Integrating Formal Methods with Model-Driven Engineering

by

Opeyemi Adesina

Thesis submitted
in partial fulfilment of the requirements for the award of

Doctor of Philosophy in Computer Science

Ottawa-Carleton Institute of Computer Science
School of Electrical Engineering and Computer Science

uOttawa

University of Ottawa
Ottawa, Canada

Thesis Advisor: Timothy C. Lethbridge
Co-Advisor: Stéphane S. Somé

© Opeyemi Adesina, Ottawa, Canada, 2017

Abstract

This thesis presents our method to integrate formal methods with model-driven engineering. Although a large amount of literature exists with the goal of facilitating the adoption of formal methods for educational and industrial practice, yet the adoption of formal methods in academia and industry is poor. The goal of this research is to improve the adoption of formal methods by automating the generation of formal methods code while maintaining scalability and bridging the gaps between formal analysis and actual implementation of the complete system.

Our approach is based on generating formal representations of software abstractions expressed in a textual language, called Umple, which is derived from UML. Software abstractions of interest include class models and state machines. For state machines, we address concerns such as composite and concurrent states separately. The resulting systems are analyzable by back-end analysis engines such as Alloy and nuXmv or NuSMV for model checking.

To ensure correctness of our approach, we have adopted simulation, empirical studies and rigorous test-driven development (TDD) methodologies. To guarantee correctness of state machine systems under analysis (SSUAs), we present methods to automatically generate specifications to analyze domain-independent properties such as non-determinism and reachability analysis. We apply these methods in various case studies; certify their conformance with sets of requirements and uncover certain flaws.

Our contributions include a) The overall approach, involving having the developer write the system in Umple and generating both the formal system for analysis and the final code from the same model; b) a novel approach to encode SSUAs even in the presence of and-cross transitions; c) a fully automated approach to certify an SSUA to be free from nondeterminism even in the presence of unbounded domains and multiple and-cross transitions within the same enclosing orthogonal state; d) an empirical study of the impact of abstraction on some performance parameters; and e) a translator from Umple to Alloy and SMV.

Acknowledgements

First and foremost, I would like to express my gratitude to God for His grace, faithfulness, wealth as well as both physical and mental health since birth until now; and most importantly in the course of my doctoral studies. May His name alone be glorified.

I would like to thank my supervisor, Professor Timothy C. Lethbridge for his fatherly and professional support during my PhD studies. Prior to my arrival from Nigeria, Tim was able to guide me remotely and upon my arrival until now he was unrelenting in supporting me physically and mentally. I appreciate your words of encouragement, financial support and absolute dedication to making your students grow and succeed.

My appreciation also goes to Professor Stéphane S. Somé for his co-supervisory role in the course of my PhD studies. I am thankful to Stéphane for his professional support prior to my arrival and during my studies. Your approach to making me understand and apply complex formal methods in solving real-life problems is commendable.

I would like to acknowledge comments and feedback of each member of my examination committee. These people include Professors Nancy Day, Douglas Howe, Amy Felty and Guy-Vincent Jourdan. Your observations are very constructive and of great importance to improving this thesis.

I am also grateful for the support I received from members of the CRUiSE group. Thank you, guys, for accepting me, including the smooth integration and criticizing of my work. In particular, I am grateful to Dr. Miguel Garzon, Dr. Hamoud Aljamaan, Dr. Vadhat Abdelzad and Sultan Almaghthawi for taking me through the processes of registration and for their invaluable contributions to my thesis.

I am also thankful to all the reviewers of my work at various capacities ranging from my thesis committee to anonymous reviewers for their constructive criticism and feedback. Although, the journey was rough, your feedback eased the tension.

I am also indebted to the Nigerian community in Ottawa for their effort in making sure I feel at home even though I am some miles away from home. Similarly, I would like to thank members of my Church for their love, support and encouragement. May God, bless you abundantly.

I would also like to thank every member of my family for their support and encouragement before and during my PhD program. Particularly, I would like to thank my brothers Babatunde, Olukayode, Ayodeji, Oluwatomiwa and Michael and my father Festus. I am thankful to God for having you in my life.

Table of Contents

ABSTRACT II
ACKNOWLEDGEMENTS III
TABLE OF CONTENTS V
LIST OF FIGURES X
LIST OF TABLES XIII
TABLE OF LISTINGS XIV
LIST OF DEFINITIONS XVI
LIST OF RESEARCH QUESTIONS XVII
LIST OF PROPOSITIONS XVIII
1 INTRODUCTION 1
1.1 MODEL-DRIVEN ENGINEERING 2
1.2 FORMAL METHODS 2
1.2.1 Theorem proving 3
1.2.2 Static analysis 3
1.2.3 Model checking 4
1.2.4 Limitations on adoption 4
1.3 PROBLEM STATEMENT AND RESEARCH DIRECTION 5
1.4 CHOICE OF TOOLS 10
1.5 CONTRIBUTIONS 12
1.6 LIMITATIONS OF THE WORK 13
1.7 THESIS OUTLINE 13
2 BACKGROUND 15
2.1 UMPLE 16
2.1.1 Umple Class Models 18
2.1.2 Associations 18
2.1.3 Other Association-Related Constructs 21
2.1.4 Umple Attributes 21
2.1.5 Keys 23

2.1.6 Specialization and Generalization ... *23*
2.1.7 Constraints in Umple ... *23*
2.1.8 Umple State Machines ... *24*
2.1.9 Transitions ... *28*
2.1.10 States ... *29*
2.1.11 Transformed Internal View of Umple State Machines ... *30*
2.1.12 Umple Template Language (UmpleTL) ... *31*
2.1.13 Templates ... *32*
2.1.14 Emit Method Specification ... *32*
2.1.15 Expression Block ... *32*
2.1.16 Code Blocks ... *33*
2.1.17 Comment Block ... *33*
2.2 ALLOY ... 33
2.2.1 Signatures ... *33*
2.2.2 Constraints ... *37*
2.2.3 Commands ... *39*
2.3 NUXMV ... 41
2.3.1 The input language of nuXmv - SMV ... *41*
2.4 BACK-END ANALYSIS TOOLS ... 49
2.5 MODEL TRANSFORMATIONS ... 51
2.5.1 Source and Target Artifacts (Q1) ... *52*
2.5.2 Characteristics of Model Transformations (Q2) ... *53*
2.5.3 Mechanisms for Model Transformation (Q5) ... *54*
2.6 SUMMARY OF BACKGROUND ... 54
3 TRANSFORMATION ENGINEERING ... 56
3.1 OVERVIEW OF THE TRANSFORMATION ARCHITECTURE ... 57
3.2 ALLOY TRANSLATOR ... 59
3.3 SMV TRANSLATOR ... 67
3.4 SUMMARY ... 72
4 TRANSFORMATION OF CLASS MODELS ... 73
4.1 ATTRIBUTES ... 74
4.2 MULTIPLICITY ... 74
4.3 TRANSFORMATION PHASES (UMPLE TO ALLOY) ... 74
4.4 OBJECT-ORIENTED DESIGN PATTERNS ... 85

4.4.1 Abstract class pattern (P1) *85*
4.4.2 Singleton class pattern (P2) *85*
4.4.3 Class hierarchy pattern (P3) *86*
4.4.4 Bidirectional association pattern (P4) *86*
4.4.5 Unidirectional association pattern (P5) *87*
4.4.6 General hierarchy pattern (P6) *88*
4.4.7 Asymmetric-reflexive association pattern (P7) *89*
4.4.8 Symmetric-reflexive association pattern (P8) *90*
4.5 SUMMARY OF CLASS MODEL SPECIFICATION 91
5 FORMALIZATION OF STATE MACHINES 93
5.1 FORMAL SEMANTICS OF UMPLE'S STATE MACHINES 95
5.1.1 Relationships Between State Machines *99*
5.1.2 Relationships Between States *101*
5.2 TRANSITIONS IN UMPLE 105
5.2.1 Base Transitions *106*
5.2.2 High-Level Transitions *107*
5.2.3 And-Cross Transitions *108*
5.3 MANAGING STATES OF UMPLE STATE MACHINES 110
5.3.1 Execution Semantics of Umple *111*
5.3.2 Enabling Simple States *113*
5.3.3 Enabling Composite States *113*
5.3.4 Default Activation *116*
5.4 FORMULATING THE DISABLING TRANSITIONS SET FOR SUB-STATE MACHINES 122
5.4.1 Enabling Transitions of a State Machine *123*
5.4.2 Disabling Non-Parallel Sub-State Machine *123*
5.4.3 Disabling Parallel Sub-State Machine (or Region) *128*
5.4.4 Generalizing Disabling Transitions for Sub-State Machines *139*
5.5 SUMMARY OF STATE MACHINES FORMALIZATION 141
6 QUALITY ASSURANCE OF UMPLE'S SSUAS 143
6.1 DISCOVERING NON-DETERMINISM 143
6.1.1 Same-Source Transitions *144*
6.1.2 Region-Cross Transitions *145*
6.1.3 Cases of Nondeterminism *150*
6.1.4 False Cases of Nondeterminism *151*

6.2 THE MATCH-MAKING ALGORITHM 153
6.2.1 Specifying Invariance for Potentially Conflicting Pairs 158
6.2.2 2-Bit Counter Case Study 158
6.2.3 Applying the Match-Maker Algorithm on 2-Bit Counter Machine 164
6.3 REACHABILITY ANALYSIS OF STATES OF AN SSUA 169
6.3.1 Limitations of this approach 170
6.4 IMPROVING QUALITATIVE ANALYSIS VIA AND-CROSS TRANSITION 171
6.4.1 Motivation for And-Cross Transition 171
6.4.2 Modeling Solutions to the 2-Bit Counter Problem (see Section 6.2.2) 172
6.5 SUMMARY OF QUALITY ASSURANCE 178
7 TRANSFORMATION OF STATE MACHINE MODELS 180
7.1 A CASE STUDY 180
7.2 FORMAL OVERVIEW OF SMV 182
7.3 TRANSFORMATION OF ATTRIBUTES 183
7.4 TRANSFORMATION OF EVENTS 184
7.5 TRANSFORMATION OF TRANSITIONS 185
7.6 TRANSFORMATION OF STATES 188
7.7 TRANSFORMATION OF STATE MACHINES 189
7.7.1 Simple State Machines 189
7.7.2 Hierarchical State Machines 192
7.8 DISCOVERING NONDETERMINISM 196
7.9 REACHABILITY ANALYSIS OF THE SSUA 197
7.10 SUMMARY OF STATE MACHINE TRANSFORMATION 197
8 VERIFICATION AND VALIDATION 199
8.1 VERIFICATION 199
8.1.1 Simulation 199
8.1.2 Test-Driven Development 202
8.2 VALIDATION 204
8.2.1 Case Study 1 – Discovering Nondeterminism 204
8.2.2 Case Study 2 – Abstraction via And-Cross Transitions 210
8.2.3 Case Study 3 – Asserting Observational Equivalence of Similar SSUAs 226
8.2.4 Case Study 4 - Electronic Seating System 235
8.3 CONCLUSION 244
9 RELATED WORK 246

9.1 LITERATURE ON ANALYSIS TOOLS 246
9.1.1 Static Analysis Tools *248*
9.1.2 Theorem Proving Approaches *249*
9.1.3 Model Checking Approaches *252*
9.1.4 Summary on Analysis Tools *257*
9.2 LITERATURE ON AND-CROSS TRANSITIONS 259
9.2.1 Support for And-Cross Transitions *259*
9.2.2 On the rejection of And-Cross Transitions *261*
9.2.3 Managing complexities of state diagrams *261*
9.3 LITERATURE ON DISCOVERING NONDETERMINISM 262
9.3.1 Algorithmic Solutions *262*
9.3.2 Solutions adopting SPIN Nondeterministic Selector *263*
9.3.3 Tool-Based Solutions *263*
9.4 SUMMARY ON AND-CROSS AND NONDETERMINISM 264
10 CONCLUSION AND FUTURE WORK 267
10.1 SUMMARY WITH RESPECT TO RESEARCH QUESTION 267
10.2 SUMMARY ON SCIENTIFIC CONTRIBUTIONS 269
10.3 FUTURE DIRECTIONS 272
REFERENCES 275
APPENDIX 287
APPENDIX 1 – HOME HEATING SYSTEM - SMV 287

List of Figures

FIGURE 1. USAGES OF FORMAL METHODS 7
FIGURE 2. VISUAL REPRESENTATION OF UMPLE CLASS MODEL (LISTING 1). 16
FIGURE 3. VISUAL REPRESENTATION OF THE HOME HEATING STATE MACHINE. 27
FIGURE 4. RELATIONSHIP BETWEEN EXPRESSIVE POWERS OF CTL AND LTL 47
FIGURE 5. TRANSFORMATION ENGINEERING 56
FIGURE 6. THE PIPELINE MODEL OF ALLOY TRANSLATOR 59
FIGURE 7. PARTIAL METAMODEL OF UMPLE FOR CLASS MODELS 60
FIGURE 8. AN OVERVIEW OF THE ALLOY METAMODEL 63
FIGURE 9. PART A - ALLOY METAMODEL 64
FIGURE 10. PART B - ALLOY METAMODEL 65
FIGURE 11. PART C - ALLOY METAMODEL 66
FIGURE 12. PART D - ALLOY METAMODEL 66
FIGURE 13. TRANSFORMATION ARCHITECTURE FOR NUXMV CODE GENERATION 67
FIGURE 14. UMPLE'S METAMODEL FOR STATE MACHINE DIAGRAMS 68
FIGURE 15. VISUAL REPRESENTATION OF NUXMV METAMODEL 71
FIGURE 16. COLLISION AVOIDANCE SYSTEM 94
FIGURE 17. SYMBOLIC EXAMPLE TO ILLUSTRATE FROM, NEXT AND IN-TRANSITIONS OF SIMPLE AND COMPOSITE STATES 103
FIGURE 18. VISUAL REPRESENTATION OF RELATIONSHIP BETWEEN TRANSITIONS IN UMPLE 106
FIGURE 19. VISUAL REPRESENTATION OF BASE TRANSITIONS 107
FIGURE 20. EXAMPLES OF HIGH-LEVEL TRANSITIONS 108
FIGURE 21. MANIFESTATIONS OF AND-CROSS TRANSITIONS 110
FIGURE 22. FLOW OF EXECUTION IN UMPLE 112
FIGURE 23. COMPOSITE STATE WITH EMPTY SET OF EMBEDDED TRANSITIONS 114
FIGURE 24. ACTIVATION-BY-DEFAULT FOR SIMPLE AND NON-ORTHOGONAL COMPOSITE STATE 117
FIGURE 25. ACTIVATION-BY-DEFAULT FOR PARALLEL STATES 118
FIGURE 26. DEMONSTRATING COMPLEXITY OF TRANSITIONS INTO CONCURRENT STATE 119
FIGURE 27. ACTIVATION-BY-DEFAULT FOR NON-ORTHOGONAL COMPOSITE STATE 120
FIGURE 28. ILLUSTRATING INCONSISTENCIES RESULTING FROM ACTIVATION-BY-DEFAULT AND HIGH-LEVEL TRANSITIONS 125

Figure 29. Demonstrating disabling transitions for non-parallel state machine ... 127
Figure 30. Symbolic example to illustrate inconsistencies with parallelism ... 129
Figure 31. Analyzing Minimal Subset to Disable Orthogonal Region ... 130
Figure 32. Analyzing a disabling transition set for an orthogonal region ... 133
Figure 33. Disabling regions of orthogonal states ... 134
Figure 34. And-cross transition examples ... 137
Figure 35. A Venn diagram to illustrate the relationships between sets of transitions necessary for disabling any region A whose immediate ancestor is B ... 139
Figure 36. Abstract example to demonstrate same-source pairs of transitions ... 144
Figure 37. Abstract example to demonstrate region-cross cases ... 146
Figure 38. Demonstrating inconsistencies with multiple and-cross transitions in the same enclosing state ... 148
Figure 39. Cases of false positives ... 151
Figure 40. 2-Bit Counter designed based on and-cross transition ... 161
Figure 41. Symbolic example to illustrate unreachability of states ... 169
Figure 42. 2-Bit Counter designed based on A1 ... 173
Figure 43. 2-Bit Counter design based on A2 ... 175
Figure 44. 2-Bit Counter design based on A3 ... 177
Figure 45. Visual Representation of Car Transmission Sub-System ... 181
Figure 46. Transformed version of Figure 20 ... 187
Figure 47. A flat version of course section state machine ... 190
Figure 48. Illustrating Under-specification in parent-child association ... 200
Figure 49. Simulation result after introducing no-cycle constraint ... 201
Figure 50. Simulation Architecture ... 202
Figure 51. The Test-Driven Development (TDD) Architecture ... 203
Figure 52. High-Level Representation of Home Heating System ... 205
Figure 53. The room sub-system ... 206
Figure 54. The controller sub-system ... 207
Figure 55. The furnace sub-system ... 208
Figure 56. The Collision Avoidance SSUA[102] ... 211
Figure 57. Comparison of modelling solutions (per number of transitions) for the Collision Avoidance Feature ... 213
Figure 58. Experimental Results for R1 Versus Methods ... 217
Figure 59. Resource Utilization for the Analysis of R2 ... 218
Figure 60. Graph of State Space Vs Number of Transitions ... 221

