
(43) OwnTrackedAltRate() <= OLEV:	8 ####
(44) OtherVRC = NoIntent:	6 ###
(45) IntentReceived INSTATE IRNo:	6 ###
(46) ModifiedTauCapped() < FRTHR():	4 ##
(47) OtherTrackedAltRate() <= OLEV:	2 #
(48) OwnTrackedAltRate() >= Zero:	2 #
(49) OtherTrackedAltRate() >= Zero:	4 ##
(50) CurrentVerticalSeparation() > MAXALTDIFF:	2 #
(51) CurrentVerticalSeparation() > MAXALTDIFF2:	2 #
(52) PTTimer INSTATE PT0:	1
Total: 390	

Figure 6.27: Partial BDD node profile for conjunction of Proximate-Traffic to Threat and to Potential-Threat with variable reordering.

```

(1) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(2) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(3) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(4) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(5) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(6) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(7) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(8) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(9) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(10) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(11) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.....
(12) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.....
(13) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(14) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(15) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.....
(16) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.....
(17) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(18) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(19) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(20) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.....
(21) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(22) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.....
(23) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(24) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(25) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(26) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.....
(27) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(28) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(29) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(30) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(31) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(32) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(33) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(34) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(35) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(36) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(37) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(38) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(39) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(40) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....
(41) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(42) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(43) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(44) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(45) ..TTF.....TTF.....TTTTTF..TTF.....FFFFFF.....FFF...FFF
(46) TTF..TTTTTTF..TTTTTTTTTF..TTF.....TTTTTTTTTF..TTF..
(47) .....TFFF.....TFFFFFFF.....TTTTTTTTTF..TTF..
(48) .....TTF.....TTF..TTF..TTF..TTF..TTF..TTF..TTF..
(49) .....TTF.....TTF..TTF..TTF..TTF..TTF..TTF..TTF..
(50) F..F..FF..FF..F..FF..F..F..F..F..F..F..F..F..F..F..F..F..
(51) .....FF.....FF..FF.....FF.....FF.....FF.....FF.....FF.....
(52) .....FFFFFFF.....F..FFFFFFF.....FFFF.FFFFFFFF.....

```

Figure 6.28: Selected samples from symbolic analysis of Proximate-Traffic to Threat and to Potential-Threat with variable reordering.

limitations); thus, the node profile without variable reordering would not help the analyst to narrow the search for missing domain axioms or to locate the relevant domain axioms in a timely manner. The analyst may be able to use the samples at this point to look for predicates that maintain the same truth values in all samples or in a majority of samples, and check these predicates to see if they are involved in domain axioms in the specification that are violated in all or a majority of the samples. In this case, there are six predicates that maintain the same truth values in all samples, and there are several other predicates that maintain the same truth values in a majority of samples. Clearly, the search is more focused and more timely when our indicator nodes technique is applied. We now show that the identified domain axiom is also required in the analysis of the guarding conditions when PVS analysis is applied.

We ran PVS analysis on the full guarding conditions without domain axioms added and using the strategy shown in Figure 6.22, on a SPARCserver 1000 with 256 MB main memory and four 85MHz CPUs. Again, as with the previously discussed transitions, we aborted the analysis after it ran on the order of a day. We added the domain axiom we identified using the indicator nodes and samples to the PVS specification of the guarding conditions, and reran the PVS analysis on a SPARCstation 20 with 64 MB of main memory and one 75MHz CPU. We used the proof commands in Figure 6.23 and proved the guarding conditions consistent in 60.69 seconds.

As in the previous case, both symbolic analysis using BDDs and PVS analysis verified that this pair of guarding conditions was mutually exclusive, but only after the appropriate information was included in the analysis process.

6.1.1.1.3 Proximate-Traffic to Other-Traffic and Proximate-Traffic to Potential-Threat

The guarding conditions for the transitions Proximate-Traffic to Other-Traffic and Proximate-Traffic to Potential-Threat were shown in Figures 6.7 and 6.24. Our

analysis of the guarding conditions for these transitions showed that the specifications of the guarding conditions were inconsistent. The results obtained from symbolic analysis without variable reordering are shown in Figure 6.29.

As in the previous two analyses for transitions out of state `Proximate-Traffic`, the BDD node profile generated when symbolic analysis without variable reordering was applied, gives no indication as to what particular predicates might be involved in the undetected contradictions that are leading to spurious error reports. Figure 6.30 shows a portion of the 400 samples reported to the analyst. Notice that there are no predicates that maintain the same truth values in all of the samples. This is because there are true inconsistencies between the two guarding conditions, that are mixed in with the spurious inconsistencies; this makes it even more difficult to identify the information that is missing from the analysis and leading to spurious errors in the analysis output.

Consider now, the results obtained from symbolic analysis with variable reordering (Figure 6.31). With variable reordering, the analysis reported 46,665,726 potential inconsistencies and the result BDD contained 932 nodes. There are still too many error reports to show to the analyst and for the analyst to manually inspect. However, the analyst can now use our method to identify any information that is missing from the analysis process and that is leading to spurious error reports in the analysis output. The BDD node profile obtained with variable reordering shows that the first three nodes are indicator nodes (they occur along every path in the BDD, and thus, occur in every reported inconsistency), and that the last node is a potential indicator node. Rather than the analyst having to check some combination of 65 predicates (the full result BDD contained 65 predicates) for undetected contradictions, the analyst can now concentrate on only four predicates, starting with the first three. The indicator predicates associated with the first three indicator nodes are:

- `Other-Air-Status IN-STATE On-Ground`,
- `Alt-Reporting IN-STATE Yes`, and
- `Other-Alt-Reporting = True`.

Number of Satisfying Paths found: 99048816

(1) PREVRAInhibit = cTrue:	1
(2) InhibitBiasedClimb() > DownSeparation():	2
(3) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP:	2
(4) UpSeparation() >= ALIM():	2
(5) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP:	2
(6) DownSeparation() >= ALIM():	2
(7) CurrentVerticalSeparation() < ZTHRTA():	8
(8) ADOT() >= nZDTHRTA:	8
(9) -((CurrentVerticalSeparation()/ADOT()))<TVTHRTATBL():	8
(10) OtherAirStatus INSTATE StateAirborne:	16 #
(11) OtherTrackedRange() < PROXR:	18 #
(12) OtherAltReporting = cTrue:	27 ##
(13) OwnTrackedAlt() >= ABOVNMC:	3
(14) CurrentVerticalSeparation() < PROXA:	16 #
(15) (OtherTrackedRange()*OtherTrackedRangeRate())<=H1TA():	28 ##
(16) OtherTrackedRangeRate() > RDTHRTA:	56 #####
(17) OtherTrackedRange() <= DMODTA():	28 ##
(18) TAURTA() < TRTHRTA():	28 ##
(19) PREV1OtherRangeValid = cTrue:	48 ####
(20) OtherRangeValid = cTrue:	100 #####
(21) OtherCapability = TARA:	144 #####
(22) ModifiedTauCapped() < TFRTHR():	144 #####
(23) OwnTrackedAlt() > OtherTrackedAlt():	192 #####
(24) AutoSL INSTATE ASL2:	110 #####
(25) ModeSelector = TAOnly:	110 #####
(26) RAIInhibitFromGround = cTrue:	110 #####
(27) CurrentVerticalSeparation() < ZT():	90 #####
(28) OtherTrackedRangeRate() > Zero:	180 #####
(29) TauRising INSTATE PLUS3:	90 #####
(30) IntruderStatus INSTATE Threat:	90 #####
(31) OtherTrackedRange() > DMOD():	90 #####
(32) (OtherTrackedRangeRate()*OtherTrackedRange())>H1():	45 ####
(33) OtherVRC = NoIntent:	25 ##
(34) OwnTrackedAltRate() <= OLEV:	20 ##
(35) OtherTrackedAltRate() > OLEV:	10 #
(36) OtherTrackedRelativeAlt() > MINSEP:	10 #
(37) TCASTCASVMD() <= Zero:	10 #
(38) OtherTrackedRelativeAlt() < nMINSEP:	10 #
(39) TCASTCASVMD() >= Zero:	10 #
(40) CurrentVerticalSeparation() > MAXALTDIFF:	15 #
(41) ModifiedTauCapped() < FRTHR():	15 #
(42) LevelWait INSTATE S3:	5
(43) AltReporting INSTATE Lost:	10 #
(44) AltReporting INSTATE No:	6
(45) OtherBearingValid = cTrue:	9
(46) OtherAirStatus INSTATE StateOnGround:	6
(47) AltReporting INSTATE Yes:	3
(48) PTTimer INSTATE PT0:	1
Total: 3491	

Figure 6.29: Partial BDD node profile for conjunction of Proximate-Traffic to Other-Traffic and to Potential-Threat without variable reordering.

(1) TTTTTTTTTTTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(2) TTTTTTTTTTTTTTTTTTFFFFFFFTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFF
(3)TTTTT.....FFFFFFFFFFFFFFFFFFFFFFFFFFFF
(4)FFFFF.....
(5) TTTTTTTTTTTTTTT.....FFFFFFFFFFFFF.....
(6) TTTTTTTTTTTTTTT.....
(7) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFF
(8)FFFFFFFFFFFFFFFFFFFFFFFFFFFF
(9)FFFFFFFFFFFFFFFFFFFFFFFFFFFF
(10) TTTTTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(11) TTTTTTTTTTTTTTTTTTTTTTTFTTTTTTTTTTTTTTFFFFFFFFFTTTTTTTTTTTTT
(12) TTTTTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFFFFFTTTTFFFFFFFFFFFFFF
(13)TTFFFTTTTTTFFFFFFFFFFFFF.....FFFFFFFFFFFFF
(14) TTTTTTTTTTTTTTT.....
(15) TTTTTTTTTTTTTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFTTTFFFFFFFFFFFFF
(16) TTTTTTTTTTTTTTTTTTFFTTTTFFTFTTFFFFFFFFFTFFFFFFFFFFFFFFFFFFFF
(17) TTTTTTTTTTTTTTT...TT.
(18)FTT....FTTT...FFFFFFFFTT.FTTTFFFFFFFFFTTTTT
(19) TTTTTTTTTTTTTTT.....
(20) TTTTTTTTTTTTTTT.TFFT..TFFTTTTTTT.FFF..FTTTTTTTTTTTTTTT
(21) TTTTTTTTTTTTTTT.....
(22) TTTTTTTTTTTTTTT.....
(23) TTTTTTTTTTTTTTT.....
(24) TTFFFFFFFFFFFFF.....FFTFFFFF.....FFFFFTT....
(25) ..TFFFFFFFFFFFFF.....TT.TFFFFF.....FTTTTT.....
(26) ...TFFFFFFFFFFFFF.....FFTT.....T.....
(27)TTTTTTTTT.....
(28)TTTTTTTTT.....
(29)TTTTTTTTT.....
(30)TTTTTTTTT.....
(31)TFFFFFFFFF.....
(32)TFFFFF.....
(33)TTTTT.....
(34)TTTTT.....
(35)TTTTT.....
(36)TTTTT.....
(37)TTTTT.....
(38)
(39)
(40)TTTFF.....
(41)TTF.....
(42)TFF.....
(43) TTFFTTFFTTF.FF...FFF....FFFTTTF.FFT....TFFTTFFFFTT
(44) .FF..FF..FF.TF...TFT....FTT..TT.FT.....FTT..FTFT..
(45) TFTFTFTFTFTFT.FT..T..FT...FFFTT.....FTT.FTFT.FTFT
(46) T.TTT.TTT.TTT.T...TT...T....TT.T.....TT.T.TT.T.T.T
(47) .T.T.T.T.T.TFT..T.TT...T.T.....TTT..TT.....T.T.
(48)FFF...FFFFFFFFFFFFFFFFF...FFFFFFFFFFFFFFFF

Figure 6.30: Portion of samples from symbolic analysis of Proximate-Traffic to Other-Traffic and to Potential-Threat without variable reordering.

Number of Satisfying Paths found: 46665726

(1) OtherAirStatus INSTATE StateOnGround:	1
(2) AltReporting INSTATE Yes:	2
(3) OtherAltReporting = cTrue:	4 #
(4) OtherAirStatus INSTATE StateAirborne:	4 #
(5) AutoSL INSTATE ASL2:	5 #
(6) ModeSelector = TAOnly:	5 #
(7) RAINhibitFromGround = cTrue:	5 #
(8) OwnTrackedAlt() > OtherTrackedAlt():	3
(9) OwnTrackedAlt() < OtherTrackedAlt():	6 #
(10) InhibitBiasedClimb() > DownSeparation():	6 #
(11) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP:	6 #
(12) UpSeparation() >= ALIM():	6 #
(13) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP:	6 #
(14) DownSeparation() >= ALIM():	6 #
(15) CurrentVerticalSeparation() > LOWFIRMZR:	6 #
(16) UpSeparation() <= SENSEFIRM():	6 #
(17) DownSeparation() <= SENSEFIRM():	6 #
(18) OtherRangeValid = cTrue:	21 #####
(19) OtherBearingValid = cTrue:	17 ####
(20) AltReporting INSTATE Lost:	22 #####
(21) ModifiedTauCapped() < TFRTHR():	22 #####
(22) LevelWait INSTATE S3:	8 ##
(23) OtherTrackedRelativeAlt() < nMINSEP:	8 ##
(24) OtherTrackedRelativeAlt() > MINSEP:	8 ##
(25) OtherTrackedRangeRate() > Zero:	26 #####
(26) CurrentVerticalSeparation() < ZT():	78 #####
(27) TimeToCoAlt() < TrueTauCapped():	26 #####
(28) TimeToCoAlt() < TVTHR():	26 #####
(29) ADOT() >= nZDTHR():	26 #####
(30) AltReporting INSTATE No:	13 ###
(31) CurrentVerticalSeparation() < ZTHRTA():	24 #####
(32) ADOT() >= nZDTHRTA:	24 #####
(33) -((CurrentVerticalSeparation()/ADOT())) < TVTHRTATBL():	24 #####
(34) CurrentVerticalSeparation() < PROXA:	32 #####
(35) OtherTrackedRange() < PROXR:	32 #####
(36) OtherTrackedRange() >= DMOD():	30 #####
(37) TauRising INSTATE PLUS3:	30 #####
(38) IntruderStatus INSTATE Threat:	30 #####
(39) OtherTrackedRangeRate() > RDTHRTA:	34 #####
(40) (OtherTrackedRange()*OtherTrackedRangeRate())<=H1TA():	28 #####
(41) OtherTrackedRange() <= DMODTA():	28 #####
(42) (OtherTrackedRangeRate()*OtherTrackedRange()) > H1():	12 ###
(43) OtherTrackedAltRate() > OLEV:	4 #
(44) OwnTrackedAltRate() <= OLEV:	8 ##
(45) OtherVRC = NoIntent:	6 #
(46) IntentReceived INSTATE IRNo:	6 #
(47) OtherTrackedAltRate() <= OLEV:	2
(48) OwnTrackedAltRate() >= Zero:	2
(49) OtherTrackedAltRate() >= Zero:	4 #
(50) CurrentVerticalSeparation() > MAXALTDIFF:	2
(51) ModifiedTauCapped() < FRTHR():	2
(52) PTTimer INSTATE PT0:	1
Total: 932	

Figure 6.31: Partial BDD node profile for conjunction of Proximate-Traffic to Other-Traffic and to Potential-Threat with variable reordering.

We know from the first example discussed (Section 6.1.1.1.1) that there are domain axioms involving the predicates

- Alt-Reporting IN-STATE Yes and Other-Alt-Reporting = True, and
- Other-Alt-Reporting = True and Other-Air-Status IN-STATE On-Ground.

The domain axioms, respectively, are:

- Other-Alt-Reporting = True *iff* Alt-Reporting IN-STATE Yes, and
- *if* Other-Alt-Reporting is FALSE, *then* Other-Air-Status *cannot be in the state* On-Ground.

If we did not have the information about the above two domain axioms from a previous analysis, we would use the same technique described in Section 6.1.1.1.1 to identify the domain axioms involving the indicator predicates. We also know that the variable Other-Air-Status cannot be in states On-Ground and Airborne at the same time, and that it must be in at least one of the states On-Ground or Airborne. The indicator predicate Other-Air-Status IN-STATE On-Ground occurs at level 0 (the root) in the result BDD, and the predicate Other-Air-Status IN-STATE Airborne occurs at level 3 in the result BDD. The root node necessarily occurs along every satisfying path in the result BDD. The related predicate at level 3 occurs four times in the result BDD; note that at level 3, the maximum number of times the predicate can occur is 2^3 or 8.

Figure 6.32 shows a portion of the 400 samples from the result BDD for the first four predicates. If we examine all of the 400 samples, we see that in the first 200 samples the root node is always TRUE and in the last 200 samples the root node is always FALSE. The majority of the first 200 samples for the three nodes following the root are TRUE. The last 200 samples of the predicate Other-Air-Status IN-STATE Airborne are all don't cares revealing that this predicate does not occur on the FALSE side of the root node Other-Air-Status IN-STATE On-Ground. This means that in most of the first 200 samples, both of the Other-Air-Status

PREDICATES

- ```
(1) OtherAirStatus INSTATE StateOnGround
(2) AltReporting INSTATE Yes
(3) OtherAltReporting = cTrue
(4) OtherAirStatus INSTATE StateAirborne
```

## SAMPLES

- [illegible]

**Figure 6.32: Selected samples from symbolic analysis of Proximate-Traffic to Other-Traffic and to Potential-Threat with variable reordering.**

in state predicates are TRUE – a contradiction that was not detected during the symbolic analysis since we did not enable our decision procedures. The last 200 samples reveal that the predicate `Other-Alt-Reporting = True` is always FALSE. In addition, the last 200 samples reveal that the predicate `Alt-Reporting IN-STATE Yes` is always TRUE in the first 100 (of the last 200) samples, and always FALSE in the last 100 (of the last 200) samples. The information revealed in the last 200 samples showed that the identified domain axiom involving `Other-Alt-Reporting = True` and `Alt-Reporting IN-STATE Yes` was violated in 100 of the 200 samples. Information from the samples also revealed that the domain axiom involving `Other-Alt-Reporting` and `Other-Air-Status` was violated in only twelve of the 400 samples.

We added the domain axiom `Other-Alt-Reporting = True iff Alt-Reporting IN-STATE Yes` (Figure 6.21), enabled the decision procedures, and reran the symbolic analysis. We did not add the third domain axiom, *if Other-Alt-Reporting is FALSE, then Other-Air-Status cannot be in the state On-Ground*, identified above because it was only violated in twelve of the 400 samples. The results from the symbolic analysis run with variable reordering, decision procedures enabled, and with the domain axiom included, are shown in

Figure 6.33.

As the results show, the total number of inconsistencies detected is now 7,540,053, a reduction of 39,125,673; this means that adding the domain axioms identified above and enabling the decision procedures eliminated 39,125,673 spurious inconsistency reports. While this reduction is significant and exemplary, there are still too many inconsistency reports to be presented to the analyst and for the analyst to manually inspect.