Figure 61. Graph of No. of Transitions Vs. Reachable Number of States 222
Figure 62. Graph of Average Time and Reachable States of the SSUAs 223
Figure 63. Graph of Average Number of BDD Nodes and Reachable States of the SSUAs 223
Figure 64. Graph of Average Memory Usage and Reachable States of the SSUAs 224
Figure 65. Graph of execution times for questions for simple and hierarchical SSUAs of Xholon watch 232
Figure 66. Graph of required number of BDDs for SSUAs (i.e., simple and hierarchical) of Xholon watch 233
Figure 67. Graph of memory utilization for Xholon watch SSUAs (i.e., simple and hierarchical) 233
Figure 68. Model of the Electronic Seating System 236
Figure 69. Alloy Equivalent of the Electronic Seating System 237
Figure 70. Counterexample showing multiple students with the same identities 242
Figure 71. Counterexample produced as a consequence of P2 244

List of Tables

TABLE 1. TABLE OF SYMBOLS xix
TABLE 2. MAPPING OF NOTATIONS AND MEANINGS 15
TABLE 3. UMPLE MULTIPLICITIES 19
TABLE 4. MULTIPLICITY CONSTRAINTS MAPPING 36
TABLE 5. SEMANTICS OF LTL OPERATORS FOR NUXMV [69] 48
TABLE 6. SEMANTICS OF CTL OPERATORS IN NUXMV [69] 49
TABLE 7. SUMMARY OF ATTRIBUTES OF ANALYSIS ENGINES 50
TABLE 8. ATTRIBUTE MAPPING FROM UMPLE TO ALLOY 73
TABLE 9. MULTIPLICITY MAPPINGS FROM UMPLE TO ALLOY 74
TABLE 10. CASES OF NONDETERMINISM 151
TABLE 11. SYMBOLIC EXAMPLE TO ILLUSTRATE MATCHING BETWEEN DIFFERENT SETS 156
TABLE 12. SYMBOLIC EXAMPLE TO ILLUSTRATE MATCHING AND FILTERING 156
TABLE 13. ANALYSIS RESULT FOR NON-DETERMINISM 209
TABLE 14. SUMMARY OF RESULTS OF NON-DETERMINISM ANALYSIS 210
TABLE 15. ANALYSIS AND PERFORMANCE RESULTS FOR VARIOUS METHODS AND REQUIREMENTS 216
TABLE 16. SUMMARY OF ANALYSIS RESULTS 222
TABLE 17. ANALYSIS RESULTS FOR XHOLON WATCH STATE MACHINE DIAGRAMS 231
TABLE 18. SUMMARY OF ANALYSIS RESULTS FOR CASE STUDY 3 234
TABLE 19. ANALYSIS OF ASSOCIATIONS OF THE E-SEAT SYSTEM 238
TABLE 20. COMPARISON BETWEEN UMPLE AND ALLOY OF E-SEAT SYSTEM 240
TABLE 21. RESEARCH QUESTION FOR THE SURVEY 246
TABLE 22. SELECTED SOURCES FOR LITERATURE SEARCH 247
TABLE 23. CATEGORIES OF KEYWORDS 247
TABLE 24. INCLUSION CRITERIA FOR THIS SURVEY 248
TABLE 25. COMPARISON OF SOFTWARE VERIFICATION TOOLS WITH "OUR WORK" 258
TABLE 26. SUMMARY OF TOOL ON SUPPORT FOR AND-CROSS TRANSITIONS AND NONDETERMINISM 266

Table of Listings

LISTING 1. EXAMPLE MODEL OF CLASS DIAGRAMS IN UMPLE....17

LISTING 2. UMPLE'S GRAMMAR FOR VARIABLE DECLARATION....21

LISTING 3. GENERAL SYNTAX FOR SPECIFYING KEYS IN UMPLE....23

LISTING 4. PARTIAL MODEL OF HOME HEATING SYSTEM....26

LISTING 5. UMPLE'S TRANSITION NOTATION....28

LISTING 6. UMPLE'S STATE NOTATION....30

LISTING 7. UMPLE'S NOTATION FOR STATE MACHINE....30

LISTING 8. LETTER TEMPLATE EXPRESSED IN UMPLE....31

LISTING 9. THE GRAMMAR OF ALLOY'S LANGUAGE....34

LISTING 10. MODELLING EXAMPLE IN ALLOY....35

LISTING 11. HEAT CONTROLLER STATE TRANSITION SYSTEM....44

LISTING 12. MODULE'S GRAMMAR IN NUXMV....46

LISTING 13. TEXTUAL REPRESENTATION OF THE PARTIAL METAMODEL OF UMPLE....61

LISTING 14. TEXTUAL REPRESENTATION OF THE PARTIAL METAMODEL OF ALLOY....62

LISTING 15. PARTIAL METAMODEL OF STATE MACHINE IN UMPLE....69

LISTING 16. TEXTUAL REPRESENTATION OF THE NUXMV METAMODEL....70

LISTING 17. EXAMPLE OF ASSOCIATIONFACT CONSTRAINT (NON-NUMERIC)....76

LISTING 18. EXAMPLE OF ASSOCIATIONFACT CONSTRAINT (NUMERIC)....77

LISTING 19. ALLOY SIGNATURE FROM CLASSES....78

LISTING 20. ASYMMETRIC ASSOCIATION EXAMPLE....80

LISTING 21. SYMMETRIC ASSOCIATION EXAMPLE....81

LISTING 22. AN EXAMPLE OF BI-DIRECTIONALITY FACT....82

LISTING 23. AN EXAMPLE OF NUMERICBOUNDFACT (EXACT VALUE)....83

LISTING 24. AN EXAMPLE OF NUMERICBOUNDFACT (RANGE VALUE)....83

LISTING 25. THE GENERAL HIERARCHY CONSTRAINT....84

LISTING 26. TEXTUAL REPRESENTATION OF COLLISION AVOIDANCE SYSTEM....94

LISTING 27. THE MATCH-MAKING ALGORITHM....153

LISTING 28. THE EXPLORATION ALGORITHM....154

Listing 29. The filtering algorithm 157
Listing 30. Umple representation of 2-bit counter system 159
Listing 31. Textual Representation of Car Transmission Sub-System 181
Listing 32. The smv equivalent of course registration system 191
Listing 33. The root state machine of transmission sub-system 193
Listing 34. Transmission module 195
Listing 35. ParkAndNeutral module 195
Listing 36. Wrapper and Main modules for instantiation purposes 196
Listing 37. Reachability specification for transmission system 197

List of Definitions

Definition 1. Global Configuration 29
Definition 2 - Transformation 51
Definition 3 – Transformation Definition 51
Definition 4 – Transformation Rule 51
Definition 5 – Endogenous Transformation 52
Definition 6 – Exogenous Transformation 52
Definition 7 - Vertical Transformation 53
Definition 8 – Horizontal Transformation 53
Definition 9. State Machine System Under Analysis (SSUA) in Umple 95
Definition 10. State Machine in Umple 97
Definition 11. Source State (or from state) 102
Definition 12. Destination State (or next state) 102
Definition 13. In-transition of State 102
Definition 14. Embedded Transitions of State 105
Definition 15. High-Level Transition 107
Definition 16. And-Cross Transitions 109
Definition 17. Unusual Transitions 109
Definition 18. Enabling Transitions of a State 110
Definition 19. Step of Execution 110
Definition 20. Micro-Step of Execution 111
Definition 21. Macro-Step of Execution 111
Definition 22. Active State of a State Machine 116
Definition 23. And-cross transition with a target region other than the 136
Definition 24. Parallel Transitions of an Orthogonal State 145
Definition 25. Enabledness of transitions of an SSUA 150

List of Research Questions

RESEARCH QUESTION 1 6
RESEARCH QUESTION 2 120
RESEARCH QUESTION 3 124
RESEARCH QUESTION 4 129

List of Propositions

PROPOSITION 1. SUFFICIENT CONSTRAINT FOR ACTIVATION-BY-DEFAULT 120

PROPOSITION 2. SUFFICIENT SUBSET OF TRANSITIONS OF THE SSUA FOR DISABLING A NON-PARALLEL SUB-STATE MACHINE. 124

TABLE 1. TABLE OF SYMBOLS

Symbol	Formula	Explanation
	V_{SSUA}	A finite set of attributes of the SSUA
	S_{SSUA}	The universal set of states of the SSUA
	M_{SSUA}	The universal set of state machines forming the SSUA
	m_{SSUA}^0	The root state machine of the SSUA
	$\mathrm{R_{SSUA}}$	The universal set of transitions of the SSUA
	R_A	The universal set of transitions of state machine A.
	n_A	The name of the state machine A
	S_A	Finite set of top-level states of state machine A
	s_A^0	The initial state of state machine A
	l_A	A finite set of labels enclosed in A
	U_{SA}	A universe of states embedded in a state machine A
	E_A	A set of embedded transitions of state machine A
γ	$\gamma(l)$	The trigger e (or event) on transition label $l = \langle g, e, a \rangle$
X	$X(t)$	X maps transition t to its destination state (see Definition 12)
F	$F(t)$	F maps transition t to its set of source state(s) (see Definition 11)
α	$\alpha(M)$	α maps state machine M to its type (i.e., simple or hierarchical)
β	$\beta(s)$	β maps state s to its number of sub-state machines
ρ	$\rho(M)$	ρ maps state machine M to its parent state
L	L(t)	L maps transition t = (x, y, z) to y (i.e., its label)
$\rightarrow$	$a \rightarrow b$	If a then b
$\leftrightarrow$	$a \leftrightarrow b$	a is true if and only if b is true
$\sqsubset$	$A \sqsubset B$	A is an immediate descendant of state machine B
$\overleftarrow{\sqsubseteq}$	$A \overleftarrow{\sqsubseteq} B$	A is an element of the set of descendants of state machine B (including B)
$\overleftarrow{\sqsubset}$	$A \overleftarrow{\sqsubset} B$	A is a descendant state machine of B
$\sqsubset_{\|\|}$	$A \sqsubset_{\|\|} B$	A is parallel to state machine B
$\lhd$	$a \lhd b$	a is an immediate descendant of state b
$\overleftarrow{\unlhd}$	$a \overleftarrow{\unlhd} b$	a is an element of the set of descendants of state b (including b)
$\overleftarrow{\lhd}$	$a \overleftarrow{\lhd} b$	a is a descendant of b
$\in$	$x \in X$	x is defined as an element of set X
$\forall$	$\forall_{x \in X}$	x is defined as any element of set X
$\exists$	$\exists_{x \in X}$	x is defined as some elements of set X
Δ	$\Delta(x)$	The set of in-transitions of state x (see Definition 13)
∇	$\nabla(x)$	The set of embedded transitions of state x (see Definition 14)
H	H(s)	The set of high-level transitions of state s (see Definition 15)
$\mathfrak{a}$	$\mathfrak{a}(s)$	The set of and-cross transitions of an orthogonal state s (see Definition 16)
$\mathfrak{u}$	$\mathfrak{u}(s)$	The set of unusual transitions for an orthogonal state s (see Definition 17)
φ	$\varphi(x)$	The set of enabling transitions of state x (see Definition 18)
$\$$	$\$A$	The fully qualified name of state (or sub-state) machine A
υ	$\upsilon(A, s^i)$	The active state of state (or sub-state) machine A at step i (see Definition 21)
DH	DH(M)	The set of high-level transitions that will disable state machine M (see Definition 22)
IH	IH(A)	The set of ignorable high-level transitions for sub-state machine A (see Equation 14)
$\mathscr{b}$	$\mathscr{b}$(A, s)	The set of and-cross transitions with target regions other than A
D	$D(M)$	The set of disabling transitions for state (or sub-state) machine M
$\mathfrak{p}$	$\mathfrak{p}(s, t_i)$	The set of parallel transitions of t_i in state s (see Definition 23)
$\mathscr{v}$	$\mathscr{v}(l)$	The guard g on transition label $l = \langle g, e, a \rangle$
$\not\equiv$	$a \not\equiv b$	a and b are non-overlapping guard statements
$\mathfrak{P}$	$\mathfrak{P}(s)$	The set of embedded states of s (see Equation 23)
$\mathfrak{m}$	$\mathfrak{m}$(s)	$\mathfrak{m}$ maps a top-level state s to its state machine.

1 Introduction

In this thesis, we demonstrate that developers can use a single formalism (a derivative of UML) to generate code for both formal methods and executable systems. Furthermore, we use formal methods to prove some properties of such systems and demonstrate usefulness of and-cross transitions as defined by Harel's statechart semantics [1].

As the complexity of real-world software systems grows relentlessly, the risk of project and system failure remains unabated. This phenomenon is domain independent, as automotive [2], health [3], and business [4] examples attest.

Christel and Joost-Pieter [5] stated:

> "Our reliance on the functioning of ICT systems (Information and Communication Technology) is growing rapidly. These systems are becoming more and more complex and are massively encroaching on daily life via the Internet and all kinds of embedded systems such as smart cards, hand-held computers, mobile phones, and high-end television sets."

Unfortunately, expecting human beings developing such systems to prevent failures by detecting faults is unreasonable unless the humans are supported by sophisticated tools. Such tools must match increasing complexity by increasing the use of abstractions with rigorous mathematical underpinnings.

Tools enabling sound mathematical analysis of software, collectively called *formal methods*, (discussed below) have been available for decades. However, their uptake has been slow since they tend to be too hard for all but the most accomplished computer scientists to use, tend not to scale well, and tend to be somewhat special-purpose.

Another set of tools and techniques in the field called *Model-Driven Engineering* (MDE), also discussed below, combats complexity by allowing relatively easy specification and generation of systems, bypassing the need for humans to understand what is being generated.

To some extent, MDE and formal methods are becoming connected, since MDE tools are more and more being given solid formal semantics. However, the easiest-to-use modeling techniques tend not to be well integrated with state of the art formal methods. This is the issue we address in this thesis.

Our objective is to allow developers to employ the easy-to-use modeling language technology Umple to generate systems, while delegating to state-of-the-art formal methods tools to transparently analyse such systems. In doing so, we hope to increase the applicability of formal methods, and hence improve the quality of software systems.

1.1 Model-Driven Engineering

MDE [6] has emerged as a disciplined approach to addressing software complexity and effectively representing domain concepts, architectural concepts and other aspects of abstract design. MDE advocates both general-purpose and domain-specific modeling languages, model transformation and code generators as means of msanaging complexity of the software industry.

A domain-specific language helps formalize structural and behavioral requirements of objects in target domains. Transformation engines and generators provide mechanisms that facilitate analysis and synthesis of domain-specific artefacts.

Advances in MDE yielded the Unified Modelling Language (UML) [7] in the mid 1990's. UML remains a key standard for representing static and dynamic aspects of software systems, but its current semantics is semi-formal [8]. UML is widely taught because its core concepts, such as class diagrams and state diagrams are simple, yet powerful. Much of its use is informal (i.e., simply drawing of pictures) however, due to tool weaknesses [9]. The Umple technology has been developed to solve certain UML tool problems while retaining and enhancing the ability to generate software. We will discuss Umple in detail in Section 2.1.

1.2 Formal Methods

Formal methods provide strong mathematical discipline with the promise of ensuring correctness of software systems [10], when it is applied correctly in contexts where proof is possible. They

offer notations to express software abstractions with unambiguous semantics and sound mathematical principles to reason about correctness of software.

According to Ouimet and Landquist [11], formal verification of systems (i.e. hardware and software) has gained tremendous attention since the advent of the famous "Pentium bug" which costs Intel Corporation a loss of $475 million in recall of faulty chips in 1994 [12].

Advances in formal methods for software engineering have given birth to various approaches including theorem proving, model checking, and static analysis.

1.2.1 Theorem proving

Theorem proving [13]–[22] provides a deductive approach to the certification of program correctness. It requires the definition of a set of calculi targeted to programming language constructs. Verification of a correctness property is thus subjected to proofs using these calculi (i.e., the deductive verification approach [23]). A program is correct with a set of calculi if and only if the established pre-condition holds before program execution and the post-condition holds after the program execution terminates. Advances in theorem proving have given birth to interactive (e.g., [16], [17], [19]–[21]) and automated (e.g., [22]) theorem proving approaches. Although theorem proving approaches guarantee absolute correctness for certain aspects of a system, their adoption in the industrial settings is low. Their major limitation is that they demand user guidance in the search for solutions and have inadequate support for automation. Another limitation is that each technique considers only a subset of aspects of a program.

1.2.2 Static analysis

Static analysis [24]–[29] is another formalizable approach. A goal of some static analysis techniques is to automatically compute *necessary preconditions* along all paths of a program without execution. According to [26], necessary preconditions are a set of constraints (any violation of which will lead to an error in subsequent program execution). The major benefit of this approach is its potential to address *scalability issues* in certifying the correctness of large-scale software systems. Despite its potential to address scalability, the approach is limited in several ways. Current solutions of this approach target program source code but not higher-level software abstractions such as state machines and class models, etc. Another limiting factor is that to realize

precision and *scalability,* solutions are tailored to a specific problem domain (e.g., avionics, automotive, etc.). To make matters worse, most interesting questions about program correctness are *undecidable* [30].

1.2.3 Model checking

Model checking [5], [31]–[36] is a formal technique with the goal of automatically executing and analyzing software abstractions before the actual system is built. The approach is model-based and is rooted in the principle of *exhaustive exploration*. According to [5], model checking explores all possible system states in a *brute-force* manner. It requires the model and requirements of the system under analysis (SUA) model to be expressed in the dialect of the particular model checking engine and logic formalism such as CTL [37] or LTL [38]. The model and its requirements are fed into the model checker to determine whether the model conforms to its requirements. The failure of the model to conform to its requirements will produce a *counterexample*.

Counterexamples are used as feedback to assist the user in certifying or correcting the SUA. They describe execution paths that lead from the initial system state to states that violate the requirement.

Analysis engines either depend on explicit-state enumeration or symbolic approaches. With the explicit-state enumeration, program states tend to increase exponentially. On the other hand, symbolic approaches (e.g. SAT [39], OBDDs [40], AIGs [41], [42]) efficiently represent sets and relations as Boolean formulas.

Model checking approaches have strong mathematical underpinnings and give results that can be trusted (are sound). The benefits of the approaches include the ease of integration with an existing development cycle and automatic verification. However, they have limited capability to support verification of data-intensive systems and suffer from a state explosion problem.

1.2.4 Limitations on adoption

Despite the attention and potential of formal methods to guarantee bug-free software systems, their adoption for industrial and teaching purposes is poor. The following are the major problems we observed that limit the adoption of formal methods for software verification purposes:

There are numerous formal languages, each with its own advantages and areas of applicability (e.g., some languages are specialized for specific domains, while others are focused on dynamic or static aspects of design).

The languages and tools for formal methods are complex, resulting in a high level of expertise and cognitive effort required to write and understand them.

Manual creation of formal specifications becomes increasingly error-prone as systems become large due to the sheer amount of formal language 'coding' required. It is, we believe not feasible to expect engineers to create formal specifications for large modern systems that are both correct and can be correctly manually translated into a correct functioning system. Automation is needed for both correctness checking, and translation into correct systems. A key objective of this thesis is to allow both to be achieved, starting from one common specification, written in a simpler language.

Formal methods are taught to some extent in universities, but not universally due to their complexity and the lack of needed support tools that can be easily used by students.

Nonetheless, a report by the National Aeronautics and Space Administration (NASA) and the Federal Aviation Authority (FAA) after an investigation into the use of formal methods stated (see pg. 7 of [5]):

> "Formal methods should be part of the education of every computer scientist and software engineer, just as the branch of applied maths is a necessary part of the education of all other engineers."

1.3 Problem Statement and Research Direction

We summarize the problem to be tackled in this thesis as follows:

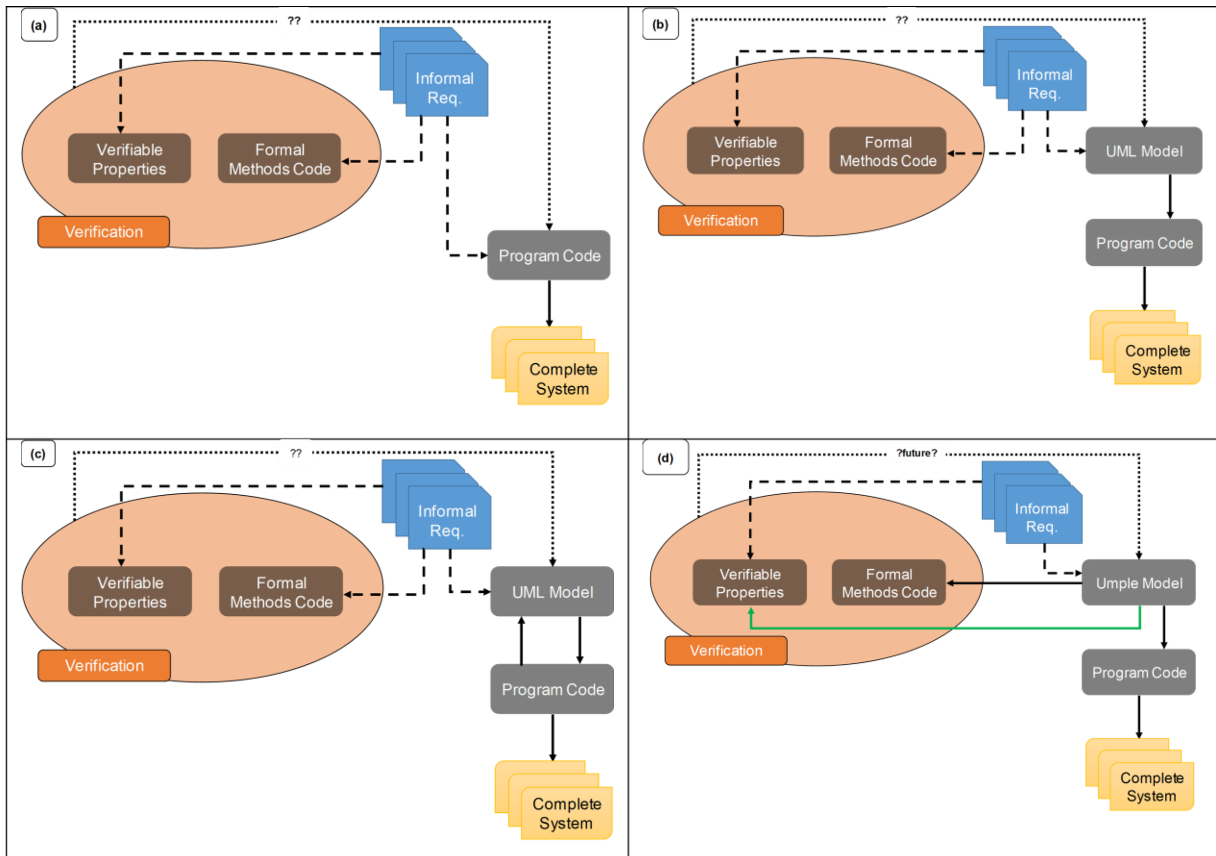
> Formal methods are too difficult to use by ordinary developers, and unsuitable to directly teach in lower-level university courses, due to complexity, scalability and tool suitability issues. Thus, formal methods have low adoption levels.

Our top-level research question derived from this is:

> ***Research Question 1***
>
> How can we improve the adoption of formal methods, both in industry and in teaching, by overcoming some of the complexity, while maintaining analysis capabilities and enhancing scalability?

Our working hypothesis for the thesis is the following:

> It should be possible to make formal methods more usable by hiding their complexity in a similar manner to how the complexity is hidden in other development technologies: By wrapping them in simpler abstractions (i.e., succinct modeling constructs), and generating both executable code and formal methods code from those simpler abstractions where necessary. For example, a complex behaviour of a system may be abstracted in a state machine (simple or hierarchical); relationships between classes can be abstracted in associations; and so on.



Key:

Feedback (dotted arrow)

Automatic transformation (solid arrow)

Manual transformation (dashed arrow)

FIGURE 1. USAGES OF FORMAL METHODS

Where:

(a) Traditional Usage;

(b) Usage with MDE;

(c) Round-trip Engineering Usage; and

(d) Proposed Usage.

By ‘formal methods code’ in this context we mean the expressions in the formal approach’s language.

More specifically, we hypothesize that we can generate formal methods code corresponding to simple models based on UML class diagrams and state diagrams, and use the power of the formal methods tools ‘behind the scenes’ to find defects in the models.

Our objective is to explore how to develop technology to show that the hypothesized capability is achievable.

To help explain the gaps our work addresses, we present various usages of formal methods for software development in Figure 1. This allows us to compare various usages of formal methods with the usage we advocate. These are: traditional usage of formal methods; usage of formal methods with MDE; roundtrip engineering and the usage advocated for using formal methods with MDE in this thesis.

In Figure 1(a), we present the traditional usage of formal methods for software development. The approach does not automate most of the transformations except the transformation to machine code or bytecodes done by a compiler. The starting point is the requirements, which are informal but later developed manually into a visual model (i.e., static and behavior). Then a formal specification of the model and properties are created manually from the model. The specification and the properties are formally analyzed to discover and fix defects. The formal specification and the model is then turned manually into code (e.g., C, Java, etc.) and compiled into a working system automatically. A typical example of this usage is applied by Chan et. al [43].

This approach tends not to be scalable and to be error-prone. For example, applying this approach on an industrial-scale automotive system is arduous. That is, the manual creation of perfectly consistent models, formal specifications and code may delay time-to-market or make the project infeasible. Similarly, keeping track of changes made to the model, code and formal specification is difficult to guarantee. Maintaining consistent model artifacts whenever there is a change may be unrealistic.

An approach to more closely connecting MDE with formal methods is presented in Figure 1(b). This approach involves manual creation of a visual (and sometimes textual) representation of the model from informal requirements in a language such as UML; this is not generally considered a formal model because formal verification (e.g., model checking) cannot be applied to such models.

Nonetheless the tool managing the model maintains a machine-processable representation of the model, which is then used for automatic generation of code (e.g., Java). To formally analyze systems with this approach, the formal specification and properties are also created manually and passed to an analyzer for analysis. The limitations of this approach for formal analysis of SUD include scalability and consistency between model and formal specifications. In other words, although the code may be synchronized with the model, the formal specification must still be manually kept in synch with the model.

Another approach often adopted for software development is *round-trip engineering* (see Figure 1(c)) with automatic code-generation. According to Klein et. al [44], round-trip engineering provides capabilities that facilitate manual editing of generated code such that the changes made to it are reflected back in the model. Round-trip engineering support attempts to guarantee consistency between code and model. However, it is still an issue to maintain consistency between model and formal specifications of the SUD.

We present an approach that maintains consistency between model and code, *as well as the* model and formal specification. This is illustrated with the diagram presented in Figure 1(d). Our approach implements the model-code duality principle such that a model and code blend into one extended model. We express the model/program textually as an Umple program (files with extension **.ump*) which is then parsed and analyzed. The Umple compiler produces an Umple Internal Representation (UIR) of the model/program. We have developed code generators to automatically generate both the final executable system, and formal specifications of static and dynamic behaviors of the SUD from the UIR. The Umple model becomes the 'master'; all changes are applied to it; formal verification is performed on a formal model that is automatically kept in synch with the Umple model.

We use Umple in our work for a variety of reasons. It implements model-code duality as described above; we have full access to it as an experimental testbed, and it meshes well with the text-based software development methods, such as automated testing, that are widely used in industry. Its capability to automatically generate formal specifications enhances scalability of formal representations and guarantees consistency between model and formal specifications.

To limit the scope our research, we decided to limit the models we would investigate to class diagrams and state diagrams. These specific modeling abstractions are widely taught, readily

understandable and, taken together, are at the core of the designs of a large proportion of systems. We chose to work on both to generalize our work to more than one type of model and formal method.

Despite the improvements our work offers, we deem it important to discuss its limitations. Currently, the analyst is required to add *actions* (e.g., assignments to variables in response to events) manually to the generated formal specifications to enable exhaustive analysis. Similarly, the analysis of algorithmic logic embedded in the model is out of the scope of this work. The focus of our work is the discovery of defects at the modeling (i.e., abstract or high) level but not at the generated-code (i.e., low) level. Hence, we encourage analysts to leverage Umple's infrastructure for test-driven development (TDD) to manage this aspect.

1.4 Choice of Tools

We had to choose the tools on which we would base our work. For the formal methods tools our selection criteria included that the tool must be actively developed and researched, capable of a wide range of mathematical reasoning. It was also important to choose only analysis engines that require no user guidance or invention to create mathematical lemmas in the process of proof search. For the selection of the model-driven tool, we needed it to be open source (so we can modify it to integrate the formal methods), easy to use and capable of full system generation.

For all tools, we needed them to be scalable to systems of very large size (with the understanding that the formal method code would have to be automatically generated).

We have developed a modular encoding approach that allows reasoning about temporal properties on moderately complex state machine models, and static properties in class diagrams. Our solution automatically generates the formal representation of systems; thus, the analysis of large systems is possible and users are shielded from mathematical notations.

As a formal method and tool set for analysing dynamic properties of systems, we selected *nuXmv* [45] or its variant NuSMV [32] for systems with unbounded variable(s). *nuXmv* is a new symbolic model checking tool for the verification of fair, finite- and infinite state synchronous systems. Its major goal is to address complexity characterizing *data-intensive* systems and preserve automated analysis benefits of model checking approach. It extends NuSMV [32], a state-of-the-art model checker for the specification and verification of finite state systems. *nuXmv* inherits basic

verification techniques of NuSMV but extends its native language with *Unbounded Integer* and *Real* data types for the specification of infinite domains. To enable verification of the newly supported domains, *nuXmv* integrates Satisfiability Modulo Theory (SMT [46]) algorithms.

nuXmv had been adopted for the verification of various applications in academic and industrial contexts [31], [47]–[50]. Among other state-of-the-art model checking tools, performance results show that nuXmv is highly competitive [45], [51].

To enable analysis of static aspects of systems, we selected Alloy [52]. This implements a first-order logic language for expressing software abstractions, simulating and checking requirements of software systems [53]. It provides mechanisms for expressing transitive closure, universal and existential quantifications, predicates, functions, relations, invariance, multiplicities, inheritance, and so on. With these mechanisms, Alloy is suitable for representing object models, simple and complicated constraints, and operations manipulating the structures dynamically. Hence, it is mostly suitable for specifying and validating structural properties of software [53].

Verification and validation of systems with Alloy is fully automatic with instant feedback from its SAT-based analyzer. It adopts a *bounded verification strategy* as a means of handling *undecidability issues*. Hence, Alloy is *sound* but *incomplete*. The Alloy analyzer is capable of discovering inconsistencies via simulations, and counterexamples by checking assertions. The simulation mechanism provided by Alloy allows detection of situations when no instance of the specified model exists within the defined scope. On the other hand, for cases when the analyst's intention is compromised, the analyzer generates a counterexample. Alloy has gained significant attention in academic research work on formalizing UML class diagrams [54]. System construction with Alloy is based on the following notions: signature, fields, facts, functions, predicates, and assertions.

As the MDE tool we will use Umple [55]–[58] as the master language for representing and generating real-world software systems. It supports the 'model-code duality' principle meaning that it represents software models, not only as diagrams but also equally easily and interchangeably in textual form [55].

Umple allows developers to model static and dynamic views of software systems and automatically generates Java, C++, Ruby, Php, etc. code from the system model. Umple achieves this by providing constructs and environments to express a rich subset of Unified Modeling

Language (UML) [7], such as class models; state machine models; and composite structure models. It provides code generation for UML associations that fully supports referential integrity and multiplicity constraints; and it supports unlimited hierarchically nested and concurrent state diagrams.

1.5 Contributions

As discussed above, the goal of this work is to integrate formal methods in a usable way with model-driven engineering to allow formal analysis of software abstractions (e.g., UML state machine diagrams and static class models); and enhance adoption of formal methods for software engineering education and industrial practices. To be specific, we intend to bridge the gaps between model and code, and model and its formal representations. This general goal requires solving various sub-problems, each of which is a distinct contribution. Therefore, we highlight the contributions of this thesis as follows:

a) The overall approach of facilitating formal analysis by having the developer model the complete system in a simple modeling language (in our case, Umple), and systematically generating formal methods from this for verification purposes, while at the same time generating the final system from the same model, thus preventing the need for re-implementation;

b) A novel approach to encode state machine-based systems even in the presence of and-cross transitions for symbolic model verification (see [59]);

c) A fully automated approach to certify a state-machine based system to be free of non-determinism even in the presence of unbounded variables and multiple and-cross transitions in the same enclosing state (see [60]);

d) A comparative study of and-cross transitions and various alternative approaches that can substitute and-crossing for modeling state machine diagrams (see [61]);

e) An empirical study of the impact of abstraction on some performance parameters (e.g., execution time, memory usage and the number of Binary Decision Diagrams - BDDs); and

f) Transformation tools from Umple (and hence from UML) to SMV and Alloy.

We summarize the contributions of this work in the following thesis statement:

> **Thesis Statement:** Both executable code and formal methods code can be generated from the same high-level model, therefore allowing developers to perform formal analysis and production of a final system from the same abstract source. It is possible to automatically analyze models corresponding to the final system for nondeterminism and consistency even in the presence of unbounded variables and and-cross transitions. And-cross transitions provide useful high-level abstraction and are applicable to modeling real-world problems that characterize software products. A set of non-conflicting transitions is computable to enable and disable states and sub-state machines of these systems for model checking purposes. By comparatively studying performance of various means of high-level abstractions, it can be shown that high-level abstractions do not always translate to performance benefits during model checking.

1.6 Limitations of the work

We tackle a general problem, but our work is limited in the following ways:

- We are focusing on class diagrams and state diagrams as representable in pure Umple. This means that any concept not representable in Umple will not be representable or analysable when we generate formal methods code. Umple models, although broadly capable of representing many aspects of systems, are not as generalized as a typical programming language. Our approach does not attempt to generate formal methods from programming language code embedded in Umple. And our approach does not purport to be able to analyse systems where injected programming language code alters the core Umple modeling semantics, which is possible using Umple's aspect-oriented code injection capabilities.
- Umple assumes certain semantics for state machines and class diagrams; other tools may adopt slightly different semantics. We have focused on generating analysis code to match Umple semantics.

1.7 Thesis Outline

The following is an outline that summarizes contents of chapters of this work:

Chapter 2 presents background information about techniques and technologies required to understand this research. The technologies we survey are Umple, Alloy and nuXmv.

Chapter 3 presents processes involved in transformation engineering. Particularly, we present our approach to automatically generate formal specifications of software abstractions expressed in Umple.

Chapter 4 presents formal specification of Umple class models in Alloy for the purpose of analysis. Umple class models include classes, attributes, and associations. These modeling constructs are formally specified in Alloy's input language. Our discussion focuses on attribute and multiplicity mappings, constraint specifications, and object-oriented design patterns.

Chapter 5 presents our approach to formalize state machines in Umple. Particularly, it discusses our methods to compute the set of enabling and disabling transitions for states and sub-state machines in Umple.