The BDD node profile that results from the symbolic analysis with domain axioms cannot be used further to locate potential indicator nodes, since the predicates involved in the domain axiom have now become the most dominant predicates in the analysis output and overshadow any remaining predicates that might be involved in undetected contradictions. Recall that we did not add all of the domain axioms identified above, since the domain axiom involving the predicates `Other-Alt-Reporting = True` and `Other-Air-Status IN-STATE On-Ground` was only violated in twelve of the 400 samples. We examined the new samples generated from the symbolic analysis run with the other domain axiom and with the information about enumerated type predicates added, and found that 48 (half of which were redundant) of the 400 samples now showed the previously excluded domain axiom violated. Though the number of violations is not significant, we added the domain axiom (Figure 6.34) along with the previously added domain axiom, turned on the decision procedures, and reran the symbolic analysis. As expected, the results were not much different from the run without this last domain axiom added. The results reported 7,540,029 potential inconsistencies versus the previously reported 7,540,053; a reduction of only 24.

Since we could not identify any additional domain axioms, we concluded that the symbolic analysis was generating many redundant true inconsistency reports that could be collapsed into fewer reports if the proper Boolean reduction techniques were applied. Our symbolic analysis component

Number of Satisfying Paths found: 7540053

|                                                            |         |
|------------------------------------------------------------|---------|
| AltReporting INSTATE Lost:                                 | 1       |
| AltReporting INSTATE No:                                   | 2 #     |
| OtherAirStatus INSTATE StateOnGround:                      | 3 #     |
| OtherAltReporting = cTrue:                                 | 5 ##    |
| OtherAirStatus INSTATE StateAirborne:                      | 5 ##    |
| AltReporting INSTATE Yes:                                  | 5 ##    |
| OtherBearingValid = cTrue:                                 | 3 #     |
| AutoSL INSTATE ASL2:                                       | 3 #     |
| ModeSelector = TAOnly:                                     | 3 #     |
| OwnTrackedAlt() > OtherTrackedAlt():                       | 1       |
| OwnTrackedAlt() < OtherTrackedAlt():                       | 2 #     |
| ( OwnTrackedAlt() - OtherTrackedAlt() ) >= MINSEP:         | 2 #     |
| UpSeparation() >= ALIM():                                  | 2 #     |
| ( OwnTrackedAlt() - OtherTrackedAlt() ) <= nMINSEP:        | 2 #     |
| UpSeparation() <= SENSEFIRM():                             | 2 #     |
| DownSeparation() <= SENSEFIRM():                           | 2 #     |
| OtherRangeValid = cTrue:                                   | 8 ####  |
| PREVRAInhibit = cTrue:                                     | 1       |
| TCASTCASVMD() >= Zero:                                     | 2 #     |
| OtherTrackedRelativeAlt() < nMINSEP:                       | 2 #     |
| TCASTCASVMD() <= Zero:                                     | 2 #     |
| OtherTrackedRelativeAlt() > MINSEP:                        | 2 #     |
| OtherCapability = TARA:                                    | 7 ###   |
| TimeToCoAlt() < TrueTauCapped():                           | 7 ###   |
| TimeToCoAlt() < TVTHR():                                   | 7 ###   |
| ADOT() >= nZDTHR():                                        | 7 ###   |
| CurrentVerticalSeparation() < ZTHRTA():                    | 8 ####  |
| ADOT() >= nZDTHRTA:                                        | 8 ####  |
| -((CurrentVerticalSeparation() / ADOT())) < TVTHRTATBL():  | 8 ####  |
| CurrentVerticalSeparation() < PROXA:                       | 16 #### |
| OwnTrackedAlt() >= ABOVNMC:                                | 3 #     |
| OtherTrackedRange() < PROXR:                               | 16 #### |
| PREV1OtherRangeValid = cTrue:                              | 8 ####  |
| PREV2OtherRangeValid = cTrue:                              | 8 ####  |
| OtherTrackedRange() >= DMOD():                             | 12 #### |
| TauRising INSTATE PLUS3:                                   | 12 #### |
| IntruderStatus INSTATE Threat:                             | 12 #### |
| OtherTrackedRangeRate() > RDTHRTA:                         | 14 #### |
| (OtherTrackedRange() * OtherTrackedRangeRate()) <= H1TA(): | 8 ####  |
| OtherTrackedRange() <= DMODTA():                           | 8 ####  |
| ModifiedTauCapped() < TRTHR():                             | 6 ###   |
| OtherTrackedRange() > DMOD():                              | 6 ###   |
| ( OtherTrackedRangeRate() * OtherTrackedRange() ) > H1():  | 6 ###   |
| OtherTrackedAltRate() > OLEV:                              | 2 #     |
| OwnTrackedAltRate() <= OLEV:                               | 4 ##    |
| OtherVRC = NoIntent:                                       | 3 #     |
| IntentReceived INSTATE IRNo:                               | 3 #     |
| ModifiedTauCapped() < FRTHR():                             | 2 #     |
| OtherTrackedAltRate() >= Zero:                             | 2 #     |
| CurrentVerticalSeparation() > MAXALTDIFF:                  | 1       |
| PTTimer INSTATE PT0:                                       | 1       |
| Total: 333                                                 |         |

Figure 6.33: Partial BDD node profile for conjunction of Proximate-Traffic to Other-Traffic and to Potential-Threat with variable reordering and domain axioms.

## Macro: Other-Alt-Reporting-Other-Air-Status-Assertion

**Definition:**

|               |                                                      |    |   |
|---------------|------------------------------------------------------|----|---|
| $\frac{A}{D}$ | Other-Alt-Reporting <sub>v-113</sub> = True          | OR |   |
|               | Other-Air-Status <sub>s-101</sub> in state On-Ground | T  | F |

Figure 6.34: Domain axiom for Other-Alt-Reporting, Other-Air-Status assertion

with its simple decision procedures for enumerated type predicates is not capable of handling this reduction task, so we converted the original guarding conditions to PVS specifications and ran PVS analysis on them with no domain axioms added to the PVS specification. The analysis ran on a Ross-based SPARC 20 with two 150MHz CPUs for over three hours<sup>3</sup>, and yielded 1129 unprovable subgoals (each unprovable subgoal represents a potential inconsistency in the specification of the guarding conditions). Though this is still a large number of potential inconsistencies, it is a significant reduction from what we were able to achieve using symbolic analysis with added domain axioms and our decision procedures. The analysis report is now at least within reasonable size for an analyst to manually inspect. Figure 6.35 shows some of the 1129 unprovable subgoals.

We assumed that many of the inconsistency reports might be spurious because the PVS specification lacked the knowledge about the system that was related to the domain axioms identified using the results from the symbolic analysis. We added the domain axioms into the PVS specification and reran the PVS analysis using proof commands to introduce the axioms into the proof process. The proof process (shown in Figure 6.36) yielded 808 unprovable subgoals (representing 808 potential inconsistencies). The proof completed in a little over 2 hours on a 200MHz Pentium Pro machine with 64MB of RAM running Redhat Linux. We manually checked all of the reported inconsistencies and they appeared to be true inconsistencies.

Manually analyzing the predicates and the predicate's truth values in the reported inconsistencies, reveals that only a few predicates may be leading to all of the inconsistencies, but the number

<sup>3</sup>The time to complete the proof is dependent on the system load. The time reported was the fastest time; a subsequent run ran for a little over nine hours.

```

SUBGOAL.1 :
{-1} StateOnGround?(OtherAirStatus!1)
{-2} cTrue?(OtherRangeValid!1)
{-3} cTrue?(OtherBearingValid!1)
|-----
{1} PT0?(PTTimer!1)

SUBGOAL.2 :
{-1} StateOnGround?(OtherAirStatus!1)
|-----
{1} No?(AltReporting!1)
{2} Lost?(AltReporting!1)
{3} PT0?(PTTimer!1)

SUBGOAL.6 :
{-1} StateOnGround?(OtherAirStatus!1)
{-2} (OtherTrackedRange!1 * OtherTrackedRangeRate!1) <= H1TA!1
{-3} OtherTrackedRangeRate!1 > 10
{-4} OtherTrackedRange!1 <= DMODTA!1
{-5} cTrue?(OtherRangeValid!1)
{-6} cTrue?(OtherBearingValid!1)
|-----
{1} cTrue?(OtherAltReporting!1)
{2} OwnTrackedAlt!1 >= 15500

SUBGOAL.23 :
{-1} TAOnly?(ModeSelector!1)
{-2} No?(AltReporting!1)
|-----
{1} OtherTrackedRangeRate!1 > 10
{2} TAURTA!1 < TRTHRTA!1
{3} cTrue?(OtherRangeValid!1)
{4} cTrue?(PREVRAInhibit!1)
{5} PT0?(PTTimer!1)

SUBGOAL.24 :
{-1} ASL2?(AutoSL!1)
{-2} No?(AltReporting!1)
|-----
{1} OtherTrackedRangeRate!1 > 10
{2} TAURTA!1 < TRTHRTA!1
{3} cTrue?(OtherBearingValid!1)
{4} cTrue?(PREVRAInhibit!1)
{5} PT0?(PTTimer!1)

SUBGOAL.45 :
{-1} cTrue?(OtherAltReporting!1)
{-2} ADOT!1 >= -1
{-3} IRNo?(IntentReceived!1)
{-4} InhibitBiasedClimb!1 > DownSeparation!1
{-5} OwnTrackedAlt!1 < OtherTrackedAlt!1
{-6} CurrentVerticalSeparation!1 > 150
|-----
{1} CurrentVerticalSeparation!1 < 1200
{2} CurrentVerticalSeparation!1 < ZTHRTA!1
{3} ModifiedTauCapped!1 < TFRTHR!1
{4} No?(AltReporting!1)
{5} Lost?(AltReporting!1)
{6} PT0?(PTTimer!1)

SUBGOAL.1128 :
|-----
{1} OtherTrackedRange!1 < 6
{2} cTrue?(OtherAltReporting!1)
{3} OtherTrackedRangeRate!1 > 10
{4} TAURTA!1 < TRTHRTA!1
{5} PT0?(PTTimer!1)

SUBGOAL.1129 :
{-1} OtherTrackedRangeRate!1 > 10
|-----
{1} OtherTrackedRange!1 < 6
{2} cTrue?(OtherAltReporting!1)
{3} (OtherTrackedRange!1 *
 OtherTrackedRangeRate!1) <= H1TA!1
{4} PT0?(PTTimer!1)

```

Figure 6.35: Selected unprovable subgoals from PVS analysis of Proximate-Traffic to Other-Traffic and to Potential-Threat.

```
(skolemizeandrewrite$) strategy
(lemma "OtherAltReporting_AltReporting_Assertion")
(lemma "OtherAltReporting_OtherAirStatus_Assertion")
(apply (repeat* (inst?)))
(apply (repeat* (try (bddsimp) (record) (postpone)))))
```

Figure 6.36: The PVS proof commands used to include the domain axioms into the analysis process to attempt to prove two guarding conditions consistent.

of permutations of the other predicates (specifically, those predicates that comprise specific macro conditions), may be leading to the large number of inconsistency reports. We will examine this theory in future experiments by not expanding the macro conditions; our translation tool to translate from RSML specifications to PVS specifications does not currently provide the option to disable macro expansion. The selective macro expansion option may be added in a future version of the tool.

#### 6.1.1.2 Transitions out of State Other-Traffic

When Intruder-Status is in state Other-Traffic, other aircraft in the airspace are being monitored, but no hazard currently exists. Transitions out of state Other-Traffic signify that an intruder aircraft has moved closer to the aircraft monitoring the intruder, and thus, the intruder aircraft status is upgraded.

We ran our consistency analysis on each of the three pairs of guarding conditions individually. All three pairs of the guarding conditions proved to be consistent.

##### 6.1.1.2.1 Other-Traffic to Threat and Other-Traffic to Potential-Threat

Figures 6.6 and 6.24 showed the guarding conditions for the transitions from Other-Traffic to Threat and from Other-Traffic to Potential-Threat. Figure 6.37 shows the results obtained when we applied symbolic analysis with variable reordering; the analysis reported 4,909,890 potential inconsistencies.

Number of Satisfying Paths found: 4909890

|                                                               |          |
|---------------------------------------------------------------|----------|
| (1) AltReporting INSTATE Yes:                                 | 1        |
| (2) OtherAltReporting = cTrue:                                | 1        |
| (3) OtherAirStatus INSTATE StateAirborne:                     | 1        |
| (4) AutoSL INSTATE ASL2:                                      | 1        |
| (5) RAINhibitFromGround = cTrue:                              | 1        |
| (6) ModeSelector = TAOnly:                                    | 1        |
| (7) OwnTrackedAlt() > OtherTrackedAlt():                      | 1        |
| (8) OwnTrackedAlt() < OtherTrackedAlt():                      | 2 #      |
| (9) InhibitBiasedClimb() > DownSeparation():                  | 2 #      |
| (10) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP:         | 2 #      |
| (11) UpSeparation() >= ALIM():                                | 2 #      |
| (12) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP:        | 2 #      |
| (13) CurrentVerticalSeparation() > LOWFIRMZ:                  | 2 #      |
| (14) CurrentVerticalSeparation() < ZTHRТА():                  | 4 ##     |
| (15) ADOT() >= nZDTHRТА:                                      | 4 ##     |
| (16) -((CurrentVerticalSeparation()/ADOT())) < TVTHRТАTBL():  | 4 ##     |
| (17) ModifiedTauCapped() < TFRTHR():                          | 8 ####   |
| (18) LevelWait INSTATE S3:                                    | 4 ##     |
| (19) UpSeparation() <= SENSEFIRM():                           | 4 ##     |
| (20) DownSeparation() <= SENSEFIRM():                         | 4 ##     |
| (21) OtherTrackedRangeRate() > Zero:                          | 10 ##### |
| (22) ThreatAltVMD() < ZT():                                   | 10 ##### |
| (23) CurrentVerticalSeparation() < ZT():                      | 30 ##### |
| (24) TimeToCoAlt() < TrueTauCapped():                         | 10 ##### |
| (25) TimeToCoAlt() < TVTHR():                                 | 10 ##### |
| (26) ADOT() >= nZDTHR():                                      | 10 ##### |
| (27) OtherCapability = TARA:                                  | 10 ##### |
| (28) PREV1OtherRangeValid = cTrue:                            | 8 ####   |
| (29) PREV2OtherRangeValid = cTrue:                            | 8 ####   |
| (30) OtherRangeValid = cTrue:                                 | 14 ##### |
| (31) OtherTrackedRange() >= DMOD():                           | 15 ##### |
| (32) TauRising INSTATE PLUS3:                                 | 15 ##### |
| (33) IntruderStatus INSTATE Threat:                           | 15 ##### |
| (34) OtherBearingValid = cTrue:                               | 12 ##### |
| (35) OtherTrackedRangeRate() > RDTHRТА:                       | 18 ##### |
| (36) OtherTrackedRange() <= DMODТА():                         | 12 ##### |
| (37) (OtherTrackedRange()*OtherTrackedRangeRate()) <= H1ТА(): | 12 ##### |
| (38) TAURТА() < TRTHRТА():                                    | 12 ##### |
| (39) OtherTrackedRange() <= RMAX:                             | 12 ##### |
| (40) ModifiedTauCapped() < TRTHR():                           | 12 ##### |
| (41) OtherTrackedRange() > DMOD():                            | 12 ##### |
| (42) (OtherTrackedRangeRate() * OtherTrackedRange()) > H1():  | 12 ##### |
| (43) OwnTrackedAltRate() <= OLEV:                             | 8 ####   |
| (44) OtherVRC = NoIntent:                                     | 6 ###    |
| (45) IntentReceived INSTATE IRNo:                             | 6 ###    |
| (46) ModifiedTauCapped() < FRTHR():                           | 4 ##     |
| (47) OtherTrackedAltRate() <= OLEV:                           | 2 #      |
| (48) OwnTrackedAltRate() >= Zero:                             | 2 #      |
| (49) OtherTrackedAltRate() >= Zero:                           | 4 ##     |
| (50) CurrentVerticalSeparation() > MAXALTDIFF:                | 2 #      |
| (51) CurrentVerticalSeparation() > MAXALTDIFF2:               | 2 #      |
| (52) PTTimer INSTATE PT0:                                     | 1        |
| Total: 390                                                    |          |

Figure 6.37: Partial BDD node profile for conjunction of Other-Traffic to Threat and to Potential-Threat with variable reordering.

From the BDD node profile shown in Figure 6.37, we identified the first seven nodes and the last node as potential indicator nodes. We already know (Section 6.1.1.1) of several domain axioms related to the first three indicator predicates shown in the BDD node profile. The next step is to examine samples of the result BDD to see if any of the known domain axioms are violated in all or a majority of the samples. Selected samples from the symbolic analysis with variable reordering are shown in Figure 6.38. As the samples show, all satisfying paths in the BDD pass through the first eight predicates, and the first six predicates maintain the same truth value on every path. Recall (Section 6.1.1.1.1), that one of the domain axioms involving the predicates

Other-Alt-Reporting = True and Alt-Reporting IN-STATE Yes

is

Other-Alt-Reporting = True *iff* Alt-Reporting IN-STATE Yes.

The samples show that

Alt-Reporting IN-STATE Yes, predicate (1), is always FALSE and that

Other-Alt-Reporting = True, predicate (2), is always TRUE;

thus, the domain axiom involving these two predicates is violated in all of the samples. We introduced this domain axiom into the specification and reran the symbolic analysis. The analysis output reported the guarding conditions consistent. Thus, all of the above reported inconsistencies were spurious and were all due to the violation of a single domain axiom involving the first two predicates in the result BDD obtained using variable reordering. Without our technique for identifying the missing information, the analyst would have no indication of which of the 58 predicates (the full result BDD contained 58 predicates) to examine to see if they are involved in any domain axioms in the specification that do not hold in the analysis output.

We attempted PVS analysis without domain axioms, on the full guarding conditions using the strategy in Figure 6.22, on a Ross-based SPARC 20 with two 150MHz CPUs. We aborted the proof



```

(1) FF
(2) TTTTNTT
(3) TTTTNTT
(4) FF
(5) FF
(6) FF
(7) TTTTNTT
(8) TTTTNTT
(9) TTT
(10)TTT
(11)FF
(12)FF
(13) TTTTNTT
(14) TTTTNTT
(15)FF
(16)FF
(17) TTTTNTT
(18) TTTTNTT
(19)TTT
(20)FF
(21) TTTTNTT
(22)FF
(23) TTTTNTT
(24)TTT
(25)TTT
(26)FF
(27) TTTTNTT
(28) TTTTNTT
(29)TTT
(30) TTTTNTT
(31) TTTTNTT
(32) TTTTNTT
(33) TTTTNTT
(34) TTTTNTT
(35) TTTTNTT
(36) TTTTNTT
(37) TTTTNTT
(38)TTFFF...FFFFFTTTFFF...FFFF...FF
(39)TFTT...FTFTFTFTFT...FFTT...FT
(40)TF.TF...T..T.TF.TF...FT...F
(41) FFFFFFFFFFFFFFFFFF.FF.FFFFF.FF.F.FF.FFFFFF.FFFFF
(42) FFFFFFFFFFFFFFFFFF.FF.FFFFF.FF.F.FF.FFFFFF.FFFFF
(43) TTTTFTT
(44) TTTTFTT
(45) ..TTF...TTTTTF...FFF...FFF
(46) TTF.TTTTTTTTTT...
(47)TFFFTFFF...
(48)TTF.TTF...
(49)TFT.TFT...
(50) F.F..FF..FF..F...
(51)FF...FF...
(52)FFFFF..FFFF.FFFFFF...FFFF.FFFFFFFFFFFF

```

Figure 6.38: Selected samples from symbolic analysis of Other-Traffic to Threat and to Potential-Threat with variable reordering.

attempt after it ran for over a day with no results. We added the domain axiom we identified using our indicator nodes method to the PVS specification and ran PVS analysis on a SPARCstation 20 Model 70 with 64MB of main memory and one 75MHz CPU. PVS reported the guarding conditions consistent in 107.85 seconds, using the proof commands shown in Figure 6.23.