Chapter 6 presents our approach to raise the quality of state machine systems under analysis (SSUAs) expressed in Umple. We focus on discovering non-determinism and reachability of states.

Chapter 7 presents the transformation of state machine models by example. This involves the mapping of each Umple state machine construct to its equivalent SMV construct.

Chapter 8 presents our approach to verify and validate our work. For verification purposes, we proposed simulation and rigorous test-driven development.

Chapter 9 presents a survey of related work and a comparative study of solutions closely related to our work.

Chapter 10 presents concluding remarks and directions for future research.

2 Background

In this chapter, we present background information about techniques and technologies required to understand the rest of this thesis. The technologies we will survey are Umple, Alloy and nuXmv.

When discussing grammars we will use a simplified notation developed for this thesis to facilitate unified representations across the three technologies; it is based on the core Umple grammar [62], although when we present the Umple grammar we will present a simplified view. We will illustrate some concepts with simple examples. Table 2 introduces notations and semantics adopted for discussion.

TABLE 2. MAPPING OF NOTATIONS AND MEANINGS

SYMBOL	MEANING
?	optional (zero or one; UML equivalent – '0..1')
*	any number (zero or more; UML equivalent – '0..*')
+	mandatory (one or more; UML equivalent – '1..*')
[[…]]	non-terminal symbol, referring to another rule
[…]	terminal symbol that can match any simple alphanumeric identifier
|	logical OR-operator
&	logical AND-operator
...	other things of no interest to us when discussing concepts
[xxx]*	any number of "xxx"
[= typeName: value-1 …]	an enumeration of possible values

To ease readability of code in listings, simple state names and terminal symbols are in red; composite state names and non-terminal symbols are in green, class, signature, and module names are in brown and keywords are in blue.

2.1 Umple

Umple is a model-oriented programming technology for the development of real-world software systems. It supports the *model-code duality* principle by representing software models, not only as diagrams but also as text [55]. Umple allows developers to model static and dynamic views of software systems and automatically generates code in languages like Java, C++, Ruby, Php from the model. Umple achieves this by providing constructs and environments to express a rich subset of Unified Modeling Language (UML) [7], such as class models; state machine models; and composite structure models. It also allows direct expression of some concepts not in UML, such as certain patterns, and algorithmic code in native languages like Java.

Umple was explicitly designed to be simple to use while generating high-quality code. People used to UML diagrams can draw them using Umple (or can import them into Umple from other UML tools), but many people who are used to textual coding can also use Umple, since its lightweight syntax for UML constructs can be blended with programming language code.

We will discuss below how Umple handles the notions of state machines and class models, since these will be particularly relevant to this work.

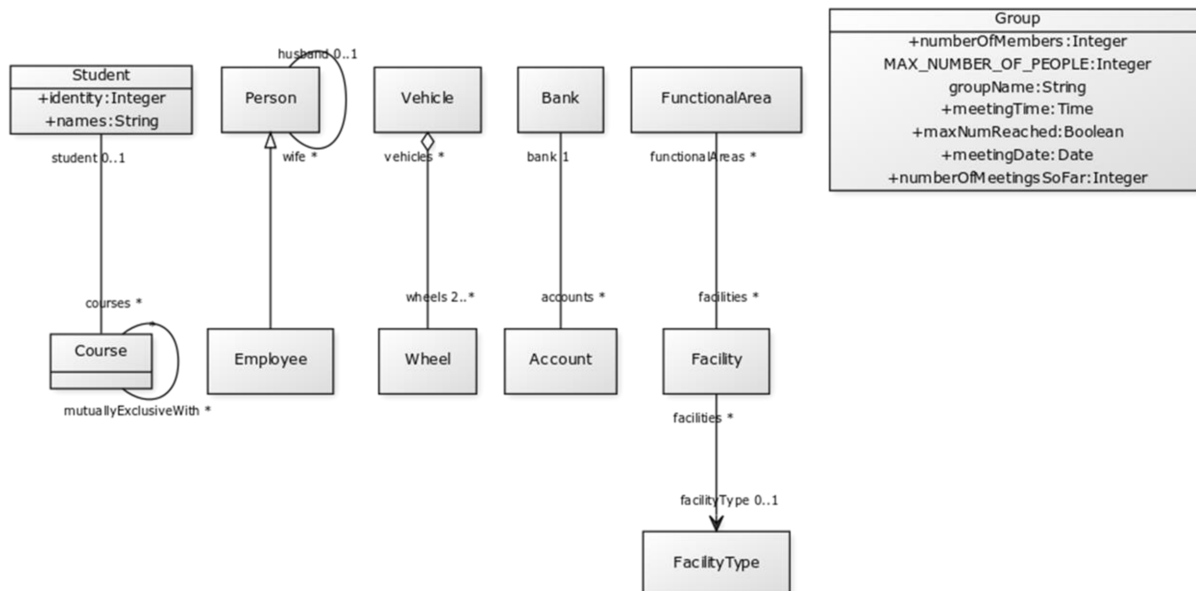


FIGURE 2. VISUAL REPRESENTATION OF UMPLE CLASS MODEL (LISTING 1).

```
1   //Umple Classes
2   class Person { abstract; }
3   class Bank { singleton; }
4   class Account { }
5
6   //Directed associations in Umple
7   class FacilityType { }
8   class Facility { * -> 0..1 FacilityType; }
9
10  //Bi-directional association
11  class FunctionalArea { * -- * Facility; }
12
13  //Independent association
14  association { 1 Bank -- * Account; }
15
16  //Composition association
17  class Wheel { }
18  class Vehicle { * <@>- 2..* Wheel; }
19
20  //Reflexive associations
21  class Person { 0..1 husband -- 0..* Person wife; } // (asymmetric)
22  class Course { * self mutuallyExclusiveWith; } // (symmetric)
23
24  //Umple Attributes
25  class Group {
26   Integer numberOfMembers;
27   const Integer MAX_NUMBER_OF_PEOPLE = 20;
28   immutable groupName;
29   Time meetingTime;
30   Boolean maxNumReached;
31   Date meetingDate;
32   autounique numberOfMeetingsSoFar;
33  }
34
35  //Specialization and generalization
36  class Employee { isA Person; }
37
38  //Other associations
39  class Student {
40   0..1 sorted { identity } -- * Course;  //sorted association
41   Integer identity;
42   String [] names; //arrays
43   key { identity }
44  }
```

LISTING 1. EXAMPLE MODEL OF CLASS DIAGRAMS IN UMPLE

2.1.1 Umple Class Models

Umple provides constructs for representing most constituents of the UML class models, and some concepts that go beyond class models. The constructs are sufficient to express structural properties of any kind of object-oriented system. These include attributes, associations with multiplicity, keys and various patterns such as singleton.

Figure 2 is the visual representation of the code presented in Listing 1 (see below). As is standard in Object-Oriented programming and modeling, a class defines a reusable entity in an object-oriented system whose instances can be created at run-time. A class in Umple may be stereotyped in various ways, such as *abstract* or *singleton*. An *abstract class* is a class that cannot be instantiated at run-time, but instances of its non-abstract subclasses can be created. This can be realized in Umple with the *abstract* keyword. A singleton class is constrained to have at most one instance at run-time, and is specified by the *singleton* keyword in the Umple class definition. Classes without any of the above keywords are not limited in terms of the number of instances created at run-time. Lines 2-4 of Listing 1 demonstrate the specification of various kinds of classes in Umple. This example defines the structural relationships of some objects in a bank subsystem with emphasis on Umple syntax for representing highlighted class types. In the discussion that follows we will use the textual representation of Umple presented in Listing 1 as a basis for explanations.

2.1.2 Associations

As in UML, an association models the mapping of instances of one class to instances of another (or possibly the same) class. Umple supports only binary associations. A reflexive association involves two ends with the same class; while a non-reflexive association involves two different classes. Reflexive and non-reflexive associations can either be unidirectional ('->', e.g. see line 8), meaning that they can be navigated in only one direction, or else bidirectional ('--', e.g. see line 11). An association may be defined as composition ('-<@> or <@>-', e.g. see line 18) meaning that the composed objects are to be destroyed when the composing object is destroyed.

Each association end defines an optional role name, and a mandatory multiplicity of the class it describes. Multiplicities in Umple allow developers to constrain the cardinalities of objects collaborating with each other in an association. A multiplicity defines both the lower and upper

bounds of the number of instances of objects allowed at run-time. Table 3 presents the syntax and semantics of multiplicities in Umple.

TABLE 3. UMPLE MULTIPLICITIES

UMPLE SYNTAX	MEANING OR SEMANTICS
1..*	Mandatory many
* or 0..*	Any number
1	Mandatory
0..1	Optional
n..m	Range()
n..n or n	Exact()

Umple supports both inline and independent definitions of associations. An inline association refers to an association defined within one of the collaborating classes. On the other hand, an independent association refers to association defined outside any of the collaborating classes. The following is a discussion on Umple constructs for the representation of various types of association: unidirectional, bidirectional, composition, reflexive, and symmetric-reflexive; and independent.

It should be noted that Umple associations, along with generalization and attributes (discussed shortly) have been part of the Umple language since its beginning in 2006, and were originally developed by PhD student Andrew Forward [63], and Masters student Dusan Brestovansky [64]. Many other students contributed enhancements to them over the years.

2.1.2.1 Unidirectional associations

The example on line 8 illustrates the syntax of unidirectional association in Umple. It models the relationship between facility (Facility) and its type (i.e. FacilityType). The representation of unidirectional association is orientation-specific. A right navigation implies right orientation and left navigation implies left orientation. The implication of unidirectionality in this example is that

Facility stores information about its type; but the reference to Facility is insignificant to FacilityType, so FacilityTypes do not store lists of their Facilities.

2.1.2.2 Bidirectional associations

As opposed to the unidirectional association, bi-directional associations (denoted as "--") involve classes where references to the instances of each associated class are accessible to each other. Line 11 of Listing 1 illustrates the syntax for representing this kind of association in Umple. It defines a bidirectional relationship between classes Facility and FunctionalArea.

2.1.2.3 Independent association definition

Associations in Umple may not necessarily be specified as inline (e.g., as specified on line 11 Listing 1). Their specification may be defined independently. Line 14 of Listing 1 is an example of independent association definition between Bank and Account.

2.1.2.4 Composition

A composition is a kind of association. Like other associations, it involves two collaborating objects (composite and parts). The composite object is composed of simpler parts. Its semantics implies that instances of simpler object should be deleted upon the deletion of the composite. Generally speaking, this implies is-part-of relationship in the software engineering context. To specify composition; the diamond goes on the composite end, which could be visually located on the right or the left. With this kind of association, Umple allows the definition to be specified independently. Line 18 illustrate the syntax of composition association (right-oriented) such that: Vehicle is the composite class and Wheel is one its elements.

2.1.2.5 Asymmetric-reflexive associations

A reflexive association is a kind of association whose ends reference the same class. A typical example is a model of class Person such that husband and wife are different kinds of persons playing different roles. The illustration on line 21 of Listing 1 demonstrates Umple's syntax and for representing this scenario. Note that this is an asymmetric reflexive association, since the two ends have different role names.

2.1.2.6 Symmetric-reflexive association

A symmetric-reflexive association is a special kind of reflexive association in which both ends act in the same capacity. For example, a set of mutually-exclusive courses in a university can be represented with this association. The code in line 22 of Listing 1 illustrates its syntax and diagram in Umple.

2.1.3 Other Association-Related Constructs

Umple provides the *sorted* construct with the intention of succinctly representing the notion of association without specifying multiplicity elements such as: multi-set, sequence, set, ordered-set, array, etc. For example, on line 40 of Listing 1 we present an example of a sorted association. The association models the relationship between classes Student and Course, such that instances of students in the association are sorted based on the values of identity attribute. Similarly, the case of array is expressed on line 42. The example presents array of names (of type *String*) as an attribute of Student.

```
1  AttributeDefinition-: [[AttributeStereotype]]?[[AttributeType]]? AttribteName;
2  AttributeStereotype-: [=type: const|immutable|lazy|settable|autounique|
3    defaulted ...]
4  AttributeType-: [= String|Integer|Boolean|Double|Float|Date|Time ...]
```

LISTING 2. UMPLE'S GRAMMAR FOR VARIABLE DECLARATION

2.1.4 Umple Attributes

An attribute defines a property of a class. As in UML, the Umple notion of attributes extends fields in object-oriented programming languages. The extension adds methods for altering or constraining values or state of an attribute. Umple allows developers to describe attributes of a class using different data types (primitive and non-primitive). These include: *String*, *Integer*, *Boolean*, *Double*, *Time*, *Date*, etc. Lines 26-32 of Listing 1 illustrate attribute definitions in Umple. We illustrate usage of attribute types and stereotypes in Umple by this example. It defines a class – Group with various properties. The general syntax for representing attributes in Umple is given in Listing 2. It

presents the grammar for attribute definitions in Umple. The following discussion focuses on Listing 2 unless explicitly stated.

A *const* attribute implies that the value of the attribute is fixed for all instances (the notion of 'static' in C and Java). The semantics of *immutable* qualifier implies that the attribute value is set during construction and remains unchanged throughout the life of the object. In particular, these attributes are non-static with private access and an assumption that the attribute cannot be modified within the class. Umple achieves this by ensuring no 'set' method is generated for attributes qualified as immutable. It is technically possible for a programmer to violate immutability by directly modifying a variable, although Umple best practice is that they must always use set methods. Violation of similar best practices would also render the formal analysis discussed in other parts of this thesis invalid. This is much the same as if a Java programmer were to use reflection to bypass the 'private' declaration on a method. As future work, we have considered detecting violations of this best practice by scanning user-written methods.

The semantics of an *autounique* attribute implies that every object of the class created at run-time is allocated a unique value for the attribute. The value can be queried (i.e., via the 'get' method) but cannot be set. For example, on line 32 the numberOfMeetingsSoFar is unique because the creation of a new group implies an increment in the number of meetings that take place.

As in typical C-family programming languages, a developer would normally explicitly specify a data type for an attribute (e.g., line 26 of Listing 1). Acceptable data types are defined on line 4. It is also possible to specify any other class as an attribute type, but the general recommendation is to only do this with classes that themselves have no associations. An example might be an Address class that has street, city and postalCode attributes.

By default, when the type of an attribute is omitted in a declaration, the Umple compiler sets the type as String (e.g., see line 28 of Listing 1). This allows rapid free-form modeling. An exception to default attribute type is a case with *autounique* qualifier (e.g., see line 32 of Listing 1). The Umple compiler defaults *autounique* attributes to *Integer*.

Another kind of attribute in Umple is the state machine. Umple considers state machines embedded in a class as attributes of the class, whose values are an enumeration of the possible states. The

discussions of state machines are deferred to Section 2.1.8. For more details on the syntax and semantics of Umple attributes, readers should consult the Umple user manual [62].

1	DataType attributeName1, attributeName2, …, attributeNameN;
2	*key* { attributeName1, attributeName2, …, attributeNameN }

LISTING 3. GENERAL SYNTAX FOR SPECIFYING KEYS IN UMPLE

2.1.5 Keys

Umple provides constructs for the specification of which attributes (or associations with a 1 end) make up the primary key. This specification requires a pre-defined attribute to be qualified with keyword '*key*'. Listing 3 is a general syntax for specifying a key.

The syntax implies that the attributes tagged as *key* must exist in the class. For example, attribute identity (see line 41 of Listing 1) is further qualified as key attribute (line 43 of Listing 1) for class Student. The semantics implies that no two students can be associated with the same identity.

2.1.6 Specialization and Generalization

As in UML and object-orientation in general, the notion of generalization involves creating a new class (known as superclass) to represent characteristics common to a group of classes. On the other hand, specialization involves making a new class as a subclass of another class (its superclass) whenever the new class shares some properties (e.g. attributes, associations, methods) with the existing class (i.e. superclass). Umple's notation for indicating generalization is the '*isA*' keyword.

Lines 35-36 illustrate the syntax of specialization in Umple. It models class Employee as a special kind (i.e. subclass) of Person (i.e. superclass). This representation applies to the notion of generalization; but the semantics must be preserved.

2.1.7 Constraints in Umple

Umple facilitates the representation of various kinds of constraints. Some, such as multiplicity constraints are built in to Umple's core notation. Others, including state machine guards, class invariants and method preconditions, are written in as Boolean expressions and can be mapped to a subset of Object Constraint Language (OCL); they can constrain various Umple constructs and

appear surrounded by square brackets [58]. The following are the types of constraints present in Umple, organized by the constrained element.

1. **Association**: The multiplicities on associations constrain the upper and lower bounds of the number of collaborating objects. The directionality of an association constrains whether an end should store information about its collaborator or not. Reflexivity and symmetricity define further constraints on associations. Class invariants (written in square brackets) can also constrain associations.

2. **Attributes**: Properties (stereotypes) of attributes such as immutability, uniqueness, constant, and laziness constrain various aspects of attribute changeability: Immutability constrains variables such that no change can be made after initial setting. Laziness relaxes the normal requirements that the attribute be set at instance creation. Uniqueness constrains two objects of the same class to have the same value for the attribute. Auto-uniqueness constrains the system to determine the value of the attribute for every given instance of its containing class. In addition to the above any class invariant can constrain values of the attribute during system execution.

3. **State Machines**: State transitions (see next section) may be controlled by a guard; a Boolean expression whose evaluation determines whether the transition executes or not. A transition controlled by a guard executes only if the controlling guard condition is satisfied.

4. **Method preconditions**: A method precondition constrains whether the method is allowed to run.

2.1.8 Umple State Machines

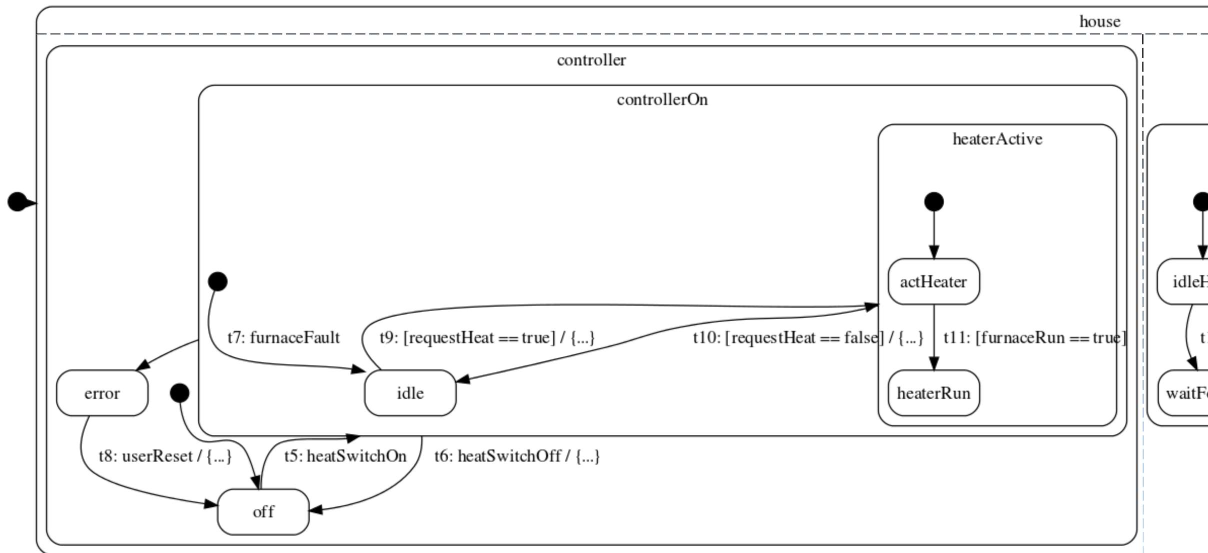
The representation of dynamic aspects of software systems is facilitated in Umple by providing support for an extended subset of UML state machines. These are graphs of states and transitions [7]. The notion of state machines as facilitated by Umple provides constructs to represent states (simple, composite, and orthogonal), transitions (regular, guarded, high-level), and events. State machines in Umple can either be simple or hierarchical. A simple state machine is composed of a set of simple states. For a hierarchical state machine, there are one or more composite or orthogonal sub-states.

State machines are designed to be textually specified in Umple, while their diagrammatic representations are automatically generated as Graphviz [65] images. We present an extract of the home heating system state machine by Lu et. al [66] to facilitate readers' understanding of Umple's state machine representation both in textual and diagrammatic forms. Our discussions of the notions of state machine and syntax will be based on this example. It expresses some of the notions facilitated by Umple but relevant to our work. These include concurrency, transitions, states, guards, and actions.

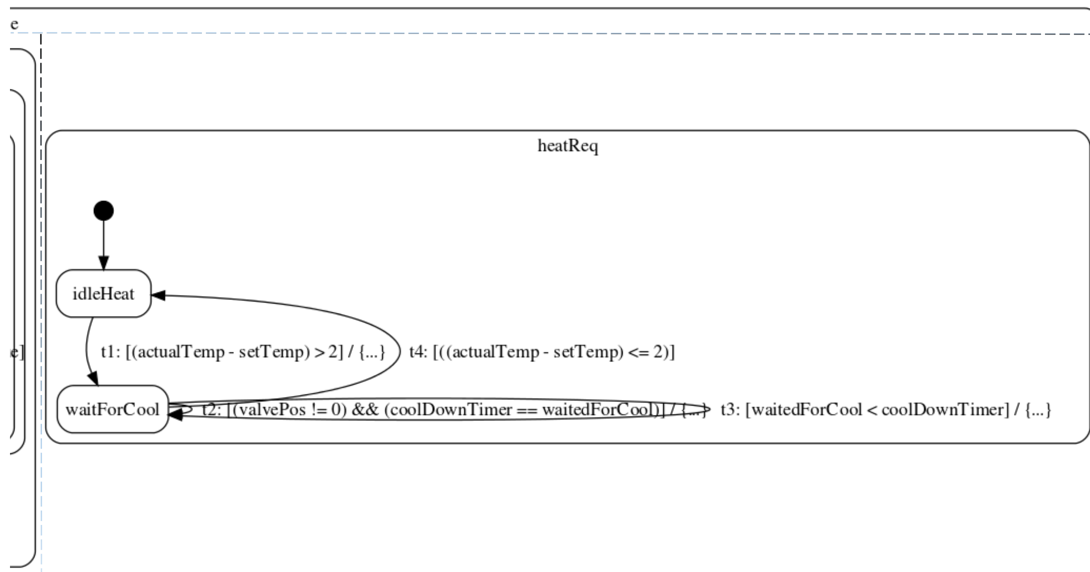
Listing 4 is a textual representation of the system under discussion. State machine – sm is defined as an attribute of class 'HeatController'. Figure 3 is a diagrammatic representation of the state machine automatically generated from the code in Listing 4. This will be used to discuss some of the notions of state machine as facilitated by Umple.

```
1   class HeatController {
2    Integer setTemp;
3    Integer actualTemp;
4    Integer valvePos;
5    Integer waitedForCool;
6    Integer coolDownTimer;
7    Boolean furnaceRunning;
8    Boolean activate;
9    Boolean deactivate;
10   Boolean requestHeat;
11    Boolean furnaceReset;
12
13   sm {
14      house {
15         heatReq {
16            idleHeat {
17                  [(actualTemp - setTemp) > 2]/ { valvePos--; waitedForCool = 0; }  -> waitForCool; //t1
18            }
19            waitForCool {
20                [(valvePos != 0) & (coolDownTimer == waitedForCool)] /{  valvePos--; waitedForCool = 0; }
21                                -> waitForCool; //t2
22                [waitedForCool < coolDownTimer] / { waitedForCool++; }  -> waitForCool; //t3
23                [!((actualTemp - setTemp) > 2)] -> idleHeat; //t4
24            } }
25            | |
26            controller {
27               off { heatSwitchOn -> controllerOn; } //t5
28               controllerOn {
29                  heatSwitchOff / { deactivate = true; } -> off; //t6
30                  furnaceFault -> error; //t7
31                  idle {
32                     [requestHeat == true] / { activate = true; } -> heaterActive; //t9
33                  }
34                  heaterActive {
35                     [requestHeat == false] / { deactivate = true; } -> idle; //t10
36                     actHeater {
37                          [furnaceRunning == true] -> heaterRun; //t11
38                  }
39                  heaterRun { }
40                } }
41                error { userReset / { furnaceReset = true; } -> off; } //t8
42    }}}}
```

LISTING 4. PARTIAL MODEL OF HOME HEATING SYSTEM



(a)



(b)

FIGURE 3. VISUAL REPRESENTATION OF THE HOME HEATING STATE MACHINE.

where:

(a) The Left-View of the State Machine.

(b) The Right-View of the State Machine.

To allow detailed discussions on the notions of state machines supported in Umple, we will in the subsequent sections provide details about states (terminal, simple, non-orthogonal and orthogonal composite), transitions, and Umple's internal representation of state machines.

State machines were added to Umple by PhD student Omar Badreddin [55] and have been improved by many others in subsequent years.

2.1.9 Transitions

As in UML, an Umple transition has a source state and a destination state; the transition is 'fired' (i.e. executes) when a specified event occurs and any associated guard evaluates to *true*. Transitions in Umple can either be: auto, or normal transitions with the former being triggered immediately upon entry into the source state. Listing 5 presents the grammar of the various transitions supported by Umple for state machine definition.

Several transitions, indicated by "->" are shown in Listing 4. The state enclosing the transition is referred to as the 'source'; while the target state is referred to as the 'destination'. For transition t_9 on line 28 of Listing 4, "idle" and "heaterActive" are its source and destination states respectively. t_9 is a basic transition because it has no guard statement.

```
1   transition:- [[standAloneTransition]]|[[autoTransition]]|[[normalTransition]]
2   autoTransition:- [[guard]]?[[transitionCore]][stateName];
3   normalTransition:- [[activity]]?[[eventSpecification]]?[[transitionCore]]
4     [stateName];
5   eventSpecification:- ( [[eventDefinition]][[guard]]|[[guard]]
6     [[eventDefinition]]|[=unspecified][[guard]]|[[eventDefinition]] )
7   transitionCore:- ( [[action]]->|->[[action]]|-> )
8   eventDefinition-: [[afterEveryEvent]]|[[afterEvent]]|[event]|…
9   guard: [[**guardRepresentation]]
10  action: /[**actionRepresentation]+
```

LISTING 5. UMPLE'S TRANSITION NOTATION

A guarded transition has a Boolean expression controlling whether or not a transition is taken whenever an event occurs or automatically taken whenever it is an auto-transition. A transition without a trigger (or a controlling event) is regarded as an auto-transition. For example, t_{11} on line 33 of Listing 4 is an auto transition, because it is not controlled by any event, and guarded, because

a guard statement controls its execution. In particular, the guard controls its execution: whenever the transition is enabled and the guard evaluates to *true* then the transition executes.

On the other hand, a transition with a triggering event becomes enabled whenever the event occurs and source state is in the global configuration (or active), and the guard (if any) evaluates to *true*. At this point, the transition executes and the global configuration reflects the target state of the transition in the next step.

Definition 1. Global Configuration

A global configuration of an Umple state machine A is a quadruple $\langle M \times U_s \times E_s \times V_A \rangle$ such that M is the set of sub-state machines (including the root), U_s is the universal set of states, E_s is a set of execution steps and V_A is a finite set of pairs $\langle n_a, v_a \rangle$ such that n_a is a variable name and v_a is its value. The following expression defines the configuration of an SSUA at step i where sub-machine m_1 is in state s_1, variable n_1 is evaluated to value v_1, and k, j are the number of variables and sub-machines respectively. This definition is further used in defining notions presented in Definition 19.

$$\langle \langle \boldsymbol{m_1}, \dots, \boldsymbol{m_j} \rangle, \langle \boldsymbol{s_1}, \dots, \boldsymbol{s_j} \rangle, \boldsymbol{e^i}, \langle \langle \boldsymbol{n_1}, \dots, \boldsymbol{n_k} \rangle, \langle \boldsymbol{v_1}, \dots, \boldsymbol{v_k} \rangle \rangle \rangle$$

A high-level transition is any kind of transition defined outside of sub-states in a composite state, but which has effect in all the sub-states (e.g., t_6, line 29 of Listing 4).

2.1.10 States

Umple provides constructs for the specification of various kinds of states. These include: initial, end, simple, non-orthogonal composite, and orthogonal states.

An example of a simple state is "idle" defined in lines 31-33 of Listing 4; an example non-orthogonal composite state is "controllerOn" defined in lines 28-40 of Listing 4.

Orthogonal states (e.g., "house" defined on lines 14-42 of Listing 4) exist when two or more sub-states all become activated whenever control is transferred to their parent. For example, sub-states "controller" and "heatReq" become activated whenever "house" is activated. In other words, the execution of sub-states occurs concurrently. Child states of non-orthogonal composite and orthogonal states may be simple, composite, or orthogonal themselves.

By default, the first state (i.e., simple or composite) defined within an Umple state machine at any level of the hierarchy is regarded as its initial state (e.g., the state "idleHeat" in lines 16-18 in Listing 4). In particular, "idleHeat" is the initial of the state machine corresponding to "heatReq". Listing 6 is a grammar defining '*state*' in Umple.

```
1  state-: [final]?[stateName]{[[stateInternal]]*}
2  stateInternal-: [[stateEntity]]|[[standAloneTransition]]|…
3  stateEntity-: [[=||]]|[[entryOrExitAction]]|[[autoTransition]]|
4    [[transition]]|[[activity]]|[[state]]|; …
5  entryOrExitAction-: [=type: entry|exit]/[actionRepresentation]+
6  activity-: do [**actionRepresentation]
```

LISTING 6. UMPLE'S STATE NOTATION

2.1.11 Transformed Internal View of Umple State Machines

To facilitate execution and prevent a combinatorial explosion of states, Umple transforms a hierarchical state machine (e.g., "sm" of Listing 4) internally into a collection of state machines [67]. For each non-orthogonal composite state, there is a corresponding state machine such that its sub-states become the states of the state machine when active.

Umple introduces a special 'null' state for every state machine such that these state machines are in their 'null' state until they are activated. Similarly, for every region of orthogonal states, there is also a corresponding state machine.

A state machine is also generated for the root state machine but without the 'null' state, since the root is always active throughout its containing object's life cycle. But sub-state machines are only active when control is transferred to their parent state or any of their sub-states [55], [67]. Listing 7 defines Umple's grammar for constructing state machines.

```
1  stateMachine-: [[inlineStateMachine]]|[[referencedStateMachine]]|…
2  inlineStateMachine-: [=queued]?[=pooled]? [~name]{([[state]]|
3    [[standAloneTransition]])*}
4  referencedStateMachine-: [name] as [definitionName]({[[extendedStateMachine]]}
5    |;)
```

LISTING 7. UMPLE'S NOTATION FOR STATE MACHINE

2.1.12 Umple Template Language (UmpleTL)

The template language facilitated by Umple, called UmpleTL, aims at providing state-of-the-art support to ease the specification of templates for text generation purposes. The Umple compiler, which is written in Umple, uses UmpleTL to generate code. UmpleTL provides various elements such as templates, emit specification methods, and expression, code, and comment blocks. A detailed discussion of the language can be obtained from [62].

```
1   class RefLetterRequest {
2     // Attributes used to construct the instance
3     String fileno;
4
5     // Letter template
6     letterTemplate <<!
7   Subject: Reference request for <<=applicant>>, File #<<=fileno>>
8
9   Dear <<=recipient>>,
10  Our company, Umple Enterprises, is hiring talented software
11  engineers.
12
13  We have received an application from <<=applicant>> who named you
14  as an individual who could provide a letter of reference. Would you
15  please reply to this letter, answering the following questions:
16    * In what capacity do you know <<=applicant>>
17    * For how long have you known <<=applicant>>
18    * Describe the abilities of <<=applicant>> in software development
19    * What his or her strengths and weaknesses?
20    * Please provide your phone number and suitable times to call in
21      case we need to follow up.
22
23  Yours sincerely,
24  <<=sender>>
25  <<=senderSignature>>
26  !>>
27  <</*Specification of the method to generate*/>>
28   emit letterTemplate(String recipient, String applicant, String sender,
29     String senderSignature) (letterTemplate);
30
31   rows <<!<<# for (int i=0; i <= times; i++) {#>>
32   <<=times>> times <<=i>> is <<=times*i>><<#}#>>!>>
33   // Specification of a single method to emit the result
34   emit result(int times)(header, rows, cr); }
```

LISTING 8. LETTER TEMPLATE EXPRESSED IN UMPLE

In Umple, every other text embedded within a template block but outside any of the blocks discussed above are output “as is” by the compiler. For example, lines 19 – 23 of Listing 8 will be output “as is”.

Listing 8 is an example of a template expressed in Umple. It describes a class for generating a reference letter for job applicants. An attribute of the class is “fileno”. We added the “rows” template (see lines 31-34 of Listing 8) to facilitate discussion of code blocks.

Templates were added to Umple in 2013 by PhD students Ahmed Orabi and Mahmoud Orabi [68].

2.1.13 Templates

The template element is fundamental and essential in template creation. It begins with the template name and followed by arbitrary texts of the form: ≪ ! ... ! ≫. For example, lines 6 – 26 of Listing 8 define a template with a name “templateLetter” and its content.

The body enclosed within tags “≪ !” and “! ≫” defines the content of the template. Its content includes other elements except the emit method specification.

2.1.14 Emit Method Specification

The emit method specification is important for all templates. For every template there need to be one or more ‘*emit*’ statements that invoke the template, or else the template needs to be nested inside some other template, which is emitted. These methods specify the logic of output to be generated. Just like every other method, an emit method may be associated with a set of arguments, each separated from another by a comma. For example, lines 28, 29 of Listing 8 define an *emit* method for “letterTemplate”. Its arguments are “recipient”, “applicant”, “sender” and “senderSignature” of type *String*. The other argument (e.g. letterTemplate) references the template of interest. It must have at least an argument specified in the order of composition.

2.1.15 Expression Block

The expression block allows programmers to specify arbitrary expressions with the following tags “≪=” and “≫”. Expressions of this kind may reference attributes, associations and state machines of a class (e.g. line 9 of Listing 8), parameters of the *emit* method (e.g. line 13 of Listing 8), and method calls. The result of the expression is substituted whenever the template method is called.

2.1.16 Code Blocks

A code block is embedded within the following tags "≪ #" and "# ≫". This corresponds to the logic of the template. Its purpose is to allow conditional emission for the parts of a template, or looping within the template. For example, line 31 of Listing 8 embeds a for-loop construct within the template "row" to compute a multiplication table.

2.1.17 Comment Block

A comment block enables developers to embed comments within the template. Anything embedded within the following block "≪/∗" and "∗/≫" are treated as comments by the Umple compiler. For instance, line 27 of Listing 8 defines a comment for the emit statement.