6.1.1.2.2 Other-Traffic to Threat and Other-Traffic to Proximate-Traffic

Figures 6.6 and 6.39 show the guarding conditions for the transitions from Other-Traffic to Threat and from Other-Traffic to Proximate-Traffic. Figure 6.40 shows the results

Transition(s): Other-Traffic → Proximate-Traffic

Location: Other-Aircraft ▸ Intruder-Status<sub>s-136</sub>

Trigger Event: Air-Status-Evaluated-Event<sub>e-279</sub>

Condition:

|     |                                              |    |   |
|-----|----------------------------------------------|----|---|
| AND | Alt-Reporting <sub>s-101</sub> in state Yes  | F  | T |
|     | Other-Bearing-Valid <sub>v-120</sub> = True  | T  | . |
|     | Other-Range-Valid <sub>v-117</sub> = True    | T  | . |
|     | Proximate-Traffic-Condition <sub>m-216</sub> | T  | T |
|     | Potential-Threat-Condition <sub>m-213</sub>  | F  | F |
|     | Threat-Condition <sub>m-140</sub>            | .  | F |
|     | PT-Timer <sub>s-136</sub> in state 0         | T  | T |
|     |                                              | OR |   |

Output Action: Intruder-Status-Evaluated-Event<sub>e-279</sub>

Figure 6.39: Transition from Other-Traffic to Proximate-Traffic.

obtained when we applied symbolic analysis with variable reordering.

From the BDD node profile shown in Figure 6.40, we identified the first ten nodes and the last node as potential indicator nodes. We know that the first and fourth predicates in the node profile are involved in the domain axiom noted earlier (Section 6.1.1.2.1). Selected samples from the symbolic analysis with variable reordering are shown in Figure 6.41. From the samples, we can see that all satisfying paths in the BDD pass through the first eleven predicates, and the samples also reveal that the first nine predicates maintain the same truth value on every path. Sampling also revealed

Number of Satisfying Paths found: 3447360

|                                                               |          |
|---------------------------------------------------------------|----------|
| (1) AltReporting INSTATE Yes:                                 | 1 #      |
| (2) OtherBearingValid = cTrue:                                | 1 #      |
| (3) OtherTrackedRange() < PROXR:                              | 1 #      |
| (4) OtherAltReporting = cTrue:                                | 1 #      |
| (5) CurrentVerticalSeparation() < PROXA:                      | 1 #      |
| (6) OtherAirStatus INSTATE StateAirborne:                     | 1 #      |
| (7) AutoSL INSTATE ASL2:                                      | 1 #      |
| (8) RAINhibitFromGround = cTrue:                              | 1 #      |
| (9) ModeSelector = TAOnly:                                    | 1 #      |
| (10) OwnTrackedAlt() > OtherTrackedAlt():                     | 1 #      |
| (11) OwnTrackedAlt() < OtherTrackedAlt():                     | 2 ##     |
| (12) InhibitBiasedClimb() > DownSeparation():                 | 2 ##     |
| (13) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP:         | 2 ##     |
| (14) UpSeparation() >= ALIM():                                | 2 ##     |
| (15) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP:        | 2 ##     |
| (16) CurrentVerticalSeparation() > LOWFIRMZ:                  | 2 ##     |
| (17) UpSeparation() <= SENSEFIRM():                           | 2 ##     |
| (18) DownSeparation() <= SENSEFIRM():                         | 2 ##     |
| (19) ModifiedTauCapped() < TFRTHR():                          | 4 ####   |
| (20) LevelWait INSTATE S3:                                    | 2 ##     |
| (21) TCASTCASVMD() >= Zero:                                   | 2 ##     |
| (22) OtherTrackedRelativeAlt() < nMINSEP:                     | 2 ##     |
| (23) TCASTCASVMD() <= Zero:                                   | 2 ##     |
| (24) OtherTrackedRelativeAlt() > MINSEP:                      | 2 ##     |
| (25) OtherTrackedRangeRate() > Zero:                          | 5 #####  |
| (26) ThreatAltVMD() < ZT():                                   | 5 #####  |
| (27) CurrentVerticalSeparation() < ZT():                      | 15 ##### |
| (28) TimeToCoAlt() < TrueTauCapped():                         | 5 #####  |
| (29) TimeToCoAlt() < TVTHR():                                 | 5 #####  |
| (30) ADOT() >= nZDTHR():                                      | 5 #####  |
| (31) ModifiedTauCapped() < FRTHR():                           | 2 ##     |
| (32) CurrentVerticalSeparation() < ZTHRТА():                  | 5 #####  |
| (33) ADOT() >= nZDTHRТА:                                      | 5 #####  |
| (34) -((CurrentVerticalSeparation()/ADOT())) < TVTHRТАTBL():  | 5 #####  |
| (35) OtherCapability = TARA:                                  | 10 ##### |
| (36) OwnTrackedAltRate() <= OLEV:                             | 8 #####  |
| (37) PREV1OtherRangeValid = cTrue:                            | 6 #####  |
| (38) PREV2OtherRangeValid = cTrue:                            | 6 #####  |
| (39) OtherRangeValid = cTrue:                                 | 12 ##### |
| (40) OtherTrackedRange() >= DMOD():                           | 12 ##### |
| (41) TauRising INSTATE PLUS3:                                 | 12 ##### |
| (42) IntruderStatus INSTATE Threat:                           | 12 ##### |
| (43) OtherTrackedRangeRate() > RDTHRТА:                       | 12 ##### |
| (44) OtherTrackedRange() <= DMODТА():                         | 6 #####  |
| (45) (OtherTrackedRange()*OtherTrackedRangeRate()) <= H1ТА(): | 6 #####  |
| (46) ТАURТА() < TRTHRТА():                                    | 6 #####  |
| (47) OtherTrackedRange() <= RMAX:                             | 6 #####  |
| (48) ModifiedTauCapped() < TRTHR():                           | 6 #####  |
| (49) (OtherTrackedRangeRate() * OtherTrackedRange()) > H1():  | 6 #####  |
| (50) IntentReceived INSTATE IRNo:                             | 3 ###    |
| (51) CurrentVerticalSeparation() > MAXALTDIFF:                | 1 #      |
| (52) PTTimer INSTATE PT0:                                     | 1 #      |
| Total: 247                                                    |          |

Figure 6.40: Partial BDD node profile for conjunction of guarding conditions from Other-Traffic to Threat and to Proximate-Traffic with variable reordering.

that the last predicate was always TRUE. The samples showed that the same domain axiom that was violated in the above case (Section 6.1.1.2.1), was also violated in all samples for the two guarding conditions being analyzed here; predicates (1) and (4). We added the appropriate domain axiom and reran the symbolic analysis. The output from this iteration of the symbolic analysis augmented with the appropriate domain axiom showed the guarding conditions consistent.

We applied PVS analysis without domain axioms, on the full guarding conditions using the strategy in Figure 6.22, on a Ross-based SPARC 20 with two 150MHz CPUs. We aborted the proof attempt after it ran for over a day with no results. We added the domain axiom we identified using our indicator nodes method to the PVS specification and ran PVS analysis on a SPARCstation 20 Model 70 with 64MB of main memory and one 75MHz CPU. PVS reported the guarding conditions consistent in 39.97 seconds, using the proof commands in Figure 6.23.

#### **6.1.1.2.3 Other-Traffic to Proximate-Traffic and Other-Traffic to Potential-Threat**

Both the symbolic analysis using BDDs and the PVS analysis showed the guarding conditions consistent on the first iteration; no domain axioms needed to be added to the analysis to show the guarding conditions consistent. In addition, we did not need to invoke our decision procedures during symbolic analysis. We ran the PVS proof on a Ross-based SPARC 20 with two 150MHz CPUs using the PVS proof strategy in Figure 6.22. The proof finished in 17.28 seconds.

#### **6.1.1.3 Transitions out of State Potential-Threat**

When Intruder-Status is in state Potential-Threat, an intruder aircraft is in a potentially threatening position in relation to own aircraft. The status of the intruder may be upgraded to Threat or downgraded to Other-Traffic or Proximate-Traffic depending on the intruder's course and the course of own aircraft. A hazardous situation could result if the same conditions were satisfied for transitions that both upgrade and downgrade the threatening status of

```

(1) FF
(2) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(3) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(4) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(5) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(6) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(7) FF
(8) FF
(9) FF
(10) TTTTTTTTTTFFFFFFFFTTTTTTTTTTFFFFFFFFFFFF
(11) TTTTTTTTTTFFFFFFFFTTTTTTTTTTFFFFFFFFFFFF
(12)FFFFFFFFFTTTTTTTTTT.....
(13)TTTTTTTTTT.....
(14)FFFFFFFF.....
(15)FFFFFFFF.....
(16) TTTTTTTTTT.....
(17)TTTTTTTTTTTTTTTTTTTTTTFFFFFFFFFFFF
(18)FFFFFFFFFFFFFFFF.....
(19) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFFFFFFFF
(20) TTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFFFF.....
(21)TTTTTTTTTTTTTTTTTTTTTT.....
(22)FFFFFFFFFFFFFFFF.....
(23)TTTTTTTTTTTTTTTTTTTTTT.....
(24)FFFFFFFFFFFFFFFF.....
(25) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFFFFFFFF
(26)FFFFFFFF.....
(27) TTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(28)TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(29)TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(30)FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(31) TTTTTTTTTT.....
(32) TTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
(33)TTTTTTTTFTTTTTTTTTTTFFFFFFFFFFFF
(34)FF.....FFFFFFFFFFFFFFFF
(35) TTTTTTTTTTFFFTTTFTFTTTTTTTTTTTFFFTTTTTTT
(36) TTTTTTTTTT.....
(37) TTTTTTTTTT...FFF...TTTFFFFFFFF...FFFFFF
(38)FFF...FFFTTT...FFFTTT
(39) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
(40) TTTTTTTTTTFFFTTTFFFTTTFTFTTTTTTTFF
(41) TTTTTTTTTT....TTT.....FFF..FFFTTT...
(42) TTTTTTTTTT....TTT.....TTTT...
(43) TTTTTTFFFTFFFTFFFTFFFTFFFTFFFTFFFTFFFT
(44) TTFFF.....
(45) FFF.....
(46)FFFF.....
(47)TTT.FTT.FTFTT.FTTFTT.FTTFTT.FT
(48)TTTF..TF..TT.FT...TF.TFT..FT.FT..FT
(49) FFFFFF...FFF.FFFF..FF.FFFF.FF.F.FFF.FF.FFF.
(50) .TF.TF.TF....FFFF....FFFF....FFFF...
(51) FF.FF.FF.F.....
(52) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT

```

Figure 6.41: Portion of the samples obtained from symbolic analysis for consistency of Other-Traffic to Threat and to Proximate-Traffic with variable reordering.

the intruder aircraft.

We ran our consistency analysis on each of the three pairs of guarding conditions individually.

All three pairs of the guarding conditions proved to be consistent.

#### 6.1.1.3.1 Potential-Threat to Threat and Potential-Threat to Proximate-Traffic

Figures 6.6 and 6.42 show the guarding conditions for the transitions from Potential-Threat to Threat and from Potential-Threat to Proximate-Traffic.

Figure 6.43 shows the results obtained when we applied symbolic analysis with variable reordering.

**Transition(s):** Potential-Threat → Proximate-Traffic

**Location:** Other-Aircraft ▸ Intruder-Status<sub>s-136</sub>

**Trigger Event:** Air-Status-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                              |   |   |   |   |
|-----|----------------------------------------------|---|---|---|---|
| AND | Alt-Reporting <sub>s-101</sub> in state Lost | T | · | · | · |
|     | RA-Mode-Canceled <sub>m-218</sub>            | · | T | F | · |
|     | Alt-Reporting <sub>s-101</sub> in state No   | · | T | T | · |
|     | Alt-Reporting <sub>s-101</sub> in state Yes  | · | · | · | T |
|     | Other-Range-Valid <sub>v-117</sub> = True    | T | T | · | · |
|     | Other-Bearing-Valid <sub>v-120</sub> = True  | T | T | · | · |
|     | PT-Timer <sub>s-136</sub> in state 0         | T | T | T | T |
|     | Proximate-Traffic-Condition <sub>m-216</sub> | T | T | T | T |
|     | Potential-Threat-Condition <sub>m-213</sub>  | F | F | F | F |
|     | Threat-Condition <sub>m-140</sub>            | · | · | · | F |

OR

**Output Action:** Intruder-Status-Evaluated-Event<sub>e-279</sub>

Figure 6.42: Transition from Potential-Threat to Proximate-Traffic.

From the BDD node profile in Figure 6.43, we identified the first two nodes and the last four nodes as potential indicator nodes. This example is not as clear cut as the previous examples, therefore, we provide a more detailed explanation. For this example, we assume that the analyst has no prior information about any domain axioms in the specification; i.e., the analyst does not know about the domain axioms we identified in the previous test cases.

Using our method to attempt to identify the relevant domain axioms, the analyst begins with the

Number of Satisfying Paths found: 6655740

|                                                               |          |
|---------------------------------------------------------------|----------|
| (1) OtherBearingValid = cTrue:                                | 1        |
| (2) AltReporting INSTATE No:                                  | 2 #      |
| (3) AutoSL INSTATE ASL2:                                      | 2 #      |
| (4) RAINhibitFromGround = cTrue:                              | 2 #      |
| (5) ModeSelector = TAOOnly:                                   | 2 #      |
| (6) AltReporting INSTATE Lost:                                | 1        |
| (7) OwnTrackedAlt() > OtherTrackedAlt():                      | 2 #      |
| (8) OwnTrackedAlt() < OtherTrackedAlt():                      | 4 ##     |
| (9) InhibitBiasedClimb() > DownSeparation():                  | 4 ##     |
| (10) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP:         | 4 ##     |
| (11) CurrentVerticalSeparation() > LOWFIRMZ:                  | 4 ##     |
| (12) ModifiedTauCapped() < TFRTHR():                          | 8 ####   |
| (13) LevelWait INSTATE S3:                                    | 4 ##     |
| (14) UpSeparation() <= SENSEFIRM():                           | 4 ##     |
| (15) ModifiedTauCapped() < FRTHR():                           | 4 ##     |
| (16) OtherRangeValid = cTrue:                                 | 9 ####   |
| (17) TCASTCASVMD() >= Zero:                                   | 2 #      |
| (18) TCASTCASVMD() <= Zero:                                   | 2 #      |
| (19) OtherTrackedRelativeAlt() > MINSEP:                      | 2 #      |
| (20) OtherCapability = TARA:                                  | 7 ###    |
| (21) OtherTrackedRangeRate() > Zero:                          | 7 ###    |
| (22) ThreatAltVMD() < ZT():                                   | 7 ###    |
| (23) CurrentVerticalSeparation() < ZT():                      | 21 ##### |
| (24) TimeToCoAlt() < TrueTauCapped():                         | 7 ###    |
| (25) TimeToCoAlt() < TVTHR():                                 | 7 ###    |
| (26) ADOT() >= nZDTHR():                                      | 7 ###    |
| (27) OtherAltReporting = cTrue:                               | 7 ###    |
| (28) OwnTrackedAltRate() <= OLEV:                             | 4 ##     |
| (29) PREV1OtherRangeValid = cTrue:                            | 3 #      |
| (30) PREV2OtherRangeValid = cTrue:                            | 3 #      |
| (31) -((CurrentVerticalSeparation()/ADOT())) < TVTHRTATBL():  | 6 ###    |
| (32) ADOT() >= nZDTHRTA:                                      | 6 ###    |
| (33) CurrentVerticalSeparation() < ZTHRTA():                  | 6 ###    |
| (34) OtherTrackedRange() >= DMOD():                           | 12 ##### |
| (35) TauRising INSTATE PLUS3:                                 | 12 ##### |
| (36) IntruderStatus INSTATE Threat:                           | 12 ##### |
| (37) OtherTrackedRangeRate() > RDTHRTA:                       | 12 ##### |
| (38) OtherTrackedRange() <= DMODTA():                         | 6 ###    |
| (39) (OtherTrackedRange()*OtherTrackedRangeRate()) <= H1TA(): | 6 ###    |
| (40) TAURTA() < TRTHRTA():                                    | 6 ###    |
| (41) OtherTrackedRange() <= RMAX:                             | 6 ###    |
| (42) ModifiedTauCapped() < TRTHR():                           | 6 ###    |
| (43) OtherTrackedRange() > DMOD():                            | 6 ###    |
| (44) (OtherTrackedRangeRate() * OtherTrackedRange()) > H1():  | 6 ###    |
| (45) OtherVRC = NoIntent:                                     | 3 #      |
| (46) IntentReceived INSTATE IRNo:                             | 3 #      |
| (47) CurrentVerticalSeparation() > MAXALTDIFF:                | 1        |
| (48) CurrentVerticalSeparation() < PROXA:                     | 1        |
| (49) OtherAirStatus INSTATE StateAirborne:                    | 1        |
| (50) OtherTrackedRange() < PROXR:                             | 1        |
| (51) PTTimer INSTATE PT0:                                     | 1        |
| Total: 288                                                    |          |

Figure 6.43: Partial BDD node profile for conjunction of Potential-Threat to Threat and to Proximate-Traffic with variable reordering.

predicate at the root of the result `BDD Other-Bearing-Valid = True`, and examines the index of the specification to find all the locations where the variable `Other-Bearing-Valid` is referenced. The analyst finds that this variable is referenced twelve times in the specification, and checks each of the references to see if there are any domain axioms associated with this variable. The analyst quickly determines that only two of the twelve references are relevant to the two guarding conditions being analyzed. The analyst also determines that there are no relevant domain axioms in the specification involving the variable. Figures 6.7 and 6.42 show the only two references of the variable under consideration; it is easy to see that there are no relevant domain axioms involving `Other-Bearing-Valid` since the truth values shown for the predicate `Other-Bearing-Valid = True` do not remain constant in the specification of the guarding conditions.

The analyst next examines the predicate `Alt-Reporting IN-STATE No`, and finds seventeen references to the state `Alt-Reporting` in the specification. Only four of the seventeen references are relevant to the guarding conditions being analyzed, and three of the four lead the analyst to the information that is missing from the symbolic analysis and resulting in the spurious errors; Figures 6.15 through 6.20 show the relevant references to the state `Alt-Reporting` and the guarding conditions associated with transitions within the state `Alt-Reporting`. The domain axiom related to the state `Alt-Reporting` is `Other-Alt-Reporting = True iff Alt-Reporting IN-STATE Yes`. Once the analyst identifies a relevant domain axiom, she looks at the samples to see if the domain axiom is violated in all or a majority of the samples. Selected samples from the symbolic analysis with variable reordering are shown in Figure 6.44. In the figure, the predicates involved in the domain axiom are triple starred to offset them. As one can see from the samples shown, it is difficult to identify the relevant domain axiom and to verify that the domain axiom is violated in all or a majority of the samples without some indication of what



predicates to look at in the first place. To make it easier to see that the identified domain axiom is violated in all of the samples, I have isolated the three predicates involved in the domain axiom (Figure 6.45). Note that the predicate `Alt-Reporting IN-STATE Yes` did not appear in the conjunction of the guarding conditions, yet is part of the relevant domain axiom; this makes it even more difficult to determine the relevant domain axiom without some indication of where to look. With a knowledge of the all-inclusive and mutually exclusive nature of enumerated type predicates, the analyst can determine from the samples of the three isolated predicates shown in the figure, that the domain axiom is violated in all samples. The samples show that `Alt-Reporting` is either in state `No` or in state `Lost` all of the time. Given this information and the fact that enumerated type predicates are mutually exclusive, we can conclude that `Alt-Reporting` is never in state `Yes`, which violates the domain axiom identified above, since `Other-Alt-Reporting = True` is always `TRUE`. When we added the identified domain axiom `Other-Alt-Reporting = True iff Alt-Reporting IN-STATE Yes` and reran the symbolic analysis, the output reported that the guarding conditions were consistent.