2.2 Alloy

Alloy [52] implements a light-weight modeling language based on first-order relational logic with the goal of expressing software abstractions, as well as simulating and checking requirements of software systems [53]. It has formal syntax and semantics, thus positioning it as a language capable of specifying, verifying and validating safety and correctness requirements of software. Requirements checking, simulations and visualizations are realizable by the aid of its SAT-based analyzer.

As mentioned earlier (see Section 1.4), Alloy is considered most suitable for the representation of class models due to its provisions of mechanisms (e.g., signatures, transitive closure, universal/existential quantifications, facts, etc.) necessary to represent domain entities, constraints and operations required to manipulate the structures dynamically. In the following, we present a grammar of the Alloy language (i.e., Listing 9) to formally describe its constituents and the notions it facilitates. The subset of Alloy relevant to our work is derivable from the grammar in Listing 9. Similarly, we have categorized its core mechanisms as signatures, constraints and commands.

2.2.1 Signatures

Alloy provides the notion of signatures that can be used to represent UML classes. A signature introduces a set of atomic objects. It is defined with the keyword *sig*. Any signature defined

independently is regarded as a top-level signature. For example, line 10 of Listing 10 is a top-level signature, equivalent to a UML class with name 'B'.

```
1   paragraph-: [[sigdecl]]|[[factdecl]]|[[predecl]]|[[fundecl]]|[[assertdecl]]
2    | [[cmdecl]]
3   sigdecl-: [abstract]? [[mult]]? sig [[varnames]] [[sigExt]]? {[[body]]?}
4    [[block]]?
5   mult-: [=type: lone|one|some]
6   varnames-: [name][,name]*
7   sigExt-: extends [name]|in [name][+ name]*
8   body-: [[vardecl]][,[[vardecl]][[expr]]
9   vardecl-: [disj]?[[varnames]] : [disj]?[[typename]]
10  typename-: [=type: String|Int|ObjectName]
11  expr-:[[const]]|@name|this|[[unop]][[expr]]
12   | [[expr]] [[binop]] [[expr]]
13   | [[expr]] [[arrowop]] [[expr]]
14   | [[expr]] [comma: ,] [[expr]]
15   | [[expr]] [!|not]? [[compareOp]] [[expr]]
16   | [[expr]] [[implication]] [[expr]]
17   | [[expr]] else [[expr]]
18   | let [[letdecl]] [,[[letdecl]]]* [[blockorbar]]
19   | [[quant]] [[decl]] [,[[decl]]]* [[blockorbar]]
20   | { [[decl]][,[[decl]]* [[blockorbar]] }
21   | ( expr ) | [[block]]
22  const-: [-]?[[number]]|none|univ|iden
23  unop-: !|not|no|[[mult]]|set|#|~|*|^
24  binop-: ||or|&&|and|<=>|iff|=>|implies|&|+|-|++|<:|:>|.
25  arrowop-: [[[mult]]|set]? -> [[[mult]]|set]?
26  compareop-: in|=|<|>|<=|>=
27  implication-: [type: =>|implies]
28  letdecl-: [name] = [[expr]]
29  block-: { [[expr]]* }
30  blockorbar-: [[block]] | |[[expr]]
31  quant-: all|no|sum|mult
32
33  -- grammar of various kinds of constraints in Alloy...
34  factdecl-: fact [name]? [[block]]
35  predecl-: pred [[[qualName]].]? [name] [[paradecls]]? [[block]]
36  qualName-: [this/]? [( name/ )]* [name]
37  paradecls-: ( [[decl]]?[,[[decl]]]* )|[ [[decl]][[,decl]] ]?
38  fundecl-: fun [[[qualname]].] [name] [[paradecls]]? : [[expr]] { [[expr]] }
39  assertdecl-: assert [name] [[block]]
40  cmdecl-: [[name]:]? [run|check]? [ [[qualname]]|[[block]] ]? [[scope]]
41  scope-: for [[number]] [ but [[typescope]][,[[typescope]]]* ]
42    | for [[typescope]][,[[typescope]]]*
43  typescope-: [exactly]? [[number]] [[qualname]]
```

LISTING 9. THE GRAMMAR OF ALLOY'S LANGUAGE

```
1   -- notion of singleton
2   one sig Organization { }
3   -- multiple inheritance
4   abstract sig Animal { } -- notion of generalization  (superclass)
5   sig Mammal extends Animal { } -- notion of generalization (subclass)
6   sig WingedAnimal extends Animal { }
7   sig Bat extends Mammal { }
8   fact { Bat in WingedAnimal }
9   -- notion of association
10  sig B { }
11  sig A { roleName : m B }
12
13  -- notion of attribute
14  sig Student { age : Int, firstName : String, takes : some Course,
15    supervisor: lone Professor, studentNumber: Int, identity: Int }
16  { all studentA, studentB: Student | studentA.identity != studentB.identity }
17  sig Course { }
18  sig Professor { students: set Student }
19  abstract sig Object { }
20  sig Directory extends Object { }
21  sig File extends Object { }
22  sig Alias extends File { }
23  assert A { … }
24  -- notion of fact
25  fact Numeric-Bounds { no student: Student | #student.takes < 1 ||
26    #student.takes > 7 }
27
28  -- notion of function
29  fun grandMothers[grandChild: Person] set Person
30  { grandChild.(mother + father).mother }
31  -- notion of predicates
32  pred coSupervision[ supA: Professor, supB: Professor, s, s', s'': Student]
33  { s'.supervisor = s.supervisor + supA && s''.supervisor = s'.supervisor + supB
34    && #s''.supervisor > 1 }
35  -- notion of assertion
36  assert UniqueIdentity { no student1, student2: Student |
37    student1.identity = student2.identity }
38
39  -- check command
40  check A
41  check A for 10
42  check A for 5 Object
43  check A for 5 but 3 Directory
44  check A for exactly 3 Directory, exactly 3 Alias, 5 File
45
46  -- run command
47  run { }
48  run { } for 10
49  run UniqueIdentity
50  run UniqueIdentity for 10
51  run UniqueIdentity for 5 Student, 1 Professor
```

LISTING 10. MODELLING EXAMPLE IN ALLOY

To facilitate the principle of inheritance, Alloy provides the '*extends*' keyword. This implies the definition of a set of objects that are a subset of the set defined by the extended signature. The example on lines 4, 5 of Listing 10 defines atomic signature Mammal (see line 5 of Listing 10) as a subclass of Animal (see line 4 of Listing 10).

Similarly, to support the UML notion of abstract object, Alloy provides the 'abstract' keyword. We illustrate the definition of an abstract signature in Alloy with the example on line 4 of Listing 10. The example enforces that no instance of signature 'Animal' is created at run-time except those extending it.

TABLE 4. MULTIPLICITY CONSTRAINTS MAPPING

ALLOY SYNTAX	MEANING OR SEMANTICS	UML NOTATION
some	Mandatory many	1..*
set	Any number	*
one	Mandatory	1
lone	Optional	0..1

Alloy provides keywords such as *lone*, *one*, *some*, *set* for the specification of multiplicity constraints. Table 4 shows the mapping between Alloy and UML multiplicity constraints.

Singleton sets can be realized by providing the notation 'one' before a signature. This restricts the number of objects that can be instantiated to 'one' at run-time. The example on line 2 of Listing 10 illustrates this notion for the 'Organization' set.

Multiple inheritance is allowed in Alloy, but it is supported in an indirect way with an additional constraint (e.g., line 8 of Listing 10). We illustrate the specification of multiple inheritance in Alloy with the example describing Bat as a WingedAnimal (see line 8 of Listing 10) and Mammal (see line 7 of Listing 10). The declaration of variables is achieved by the concepts of fields and attributes.

The notion of fields in Alloy defines relations among objects of the domain under specification. It equivalently represents the notion of attributes and associations in the UML context. Fields of a set are separated with a comma operator.

The example on lines 10, 11 of Listing 10 illustrates the use of field declarations in the realization of UML associations. It maps set "A" to a corresponding set "B" (see line 11 of Listing 10). 'm' defines the multiplicity between the collaborating sets. The roleName defines the role of set B in A. The notation 'takes: *some* Course' in line 14 of Listing 10 is another association.

The types of attributes supported by Alloy include string and integer types. String types are qualified with keyword *String* and integer types are qualified with keyword *Int*. The example on lines 14, 15 of Listing 10 demonstrates the specification of attribute types in Alloy. It defines a set of type Student. Every instance of Student has an attribute age of type *Int* and attribute "firstName" of type *String* and so on.

2.2.2 Constraints

Alloy allows the specification of constraints with the following notions: facts, assertion, predicates, and functions. According to [52], the notion of fact semantically correlates to invariant but is richer than invariant due to set navigation capabilities; functions are reusable constraints or expressions; assertions are implications to be checked on the model; and predicates represent constraints used in different contexts of correctness certification.

2.2.2.1 Facts

Constraints that are assumed to hold in all cases of execution are recommended to be placed in a fact paragraph. Alloy allows the specification of invariants to be quantified over the set of objects. This can be actualized by directly associating the paragraph of the fact to the set. A fact may be named or not. There can be any number of facts in a model. The order of occurrence of fact or the content of its paragraphs are irrelevant.

The example on lines 25, 26 of Listing 10 illustrates a named independent paragraph of fact in a modeling context. It models the relationship between Student and Course sets in the school domain. Each student must take a minimum of one course but a maximum of seven courses. The constraint is specified in the fact paragraph with a name – Numeric-Bounds. A dependent fact is a constraint

that is directly attached to the signature. The semantics implies that the constraint is associated with every instance of the signature created at run-time. An example illustrating the specification of this kind of fact is on line 16 of Listing 10. The *fact* is directly associated with signature Student (see lines 16 of Listing 10). The keyword *fact* and the name are not necessarily required. For example, lines 14-16 of Listing 10 define a signature and an associated fact. The fact is directly associated with the signature on line 16. Thus, the keyword *fact* has been omitted as well as the name.

2.2.2.2 Functions

A function houses a set of expressions for reuse purposes. Like functions in other programming languages, Alloy functions may have any number of arguments. Functions are referenced by their unique names. The body of a function defines constraints binding results of the function. An example illustrating the use of function is presented on lines 29, 30 of Listing 10.

The example (see lines 29, 30 of Listing 10) illustrates a function expression for determining a set of grandmothers of a Person. It is observable from the example that every Person is associated with a mother and a father both of type Person. The function can be referenced as grandMothers. It takes an instance of Person - grandChild as an argument. The function returns all the grandmothers of grandChild.

2.2.2.3 Predicates

Alloy predicates are named expressions with at least an argument. According to [52], their uses for model analysis include:

- to check for inclusion or exclusion of a constraint;
- to check whether a constraint is a consequence of others; or
- to define a reusable constraint in other contexts.

To use a predicate, the basic requirement is that an expression is provided for each of its arguments. It returns either TRUE or FALSE.

The example on lines 32-34 of Listing 10 illustrates the use of a predicate. We define a predicate to verify whether the model satisfies co-supervision requirement. The predicate assumed that

‘supA’ and ‘supB’ are the supervisors of ‘student’; then if that is valid, the implication is that the cardinality of supervisors of ‘student’ must be ‘two’.

2.2.2.4 Assertions

According to [52], assertions are constraints derivable from the facts of the model. The Alloy analyzer explores all the states of the model to verify conformance of the assertion. It achieves this by negating the assertion and conjoining the result with the rest of the constraints of the model. The result of the state exploration is a counterexample, whenever the negated assertion produces a model with constraints defined on the model. A counterexample implies the presence of a model error or a wrong formulation of the assertion expressions. For example, in a student management system’s model, a student should have a unique identity. We demonstrated the use of assertion to verify whether the model conforms to the requirement of uniqueness of identity.

Lines 36, 37 of Listing 10 define an assertion with the intention of verifying whether the student model satisfies the uniqueness property of students’ identities.

2.2.3 Commands

Alloy provides command statements for the analysis of an abstract model expressed in its native dialect. These include *run* and *check* commands. The run command instructs the Alloy analysis engine to search for an instance of a predicate. On the other hand, the check command instructs the analysis engine to search for a counterexample of a given assertion. To address decidability issues, each command requires the specification of scope for searching or exploration. However, in the case where the user specified no scope for exploration, Alloy defaults the exploration scope to three for all top-level signatures. Defining exact size of top-level signatures to be created for analysis requires the usage of keyword *exactly*. The following illustrate usage of commands *check* and *run* in modeling context. A good discussion on the use of these commands can be obtained from [52].

2.2.3.1 Check commands

The example on lines 40-44 of Listing 10 illustrates various ways of using the *check* command. We obtained the example from [52].

The following explains the semantics of the commands defined using the associated label and references focus on Listing 9Listing 10.

- Line 40 instructs the analyzer to check the assertion with the default of 3 instances of Object.
- On line 41, the assertion overrides the default scope to create 10 instances of Object for analysis.
- The assertion on line 42 specifically binds the scope of analysis to 5 instances of Object. This is semantically equivalent to the assertion on line 41 if and only if the scope was bound to 5 instead of 10 instances.
- The assertion on line 43 will create 5 instances of Object but 3 of the instances will be Directory; an Alias and File will be created as well.
- On line 44, the command will create 3 instances of Directory and Alias, and 5 instances of File.

2.2.3.2 Run commands

The example on lines 47-51 illustrates varieties of usage of the *run* command. This command can be used with predicates.

In the following, we explain informally the semantics of commands using associated indexes. Readers should note that references focus on Listing 10.

- The command on line 47 runs the entire model in search of possible examples using the default scope (i.e., 3 instances of every top-level signature). The top-level signatures according to the model given are: Object, Animal, Professor, Course, Student, A, and B. The run command is a mechanism for simulating models within Alloy environment. Suppose there is no example or the model cannot be instantiated; in that case the analyzer will report "no instance found".
- The command on line 48 will be semantically equivalent to the command on line 47 if and only if the scope of 10 instances has not been specified for every top-level signature.
- The command on line 49 runs the UniqueIdentity assertion with 3 instances (the default) of Person.
- The command on line 50 is semantically equivalent to line 49 except the default scope is not overridden to 10.

- The command on line 51 runs UniqueIdentity assertion with 5 instances of Student and 1 instance of Professor.

2.3 nuXmv

nuXmv, a symbolic model checker for the verification of fair finite- and infinite-state synchronous systems extends NuSMV [32], a state-of-the-art model checker for the specification and verification of finite state systems. As earlier mentioned, it adopts the basic verification techniques of NuSMV and extends its native language with unbounded *integer* and *real* data types for the specification of infinite domains. For the verification of the newly supported domains, nuXmv integrates Satisfiability Modulo Theory (SMT [46]) algorithms.

We have adopted nuXmv as a back-end engine for the analysis and simulation of correctness properties of state machines expressed in Umple based on its capabilities to represent and analyze infinite-state systems. The following section presents an overview of the nuXmv specification language.

2.3.1 The input language of nuXmv - SMV

In this section, we present the syntax and semantics of Symbolic Model Verifier (SMV), the specification language of the nuXmv model checker. We facilitate easy reading and thorough understanding by focusing on the subset of the language relevant to this thesis. As we did with Umple and Alloy, we have adapted the grammar of SMV to the notational style introduced in this chapter so readers do not have to deal with more than one grammar notation.