We ran PVS analysis on the full guarding conditions without the domain axiom added, using the proof strategy in Figure 6.22 on a Ross-based SPARC 20 with two 150MHz CPUs. We aborted the proof attempt after it ran on the order of a day. We then added the domain axiom we identified using our indicator nodes approach to the PVS specification of the guarding conditions. Using the proof commands in Figure 6.23, we ran PVS analysis on a SPARCstation 20 Model 70 with 64MB main memory and one 75MHz CPU. The guarding conditions were proven consistent in 92.85 seconds.

#### 6.1.1.3.2 Potential-Threat to Threat and Potential-Threat to Other-Traffic

Figures 6.6 and 6.46 show the guarding conditions for the transitions from Potential-Threat to Threat and from Potential-Threat to Other-Traffic.

[illegible]

**Figure 6.44: Selected samples from symbolic analysis of Potential-Threat to Threat and to Proximate-Traffic with variable reordering.**



Number of Satisfying Paths found: 15095745

```

(1) PTimer INSTATE PT0: 1
(2) AutoSL INSTATE ASL2: 2 #
(3) ModeSelector = TAOnly: 2 #
(4) RAIInhibitFromGround = cTrue: 2 #
(5) OtherAltReporting = cTrue: 2 #
(6) OtherAirStatus INSTATE StateAirborne: 2 #
(7) OwnTrackedAlt() > OtherTrackedAlt(): 2 #
(8) OwnTrackedAlt() < OtherTrackedAlt(): 4 ##
(9) InhibitBiasedClimb() > DownSeparation(): 4 ##
(10) (OwnTrackedAlt() - OtherTrackedAlt()) >= MINSEP: 4 ##
(11) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP: 4 ##
(12) UpSeparation() <= SENSEFIRM(): 4 ##
(13) DownSeparation() <= SENSEFIRM(): 4 ##
(14) ModifiedTauCapped() < TFRTHR(): 8 ####
(15) LevelWait INSTATE S3: 4 ##
(16) OtherTrackedRelativeAlt() < nMINSEP: 4 ##
(17) OtherTrackedRelativeAlt() > MINSEP: 4 ##
(18) OtherTrackedRangeRate() > Zero: 10 #####
(19) ThreatAltVMD() < ZT(): 10 #####
(20) CurrentVerticalSeparation() < ZT(): 30 #####
(21) TimeToCoAlt() < TrueTauCapped(): 10 #####
(22) TimeToCoAlt() < TVTHR(): 10 #####
(23) ADOT() >= nZDTHR(): 10 #####
(24) CurrentVerticalSeparation() < PROXA: 5 ##
(25) OtherTrackedRange() < PROXR: 5 ##
(26) AltReporting INSTATE Lost: 10 #####
(27) OtherRangeValid = cTrue: 14 #####
(28) OtherBearingValid = cTrue: 10 #####
(29) OtherTrackedRangeRate() > RDTHRTA: 21 #####
(30) OtherTrackedRange() <= DMODTA(): 7 ###
(31) (OtherTrackedRange()*OtherTrackedRangeRate())<=H1TA(): 7 ###
(32) TAURTA() < TRTHRTA(): 7 ###
(33) OtherTrackedRange() <= RMAX: 14 #####
(34) ModifiedTauCapped() < TRTHR(): 14 #####
(35) OtherTrackedRange() > DMOD(): 14 #####
(36) (OtherTrackedRangeRate()*OtherTrackedRange()) > H1(): 14 #####
(37) OtherCapability = TARA: 14 #####
(38) OwnTrackedAltRate() <= OLEV: 8 ####
(39) PREV1OtherRangeValid = cTrue: 6 ###
(40) PREV2OtherRangeValid = cTrue: 6 ###
(41) OtherTrackedRange() >= DMOD(): 12 #####
(42) TauRising INSTATE PLUS3: 12 #####
(43) IntruderStatus INSTATE Threat: 12 #####
(44) OtherVRC = NoIntent: 6 ###
(45) IntentReceived INSTATE IRNo: 6 ###
(46) ModifiedTauCapped() < FRTHR(): 4 ##
(47) OtherTrackedAltRate() <= OLEV: 2 #
(48) OwnTrackedAltRate() >= Zero: 2 #
(49) OtherTrackedAltRate() >= Zero: 4 ##
(50) CurrentVerticalSeparation() > MAXALTDIFF: 2 #
(51) CurrentVerticalSeparation() > MAXALTDIFF2: 2 #
(52) OtherAirStatus INSTATE StateOnGround: 1
Total: 403

```

Figure 6.47: Partial BDD node profile for conjunction of Potential-Threat to Threat and to Other-Traffic with variable reordering.

domain axioms associated with this state, but does not identify any. The analyst also determines that there are no relevant domain axioms in the specification involving the state Auto-SL.

Looking at the last indicator predicate Other-Air-Status IN-STATE On-Ground, the analyst identifies two domain axioms:

1. *if Other-Alt-Reporting = True is FALSE then NOT Other-Air-Status IN-STATE On-Ground, and*
2. *Other-Air-Status cannot be in both On-Ground and Airborne at the same time, and it has to be in one of On-Ground or Airborne at any time.*

The analyst examines the samples to see if either or both of the identified domain axioms are violated in all or in a majority of the samples. Selected samples are shown in Figure 6.48. The analyst finds that the first domain axiom she identified is not violated in any of the samples, but finds that the axiom related to the mutually exclusive nature of the enumerated type predicate is violated in most of the samples; the samples show that Other-Air-Status is in both of its sub-states at the same time; predicates (6) and (52). We reran the symbolic analysis with variable reordering and decision procedures enabled. The analysis output reported 3,982,320 potential inconsistencies. Though this reduction in the number of reported inconsistencies is substantial (a reduction of almost 12 million spurious inconsistencies), it does not facilitate the analyst in locating the true inconsistencies (if any).

At this point, the analyst has a couple of options with our method. First, she can continue checking each individual predicate in the node profile starting from where she left off at the third predicate, or, second, she can look at the sample output obtained from either the first iteration with variable ordering, or she can look at the sample output obtained from the second iteration with variable reordering and decision procedures enabled. Either of these outputs will reveal that five of the top six predicates in the node profile maintain the same truth values in all samples. Constant truth values are indicative of domain axioms that are being violated. The analyst has already examined the second predicate in the node profile, so she can concentrate on the next four predicates rather than having

(1) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFFFFFFFFFFFFFFFFFFFFFF  
(2) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
(3) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
(4) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
(5) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFTTTTTTTTTTTTTTTTTTT  
(6) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFTTTTTTTTTTTTTTTTTTT  
(7) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFTTTTTTTTTTTTTTTTTTT  
(8) TTTTTTTTTTTTTTTTTTTTTTTTTTTFFFTTTTTTTTTTTTTTTTTTTFFFT  
(9) .....TTTTTTTTTTFFFTTTTTTTTTTTTTTTTTTT.....  
(10) .....TTTTTTTTTT.....  
(11) .....FFFFFFFF.....  
(12) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTFFFT  
(13) .....FFFFFFFFFFFFFFFFFFFF.....  
(14) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(15) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(16) .....FFFFFFFFFFFFFFFFFFFF.....  
(17) .....FFFFFFFFFFFFFFFFFFFF.....  
(18) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(19) .....FFFFFFFF.....  
(20) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(21) .....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(22) .....:.....TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(23) .....FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
(24) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(25) TTTTTTTTTTTTTTTTTTTTTTTT.....  
(26) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(27) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(28) TTTTTTTTTTTTTTTTTTTTTTT.....F.....  
(29) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(30) .....FFFFFTTT.....T.....  
(31) .....TTTT.....T.....  
(32) .....FFFFFFFFTTTT.....  
(33) .....TTTTTTTTTT.TTTTTTTTTTT.....  
(34) .....FFFFFFFF.TTTT.....  
(35) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF....FFFF  
(36) FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF....FFFF  
(37) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(38) TTTTTTTTTTTTTTTTTTTTTTT.....  
(39) TTTTTTTTTTTTTTTTTTTTTTT.....TTTTTTTT.T.....  
(40) .....TTTTTTTTTT.....TTTTTT.....  
(41) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(42) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(43) TTTT.....T..T..T..T..T..T..T..T..T..T..  
(44) TTFTTTFTFTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT  
(45) ..TFT..TFT..TTTTTTT.FFT.....F...FFT.....FFT..FT  
(46) FTFT.FTFT.FTTTTTTT.TT.....  
(47) .....F.TFTTT..FT.....  
(48) .....F..TFTT..T.....  
(49) .....F..TFTT..T.....  
(50) F.F..F.F..F.FT..F..FT.....  
(51) .....FF.....  
(52) TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT.....TTTTTTTTTT.TTTT.....

**Figure 6.48: Selected samples from symbolic analysis of Potential-Threat to Threat and to Other-Traffic with variable reordering.**

to potentially examine all 55 of the remaining unexamined predicates. The analyst does not find any domain axioms related to the third or fourth predicates, but finds the following domain axiom related to the fifth predicate `Other-Alt-Reporting = True: Other-Alt-Reporting = True iff Alt-Reporting IN-STATE Yes`<sup>4</sup>. The samples revealed that this domain axiom was violated in a majority of cases; predicates (5) and (26). We added the relevant domain axiom to the specification and reran the symbolic analysis with decision procedures enabled. The analysis reported the guarding conditions consistent.

We attempted PVS analysis without domain axioms, on the full guarding conditions using the strategy in Figure 6.22, on a Ross-based SPARC 20 with two 150MHz CPUs. We aborted the proof attempt after it ran for over a day with no results. We added the domain axiom we identified using the process described above to the PVS specification and ran PVS analysis on a 200MHz Pentium Pro machine with 64MB RAM running Redhat Linux. PVS reported the guarding conditions consistent in 32.72 seconds, using the proof commands in Figure 6.23.

#### 6.1.1.3.3 Potential-Threat to Proximate-Traffic and Potential-Threat to Other-Traffic

Figures 6.42 and 6.46 show the guarding conditions for the transitions from Potential-Threat to Proximate-Traffic and from Potential-Threat to Other-Traffic. Figure 6.49 shows the results obtained when we applied symbolic analysis with variable reordering.

From the BDD node profile (Figure 6.49) we identified the first three nodes and the last node as potential indicator nodes. There are some very obvious domain axioms associated with the `Alt-Reporting` and `Other-Air-Status` in state predicates that showed up as indicator predicates. The samples (Figure 6.50) showed that these in state predicates (predicates (2), (3),

---

<sup>4</sup>Section 6.1.1.1.1 discussed this domain axiom in detail.

Number of Satisfying Paths found: 4787831

|                                                             |          |
|-------------------------------------------------------------|----------|
| (1) PTimer INSTATE PT0:                                     | 1        |
| (2) AltReporting INSTATE No:                                | 1        |
| (3) AltReporting INSTATE Yes:                               | 2        |
| (4) PREVRAInhibit = cTrue:                                  | 2        |
| (5) AutoSL INSTATE ASL2:                                    | 3 #      |
| (6) RAINhibitFromGround = cTrue:                            | 3 #      |
| (7) ModeSelector = TAOnly:                                  | 3 #      |
| (8) OwnTrackedAlt() > OtherTrackedAlt():                    | 1        |
| (9) OwnTrackedAlt() < OtherTrackedAlt():                    | 2        |
| (10) InhibitBiasedClimb() > DownSeparation():               | 2        |
| (11) (OwnTrackedAlt() - OtherTrackedAlt()) <= nMINSEP:      | 2        |
| (12) ModifiedTauCapped() < TFRTHR():                        | 4 #      |
| (13) LevelWait INSTATE S3:                                  | 2        |
| (14) UpSeparation() <= SENSEFIRM():                         | 2        |
| (15) AltReporting INSTATE Lost:                             | 7 ##     |
| (16) OtherRangeValid = cTrue:                               | 11 ###   |
| (17) OtherBearingValid = cTrue:                             | 7 ##     |
| (18) OtherCapability = TARA:                                | 14 ####  |
| (19) OtherTrackedRangeRate() > Zero:                        | 14 ####  |
| (20) ThreatAltVMD() < ZT():                                 | 14 ####  |
| (21) CurrentVerticalSeparation() < ZT():                    | 42 ##### |
| (22) TimeToCoAlt() < TrueTauCapped():                       | 14 ####  |
| (23) TimeToCoAlt() < TVTHR():                               | 14 ####  |
| (24) ADOT() >= nZDTHR():                                    | 14 ####  |
| (25) OtherAltReporting = cTrue:                             | 16 ##### |
| (26) OwnTrackedAlt() >= ABOVNMC:                            | 2        |
| (27) CurrentVerticalSeparation() < ZTHRТА():                | 14 ####  |
| (28) -((CurrentVerticalSeparation()/ADOT()))<TVTHRТАТBL():  | 14 ####  |
| (29) ADOT() >= nZDTHRТА:                                    | 14 ####  |
| (30) OtherTrackedRange() >= DMOD():                         | 24 ##### |
| (31) TauRising INSTATE PLUS3:                               | 24 ##### |
| (32) IntruderStatus INSTATE Threat:                         | 24 ##### |
| (33) OtherTrackedRangeRate() > RDTHRТА:                     | 28 ##### |
| (34) OtherTrackedRange() <= DMODТА():                       | 16 ##### |
| (35) (OtherTrackedRange()*OtherTrackedRangeRate())<=H1ТА(): | 16 ##### |
| (36) ТАURТА() < TRTHRТА():                                  | 16 ##### |
| (37) OtherTrackedRange() <= RMAX:                           | 12 ####  |
| (38) ModifiedTauCapped() < TRTHR():                         | 12 ####  |
| (39) OtherTrackedRange() > DMOD():                          | 12 ####  |
| (40) (OtherTrackedRangeRate()*OtherTrackedRange()) > H1():  | 12 ####  |
| (41) OtherVRC = NoIntent:                                   | 6 ##     |
| (42) IntentReceived INSTATE IRNo:                           | 6 ##     |
| (43) CurrentVerticalSeparation() < PROXA:                   | 2        |
| (44) OtherTrackedRange() < PROXR:                           | 4 #      |
| (45) OtherAirStatus INSTATE StateAirborne:                  | 2        |
| (46) OtherAirStatus INSTATE StateOnGround:                  | 1        |
| Total: 523                                                  |          |

Figure 6.49: Partial BDD node profile for conjunction of Potential-Threat to Proximate-Traffic and to Other-Traffic with variable reordering.



(15), (45), and (46)) violated the enumerated type properties in a majority of cases. We chose to rerun the symbolic analysis with decision procedures enabled rather than to continue looking for more domain axioms; this choice was made because it was obvious that in most of the samples, the spurious inconsistency reports resulted because the symbolic analysis lacked the information about enumerated type predicates that was necessary to eliminate the spurious inconsistencies associated with these predicates. The results from the second iteration of symbolic analysis with decision procedures reported 39 potential inconsistencies. This number of inconsistency reports can easily be managed by an analyst, but we converted the output to a PVS specification and ran PVS analysis to see if PVS' decision procedures could collapse the spurious inconsistency reports. The PVS analysis reduced the 39 inconsistency reports from the symbolic analysis to 12 inconsistency reports. All of the 12 reported inconsistencies violated the domain axiom described previously (Section 6.1.1.3.2),

```
if Other-Alt-Reporting = True is FALSE then NOT Other-Air-Status
IN-STATE On-Ground.
```

We ran PVS analysis without domain axioms, on the full guarding conditions using the strategy in Figure 6.22, on a Ross-based SPARC 20 with two 150MHz CPUs. The PVS proof finished in 26.33 seconds and reported the same 12 spurious inconsistencies as reported above when we ran PVS analysis on the results from the symbolic analysis.

We added the domain axiom NOT (Other-Alt-Reporting = True) IMPLIES Other-Air-Status IN-STATE Airborne to the PVS specification of the guarding conditions. We used the proof commands in Figure 6.51 to prove the guarding conditions consistent. The proof finished in 106.29 seconds on a SPARCstation 20 Model 70 with 64MB main memory and one 75MHz CPU.

1

1

```
(skolemizeandrewrite$) strategy
(lemma "OtherAltReporting_OtherAirStatus_Assertion")
(apply (repeat* (inst?)))
(apply (repeat* (try (bddsimp) (record) (postpone)))))
```

Figure 6.51: Specific PVS commands to include a domain axiom into the analysis process and to prove two guarding conditions consistent.

#### 6.1.1.4 Transitions out of State Threat

When Intruder-Status is in state Threat (Figure 6.4), there is another aircraft on collision course with own aircraft. We analyzed the transitions out of the atomic states Passed and Failed within the state Threat, for consistency. It is important to ensure that a transition out of state Threat does not happen prematurely (i.e., when a threat still exists), since such a premature transition could lead to a collision. The transitions and their corresponding guarding conditions are shown in Figures 6.52 through 6.57.

**Transition(s):** Threat → Other-Traffic

**Location:** Other-Aircraft ▷ Intruder-Status<sub>s-136</sub>

**Trigger Event:** Air-Status-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                              |    |   |
|-----|----------------------------------------------|----|---|
| AND | Alt-Reporting <sub>s-101</sub> in state Lost | OR |   |
|     | Other-Bearing-Valid <sub>v-120</sub> = True  | T  | T |
|     | Other-Range-Valid <sub>v-117</sub> = True    | F  | · |
|     | Potential-Threat-Range-Test <sub>m-214</sub> | ·  | F |
|     |                                              | T  | T |

**Output Action:** Intruder-Status-Evaluated-Event<sub>e-279</sub>

Figure 6.52: Transition from Threat to Other-Traffic.

For each pair of transitions, we performed both symbolic analysis using BDDs, and PVS analysis. The symbolic analyses included the macro expansion, sift reordering, and path count options. The symbolic analyses either showed the guarding conditions consistent on the first iteration, or yielded a manageable number of error reports. We performed PVS analysis on both the original

**Transition(s):** Threat → Potential-Threat

**Location:** Other-Aircraft ▷ Intruder-Status<sub>s-136</sub>

**Trigger Event:** Air-Status-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                              |   |   |   |
|-----|----------------------------------------------|---|---|---|
| AND | Alt-Reporting <sub>s-101</sub> in state Lost | T | T | F |
|     | Other-Bearing-Valid <sub>v-120</sub> = True  | T | . | . |
|     | Other-Range-Valid <sub>v-117</sub> = True    | T | . | . |
|     | Potential-Threat-Range-Test <sub>m-214</sub> | . | F | . |
|     | RA-Inhibit <sub>m-219</sub>                  | . | . | T |

**Output Action:** Intruder-Status-Evaluated-Event<sub>e-279</sub>

Figure 6.53: Transition from Threat to Potential-Threat.