We assume that identifiers are well-formed and composed from the set {A-Z, a-z, 0-9, _, $, #, -}. A comprehensive knowledge of this specification language can be obtained from the nuXmv user manual [69]. We will focus our discussion on the following notions: variable declaration, assign constraint, module concepts (e.g. declarations, instantiations), and logic specifications (e.g. LTL, CTL, INVARSPEC).

```
1    MODULE HeatControllerSm (smHeatReq, smController, smControllerControllerOn,
2     smControllerControllerOnHeaterActive)
3
4     VAR
5      state : { Sm_house, null };
6      event : { heatSwitchOn, heatSwitchOff, autotransition, userReset,
7       furnaceFault, null };
8      setTemp : integer;
9      actualTemp : integer;
10     valvePos : integer;
11     waitedForCool : integer;
12     coolDownTimer : integer;
13     furnaceRunning : boolean;
14     activate : boolean;
15     deactivate : boolean;
16     requestHeat : boolean;
17     furnaceReset : boolean;
18
19    DEFINE
20     sm_stable := !(event = heatSwitchOff | event = userReset
21      | event = heatSwitchOn | event = autotransition | event = furnaceFault);
22     t1 := event = autotransition & smHeatReq.state = SmHeatReq_idleHeat & g1;
23     t2 := event = autotransition & smHeatReq.state = SmHeatReq_waitForCool &
24       g2;
25     t3 := event = autotransition & smHeatReq.state = SmHeatReq_waitForCool &
26       g3;
27     t4 := event = autotransition & _smHeatReq.state = SmHeatReq_waitForCool &
28       g4;
29     t5 := event = heatSwitchOn & smController.state = SmController_off;
30     t6 := event = heatSwitchOff & smController.state =
31       SmController_controllerOn;
32     t7 := event = furnaceFault &
33       smController.state = SmController_controllerOn;
34     t8 := event = userReset & smController.state = SmController_error;
35     t9 := event = autotransition &
36       smControllerControllerOn.state = SmControllerControllerOn_idle & g5;
37     t10 := event = autotransition &
38       smControllerControllerOn.state = SmControllerControllerOn_heaterActive &
39       g6;
40     t11 := event = autotransition & smControllerControllerOnHeaterActive.state
41       = SmControllerControllerOnHeaterActive_actHeater & g7;
42     g1 := (actualTemp - setTemp) > 2;
43     g2 := (valvePos != 0) & (coolDownTimer = waitedForCool);
44     g3 := waitedForCool < coolDownTimer;
45     g4 := ((actualTemp - setTemp) <= 2);
46     g5 := requestHeat = TRUE;
47     g6 := requestHeat = FALSE;
48     g7 := furnaceRunning = TRUE;
49
50    ASSIGN
51     init( state ) := Sm_house;
52     next( state ) := case
53      t1 | t3 | t8 | t10 | t11 | t4 | t2 | t6 | t5 | t9 | t7 : Sm_house;
54      TRUE : state;
55     esac;
56     init( event ) := null;
57     next( event ) := case
58      sm_stable : { heatSwitchOn, heatSwitchOff, autotransition, userReset,
59        furnaceFault };
```

```
60    TRUE : null;
61   esac;
62   init( setTemp ) := 0;
63   init( actualTemp ) := 0;
64   init( valvePos ) := 0;
65   init( waitedForCool ) := 0;
66   init( coolDownTimer ) := 0;
67   init( furnaceRunning ) := FALSE;
68   init( activate ) := FALSE;
69   init( deactivate ) := FALSE;
70   init( requestHeat ) := FALSE;
71   init( furnaceReset ) := FALSE;
72
73 MODULE HeatControllerSmHeatReq ( sm )
74
75  VAR
76   state : { SmHeatReq_idleHeat, SmHeatReq_waitForCool, null };
77
78  ASSIGN
79   init( state ) := null;
80   next( state ) := case
81    sm.t4 : SmHeatReq_idleHeat;
82    sm.t2 | sm.t1 | sm.t3 : SmHeatReq_waitForCool;
83    sm.state = Sm_house & state = null : SmHeatReq_idleHeat;
84    TRUE : state;
85   esac;
86
87 MODULE HeatControllerSmController ( sm )
88
89  VAR
90   state : { SmController_off, SmController_controllerOn, SmController_error,
91     null };
92
93  ASSIGN
94   init( state ) := null;
95   next( state ) := case
96    sm.t6 | sm.t8 : SmController_off;
97    sm.t5 | sm.t9 | sm.t10 | sm.t11 : SmController_controllerOn;
98    sm.t7 : SmController_error;
99    sm.state = Sm_house & state = null : SmController_off;
100   TRUE : state;
101  esac;
102
103 MODULE HeatControllerSmControllerControllerOn ( sm , smController )
104
105  VAR
106   state : { SmControllerControllerOn_idle,
107     SmControllerControllerOn_heaterActive, null };
108
109  ASSIGN
110   init( state ) := null;
111   next( state ) := case
112    sm.t6 | sm.t8 | sm.t5 | sm.t7 | sm.t9 : null;
113    sm.t10 : SmControllerControllerOn_idle;
114    sm.t9 | sm.t11 : SmControllerControllerOn_heaterActive;
115    smController.state = SmController_controllerOn & state = null :
116      SmControllerControllerOn_idle;
117    TRUE : state;
118   esac;
119
120 MODULE HeatControllerSmControllerControllerOnHeaterActive ( sm,
121  smControllerControllerOn )
```

```
122   VAR
123    state : { SmControllerControllerOnHeaterActive_actHeater,
124     SmControllerControllerOnHeaterActive_heaterRun, null };
125
126   ASSIGN
127    init( state ) := null;
128    next( state ) := case
129     sm.t9 | sm.t10 : null;
130     sm.t11 : SmControllerControllerOnHeaterActive_heaterRun;
131     smControllerControllerOn.state = SmControllerControllerOn_heaterActive &
132      state = null : SmControllerControllerOnHeaterActive_actHeater;
133     TRUE : state;
134    esac;
135
136  MODULE HeatControllerSm_Machine
137   VAR
138    hcsm : HeatControllerSm(hcsmHeatReq, hcsmController,
139     hcsmControllerControllerOn, hcsmControllerControllerOnHeaterActive);
140    hcsmHeatReq : HeatControllerSmHeatReq(hcsm);
141    hcsmController : HeatControllerSmController(hcsm);
142    hcsmControllerControllerOn : HeatControllerSmControllerControllerOn(hcsm,
143     hcsmController);
144    hcsmControllerControllerOnHeaterActive :
145     HeatControllerSmControllerControllerOnHeaterActive(hcsm,
146     hcsmControllerControllerOn );
147
148  -- The following properties are specified to certify that this model is free of non-determinism.
149   INVARSPEC( hcsm_Machine.hcsm.t2 & hcsm_Machine.hcsm.t4
150    -> next( hcsm_Machine.hcsmHeatReq.state = SmHeatReq_waitForCool &
151    hcsm_Machine.hcsmHeatReq.state = SmHeatReq_idleHeat ) )
152  ...
153
154  -- this defines the initial configuration of the SUA
155   LTLSPEC ( hcsm_Machine.hcsm.state = Sm_house
156    & hcsm_Machine.hcsmHeatReq.state = null
157    & hcsm_Machine.hcsmController.state = null
158    & hcsm_Machine.hcsmControllerControllerOn.state = null
159    & hcsm_Machine.hcsmControllerControllerOnHeaterActive.state = null )
160    SmControllerControllerOnHeaterActive_heaterRun )
161  -- The following properties are specified to certify that non-symbolic state(s) of this model is (or are) reachable.
162   CTLSPEC EF( hcsm_Machine.hcsmHeatReq.state = SmHeatReq_idleHeat )
162   CTLSPEC EF( hcsm_Machine.hcsmHeatReq.state = SmHeatReq_waitForCool )
163   CTLSPEC EF( hcsm_Machine.hcsmController.state = SmController_off )
164   CTLSPEC EF( hcsm_Machine.hcsmController.state = SmController_controllerOn )
165   CTLSPEC EF( hcsm_Machine.hcsmController.state = SmController_error )
166   SPEC EF( hcsm_Machine.hcsmControllerControllerOn.state =
167    SmControllerControllerOn_idle )
168   CTLSPEC EF( hcsm_Machine.hcsmControllerControllerOn.state =
169    SmControllerControllerOn_heaterActive )
170   CTLSPEC EF( hcsm_Machine.hcsmControllerControllerOnHeaterActive.state =
171    SmControllerControllerOnHeaterActive_actHeater )
172   CTLSPEC EF( hcsm_Machine.hcsmControllerControllerOnHeaterActive.state =
173    SmControllerControllerOnHeaterActive_heaterRun )
```

LISTING 11. HEAT CONTROLLER STATE TRANSITION SYSTEM

Listing 11 is an example SMV program with the purpose of facilitating readers' understanding of notions supported by the language and those relevant to our work.

The program defines a transition system for the heating system presented in Listing 4. The system contains several modules because the heating system is hierarchical. These are modules corresponding to the root state machine and sub-state machines of composite states, and the main module. However, the simplest form of an SMV program contains at least a main module. Listing 12 is a grammar for defining SMV program specifically adapted to our work.

2.3.1.1 Module Concepts

In this section, we will discuss module declaration and instantiation. Module declarations specify a transition system in SMV language; while module instantiations describe how instances of modules are created. An example of a non-main but comprehensive module is presented on lines 1-71 of Listing 11. By "non-main" module, we mean every other module in the program except the "main" module. A "main" module (see lines 136-146 of Listing 11) is an entry point of execution just like programming languages in C-family. The module represents most concepts relevant to our work. Module instantiation inherits the properties of object declaration in object-oriented systems. For example, lines 138, 139 of Listing 11 defines "hcsm" as an instance of "HeatControllerSm". It represents a parametrized instantiation of a module.

Other notions facilitated by the referenced module include: variable and define declarations, and assign constraints. Variable declarations are the SMV constructs for declaring properties in transition systems. These are synonymous with field declarations in conventional object-oriented programming languages. Variable declarations are defined within a paragraph preceded by the "*VAR*" keyword. For example, lines 4-17 of Listing 11 is a VAR paragraph with various declarations of names and types of variables local to the module. The grammar equivalent to this definition is presented on line 7 of Listing 12.

For the purpose of modularity and conciseness, a "define" declaration paragraph may be associated with a common expression. For example, lines 19-48 of Listing 11 is an example of such a declaration. Any variable declared in this paragraph can be seen as a macro. In essence, identifiers in this paragraph do not contribute to the state space of the system under analysis (SSUA). A macro binds the identifier with the expression type and value on its right-hand side. The paragraph must

be preceded with the keyword "*DEFINE*". The grammar corresponding to this definition is presented on line 11 of Listing 12.

```
1   smv-program -: [ [[module]] ]* MODULE main [[moduleBody]]
2   module -: MODULE [name] [( [[moduleParameters]] )]? [[moduleBody]]
3   moduleParameters -: [name] [, [name] ]*
4   moduleBody -: [[moduleBodyPart]]+
5   moduleBodyPart -: [[varDeclaration]] | [[defineDeclaration]]
6    [[assignConstraint]] | [[specification]] …
7   varDeclaration -: VAR [[declarationStatements]]
8   declarationStatements -: [name] : [[variableTypes]];
9   variableTypes -: [=type: integer | real | boolean] | [[enumeration]]
10  enumeration -: { val-1 [, val-n]* }
11  defineDeclaration -: DEFINE [[defineBlock]]
12  defineBlock -: [ [name] := [[expression]] ]+
13  expression -: [constant] | [identifier]
14   | [[expression]] [[binop]] [[expression]] | [[unop]] [[expression]]
15   | ( [[expression]] ) | [[case-expression]] | [=type: TRUE | FALSE]
16   | next( [[expression]] )
17  binop -: [=type: & | | | xor | xnor | -> | <-> | = | != | < | > | <=
18   | >= | + | - | * | / | mod]
19  unop -:[=type: ! | -]
20  case-expression -: case [[case-body]] esac;
21  case-body -: [[expression]] : [[expression]]
22   | [[case-body]] [[expression]] : [[expression]]
23  assignConstraint -: ASSIGN [[assignBody]]
24  assignBody -: [ [[initStmt]] [ [[nextStmt]] ]? ]+
25  initStmt -: init( identifier ) := [[value]]
26  nextStmt -: next( identifier ) := [[expression]]
27  value -: [identifier] | [integerNumber] | [realNumber]
28  specification -: [[ltl-specification]] | [[ctl-specification]]
29   | INVARSPEC [[expression]]
30  ltl-specification -: LTLSPEC [[ltl-expression]]
31  ltl-expression -: [[expression]] | [[ltlOperator]] [[ltl-expression]]
32   | [[ltl-expression]] [[specialBinaryOperator]] [[ltl-expression]]
33  ltlOperator -: [= G | X | F | Y | Z | H | O]
34  specialBinaryOperator -: [= U | V | S | T]
35  ctl-specification -: CTLSPEC [[ctl-expression]] | SPEC [[ctl-expression]]
36  ctl-expression -: [[ctlOperator]] [[expression]]
37   | [[ctlOperator]] [[ctl-expression]]
38   | [[existentialOperator]] [ [[ctl-expression]] U [[ctl-expression]] ]
39  ctlOperator -: [= AG | AX | AF | EG | EX | EF]
40  existentialOperator -: [= A | E]
```

LISTING 12. MODULE'S GRAMMAR IN NUXMV

SMV allows assignment of values to state variables within an assign paragraph. In an assign paragraph (or constraint), a variable is assigned initial and next values with the keywords "*init*" (see line 51 of Listing 11) and "*next*" (see lines 80-85 of Listing 11) respectively. The *next* statement embodies a "*case … esac*" statement which models the possible conditions (on the left-hand side) and values (on the right-hand side) to be assigned to the variable at any given step of execution of the program. In this paragraph, at least a variable must be initialized or its next values specified.

2.3.1.2 Logic Specification

The notion of logic specification allows developers to define requirements for the purpose of analysis via the analysis engine. The nuXmv analysis engine accepts both linear (i.e., LTL [38]) and branching (i.e., CTL [37]) time logics for the expression of system requirements. While LTL specifications quantify over paths, CTL specifications quantity over the global state space of the SUD. The benefit of dual support is to allow user the expressive power of each logic specification. Besides these logics, nuXmv also facilitates the representation of invariance.

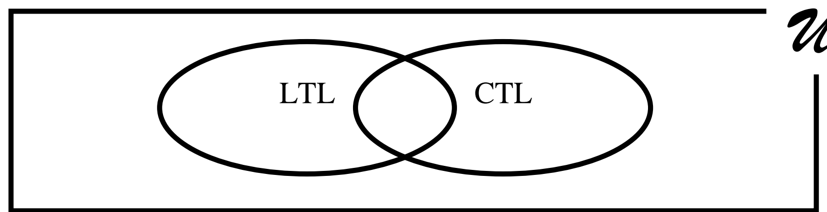


FIGURE 4. RELATIONSHIP BETWEEN EXPRESSIVE POWERS OF CTL AND LTL

Figure 4 presents the relationship between expressive powers of LTL and CTL. The implication of the figure is to demonstrate that some requirements are only expressible in either of the logics; while some requirements can be expressed in both logics.

A CTL statement begins with either keyword '*SPEC*' (see line 168, 169 of Listing 11) or '*CTLSPEC*' (see line 167 of Listing 11). An LTL statement begins with the keyword '*LTLSPEC*' (see lines 156-160 of Listing 11). To constrain the entire model to fulfil some special properties, SMV provides "*INVARSPEC*" (see lines 156-160 of Listing 11) keyword. We introduce the grammar to define these specification statements on line 29 of Listing 12.

A detailed discussion of the semantics of CTL (e.g. *EF*, *AF*, *AG*, *EG*, *A[… U …]*, *E[… U …]*) and LTL (e.g. `G`, `X`, `F`, `Y`, `Z`, `H`, `O`) operators as obtainable from [5], [69] are presented in Table 5, Table 6 respectively. p and q are considered as formulas whose syntax conforms to the grammar described in Listing 12. A CTL formula evaluates to true whenever it is true in all initial states. An LTL formula evaluates to true whenever it is true at the initial time t_0.

TABLE 5. SEMANTICS OF LTL OPERATORS FOR NUXMV [69]

Notation	Semantics
X *p*	Given times t, t', then X *p* = *true* at time t if and only if *p* is *true* at time $t' = t + 1$.
F *p*	Given times t, t', then F *p* = *true* at time t if and only if *p* is true at some time $t' \geq t$.
G *p*	Given times t, t', then G *p* = *true* at time t if and only if *p* is true at times $t' \geq t$.
Y *p*	Given times t, t', t_0, then Y *p* = *true* at time $t > t_0$, if and only if *p* is true at time $t' = t - 1$ but *p* is false at time t_0.
Z *p*	Similar to Y *p* but *p* must be true at time t_0.
H *p*	Given times t, t', then H *p* = *true* at time t if and only if *p* is *true* at all previous time steps $t' \leq t$.
O *p*	Given times t, t', then O *p* = *true* at time t if and only if *p* is *true* in some previous time steps $t' \leq t$.
p U *q*	Given times t, t', t'', then *p* U *q* = *true* at time t if and only if *q* is true at some time $t' \geq t$ and *p* is true for all time t'', such that: $t \leq t'' < t'$.
p V *q*	Given times t, t', t'', then *p* V *q* = *true* at time t if and only if *q* is true at all time steps $t' \geq t$ up to and including time step t'' where *p* is also true. Alternatively, *p* may never be true but *q* must be true in all time steps $t' \geq t$.
p S *q*	Given times t, t', t'', then *p* S *q* = *true* at time t if and only if *q* is true at time $t' \leq t$ and *p* is true at all time steps t'', such that: $t' < t'' \leq t$.
p T *q*	Given times t, t', t'', t_0, then *p* T *q* = *true* at time t if and only if *p* is true at $t' \leq t$ and *q* is true at all time steps t'', such that: $t' \leq t'' \leq t$. Alternatively, if *p* has never been true, then *q* must be true in all time steps t'', such that: $t_0 \leq t'' \leq t$.

TABLE 6. SEMANTICS OF CTL OPERATORS IN NUXMV [69]

Notation	Semantics
EX p	Given states s, s' such that: s -> s' then EX p = *true* in s if and only if p is *true* in s'.
AX p	Given states s, s' such that: s -> s' then AX p = *true* in s if and only if p is true in $x \; \forall \; x \in s'$.
EF p	Given states $s_0, s_1, s_2, \dots, s_{n-1}, s_n$ and *there exists* $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \dots, s_{n-1} \rightarrow s_n$ then EF p = *true* in s_0 if and only if p is true in s_n.
AF p	Given states $s_0, s_1, s_2, \dots, s_{n-1}, s_n$ and *for all* $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \dots, s_{n-1} \rightarrow s_n$ then AF p = *true* in s_0 if and only if p is true in s_n.
EG p	Given states s_0, s_1, s_2 and *there exists* $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \dots$ then EG p = *true* if and only if p is true in s_i.
AG p	Given states s_0, s_1, s_2 and *for all* $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \dots$ then EG p = *true* if and only if p is true in s_i.
E[p U q]	Given states $s_0, s_1, s_2, \dots, s_{n-1}, s_n$ and *there exists* $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \dots, s_{n-1} \rightarrow s_n$ then E[p U q] = *true* if and only if p is true in all states $s_0, s_1, s_2, \dots, s_{n-1}$ and q is true in s_n.
A[p U q]	Given states $s_0, s_1, s_2, \dots, s_{n-1}, s_n$ and *for all* $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \dots, s_{n-1} \rightarrow s_n$ then *A*[p *U* q] = *true* if and only if p is true in all states $s_0, s_1, s_2, \dots, s_{n-1}$ and q is true in s_n.