**Transition(s):** Failed → Potential-Threat

**Location:** Other-Aircraft ▷ Intruder-Status<sub>s-136</sub>

**Trigger Event:** Air-Status-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                                         |   |
|-----|-------------------------------------------------------------------------|---|
| AND | $t > \text{Time-Advisory-Changed}_{f-258} + 5.5s_{(TMIN)} - 1\text{ s}$ | T |
|     | Threat-Range-Test <sub>m-222</sub>                                      | F |
|     | Other-Air-Status <sub>s-101</sub> in state Airborne                     | T |
|     | Alt-Reporting <sub>s-101</sub> in state Lost                            | F |

**Output Action:** Intruder-Status-Evaluated-Event<sub>e-279</sub>

Figure 6.54: Transition from Failed to Potential-Threat.

**Transition(s):** Failed → Other-Traffic

**Location:** Other-Aircraft ▷ Intruder-Status<sub>s-136</sub>

**Trigger Event:** Air-Status-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                                         |   |
|-----|-------------------------------------------------------------------------|---|
| AND | $t > \text{Time-Advisory-Changed}_{f-258} + 5.5s_{(TMIN)} - 1\text{ s}$ | T |
|     | Other-Air-Status <sub>s-101</sub> in state Airborne                     | F |

**Output Action:** Intruder-Status-Evaluated-Event<sub>e-279</sub>

Figure 6.55: Transition from Failed to Other-Traffic.

**Transition(s):** Failed  $\rightarrow$  Passed

**Location:** Threat  $\triangleright$  Range-Test<sub>s-136</sub>

**Trigger Event:** Air-Status-Evaluated-Event<sub>e-279</sub>

**Condition:**

|                                                     |   |
|-----------------------------------------------------|---|
| Threat-Range-Test <sub>m-222</sub>                  | T |
| Other-Air-Status <sub>s-101</sub> in state Airborne | T |

**Output Action:** Range-Test-Evaluated-Event

Figure 6.56: Transition from Failed to Passed.

**Transition(s):** Passed  $\rightarrow$  Failed

**Location:** Threat  $\triangleright$  Range-Test<sub>s-136</sub>

**Trigger Event:** Air-Status-Evaluated-Event<sub>e-279</sub>

**Condition:**

|                                                      |   |   |
|------------------------------------------------------|---|---|
| Threat-Range-Test <sub>m-222</sub>                   | F | . |
| Other-Air-Status <sub>s-101</sub> in state On-Ground | . | T |

OR

**Output Action:** Range-Test-Evaluated-Event

Figure 6.57: Transition from Passed to Failed.

guarding conditions, and on the output of the symbolic analysis if the symbolic analysis reported any inconsistencies. In all PVS analyses we used the proof strategy in Figure 6.22.

To verify that the analysis components produced the same results, we compared the results output from the PVS analysis on the original guarding conditions, the symbolic analysis on the original guarding conditions, and the PVS analysis on any reported inconsistencies from the symbolic analysis. In all cases, the guarding conditions proved consistent by either the analysis procedures or by inspection.

The guarding conditions for transitions out of the atomic states Passed and Failed are not as complex as the transitions we discussed earlier out of the atomic states of Intruder-Status. Therefore, we provide only a summary of the results here.

#### 6.1.1.4.1 Summary of Results

For the atomic states `Passed` and `Failed` within the state `Threat`, we analyzed a total of fourteen pairs of guarding conditions. Of these fourteen, five proved consistent after a single iteration of both the symbolic analysis and the PVS analysis. Both the symbolic analysis and the PVS analysis required only a few seconds to report that the guarding conditions were consistent.

For the remaining nine pairs, the first iteration of the symbolic analysis reported from four to 403 inconsistencies. All of the reported inconsistencies from the symbolic analysis were automatically translated to a PVS specification and we ran PVS analysis to see if any further reductions in the reported inconsistencies could be achieved. In all cases (except for the minimum output of four inconsistencies from symbolic analysis), the PVS output was significantly reduced. The number of inconsistencies reported by PVS analysis ranged from one to 46. We ran PVS analysis on the original guarding conditions and achieved the same inconsistency reports as we achieved from applying PVS analysis to the output of the symbolic analysis component.

Manual inspection of the reported inconsistencies seemed to confirm that the guarding conditions could both be satisfied at the same time (i.e., they were inconsistent). These potential inconsistencies were reported to the maintainers of the TCAS II specification. They reviewed the reported inconsistencies and noted that in their definition of the semantics of RSML, transitions out of a super-state take precedence over transitions out of a sub-state. Thus, both transitions cannot be satisfied at the same time; the transition from the super-state overrides the transition from the sub-state. Therefore, all of the guarding conditions for transitions out of the states `Passed` and `Failed` were, in their choice of semantics, mutually exclusive.

### 6.1.2 Transitions in State Intruder-Status - TCAS II Version 7

The latest version of the TCAS II requirements specification is version 7. We ran the same tests on the Intruder-Status portion of version 7 as we ran on the Intruder-Status portion of version 6.04A. The guarding conditions on the transitions in the Intruder-Status portion of version 7 of the specification were not as complex (but were still significantly complex) as the guarding conditions from version 6.04A of the specification (i.e., the AND/OR tables in version 7 of the specification contained slightly fewer predicates and slightly fewer columns). In addition, there was one less transition out of the super-state Threat. Many of the guarding condition pairs that were inconsistent in version 6.04A of the specification, were consistent in version 7 of the specification.

To quickly identify and eliminate from future analysis the consistent guarding conditions, we initially ran symbolic analysis on the guarding conditions of all transitions at once, rather than on only two guarding conditions at a time; i.e., the machine readable specification included specifications for all guarding conditions and our analysis procedure set up and performed the consistency checks for all pairs of guarding conditions out of each state. The first run symbolic analysis showed the following guarding conditions consistent with no augmenting information required:

- Other-Traffic-TO-Proximate-Traffic  
and Other-Traffic-TO-Potential-Threat
- Potential-Threat-TO-Other-Traffic  
and Potential-Threat-TO-Proximate-Traffic
- Potential-Threat-TO-Other-Traffic  
and Potential-Threat-TO-Threat
- Proximate-Traffic-TO-Threat  
and Proximate-Traffic-TO-Other-Traffic
- Proximate-Traffic-TO-Other-Traffic  
and Proximate-Traffic-TO-Potential-Threat
- Failed-TO-Passed and Threat-TO-Potential-Threat

Initial PVS analysis also yielded the same results.

For the remaining large guarding conditions that symbolic analysis reported as inconsistent, there were millions of potential inconsistencies reported, just as in the analysis of version 6.04A of the specification. In addition, PVS analysis of the guarding conditions that involved the `Threat` macro always ran on the order of a day or more before we aborted the process. We used the same method described in Section 6.1.1 to identify domain axioms in the specification that did not hold in the result BDD. Only one domain axiom was required in all of the cases: `Other-Alt-Reporting = True iff Alt-Reporting IN-STATE Yes`. Once we added this domain axiom to the analysis process, both the symbolic analysis and PVS analysis showed the guarding conditions consistent.

## 6.2 Completeness Analysis

We applied our completeness analysis technique to the transitions out of `Auto-SL` state 1 and to the transitions out of `Effective-SL` state 4. To demonstrate the effectiveness of our method applied to completeness analysis, we first give a detailed description of the results obtained when we applied our method to `Auto-SL` state 1. Next, we provide a summary of the results obtained when we applied our analysis method to the guarding conditions on transitions out of `Effective-SL` state 4.

### 6.2.1 Transitions in State Auto-SL

Figures 6.58 through 6.64 show the guarding conditions for the transitions out of the `Auto-SL` atomic states.

First, recall (Chapter 4, Section 4.2.6.2) that when our symbolic analysis component checks the guarding conditions out of a state triggered by the same event for completeness, the paths leading to



Transition(s): ANY → 1

Location: Own-Aircraft ▸ Auto-SL<sub>s-30</sub>

Trigger Event: Descend-Inhibit-Evaluated-Event<sub>e-279</sub>

Condition:

|     |                                                  |   |   |
|-----|--------------------------------------------------|---|---|
| AND | Own-Air-Status <sub>v-36</sub> = On-Ground       | T | T |
|     | Traffic-Display-Permitted <sub>v-39</sub> = True | · | F |
|     | Mode-Selector <sub>v-34</sub> = Standby          | T | · |

OR

Output Action: Auto-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.58: Guarding condition for transition from any Auto-SL state to Auto-SL state 1.

Transition(s): ANY → 2

Location: Own-Aircraft ▸ Auto-SL<sub>s-30</sub>

Trigger Event: Descend-Inhibit-Evaluated-Event<sub>e-279</sub>

Condition:

|     |                                                              |   |   |   |   |
|-----|--------------------------------------------------------------|---|---|---|---|
| AND | Effective-SL <sub>s-30</sub> in one of {1,2}                 | T | · | · | · |
|     | Effective-SL <sub>s-30</sub> in state 3                      | · | · | · | T |
|     | Own-Alt-Radio <sub>v-31</sub> < 1100 ft <sub>(ZSL2TO3)</sub> | T | · | · | · |
|     | Own-Alt-Radio <sub>v-31</sub> ≤ 900 ft <sub>(ZSL3TO2)</sub>  | · | · | · | T |
|     | Climb-Desc.-Inhibit <sub>m-202</sub>                         | F | · | T | F |
|     | Own-Air-Status <sub>v-36</sub> = Airborne                    | T | F | T | T |
|     | Traffic-Display-Permitted <sub>v-39</sub>                    | · | T | · | · |
|     | Radarout-EQ-0 <sub>m-201</sub>                               | T | · | · | T |
|     | Mode-Selector = one of {TA/RA,TA-Only,3,4,5,6,7}             | · | T | · | · |

OR

Output Action: Auto-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.59: Guarding condition for transition from any Auto-SL state to Auto-SL state 2.

**Transition(s):** *ANY*  $\rightarrow$  3

**Location:** Own-Aircraft  $\triangleright$  Auto-SL<sub>s-30</sub>

**Trigger Event:** Descend-Inhibit-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                                      |   |   |   |   |   |   |
|-----|----------------------------------------------------------------------|---|---|---|---|---|---|
| AND | Effective-SL <sub>s-30</sub> in one of {1,2}                         | T | T | . | . | . | . |
|     | Effective-SL <sub>s-30</sub> in state 3                              | . | . | . | T | T | T |
|     | Effective-SL <sub>s-30</sub> in state 4                              | . | . | T | T | . | . |
|     | Own-Alt-Radio <sub>v-31</sub> $\geq 1100$ ft <sub>(ZSL2TO3)</sub>    | T | . | . | . | . | . |
|     | Own-Alt-Radio <sub>v-31</sub> $\leq 2150$ ft <sub>(ZSL4TO3)</sub>    | . | . | T | . | . | . |
|     | Climb-Desc.-Inhibit <sub>m-202</sub>                                 | F | F | F | F | F | F |
|     | Own-Air-Status <sub>v-36</sub> = Airborne                            | T | T | T | T | T | T |
|     | Radar-Bad-For-RADARLOST-Cycles <sub>m-217</sub>                      | . | T | . | T | T | F |
|     | Radarout-EQ-0 <sub>m-201</sub>                                       | T | F | T | F | T | F |
|     | Own-Tracked-Alt <sub>f-248</sub> $\leq 2150$ ft <sub>(ZSL4TO3)</sub> | . | . | . | T | . | . |
|     | Own-Alt-Radio <sub>v-31</sub> $< 2550$ ft <sub>(ZSL3TO4)</sub>       | T | . | . | . | T | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $< 2550$ ft <sub>(ZSL3TO4)</sub>    | . | T | . | . | T | . |
|     | Own-Alt-Radio <sub>v-31</sub> $> 900$ ft <sub>(ZSL3TO2)</sub>        | . | . | . | T | . | . |
|     |                                                                      |   |   |   |   |   |   |

OR

**Output Action:** Auto-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.60: Guarding condition for transition from any Auto-SL state to Auto-SL state 3.

**Transition(s):** *ANY*  $\rightarrow$  4

**Location:** Own-Aircraft  $\triangleright$  Auto-SL<sub>s-30</sub>

**Trigger Event:** Descend-Inhibit-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                                      |   |   |   |   |   |   |
|-----|----------------------------------------------------------------------|---|---|---|---|---|---|
| AND | Effective-SL <sub>s-30</sub> in one of {1,2,3}                       | T | T | . | . | . | . |
|     | Effective-SL <sub>s-30</sub> in state 4                              | . | . | . | T | T | T |
|     | Effective-SL <sub>s-30</sub> in state 5                              | . | . | T | T | . | . |
|     | Own-Alt-Radio <sub>v-31</sub> $\geq 2550$ ft <sub>(ZSL3TO4)</sub>    | T | . | . | . | . | . |
|     | Own-Alt-Radio <sub>v-31</sub> $\leq 4500$ ft <sub>(ZSL5TO4)</sub>    | . | . | T | . | . | . |
|     | Climb-Desc.-Inhibit <sub>m-202</sub>                                 | F | F | F | F | F | F |
|     | Own-Air-Status <sub>v-36</sub> = Airborne                            | T | T | T | T | T | T |
|     | Radar-Bad-For-RADARLOST-Cycles <sub>m-217</sub>                      | . | T | . | T | T | F |
|     | Radarout-EQ-0 <sub>m-201</sub>                                       | T | F | T | F | T | F |
|     | Own-Tracked-Alt <sub>f-248</sub> $\geq 2550$ ft <sub>(ZSL3TO4)</sub> | . | T | . | . | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $\leq 4500$ ft <sub>(ZSL5TO4)</sub> | . | . | . | T | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $< 5500$ ft <sub>(ZSL4TO5)</sub>    | . | T | . | . | T | . |
|     | Own-Alt-Radio <sub>v-31</sub> $> 2150$ ft <sub>(ZSL4TO3)</sub>       | . | . | . | . | T | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $> 2150$ ft <sub>(ZSL4TO3)</sub>    | . | . | . | . | T | . |

OR

**Output Action:** Auto-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.61: Guarding condition for transition from any Auto-SL state to Auto-SL state 4.

**Transition(s):**  $ANY \rightarrow \boxed{5}$

**Location:** Own-Aircraft  $\triangleright$  Auto-SL<sub>s-30</sub>

**Trigger Event:** Descend-Inhibit-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                                      |   |   |   |   |   |   |
|-----|----------------------------------------------------------------------|---|---|---|---|---|---|
| AND | Effective-SL <sub>s-30</sub> in one of {1,2,3,4}                     | T | . | . | . | . | . |
|     | Effective-SL <sub>s-30</sub> in state 5                              | . | . | . | T | T | T |
|     | Effective-SL <sub>s-30</sub> in state 6                              | . | T | T | . | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $\geq 5500$ ft <sub>(ZSL4TO5)</sub> | T | . | . | . | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $< 9500$ ft <sub>(ZSL6TO5)</sub>    | . | . | T | . | . | . |
|     | Own-Alt-Radio <sub>v-31</sub> $\leq 9500$ ft <sub>(ZSL6TO5)</sub>    | . | T | . | . | . | . |
|     | Climb-Desc.-Inhibit <sub>m-202</sub>                                 | F | F | F | F | F | F |
|     | Own-Air-Status <sub>v-36</sub> = Airborne                            | T | T | T | T | T | T |
|     | Radar-Bad-For-RADARLOST-Cycles <sub>m-217</sub>                      | T | . | T | T | . | F |
|     | Radarout-EQ-0 <sub>m-201</sub>                                       | F | T | F | F | T | F |
|     | Own-Tracked-Alt <sub>f-248</sub> $< 10500$ ft <sub>(ZSL5TO6)</sub>   | T | . | . | T | . | . |
|     | Own-Alt-Radio <sub>v-31</sub> $> 4500$ ft <sub>(ZSL5TO4)</sub>       | . | . | . | . | T | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $> 4500$ ft <sub>(ZSL5TO4)</sub>    | . | . | . | T | . | . |
|     |                                                                      | . | . | . | . | . | . |

OR

**Output Action:** Auto-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.62: Guarding condition for transition from any Auto-SL state to Auto-SL state 5.

**Transition(s):**  $ANY \rightarrow \boxed{6}$

**Location:** Own-Aircraft  $\triangleright$  Auto-SL<sub>s-30</sub>

**Trigger Event:** Descend-Inhibit-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                                       |   |   |   |   |   |
|-----|-----------------------------------------------------------------------|---|---|---|---|---|
| AND | Effective-SL <sub>s-30</sub> in one of {1,2,3,4,5}                    | T | . | . | . | . |
|     | Effective-SL <sub>s-30</sub> in state 6                               | . | . | T | T | T |
|     | Effective-SL <sub>s-30</sub> in state 7                               | . | T | . | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $\geq 10500$ ft <sub>(ZSL5TO6)</sub> | T | . | . | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $\leq 19500$ ft <sub>(ZSL7TO6)</sub> | . | T | . | . | . |
|     | Climb-Desc.-Inhibit <sub>m-202</sub>                                  | F | F | F | F | F |
|     | Own-Air-Status <sub>v-36</sub> = Airborne                             | T | T | T | T | T |
|     | Radar-Bad-For-RADARLOST-Cycles <sub>m-217</sub>                       | T | . | . | T | F |
|     | Radarout-EQ-0 <sub>m-201</sub>                                        | F | . | T | F | F |
|     | Own-Tracked-Alt <sub>f-248</sub> $< 20500$ ft <sub>(ZSL6TO7)</sub>    | T | . | . | T | . |
|     | Own-Alt-Radio <sub>v-31</sub> $> 9500$ ft <sub>(ZSL6TO5)</sub>        | . | . | T | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> $> 9500$ ft <sub>(ZSL6TO5)</sub>     | . | . | . | T | . |
|     |                                                                       | . | . | . | . | . |
|     |                                                                       | . | . | . | . | . |

OR

**Output Action:** Auto-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.63: Guarding condition for transition from any Auto-SL state to Auto-SL state 6.

**Transition(s):** *ANY*  $\rightarrow$  7

**Location:** Own-Aircraft  $\triangleright$  Auto-SL<sub>s-30</sub>

**Trigger Event:** Descend-Inhibit-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                                       |    |   |
|-----|-----------------------------------------------------------------------|----|---|
|     |                                                                       | OR |   |
| AND | Effective-SL <sub>s-30</sub> in one of {1,2,3,4,5,6}                  | T  | · |
|     | Effective-SL <sub>s-30</sub> in state 7                               | ·  | T |
|     | Own-Tracked-Alt <sub>f-248</sub> $\geq$ 20500 ft <sub>(ZSL6TO7)</sub> | T  | · |
|     | Climb-Desc.-Inhibit <sub>m-202</sub>                                  | F  | F |
|     | Own-Air-Status <sub>v-36</sub> = Airborne                             | T  | T |
|     | Radar-Bad-For-RADARLOST-Cycles <sub>m-217</sub>                       | T  | · |
|     | Radarout-EQ-0 <sub>m-201</sub>                                        | F  | · |
|     | Own-Tracked-Alt <sub>f-248</sub> $>$ 19500 ft <sub>(ZSL7TO6)</sub>    | ·  | T |

**Output Action:** Auto-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.64: Guarding condition for transition from any Auto-SL state to Auto-SL state 7.

FALSE in the result BDD represent potential incompletenesses in the specification of the guarding conditions. This is because if the specifications of the guarding conditions are complete, the disjunction of the guarding conditions will be TRUE (a tautology). If the disjunction of the guarding conditions does not reduce to TRUE, then there are some conditions for which a transition out of a state is not possible. These conditions are represented in the result BDD as paths leading to FALSE, and are reported to the analyst as conditions for which the specification provides no transitions out of a particular state. The analyst must then determine a way to incorporate the missing conditions into the specification of the guarding conditions for the particular state under consideration.

Recall further, that we use the same routine in both completeness and consistency analysis to convert the result BDD to AND/OR table format to present the potential incompletenesses and inconsistencies to the analyst. As we discussed in Chapter 4, Section 4.2.6, every satisfying path in the result BDD becomes a column in the AND/OR table. In completeness analysis we want to report unsatisfiable paths to the analyst. Therefore, we simply negate the result BDD before we send it to the AND/OR *table translator* portion of our tool, so that the unsatisfiable paths in the result BDD

become satisfiable paths, and vice versa. This means that if there is some contradiction among any of the predicates in a column of the AND/OR table, then that particular condition does not represent a true incompleteness in the specification; the reported condition cannot happen in the first place, so it need not be added to the specification of the guarding conditions to make the specification complete. This latter fact is significant for interpreting and explaining the results we obtained below for the guarding conditions for Auto-SL state 1.

Note that the guarding conditions for the transitions shown in Figures 6.59 through 6.64 contain many relational expressions such as,  $\text{Own-Alt-Radio} \geq 1100$  in Figure 6.60. These are simple relational expressions and it is easy to see that many combinations of them cannot be satisfied at the same time, and that many of them are mutually exclusive (for example, the predicates  $\text{Own-Alt-Radio} < 1100$  and  $\text{Own-Alt-Radio} \geq 1100$  cannot both be FALSE at the same time, and one of them has to be TRUE). In Figure 6.65 I have collected and grouped all of the relational expressions involving the variable  $\text{Own-Alt-Radio}$  (abbreviated OAR in the figure), according to obvious mutually exclusive and all-inclusive relationships.

There are also various other combinations of the relational predicates that are mutually exclusive. For example, if the predicate  $\text{Own-Alt-Radio} > 9500$  is TRUE, then the predicate  $\text{Own-Alt-Radio} < 2550$  cannot be TRUE at the same time. In addition, there are many multiple dependencies between the relational predicates, such that three or more of the relational predicates are mutually exclusive. Relational predicates such as those described here, wreak havoc for symbolic analysis methods using BDDs. It is possible to develop decision procedures to detect non-satisfiable combinations of relational predicates such as these, but we chose not to do this.

We applied our symbolic analysis using BDDs to check the guarding conditions on transitions from the Auto-SL states for completeness. First, we applied the analysis without our decision procedures for enumerated type predicates, and with variable reordering. The number of reported

|              |             |
|--------------|-------------|
| OAR <= 900,  | OAR > 900   |
| OAR < 1100,  | OAR >= 1100 |
| OAR <= 2150, | OAR > 2150  |
| OAR < 2550,  | OAR >= 2550 |
| OAR <= 4500, | OAR > 4500  |
| OAR <= 9500, | OAR > 9500  |

Figure 6.65: Mutually exclusive and all-inclusive relational expressions involving the variable Own-Alt-Radio (OAR).

incompletenesses was on the order of 120,000. We could not determine the domain axioms using our indicator nodes approach; we will explain why, shortly. We reran symbolic analysis with both decision procedures and variable reordering. The number of reported incompletenesses was on the order of 10,000; a significant reduction in reported incompletenesses, but not enough of a reduction to be helpful to the analyst in identifying any true incompletenesses in the specifications of the guarding conditions. We examined the samples output from the latter symbolic analysis and discovered some obvious contradictions between the relational predicates as noted above. Clearly, the information our analysis process lacked, was the information about the mutually exclusive and all-inclusive aspects of the relational predicates. We attempted to add the missing information to the specification and rerun the analysis, but there were too many multi-way interdependencies between too many of the relational predicates for both the Own-Alt-Radio and the Other-Tracked-Alt variables. In other words, there were too many different combinations of the relational predicates that were resulting in the spurious incompleteness reports and there was too much information missing from the analysis process. It was not feasible to attempt to add all of the required axioms describing the unsatisfiable combinations of the relational predicates, and we did not want to complicate our symbolic analysis by incorporating decision procedures for relational expressions. Especially since PVS has decision procedures that can easily deal with the problems we encountered.

We applied PVS analysis to the full guarding conditions on a 200MHz Pentium Pro machine

with 64MB RAM running Redhat Linux. The strategy we used is in Figure 6.66. PVS reported eight incompletenesses in under five minutes. The eight reported errors reduced to one because PVS split up one enumerated type IN-ONE-OF predicate and reported it as two separate unprovable subgoals in four instances. In addition, these four potential incompletenesses involved the permutations of the predicates in three macro predicates; we simply needed to specify different truth values for the macro predicates, rather than providing all possible combinations of TRUE and FALSE for the two predicates in each macro. The three macro predicates (and the one included macro) involved in the incompleteness reports are shown in Figures 6.67 through 6.70. The results from the PVS analysis are shown in Figure 6.71.

```
(defstep complete2
 (apply (then (skolem!)
 (rewrite-msg-off)
 (auto-rewrite-defs$)
 (do-rewrite$)
 (repeat* (try (bddsimp)
 (try (record) (assert) (postpone))
 (skip))))))
```

Figure 6.66: PVS strategy for proving guarding conditions complete.

### Macro: Radar-Bad-For-RADARLOST-cycles

#### Definition:

For each  $j \in \{1, 2, \dots, 10_{(\text{RADARLOST})}\}$ :

|                                           |                                      |    |   |
|-------------------------------------------|--------------------------------------|----|---|
| $\begin{matrix} A \\ N \\ D \end{matrix}$ | PREV $j$ (Radarout-EQ-0 $_{m-201}$ ) | OR |   |
|                                           | Standby-Since $_{m-202}(j)$          | F  | · |
|                                           |                                      | ·  | T |

Figure 6.67: The Radar-Bad-For-RADARLOST-cycles macro.

### Macro: Radarout-EQ-0

#### Definition:

|                                           |                                          |   |
|-------------------------------------------|------------------------------------------|---|
| $\begin{matrix} A \\ N \\ D \end{matrix}$ | Own-Alt-Radio $_{v-31}$ is credible      | T |
|                                           | Radio-Altimeter-Status $_{v-35}$ = valid | T |

Figure 6.68: The Radarout-EQ-0-macro.

**Macro: Climb-Desc.-Inhibit****Definition:**

|                                           |                                                    |                                      |
|-------------------------------------------|----------------------------------------------------|--------------------------------------|
| $\begin{matrix} A \\ N \\ D \end{matrix}$ | Climb-Inhibit <sub>s-30</sub> in state Inhibited   | $\begin{matrix} T \\ F \end{matrix}$ |
|                                           | Descend-Inhibit <sub>s-30</sub> in state Inhibited | $\begin{matrix} T \\ F \end{matrix}$ |

Figure 6.69: The Climb-Desc.-Inhibit macro.

**Macro: Standby-Since(i)****Definition:****There exists  $a$  such that**

|                                           |                                                                                |                                      |
|-------------------------------------------|--------------------------------------------------------------------------------|--------------------------------------|
| $\begin{matrix} A \\ N \\ D \end{matrix}$ | $a \leq i$                                                                     | $\begin{matrix} T \\ F \end{matrix}$ |
|                                           | PREV <sub>a</sub> (TCAS-Controller <sub>s-11</sub> in state Fully-Operational) | $\begin{matrix} T \\ F \end{matrix}$ |

Figure 6.70: The Standby-Since macro.

Recall (Chapter 2, Section 2.2.3.4.1) that the interpretation of an unprovable PVS subgoal is the conjunction of the formulas in the antecedent implies the disjunction of the formulas in the consequent. Since  $a \Rightarrow b \equiv \neg a \vee b$ , and since by DeMorgan's Law  $\neg(a_1 \wedge a_2) \equiv \neg a_1 \vee \neg a_2$ , our unprovable subgoals can be written in disjunctive form. The disjunction of the negation of the individual formulas in the antecedent forms one disjunct and the disjunction of the individual formulas in the consequent forms the other disjunct. To convert the subgoals into the proper form for our tabular notation, we simply need to apply DeMorgan's Law to the disjunctive formula. In the resulting formula, each individual formula from the original subgoal is a conjunct; thus, each unprovable subgoal may become one column in an AND/OR table. We show now that there is not necessarily a one-to-one correspondence between subgoals and columns in an AND/OR table.

An examination of the results shown in Figure 6.71 reveals that formulas  $\{-2\}$  and  $\{5\}$  in each of the reported incompletenesses, requires that the macro Radar-Bad-For-RADARLOST-Cycles (Figure 6.67) be FALSE. Formula  $\{3\}$  in each of the reported incompletenesses requires that the macro Climb-Desc.-Inhibit (Figure 6.69)



|                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>SUBGOAL.1 :</p> <pre> {-1} ESL_1?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} D_Inhibited?(Descend_Inhibit!1) {4} Valid?(Radio_Altimeter_Status!1) {5} cTrue?(Standby_Since_j!1) </pre> | <p>SUBGOAL.2 :</p> <pre> {-1} ESL_2?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} D_Inhibited?(Descend_Inhibit!1) {4} Valid?(Radio_Altimeter_Status!1) {5} cTrue?(Standby_Since_j!1) </pre> |
| <p>SUBGOAL.3 :</p> <pre> {-1} ESL_1?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} D_Inhibited?(Descend_Inhibit!1) {4} Own_Alt_Radio!1 = Credible!1 {5} cTrue?(Standby_Since_j!1) </pre>     | <p>SUBGOAL.4 :</p> <pre> {-1} ESL_2?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} D_Inhibited?(Descend_Inhibit!1) {4} Own_Alt_Radio!1 = Credible!1 {5} cTrue?(Standby_Since_j!1) </pre>     |
| <p>SUBGOAL.5 :</p> <pre> {-1} ESL_1?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} C_Inhibited?(Climb_Inhibit!1) {4} Valid?(Radio_Altimeter_Status!1) {5} cTrue?(Standby_Since_j!1) </pre>   | <p>SUBGOAL.6 :</p> <pre> {-1} ESL_2?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} C_Inhibited?(Climb_Inhibit!1) {4} Valid?(Radio_Altimeter_Status!1) {5} cTrue?(Standby_Since_j!1) </pre>   |
| <p>SUBGOAL.7 :</p> <pre> {-1} ESL_1?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} C_Inhibited?(Climb_Inhibit!1) {4} Own_Alt_Radio!1 = Credible!1 {5} cTrue?(Standby_Since_j!1) </pre>       | <p>SUBGOAL.8 :</p> <pre> {-1} ESL_2?(Effective_SL!1) {-2} cTrue?(PREV_j_Radarout_EQ!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} C_Inhibited?(Climb_Inhibit!1) {4} Own_Alt_Radio!1 = Credible!1 {5} cTrue?(Standby_Since_j!1) </pre>       |

Figure 6.71: Unprovable subgoals from PVS analysis.

be FALSE. Formula {4} in each of the reported incompletenesses requires that the macro Radarout-EQ-0 (Figure 6.68) be FALSE. This means that the eight unprovable subgoals without the macros expanded, become the two subgoals shown in Figure 6.72. Each of the two subgoals

|                                                                                                                                                                                                          |                                                                                                                                                                                                          |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>SUBGOAL.1 :</p> <pre>{-1} ESL_1?(Effective_SL!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} Climb_Desc_Inhibit {4} Radarout_EQ_0 {5} Radar_Bad_For_RADARLOST_Cycles</pre> | <p>SUBGOAL.2 :</p> <pre>{-1} ESL_2?(Effective_SL!1)  ----- {1} On_Ground?(Own_Air_Status!1) {2} ESL_3?(Effective_SL!1) {3} Climb_Desc_Inhibit {4} Radarout_EQ_0 {5} Radar_Bad_For_RADARLOST_Cycles</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Figure 6.72: Reduced unprovable subgoals from PVS analysis.

could become a separate column in an AND/OR table, thus, representing two incompletenesses. However, recall that Effective-SL can be in only one of its sub-states at a time, and that we can represent this using the IN-ONE-OF predicate; Effective-SL IN-ONE-OF {1,2}. Thus, the eight reported incompletenesses become the one incompleteness shown in Figure 6.73; i.e., this condition is reported as missing from the specification and as a condition that should be included in the specification of the guarding conditions to make the specification complete. We examined the specification and determined that the reported incompleteness was a true incompleteness. We added the missing condition to the specification of the guarding conditions and reran PVS analysis on the same machine and using the same strategy. PVS reported the specification of the guarding conditions complete in under six minutes.

|                                      |   |
|--------------------------------------|---|
| Effective-SL IN-ONE-OF {1,2}         | T |
| Effective-SL IN-STATE 3              | F |
| Own-Air-Status = On-Ground           | F |
| Climb-Descend-Inhibit-Macro          | F |
| Radarout-EQ-0-Macro                  | F |
| Radar-Bad-For-RADARLOST-Cycles-Macro | F |

Figure 6.73: Condition to add to specification of guarding conditions to make the specification complete.

### 6.2.2 Transitions in State Effective-SL

Figures 6.74 through 6.80 show the guarding conditions for the transitions in the state Effective-SL. Note that the majority of the predicates involved in the guarding conditions shown in the figures, are enumerated type predicates: **in state**, **in one of**, **equal**, and **equal one of**.

**Transition(s):** *ANY*  $\rightarrow$  1

**Location:** Own-Aircraft  $\triangleright$  Effective-SL<sub>s-30</sub>

**Trigger Event:** Auto-SL-Evaluated-Event<sub>e-279</sub>

**Condition:**

|                                           |                                           |    |   |
|-------------------------------------------|-------------------------------------------|----|---|
| $\begin{matrix} A \\ N \\ D \end{matrix}$ | Auto-SL <sub>s-30</sub> <b>in state 1</b> | OR |   |
|                                           | Mode-Selector <sub>v-34</sub> = Standby   | T  | · |
|                                           |                                           | ·  | T |

**Output Action:** Effective-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.74: Guarding condition for transition from any Effective-SL state to Effective-SL state 1.

**Transition(s):** *ANY*  $\rightarrow$  2

**Location:** Own-Aircraft  $\triangleright$  Effective-SL<sub>s-30</sub>

**Trigger Event:** Auto-SL-Evaluated-Event<sub>e-279</sub>

**Condition:**

|                                           |                                           |   |
|-------------------------------------------|-------------------------------------------|---|
| $\begin{matrix} A \\ N \\ D \end{matrix}$ | Auto-SL <sub>s-30</sub> <b>in state 2</b> | T |
|                                           | Mode-Selector <sub>v-34</sub> = Standby   | F |

**Output Action:** Effective-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.75: Guarding condition for transition from any Effective-SL state to Effective-SL state 2.

We applied symbolic analysis using BDDs to check the guarding conditions to see if their specification was complete. In the first iteration of symbolic analysis we did not enable our decision procedures. The output from the symbolic analysis reported on the order of 130,000 potential incompletenesses. We reran symbolic analysis with decision procedures enabled and the analysis reported zero satisfying paths in the result BDD, signifying that the specification of the guarding

**Transition(s):** *ANY* → 3

**Location:** Own-Aircraft ▸ Effective-SL<sub>s-30</sub>

**Trigger Event:** Auto-SL-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                          |   |   |   |   |   |
|-----|----------------------------------------------------------|---|---|---|---|---|
| AND | Auto-SL <sub>s-30</sub> in state 3                       | T | T | T | . | . |
|     | Auto-SL <sub>s-30</sub> in one of {3,4,5,6,7}            | . | . | . | T | T |
|     | Lowest-Ground <sub>f-241</sub> = one of {3,4,5,6,7,None} | T | . | . | . | T |
|     | Lowest-Ground <sub>f-241</sub> = 2                       | . | . | T | . | . |
|     | Lowest-Ground <sub>f-241</sub> = 3                       | . | . | . | T | . |
|     | Mode-Selector = one of {TA/RA,3,4,5,6,7}                 | T | . | . | T | . |
|     | Mode-Selector <sub>v-34</sub> = TA-Only                  | . | T | . | . | . |
|     | Mode-Selector = one of {TA/RA,TA-Only,3,4,5,6,7}         | . | . | T | . | . |
|     | Mode-Selector <sub>v-34</sub> = 3                        | . | . | . | . | T |

OR

**Output Action:** Effective-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.76: Guarding condition for transition from any Effective-SL state to Effective-SL state 3.

**Transition(s):** *ANY* → 4

**Location:** Own-Aircraft ▸ Effective-SL<sub>s-30</sub>

**Trigger Event:** Auto-SL-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                        |   |   |   |   |   |
|-----|--------------------------------------------------------|---|---|---|---|---|
| AND | Auto-SL <sub>s-30</sub> in state 4                     | T | T | T | . | . |
|     | Auto-SL <sub>s-30</sub> in one of {4,5,6,7}            | . | . | . | T | T |
|     | Lowest-Ground <sub>f-241</sub> = one of {4,5,6,7,None} | T | . | . | . | T |
|     | Lowest-Ground <sub>f-241</sub> = 2                     | . | . | T | . | . |
|     | Lowest-Ground <sub>f-241</sub> = 4                     | . | . | . | T | . |
|     | Mode-Selector = one of {TA/RA,4,5,6,7}                 | T | . | . | T | . |
|     | Mode-Selector <sub>v-34</sub> = TA-Only                | . | T | . | . | . |
|     | Mode-Selector = one of {TA/RA,TA-Only,3,4,5,6,7}       | . | . | T | . | . |
|     | Mode-Selector <sub>v-34</sub> = 4                      | . | . | . | . | T |

OR

**Output Action:** Effective-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.77: Guarding condition for transition from any Effective-SL state to Effective-SL state 4.

**Transition(s):** ANY → 5

**Location:** Own-Aircraft ▸ Effective-SL<sub>s-30</sub>

**Trigger Event:** Auto-SL-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                      |   |   |   |   |   |
|-----|------------------------------------------------------|---|---|---|---|---|
| AND | Auto-SL <sub>s-30</sub> in state 5                   | T | T | T | · | · |
|     | Auto-SL <sub>s-30</sub> in one of {5,6,7}            | · | · | · | T | T |
|     | Lowest-Ground <sub>f-241</sub> = one of {5,6,7,None} | T | · | · | · | T |
|     | Lowest-Ground <sub>f-241</sub> = 2                   | · | · | T | · | · |
|     | Lowest-Ground <sub>f-241</sub> = 5                   | · | · | · | T | · |
|     | Mode-Selector = one of {TA/RA,5,6,7}                 | T | · | · | T | · |
|     | Mode-Selector <sub>v-34</sub> = TA-Only              | · | T | · | · | · |
|     | Mode-Selector = one of {TA/RA,TA-Only,3,4,5,6,7}     | · | · | T | · | · |
|     | Mode-Selector <sub>v-34</sub> = 5                    | · | · | · | · | T |

OR

**Output Action:** Effective-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.78: Guarding condition for transition from any Effective-SL state to Effective-SL state 5.

**Transition(s):** ANY → 6

**Location:** Own-Aircraft ▸ Effective-SL<sub>s-30</sub>

**Trigger Event:** Auto-SL-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                    |   |   |   |   |   |
|-----|----------------------------------------------------|---|---|---|---|---|
| AND | Auto-SL <sub>s-30</sub> in state 6                 | T | T | T | · | · |
|     | Auto-SL <sub>s-30</sub> in one of {6,7}            | · | · | · | T | T |
|     | Lowest-Ground <sub>f-241</sub> = one of {6,7,None} | T | · | · | · | T |
|     | Lowest-Ground <sub>f-241</sub> = 2                 | · | · | T | · | · |
|     | Lowest-Ground <sub>f-241</sub> = 6                 | · | · | · | T | · |
|     | Mode-Selector = one of {TA/RA,6,7}                 | T | · | · | T | · |
|     | Mode-Selector <sub>v-34</sub> = TA-Only            | · | T | · | · | · |
|     | Mode-Selector = one of {TA/RA,TA-Only,3,4,5,6,7}   | · | · | T | · | · |
|     | Mode-Selector <sub>v-34</sub> = 6                  | · | · | · | · | T |

OR

**Output Action:** Effective-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.79: Guarding condition for transition from any Effective-SL state to Effective-SL state 6.

**Transition(s):**  $ANY \rightarrow \boxed{7}$

**Location:** Own-Aircraft  $\triangleright$  Effective-SL<sub>s-30</sub>

**Trigger Event:** Auto-SL-Evaluated-Event<sub>e-279</sub>

**Condition:**

|     |                                                        |    |   |   |
|-----|--------------------------------------------------------|----|---|---|
| AND | Auto-SL <sub>s-30</sub> in state 7                     | OR |   |   |
|     | Lowest-Ground <sub>f-241</sub> = 2                     | T  | T | T |
|     | Lowest-Ground <sub>f-241</sub> = one of {7, None}      | T  | . | . |
|     | Mode-Selector = one of {TA/RA, TA-Only, 3, 4, 5, 6, 7} | .  | . | T |
|     | Mode-Selector <sub>v-34</sub> = TA-Only                | T  | . | . |
|     | Mode-Selector = one of {TA/RA, 7}                      | .  | T | . |
|     |                                                        | .  | . | T |

**Output Action:** Effective-SL-Evaluated-Event<sub>e-279</sub>

Figure 6.80: Guarding condition for transition from any Effective-SL state to Effective-SL state 7.

conditions for transitions in the state Effective-SL was complete. No augmenting information (other than the knowledge about enumerated type predicates that is incorporated into our decision procedures) was required to show the specification of the guarding conditions complete.

We applied PVS analysis to the PVS specification of the original guarding conditions. PVS analysis showed the guarding conditions complete in under four seconds on a 200MHz Pentium Pro machine running Redhat Linux. We used the same strategy as in the previous completeness analysis for transitions in the state Auto-SL (Figure 6.66).

### 6.3 Summary and Heuristics

From the results we obtained and discussed in this chapter and from additional case studies we performed, we developed some general heuristics to assist analysts in applying our method efficiently and effectively. First, for guarding conditions similar to those of the Auto-SL transitions that contain a large number of relational type predicates with a large number of possible multi-way interdependencies, such as  $(x < c1, x \geq c1)$ ,  $(x \leq c2, x > c2)$  where  $x$  is an integer or integer function and  $c1$  and  $c2$  are integer constants, and  $(x < c3, x \geq c4)$  and  $(x \geq c4, x \leq c5)$

where  $x$  is an integer or an integer function and  $c3$ ,  $c4$ , and  $c5$  are constants subject to the constraint:  $c3 < c4$ ,  $c5 < c4$  where the predicates cannot be both TRUE or both FALSE at the same time, PVS analysis should be applied on the first iteration; symbolic analysis (without some additional decision procedures) will not be effective, and there will be so much missing information in the analysis process that the indicator nodes method is not feasible; there are too many multi-way dependencies between the predicates.

Second, for guarding conditions similar to those of the *Effective-SL* transitions that contain a large number of enumerated type predicates, either symbolic analysis with our decision procedures, or PVS analysis can be applied first. However, since PVS has more decision procedures available, applying PVS on the first iteration may yield fewer spurious or redundant error reports.

Third, if the guarding conditions being analyzed have many rows and columns with many levels of indirection similar to the guarding conditions of the *Intruder-Status* transitions (for both TCAS II version 6.04A and version 7), it is best to apply symbolic analysis with variable reordering on the first iteration. The symbolic analysis report may show that some of the guarding conditions are consistent (complete). When the report from the symbolic analysis shows many inconsistencies (incompletenesses) then apply our domain axiom identification process to identify any missing domain axioms. Once the missing domain axioms have been identified, it is best to generate the PVS specifications of the guarding conditions and the conjecture theories to check for consistency and completeness, and add the identified domain axioms to the PVS specification. Then, apply the PVS prover to the conjectures, augmenting the proof process with the identified domain axioms. Since PVS has more decision procedures available, the output may report fewer spurious errors and fewer redundant errors when true errors exist. We do not start with PVS analysis for complex guarding conditions since, as our results show, there are some guarding conditions that PVS analysis fails on; this failure is generally because some information is lacking from the specification and the proof

process. Once the PVS specification and proof process are augmented with the domain axioms identified using our indicator nodes approach, PVS analysis is effective.

Fourth, for guarding conditions that contain a significant number of linear arithmetic predicates, non-linear arithmetic predicates with constants, or expressions involving division by constants when the expressions are structurally equivalent, it is best to apply PVS analysis to the guarding conditions on the first iteration; symbolic analysis using BDDs cannot effectively manage any of the aforementioned problems unless all of the predicates are structurally equivalent and there are no multi-way interdependencies between predicates (for example, three or more arithmetic predicates that cannot be satisfied at the same time).

Finally, the results we obtained for both our completeness analysis of transitions in the state Auto-SL, and for our consistency analysis for the guarding conditions on the transitions Proximate-Traffic to Other-Traffic and Proximate-Traffic to Potential-Threat, suggest that a large number of reported inconsistencies or incompletenesses result because of the large number of permutations that may exist between the predicates within a macro (for example, there may be six ways the truth values of three predicates in a one column table can be permuted such that the resulting macro is unsatisfied). Rather than reporting all permutations of truth values that make a particular macro FALSE or TRUE, we can combine all of the permutations into the single macro from whence the predicates came, and just report that the macro has to be FALSE or TRUE, accordingly. Thus, making the analysis output still smaller and more manageable. We can also investigate the selective expansion of certain macro predicates, for example. Our analysis tool does not have these capabilities yet, but these are ideas that bear further consideration.



## **Chapter 7**

# **Conclusions and Future Investigations**

Statically analyzing requirements specifications to assure that they possess desirable properties is an important activity in any rigorous software development project. However, static analysis is performed on a formal model of the requirements that is an abstraction of the original requirements specification. Often, abstractions in the model lead to spurious errors in the analysis output. A high ratio of spurious errors to true errors in the analysis output makes it difficult, error-prone, and time consuming to find and correct the true errors in the specification.

Two desirable properties that certain requirements documents should satisfy are completeness and consistency. The goal of the research described in this dissertation was to develop a method to analyze state-based requirements for completeness and consistency in a way that is fast enough and automated enough to be used on a day-to-day basis by practicing engineers, that generates analysis output with a small ratio of spurious errors to true errors, and that is scalable and generalizable. Analyzing state-based requirements for completeness and consistency generalizes to analyzing large logical expressions for tautologies and contradictions.

In Chapters 2 and 3 we showed that different analysis methods had different strengths and weaknesses. In Chapter 3 we showed that all analysis techniques suffer from a common problem, namely,

spurious errors in the analysis output. Two methods for analyzing logical expressions for tautologies and contradictions are symbolic methods such as those that rely on Binary Decision Diagrams (BDDs), and reasoning methods such as theorem proving. We showed that symbolic methods are fast and fully automated, but generate output that may contain many spurious errors since the analysis model contains many abstractions. Reasoning methods tend to be slower and require more manual intervention, but generate more accurate output since the analysis model contains fewer abstractions. We showed that both symbolic analysis and reasoning methods work well for certain problems, but that neither method alone can always provide the analyst with accurate analysis output in a timely and automated manner.

In Chapter 3 we identified four different classes of spurious errors and the undetected contradictions that lead to them. We demonstrated that different analysis methods can deal effectively with different classes of spurious errors; i.e., different analysis methods can detect and eliminate different types of contradictions. We showed that no analysis method is able to detect contradictions between predicates when information about environmental constraints or when information related to the structure of the state machine is missing from the analysis model. In Chapter 4 we discussed that if one can identify the information that has been abstracted away and that is causing the spurious errors, and add the information back into the analysis process, then the spurious errors can be eliminated. In Chapter 5 we described a technique to help identify the missing information. We call the augmenting information domain axioms.

Our domain axiom identification technique uses the structure of the symbolic representation (BDDs) to point out potential predicates that may be involved in domain axioms that have been abstracted out of the analysis model. We found that nodes that occur on all or a majority of paths near the top or bottom of the BDD are indicative of predicates involved in domain axioms in the specification. We call such nodes indicator nodes, and we call the predicates associated with them,

indicator predicates. We demonstrated how we can use the indicator predicates to identify domain axioms in the specification that have been abstracted out of the analysis model and that need to be added back into the analysis model to eliminate the spurious errors.

The results of the research are: (1) an iterative approach that integrates the strengths of a symbolic and a reasoning component to analyze logical expressions for tautologies and contradictions while circumventing their weaknesses, and (2) a simple technique that uses a symbolic representation of logical expressions to help identify abstractions in a model that are causing spurious errors in the analysis output.

We applied our method to some of the most complex portions of a large real-world avionics specification, the TCAS II requirements specification. During this application we learned some heuristics to help guide analysts in using our approach efficiently and effectively. We reported these heuristics at the conclusion of Chapter 6. The results we achieved demonstrate that our approach is feasible and promising.

## **7.1 Potential Future Work**

There are many sub-tasks of our method that can be further automated. For example, the process of identifying the indicator nodes (predicates) can easily be automated. Automating the processes of identifying the relevant domain axioms and verifying that the identified domain axioms are truly axioms in the specification is more difficult and requires further research. Another sub-task that may be automated and that requires further investigation is the process of inspecting the samples from the symbolic analysis process and identifying potential domain axioms by identifying patterns in the samples. The latest version of PVS has provisions for batch mode, so it is possible that the PVS analysis process can be further automated as well.

In addition, our method of applying the different analysis techniques based on the results of

particular iterations can easily be automated. Our analysis tool allows the analysts to select the desired analysis via command line options when the analysis is initiated. We could easily set up a shell script to interact with our analysis process to decide what action to take on subsequent iterations based on the output from the most recent iteration.

Analysis of input and output interfaces is also related to this research [29]. Constraints may be placed on when an input can be received and when an output can be generated. These constraints can be in the form of logical expressions. The analysis technique discussed in this dissertation easily extends to examining input and output interfaces for completeness and consistency.

## **7.2 Suggestions for Future Investigations**

One suggestion for future investigations is to determine if our method for identifying domain axioms can be integrated into other analysis methods that use BDDs, such as PVS [44, 55], model checkers [4, 9, 10, 42], and SVC [47].

Another opportunity for future investigation of the applicability of our method is to investigate other application domains that rely on manipulation of logical expressions to check for tautologies and contradictions. For example, areas that rely on decision tables, and the SCR requirements specification formalism.

Finally, a more difficult problem that requires further investigation is what to do once the errors have been determined; for example, when incompletenesses are discovered, in what guarding conditions should the missing conditions be added; or, should conditions be removed and the guarding conditions simplified. In other words, once the inconsistencies and incompletenesses have been found, how are they removed and corrected. These are difficult questions that need to be addressed in the near future.

## APPENDICES

## **Appendix A**

### **Macro Definitions**

Macro: Alt-Separation-Test

Definition:

|     |                                                                       |   |   |   |   |
|-----|-----------------------------------------------------------------------|---|---|---|---|
| AND | Other-Capability <sub>v-111</sub> = TA/RA                             | T | T | F | F |
|     | Two-Of-Three <sub>m-222</sub>                                         | T | T | . | . |
|     | No-Vertical-Intent <sub>m-218</sub>                                   | T | T | . | . |
|     | Modified-Tau-Capped <sub>f-241</sub> < FRTHR                          | T | T | T | T |
|     | ALT-RATE-&-SEPARATION-TEST                                            | T | T | T | T |
|     | Noncrossing-Biased-Climb <sub>m-217</sub>                             | T | . | T | . |
|     | Own-Tracked-Alt <sub>f-248</sub> < Other-Tracked-Alt <sub>f-243</sub> | T | . | T | . |
|     | Noncrossing-Biased-Descend <sub>m-218</sub>                           | . | T | . | T |
|     | Own-Tracked-Alt <sub>f-248</sub> > Other-Tracked-Alt <sub>f-243</sub> | . | T | . | T |

Abbreviations:

FRTHR

Time-To-CPA-Firmness-Dependent<sub>t-274</sub>[Conflict-SL<sub>f-231</sub>,Other-Track-Firmness<sub>f-243</sub>]

ALT-RATE-&-SEPARATION-TEST

|     |                                                                                |    |   |   |   |
|-----|--------------------------------------------------------------------------------|----|---|---|---|
|     |                                                                                | OR |   |   |   |
| AND | Own-Tracked-Alt-Rate <sub>f-247</sub>   ≤ 600 ft/min <sub>(OLEV)</sub>         | T  | • | • | F |
|     | Other-Tracked-Alt-Rate <sub>f-244</sub>   ≤ 600 ft/min <sub>(OLEV)</sub>       | •  | T | • | F |
|     | SAME-VERTICAL-DIRECTION                                                        | •  | • | T | F |
|     | Current-Vertical-Separation <sub>f-231</sub> > 600 ft <sub>(MAXALTDIFF)</sub>  | T  | T | T | • |
|     | Current-Vertical-Separation <sub>f-231</sub> > 850 ft <sub>(MAXALTDIFF2)</sub> | •  | • | • | T |

SAME-VERTICAL-DIRECTION

|     |                                             |     |
|-----|---------------------------------------------|-----|
| AND | OR                                          |     |
|     | Own-Tracked-Alt-Rate <sub>f-247</sub> ≥ 0   | T F |
|     | Other-Tracked-Alt-Rate <sub>f-244</sub> ≥ 0 | T F |

Figure A.1: Alt-Separation-Test Macro

Macro: Low-Firmness-Separation-Test

Definition:

|     |                                                                             |   |   |   |   |   |   |
|-----|-----------------------------------------------------------------------------|---|---|---|---|---|---|
| AND | Other-Capability <sub>v-111</sub> = TA/RA                                   | T | T | T | F | F | F |
|     | No-Vertical-Intent <sub>m-218</sub>                                         | T | T | T | . | . | . |
|     | Two-Of-Three <sub>m-222</sub>                                               | T | T | T | . | . | . |
|     | Modified-Tau-Capped <sub>f-241</sub> < TFRTHR                               | F | F | F | F | F | F |
|     | Down-Separation <sub>f-236</sub> (low-firm) ≤ SENSEFIRM                     | T | . | . | T | . | . |
|     | Up-Separation <sub>f-261</sub> (low-firm) ≤ SENSEFIRM                       | T | . | . | T | . | . |
|     | CHANGE-ME                                                                   | . | T | F | . | T | F |
|     | Own-Tracked-Alt <sub>f-248</sub> < Other-Tracked-Alt <sub>f-243</sub>       | . | T | . | . | T | . |
|     | Own-Tracked-Alt <sub>f-248</sub> > Other-Tracked-Alt <sub>f-243</sub>       | . | . | T | . | . | T |
|     | Current-Vertical-Separation <sub>f-231</sub> > 150 ft <sub>(LOWFIRMZ)</sub> | . | T | T | . | T | T |

OR

Abbreviations:

CHANGE-ME

Inhibit-Biased-Climb<sub>f-238</sub>(low-firm) > Down-Separation<sub>f-236</sub>(low-firm)

SENSEFIRM

Low-Firmness-Alt-Threshold<sub>t-275</sub>[Alt-Layer-Value<sub>f-229</sub>]

TFRTHR

Time-To-CPA-Firmness-Dependent<sub>t-274</sub>[Conflict-SL<sub>f-231</sub>,Other-Track-Firmness<sub>f-243</sub>]

Figure A.2: Low-Firmness-Separation-Test Macro

Macro: Noncrossing-Biased-Climb

Definition:

|     |                                                                                                      |   |   |   |
|-----|------------------------------------------------------------------------------------------------------|---|---|---|
| AND | Inhibit-Biased-Climb <sub>f-238</sub> (normal) > Down-Separation <sub>f-236</sub> (normal)           | F | T | T |
|     | Own-Tracked-Alt <sub>f-248</sub> − Other-Tracked-Alt <sub>f-243</sub> ≥ 300 ft <sub>(MINSEP)</sub>   | T | . | . |
|     | Up-Separation <sub>f-261</sub> (normal) ≥ ALIM                                                       | T | . | . |
|     | Own-Tracked-Alt <sub>f-248</sub> − Other-Tracked-Alt <sub>f-243</sub> ≤ − 300 ft <sub>(MINSEP)</sub> | . | . | F |
|     | Down-Separation <sub>f-236</sub> (normal) ≥ ALIM                                                     | . | F | . |

OR

Abbreviations:

ALIM

Positive-RA-Altitude-Limit-Threshold<sub>t-274</sub>[Alt-Layer-Value<sub>f-229</sub>]

Figure A.3: Noncrossing-Biased-Climb Macro



Macro: Noncrossing-Biased-Descend

Definition:

|     |                                                                                                      |   |   |   |
|-----|------------------------------------------------------------------------------------------------------|---|---|---|
| AND | Inhibit-Biased-Climb <sub>f-238</sub> (normal) > Down-Separation <sub>f-236</sub> (normal)           | F | T | F |
|     | Own-Tracked-Alt <sub>f-248</sub> – Other-Tracked-Alt <sub>f-243</sub> ≥ 300 ft <sub>(MINSEP)</sub>   | • | • | F |
|     | Up-Separation <sub>f-261</sub> (normal) ≥ ALIM                                                       | F | • | • |
|     | Own-Tracked-Alt <sub>f-248</sub> – Other-Tracked-Alt <sub>f-243</sub> ≤ – 300 ft <sub>(MINSEP)</sub> | • | T | • |
|     | Down-Separation <sub>f-236</sub> (normal) ≥ ALIM                                                     | • | T | • |

OR

Abbreviations:

ALIM

|                                                                                           |
|-------------------------------------------------------------------------------------------|
| Positive-RA-Altitude-Limit-Threshold <sub>t-274</sub> [Alt-Layer-Value <sub>f-229</sub> ] |
|-------------------------------------------------------------------------------------------|

Figure A.4: Noncrossing-Biased-Descend Macro

Macro: No-Vertical-Intent

Definition:

|     |                                              |   |   |   |
|-----|----------------------------------------------|---|---|---|
| AND | Other-VRC <sub>v-108</sub> = No-Intent       | T | • | • |
|     | Intent-Received <sub>s-118</sub> in state No | • | T | • |
|     | Other-Capability <sub>v-111</sub> = TA/RA    | • | • | F |

OR

Figure A.5: No-Vertical-Intent Macro

Macro: RA-Mode-Canceled

Definition:

|     |                                            |   |
|-----|--------------------------------------------|---|
| AND | RA-Inhibit <sub>m-219</sub> = True         | T |
|     | PREV(RA-Inhibit <sub>m-219</sub> ) = False | T |

Figure A.6: RA-Mode-Canceled Macro

**Macro: RA-Inhibit**

**Definition:**

|     |                                         |   |   |   |
|-----|-----------------------------------------|---|---|---|
| AND | Auto-SL <sub>s-30</sub> in state 2      | T | . | . |
|     | Mode-Selector <sub>v-34</sub> = TA-Only | . | T | . |
|     | RA-Inhibit-From-Ground                  | . | . | T |

**Abbreviations:**

**RA-Inhibit-From-Ground**

|                                                                                                     |
|-----------------------------------------------------------------------------------------------------|
| there exists <i>i</i> : Mode-S-Ground-Station[ <i>i</i> ].Ground-Commanded-SL <sub>v-193</sub> = 2  |
| <b>Description:</b> There exists at least one ground station that has commanded sensitivity level 2 |

Figure A.7: RA-Inhibit Macro

**Macro: Reply-Invalid-Test**

**Definition:**

|     |                                           |   |
|-----|-------------------------------------------|---|
| AND | Other-Capability <sub>v-111</sub> = TA/RA | T |
|     | No-Vertical-Intent <sub>m-218</sub>       | T |
|     | Two-Of-Three <sub>m-222</sub>             | F |

Figure A.8: Reply-Invalid-Test Macro

**Macro: TCAS-TCAS-Crossing-Test**  
**Definition:**

|     |                                                                           |    |   |
|-----|---------------------------------------------------------------------------|----|---|
| AND | Other-Capability <sub>v-111</sub> = TA/RA                                 | OR |   |
|     | No-Vertical-Intent <sub>m-218</sub>                                       | T  | T |
|     | Two-Of-Three <sub>m-222</sub>                                             | T  | T |
|     | Modified-Tau-Capped <sub>f-241</sub> < TFRTHR                             | T  | T |
|     | Own-Tracked-Alt-Rate <sub>f-247</sub>   ≤ 600 ft/min <sub>(OLEV)</sub>    | T  | T |
|     | Other-Tracked-Alt-Rate <sub>f-244</sub>   > 600 ft/min <sub>(OLEV)</sub>  | T  | T |
|     | Other-Tracked-Relative-Alt <sub>f-246</sub> > 300 ft <sub>(MINSEP)</sub>  | T  | . |
|     | VMD ≤ 0                                                                   | T  | . |
|     | Other-Tracked-Relative-Alt <sub>f-246</sub> < −300 ft <sub>(MINSEP)</sub> | .  | T |
|     | VMD ≥ 0                                                                   | .  | T |

**Abbreviations:**

- TFRTHR**  
Time-To-CPA-Firmness-Dependent<sub>t-274</sub>[Conflict-SL<sub>f-231</sub>,Other-Track-Firmness<sub>f-243</sub>]
- VMD**  
Not defined in this document for this macro.

Figure A.9: TCAS-TCAS-Crossing-Test Macro

Macro: Threat-Alt-Test

Definition:

|     |                                                                    |   |   |   |   |
|-----|--------------------------------------------------------------------|---|---|---|---|
| AND | Other-Alt-Reporting <sub>v-113</sub> = True                        | T | T | T | T |
|     | Current-Vertical-Separation <sub>f-231</sub> < ZT                  | T | F | T | F |
|     | Other-Tracked-Range-Rate <sub>f-245</sub> > 0                      | F | . | T | F |
|     | VMD  < ZT                                                          | T | . | . | T |
|     | ADOT ≥ −1ft/s( <i>ZDTHR</i> )                                      | . | F | . | F |
|     | Time-To-Co-Alt <sub>f-258</sub> < TVTHR                            | . | T | . | T |
|     | Time-To-Co-Alt <sub>f-258</sub> < True-Tau-Capped <sub>f-261</sub> | . | T | . | . |

OR

Abbreviations:

ADOT

Other-Tracked-Relative-Alt-Rate<sub>f-246</sub> \* SIGN(Other-Tracked-Relative-Alt<sub>f-246</sub>)

VMD

Let:  
RZ = Own-Tracked-Alt<sub>f-248</sub> − Other-Tracked-Alt<sub>f-243</sub>  
RZD = Own-Tracked-Alt-Rate<sub>f-247</sub> − Other-Tracked-Alt-Rate<sub>f-244</sub>  
TRTRU = True-Tau-Capped<sub>f-261</sub>  
TAUR = Modified-Tau-Capped<sub>f-241</sub>  
TVPCMD = XTVPCTBLX<sub>f-272</sub>[Conflict-SL<sub>f-231</sub>]  
Then: VMD = Vertical-Miss-Distance<sub>f-262</sub>(RZ, RZD, TRTRU, TAUR, TVPCMD)

ZT

Threat-Alt-Threshold<sub>f-275</sub> [Alt-Layer-Value<sub>f-229</sub>]

TVTHR

Not defined in this document for this macro.

Figure A.10: Threat-Alt-Test Macro

Macro: Threat-Range-Test

|             |                                                                                         |    |   |
|-------------|-----------------------------------------------------------------------------------------|----|---|
| Definition: |                                                                                         | OR |   |
| AND         | Other-Tracked-Range-Rate <sub>f-245</sub> > 10 ft/s <sub>(RDTHR)</sub>                  | .  | F |
|             | Other-Tracked-Range <sub>f-245</sub> > DMOD                                             | F  | . |
|             | Modified-Tau-Capped <sub>f-241</sub> < TRTHR                                            | .  | T |
|             | Other-Tracked-Range <sub>f-245</sub> ≤ 12.0 nmi <sub>(RMAX)</sub>                       | .  | T |
|             | Other-Tracked-Range-Rate <sub>f-245</sub> * Other-Tracked-Range <sub>f-245</sub>   > H1 | F  | . |
|             | NUISANCE-ALARM-FILTER                                                                   | F  | F |

Abbreviations:

DMOD

Threat-Minimum-Range-Threshold<sub>t-273</sub> [Conflict-SL<sub>f-231</sub>]

TRTHR

Threat-Modified-Tau-Threshold<sub>t-274</sub> [Conflict-SL<sub>f-231</sub>]

H1

Threat-Minimum-Divergence-Threshold<sub>t-273</sub> [Conflict-SL<sub>f-231</sub>]

NUISANCE-ALARM-FILTER

|     |                                                  |   |
|-----|--------------------------------------------------|---|
| AND | Tau-Rising <sub>s-148</sub> in state 3Plus       | T |
|     | Intruder-Status <sub>s-136</sub> in state Threat | F |
|     | Other-Tracked-Range <sub>f-245</sub> ≥ DMOD      | T |

Figure A.11: Threat-Range-Test Macro

Macro: Two-Of-Three

Definition:

there exists  $j \in \{1, 2\}$  :

|     |                                                                |   |
|-----|----------------------------------------------------------------|---|
| AND | PREV <sub>j</sub> (Other-Range-Valid <sub>v-117</sub> ) = True | T |
|     | Other-Range-Valid <sub>v-117</sub> = True                      | T |

Figure A.12: Two-Of-Three Macro

## Appendix B

# PVS Commands and Strategies

In Section B.1, we describe the commands and strategies available in PVS that we used to build our own PVS strategies. Section B.2 describes how to define proof strategies in PVS. In Section B.3, we describe the proof strategies we defined to assist the analyst with proving disjunctive and conjunctive logical expressions satisfiable and mutually exclusive.

For additional details about the proof checker and the proof commands, see [55]. For details on the PVS specification language, see [44]. For information on how to use the PVS specification and verification system see [15, 45].

### B.1 Built-in PVS Commands and Strategies

The description of the PVS commands and strategies listed in this section, comes from the online PVS help facility and from [55].

- **(apply *strategy*):** Applies a proof strategy, *strategy*, in a single atomic step.
- **assert:** Simplifies the current goal using the decision procedures.
- **auto-rewrite-defs\$:** Installs all of the definitions used directly or indirectly in the original statement (i.e., relevant to the conjecture) as auto-rewrite rules.
- **bddsimp:** Performs propositional simplification using Binary Decision Diagrams (BDDs).

- **do-rewrite\$**: Applies the rewrite rules to all of the formulas in the sequent.
- **inst?**: Tries to automatically instantiate a quantifier.
- **(lemma name)**: Introduces an instance of the lemma named *name*. *name* may be an axiom, lemma, or definition.
- **postpone**: Marks the current goal as pending to be proved, and changes focus to the next remaining goal.
- **record**: Adds more assumptions to the data structures used by the decision procedures. The data structures are used to record the assumptions that are true in the current context.
- **(repeat\* step)**: A strategy that successively applies *step* until *step* does nothing.
- **rewrite-msg-off**: Turns off printing of applied auto rewrites and skips.
- **skip**: Does nothing. Primarily used in writing strategies where a step is required to have no effect unless some condition holds.
- **skolem!**: Skolemizes by automatically generating skolem constants.
- **(then step1 step2)**: A sequencing strategy that first applies *step1*, and if any subgoals are generated, then applies *step2* to each of the generated subgoals. If *step1* has no effect, then *step2* is applied to the original goal.
- **(try step1 step2 step3)**: A strategy for subgoaling and backtracking that applies *step1* to the current goal, and if *step1* succeeds and generates subgoals, then it applies *step2* to the generated subgoals. If *step1* does nothing, then *step3* is applied to the current goal.

## B.2 Defining Strategies

The user-defined strategies are saved in a file called **pvs-strategies** [55]. When the PVS prover is invoked, PVS loads the strategies from a file of the above name. A user-defined strategy definition has the form:

```
(defstep name
 (required-parameters
 &optional optional-parameters
 &rest parameter
 strategy-expression
 documentation-string
 format-string)
```

where **&rest** is a keyword indicating that zero or more values may be provided for the indicated argument. Our user-defined strategies (described in Section B.3) do not use parameters.

The strategy definition generates both a defined rule *name* and a strategy *name*\$ ( [55]). The main difference between rules and strategies is that rules are atomic. When a rule is applied to a goal, it reduces the goal to zero or more subgoals in a single step, whereas a strategy generates a tree of rule-applications while reducing a goal to subgoals.

### B.3 Our PVS Defined Strategies For Checking For Consistency and Completeness

We developed the following strategy *consistent2* to assist the analyst in checking conjunctive logical expressions for mutual exclusion. The strategy is invoked using the command *consistent2\$*.

```
(defstep consistent2
 NIL
 (apply (then (rewrite-msg-off)
 (skolem!)
 (auto-rewrite-defs$)
 (do-rewrite$)
 (repeat* (try (bddsimp)
 (record)
 (postpone)))))
 " "
 " ")
```

We developed the following strategy *complete2* to assist the analyst in checking disjunctive logical expressions for satisfiability. The strategy is invoked using the command *complete2\$*.

```
(defstep complete2
 NIL
 (apply (then (skolem!)
 (rewrite-msg-off)
 (auto-rewrite-defs$)
 (do-rewrite$)
 (repeat* (try (bddsimp)
 (try (record) (assert) (postpone))
 (skip))))))
 ""
 ""
)
```



The *skolemizeandrewrite* strategy is used to perform only skolemization and rewriting. It is used when the proof process requires the addition of lemmas, since a single automated strategy cannot be used. When lemmas need to be added to the proof process, each lemma must be specified by name and individually included. The strategy is invoked using *skolemizeandrewrite\$*.

```
(defstep skolemizeandrewrite
 NIL
 (apply (then (rewrite-msg-off)
 (skolem!)
 (auto-rewrite-defs$)
 (do-rewrite$)))
 " "
 " "
)
```

# Bibliography

- [1] R.J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. In D. Garlan, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on the foundations of Software Engineering (SIGSOFT'96)*, volume 21, pages 156–166, November 1996.
- [2] Joanne M. Atlee. Native model-checking of SCR requirements. Obtained via ftp; Dept. of Computer Science; University of Waterloo; Waterloo, Ontario.
- [3] Joanne M. Atlee. *Automated Analysis of Software Requirements*. PhD thesis, University of Maryland, 1992.
- [4] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, pages 24–40, January 1993.
- [5] BDD(3). Manual page, June 1993. Manual page for BDD package obtained via ftp from Carnegie-Mellon University.
- [6] Ramesh Bharadwaj and Constance Heitmeyer. Verifying SCR requirements specifications using state exploration. In *First ACM SIGPLAN Workshop on Automatic Analysis of Software*, Jan 1997.
- [7] Michael C. Browne et al. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1043, December 1986.
- [8] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [9] Jerry R. Burch et al. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.
- [10] J.R. Burch, E.M. Clarke, K. L. McMillan, et al. Symbolic model checking:  $10^{20}$  states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
- [11] William Chan, Richard Anderson, Paul Beame, and David Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In Orna Grumberg, editor, *Computer Aided Verification; 9th International Conference, CAV '97 Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327, June 1997.

- [12] William Chan, Richard Anderson, Paul Beame, and David Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 102–112, March 1998.
- [13] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
- [14] E. M. Clarke et al. Using temporal logic for automatic verification of finite state systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 3–26. Springer-Verlag, Berlin, 1985.
- [15] Judy Crow, Sam Owre, John Rushby, et al. A tutorial introduction to PVS. Presented at WIFT 95: Workshop on Industrial-Strength Formal Specification Techniques, 1995. Available at <http://www.csl.sri.com/fm-papers.html>.
- [16] David Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons Ltd., 1991.
- [17] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, January 1986.
- [18] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.
- [19] Stephen J. Garland and John V. Guttag. A guide to LP, the larch prover, December 1991.
- [20] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [21] David Harel et al. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, pages 403–413, April 1990.
- [22] V. Hartonas-Garmhausen et al. Automatic verification of industrial designs. In *Workshop on Industrial Strength Formal Specification Techniques*, pages 88–96, April 1995.
- [23] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe (FME '96)*, pages 662–681, 1996.
- [24] M. P.E. Heimdahl and N.G. Leveson. Completeness and Consistency Analysis of State-Based Requirements. *IEEE Transactions on Software Engineering*, TSE-22(6):363–377, June 1996.
- [25] Mats P.E. Heimdahl. Analysis of state-based models: A survey. Technical Report CPS-94-59, Michigan State University, Department of Computer Science; 3115 Engineering Building; E. Lansing, MI 48824-1027, November 1994.
- [26] Mats P.E. Heimdahl. Completeness and consistency in hierarchical state-based requirements. Technical Report CPS-94-69, Michigan State University, Department of Computer Science; 3115 Engineering Building; E. Lansing, MI 48824-1027, December 1994.
- [27] Mats P.E. Heimdahl. Experiences and lessons from the analysis of TCAS II. Technical Report CPS-95-25, Michigan State University, Department of Computer Science; 3115 Engineering Building; E. Lansing, MI 48824-1027, June 1995.

- [28] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and consistency analysis of state-based requirements. Technical Report CPS-94-53, Michigan State University, Department of Computer Science; 3115 Engineering Building; E. Lansing, MI 48824-1027, October 1994.
- [29] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of system level inter-component communication. *IEEE Computer*, pages 47–54, April 1998.
- [30] Constance Heitmeyer et al. Consistency checking of SCR-style requirements specifications. In *International Symposium on Requirements Engineering*, March 1995.
- [31] Constance Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
- [32] Constance L. Heitmeyer and Bruce G. Labaw. Consistency checks for SCR-style requirements specifications. Technical Report NRL/FR/5540-93-9586, Naval Research Laboratory, Washington, DC 20375-5320, December 1993.
- [33] Kathryn L. Heninger, J. W. Kallander, J. E. Shore, and D. L. Parnas. Software requirements for the A-7e aircraft. Technical Report 3876, Naval Research Laboratory, 1978.
- [34] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [35] Alan J. Hu et al. Higher-level specification and verification with BDDs. In *Participants' Proceedings of the fourth workshop on computer-aided verification*, pages 84–96, 1992.
- [36] *Introduction to the HOL Theorem Proving System*. Available at <http://lal.cs.byu.edu/lal/holdoc/Description/Description.html>.
- [37] D. Jackson. Exploiting symmetry in the model checking of relational specifications. Technical report, Carnegie-Mellon University, Pittsburgh, PA, 1994.
- [38] Matthew S. Jaffe et al. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, pages 241–257, March 1991.
- [39] Jeffrey J. Joyce and Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. Technical Report TR-93-18, University of British Columbia, Department of Computer Science; Vancouver, B.C. V6T 1Z2 Canada, 1993.
- [40] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. Reese. TCAS II requirements specification.
- [41] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
- [42] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [43] NASA; Office of Safety and Mission Assurance. Formal methods specification and verification guidebook for software and computer systems; draft 3.0.

- [44] S. Owre, N. Shankar, and J.M.Rushby. *The PVS Specification Language*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, April 1993. Available at <http://www.csl.sri.com/pvs.html>.
- [45] S. Owre, N. Shankar, and J.M.Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993. Available at <http://www.csl.sri.com/pvs.html>.
- [46] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.
- [47] D. Y. W. Park, J. U. Skakkebaek, M. P. E. Heimdahl, B. J. Czerny, and D. L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *FMSP '98: Second Workshop on Formal Methods in Software Practice*, pages 34–43, March 1998.
- [48] Mauro Pezze, Richard N. Taylor, and Michal Young. Graph models for reachability analysis of concurrent programs, October 1994. Available at <http://www.cs.purdue.edu/homes/young/papers/papers.html>.
- [49] Roger S. Pressman. *Software Engineering A Practitioners Approach*. McGraw-Hill, Inc., fourth edition, 1997.
- [50] Jean B. Rogers. *A Prolog Primer*. Addison Wesley, 1987.
- [51] J. M. Rushby, 1995. personal communication.
- [52] John Rushby. Formal specification and verification for critical systems: Tools, achievements, and prospects. In *Electric Power Research Institute (EPRI) Workshop on Methodologies for Cost-Effective, Reliable Software Verification and Validation*, pages 9–1 to 9–14, January 1992.
- [53] John Rushby. Model checking and other ways of automating formal methods. Position paper for panel on Model Checking for Concurrent Programs; Software Quality Week, May/June 1995.
- [54] Carl-Johan Seger. An introduction to formal hardware verification. Technical Report TR-92-13, University of British Columbia, Department of Computer Science; Vancouver, B.C., Canada V6T 1Z2, 1992.
- [55] N. Shankar, S. Owre, and J.M.Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993. Available at <http://www.csl.sri.com/pvs.html>.
- [56] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company Inc., fourth edition, 1992.
- [57] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements. In *11th Annual Conference on Computer Assurance COMPASS*, pages 77–88, June 1996.
- [58] *Tutorial for the HOL Theorem Proving System*. Available at <http://lsl.cs.byu.edu/lsl/holdoc/tutorial.html>.

- [59] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, pages 8–24, Sept 1990.
- [60] Jeannette M. Wing and Mandana Vaziri-Farahani. Model checking software systems: A case study. Technical Report CMU-CS-95-128, Carnegie Mellon University, March 1995.
- [61] Larry Wos et al. *Automated Reasoning Introduction and Applications*. McGraw-Hill, Inc., second edition, 1992.
- [62] Wei Jen Yeh. *Controlling State Explosion in Reachability Analysis*. Doctoral thesis, Purdue University, December 1993.
- [63] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Symposium on Software Testing, Analysis, and Verification TAV4*, pages 49–59. ACM SIG-SOFT, ACM Press, October 1991.
- [64] Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.
- [65] Michal Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques, September 1991. Available at <http://www.cs.purdue.edu/homes/young/papers/papers.html>.
- [66] Michal Young, Richard N. Taylor, Kari Forester, and Debra Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the 3rd International Workshop on Testing, Analysis, and Verification*, 1989.
- [67] Michal Young, Richard N. Taylor, David L. Levine, Kari Forester, and Debra Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. Technical Report TR-128-P, Software Engineering Research Center, November 1994.

MICHIGAN STATE UNIV. LIBRARIES



31293017128830