2.4 Back-end Analysis Tools

In the solution space, varieties of formal analysis tools exist. These range from model checking to theorem proving technologies. We will limit our survey to tools that requires no user-guidance in the search for solution. In this section, we will summarize these technologies using some parameters such as the underlying technology (e.g. SAT [39], BDDs [40], AIGs [42], SMT [46]); specification logic supported (e.g. CTL, LTL); domain of analysis supported (e.g. infinite or finite); open-source or not; orientation (state-based or event-based).

Our analysis is presented in Table 7; this indicates that nuXmv [45] and SAL [70] are the only tools with capabilities of analyzing unbounded infinite types (e.g. integer and floating-point numbers). Although SAL is open-source, it is not being actively developed. On the other hand, while other tools surveyed facilitate the analysis dynamic aspects, Alloy is the only tool surveyed that facilitates the analysis of static aspects.

TABLE 7. SUMMARY OF ATTRIBUTES OF ANALYSIS ENGINES

Analysis Engine	Open Source	Orientation		Verification Domain		Spec. Language			Software Aspects Supported		Underlying Technology			
		State	Event	Finite	∞	LTL	CTL	FOL	Static	Dynamic	SAT	BDD	AIG	SMT
nuXmv [45]		+		+	+	+	+			+++	+	+		+
SAL [70]	+	+		+	+	+	+			+++	+	+		+
NuSMV [32]	+	+		+		+	+			+++	+	+		
ABC [41]	+	+	+	+		+				+++	+	+	+	
Magic [42]	+	+	+	+		+				+++	+	+	+	
Spin [149]	+	+		+		+				+++				
Bebop [154]		+		+						+++		+		
Moped [185]	+	+		+		+				+++		+		
Zing [186]		+	+	+		+				+++				
Alloy [52]	+	+	+	+				+	+	++	+			
FDR [187]		+	+	+						+++				

Verification domain: ∞ = unbounded infinite types; Specification language: LTL, CTL, FOL = Linear Temporal Logic, Computational Tree Logic, First-Order Logic; Underlying Technology: SAT, BDD, AIG, SMT = Boolean Satisfiability, Binary Decision Diagrams, And-Inverter Gate, Satisfiability Modulo Theory; Other Notations: +, ++, +++ = Supported, Partly Supported, Fully Supported.

2.5 Model Transformations

In this section, we present some background information on model transformations. This will enable us to discuss in detail and position our work with related concepts. In the literature a large amount of work has been done on this topic; we will adopt definitions from Kleppe et al. [71] because most literature (e.g. [72]–[74]) relies on these definitions.

> ***Definition 2 - Transformation***
>
> *"A transformation is an automatic generation of a target model from a source model, according to a transformation definition."* [75]
>
> ***Definition 3 – Transformation Definition***
>
> *"A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into one or more constructs in the target language".* [75]
>
> ***Definition 4 – Transformation Rule***
>
> *"A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."* [75]

According to the report of the working group of the Dagstuhl Seminar on Language engineering for Model-Driven Software Development (i.e. Mens et al. [76]) five questions are key to the discussion of model transformation. These include:

> ***Q1.*** *What needs to be transformed into what?*
>
> ***Q2.*** *What are the important characteristics of model transformation?*
>
> ***Q3.*** *What are the success criteria for a transformation language or tool?*
>
> ***Q4***. *What are the quality requirements for a transformation language or tool?*
>
> ***Q5.*** *Which mechanisms can be used for model transformation?*

We will focus on (Q1), (Q2) and (Q5) because our work is not a transformation tool but it applies model transformation for the purpose of code generation. In addition, our discussion will be based on Mens et al. [76].

2.5.1 Source and Target Artifacts (Q1)

This question concerns the artifacts for transformation. For example, the source and target programs or models. If the artifacts are source code, bytecode or machine code, they are referred to as programs and the process is termed *program transformation*. On the other hand, if the artifacts are models (e.g. state, class, composite structure, or activity diagram), the process is termed *model transformation*. A hybrid of these elements is possible such as model-to-text (M2T) or text-to-model (T2M). Consequently, program transformation may be considered a subset of model transformation because a model may range from abstract analysis models to concrete models of source code.

Model transformation involves the expression of models in some modeling language such as UML diagrams or programming languages for source code. The syntax and semantics of a modeling language (e.g. Umple, Alloy, SMV) is expressed with a metamodel. Transformation processes can be categorized into exogenous or endogenous based on the language for expressing the source and target.

> ***Definition 5 – Endogenous Transformation***
>
> *"Endogenous transformations are transformations between models expressed in the same language"* [75]. *In particular, the source and target models conforms to a meta-model. For example, a model transformation between UML state diagrams and UML activity diagrams is regarded as endogenous transformation since both must conform to the UML meta-model at large.*
>
> ***Definition 6 – Exogenous Transformation***
>
> *"Exogenous transformations are transformations between models expressed using different languages."* [75] *In particular, the source and target models conform to different meta-models. For example, a model transformation from Umple to Alloy is regarded as an exogenous transformation because Umple's meta-model differs from Alloy's meta-model.*

Transformations may also be categorized based on the abstraction levels of source and target models. These are called *vertical* and *horizontal* transformations.

Definition 7 - Vertical Transformation

"A vertical transformation is a transformation where the source and target models reside at different abstraction level." [75] *A different abstraction level may be model or source code. The model is considered more abstract than source code. The transformation at a parsing phase (i.e. T2M) such that a program text is transformed to an abstract syntax tree (AST) is a typical example of vertical transformation.*

Definition 8 – Horizontal Transformation

"A horizontal transformation is a transformation where source and target models reside at the same abstraction level." [75] *For example, the overall process of transforming Umple's source code to Alloy's source code is horizontal. Similarly, the transformation between Umple Internal Representation (UIR) to Alloy Internal Representation (AIR) is horizontal.*

2.5.2 Characteristics of Model Transformations (Q2)

Mens et al. [76] identified *level of automation*, *complexity of transformation, and preservation* as key properties of model transformations. A clear distinction must be made on what can and should be automated or executed manually. In particular, what demands user-intervention? For example, a transformation between requirements documents and analysis models demands user-intervention – most importantly to resolve ambiguities, incompleteness and inconsistencies in the requirements.

Transformations such as *refactoring* (e.g. model or code level) may be considered minor transformations but transformations involving parsers, compilers, and code generators are major transformations.

The question of what must be preserved is important for model transformations. In particular, what aspects (e.g. structure or behavior) must be preserved such that these elements of the input model are unchanged in the output model. *Refactoring* and *restructuring* demand behavioral preservation while the structure may be modified. On the other hand, *refinement* demands semantics preservation.

2.5.3 Mechanisms for Model Transformation (Q5)

Mechanisms for transformation are not limited to techniques, languages, and methods adopted for development to actualize the transformation process. A programming paradigm like procedural, object-oriented, functional or logic-based approaches or a hybrid may be applied to specify transformations.

These mechanisms can be categorized into *declarative* and *operational* approaches. Declarative approaches define relationships between the source and target models. In particular, they map elements of a source model to that of a target model. They are centered on "*what"* should be transformed and what it should be transformed to. Declarative approaches offer particular services like source model traversal, traceability management and automatic bi-directionality [73] via the underlying reasoning engine. Thus, they tend to be more attractive to software engineers. Examples include functional programming and logic programming [76].

Operational approaches define *steps* for executing transformation process from source to target models. They are centred on "*how"* the transformation is executed. They are most suitable for transformations that incrementally update a model. This is achievable by its built-in support for sequence, selection and iteration and an operational approach that are most beneficial when it is necessary to control the order of applying a set of transformations [76].

2.6 Summary of Background

In this chapter, we have presented background on the technologies involved in this thesis. The technologies discussed were Umple, Alloy, and nuXmv.

Umple, an MDE technology designed to be highly usable in both textual and visible form provides constructs for the representation of class models and state machine models. It is designed for integration with programming languages and code generation. We discussed the various notions of class and state machine models that are relevant to this thesis by giving some examples (textually and diagrammatically). Techniques discussed under class model representation include: class diagrams, associations, attribute, specializations, and keys. The notions discussed under state machines of Umple include: transitions, states, and state machines themselves.

Alloy is a specification technology for representing and analyzing structural models of software systems. We presented a detailed discussion of Alloy, covering the following notions: signature, field, constraint, and commands. Our discussion included textual examples for each notion. The notion of fields focuses on how to specify associations and attributes. We discussed the notion of constraints based on specifications of facts, functions, predicates, and assertions. The notion of commands was discussed; in particular, the *check* and *run* concepts.

nuXmv is a model checking tool for the analysis of state transition systems. We began our discussion on nuXmv with an example to illustrate notions supported by the tool. The notions discussed include: variable declaration, assign constraint, trans constraint, module concepts, and logic specification. We discussed each notion with its grammar and semantics were discussed succinctly. Discussions of trans constraints involve how transitions and states are represented in nuXmv. We discussed module concepts based on how a module can be designed and instantiated. We presented a detailed discussion on logic specification with nuXmv. Our discussion on logic specification focuses on linear time logic (LTL) and computational tree logic (CTL). Finally, we presented a table comparing underlying concepts of model checking tools surveyed in the literature.

3 Transformation Engineering

In this chapter, we give an overview of our solution to automatically generate formal specifications of models (state machine and class models) expressed in Umple. We will present an architecture we have adopted for the realization of our goal.

Similarly, we will present partial metamodels of Umple, Alloy and SMV to facilitate the process of model transformations and relevant architectures to illustrate specific transformations. Readers should note that the metamodels presented follow the semantics of UML class modeling. To simplify the representation, we exclude algorithmic logic and templates from the Umple code.

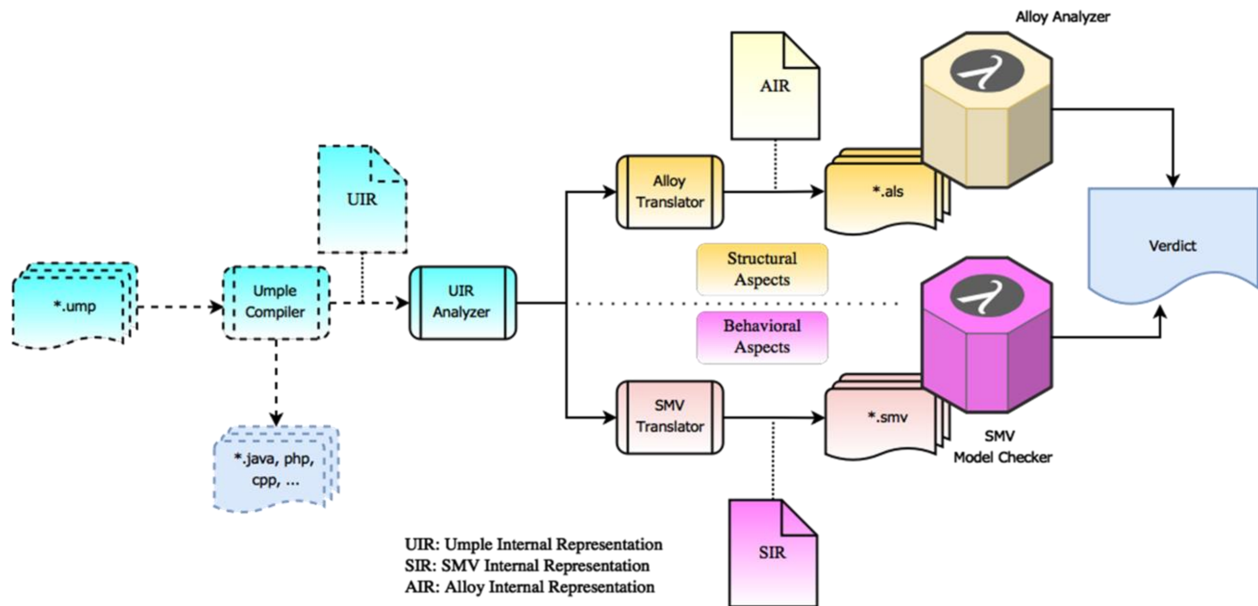


FIGURE 5. TRANSFORMATION ENGINEERING

3.1 Overview of the Transformation Architecture

The overall process of translating models expressed in Umple to target formal languages and analyzing them for correctness according to our approach is presented in Figure 5. To facilitate detailed discussion, we have divided the overall process into five (5) phases.

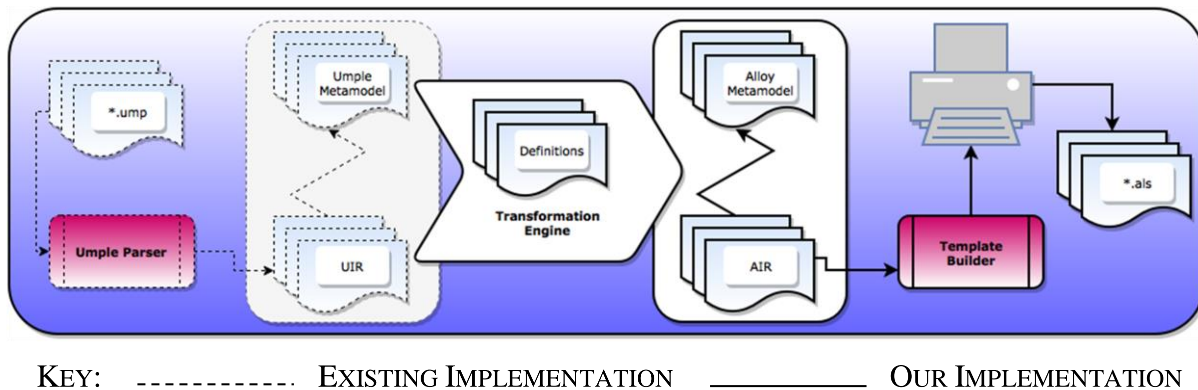
- **Phase - 1** involves writing models in code or diagrammatically (following the model-code duality principle) or importing existing models into an Umple development environment. The user needs to be familiar with Umple syntax and semantics. The artefact produced at this phase is a set of *.ump files representing the system.

- **Phase – 2** involves compilation of the *.ump file prouced in the first phase. At this stage, the model is parsed and analyzed to produce an Umple internal representation (UIR) of the model under design (MUD). The parsing phase tokenizes the model and forwards the resulting token set to the analyzer. The analyzer processes the tokens generated by the parser and converts them to an internal representation which conforms to Umple's meta-model. A detailed discussion and architecture of compilation in Umple can be obtained from [55], [63], [77]. During compilation, the Umple compiler detects many types of syntactic and semantic errors, reporting them in a manner identical to how a programming language compiler works. It should be noted that the compilation phase, with error reporting, occurs immediately (i.e., within 3 seconds of the user ceasing to type or draw) when the user is building an Umple model with UmpleOnline. Therefore, Phases 1 and 2 occur iteratively, and from the user's perspective largely concurrently. in this context.

- **Phase – 3** involves the selection of the code generator by the user (or analyst). The user first selects the type of code to be generated, which may be Alloy or SMV. If the user intends to analyze class models, the Alloy option must be selected for code generation. Otherwise, the SMV option must be selected for code generation. The user also has an option of issuing similar directives as command-line arguments or textually in code. However, if the UIR does not facilitate the generation of the chosen option, appropriate warnings are generated for the user. At this phase, the underlying translator is invoked base on the type of analysis selected and concerns are separated.

- **Phase - 4** involves the translation of modeling components into the target language; i.e. taking the elements from Phase 2 and transforming them using the translator selected in Phase 3. The meta-model of the target language is instantiated, then the resulting instance is written into a file depending on the aspects (e.g., static and dynamic) of interest. For modeling components like classes, attributes (but not state machines), and associations (i.e., modeling constructs related to static aspects of systems), the Alloy meta-model is instantiated. We collectively refer to these modeling components as class models. The corresponding Alloy model is written in an *.als file. On the other hand, the SMV meta-model is invoked whenever the modeling components are state machines and its constituents (i.e., dynamic aspects of the system). The resulting SMV model in that case will be written in an *.smv file. The details of each translator will be discussed shortly in Chapters 4 and 7.

- **Phase – 5** involves the analysis of the systems generated from Phase – 4. At this stage, the specification of the system is already generated but it is the responsibility of the user to specify the properties of the system to be verified. The properties can be expressed in LTL or CTL notations and added to *.smv files for state machine analysis. On the other hand, the properties can be expressed in FOL notations and added to *.als files for class models analysis. The resulting system is fed into the back-end analyzer and a verdict is generated. A *counterexample* is generated as a verdict whenever the model does not conform to the property verified. But whenever the model conforms to the property verified, the verdict is *true*.



KEY: - - - - - - - - - - - - EXISTING IMPLEMENTATION ———————— OUR IMPLEMENTATION

FIGURE 6. THE PIPELINE MODEL OF ALLOY TRANSLATOR

3.2 Alloy Translator

We present the discussion of our translator for the purpose of generating Alloy specifications. Figure 6 is the model we designed to specifically explain the processes involved (or internal components) in translating class models in *.ump file(s) to *.als file(s). In particular, an *.als file is a *textual representation* of an Alloy internal representation (AIR) resulting from the transformations. The overall process is a *model transformation*. To realize the target artifact, the pipeline transformation combines *model-to-model* (M2M) and *model-to-text* (M2T) transformations such that a corresponding Alloy code (i.e., textual representation) of the input class models (i.e., Umple Internal Representation) is produced as an end product. In the same vein, we have created metamodels for both languages. The details of the transformations are presented in Chapters 4 and 7 for class models and state machines respectively.

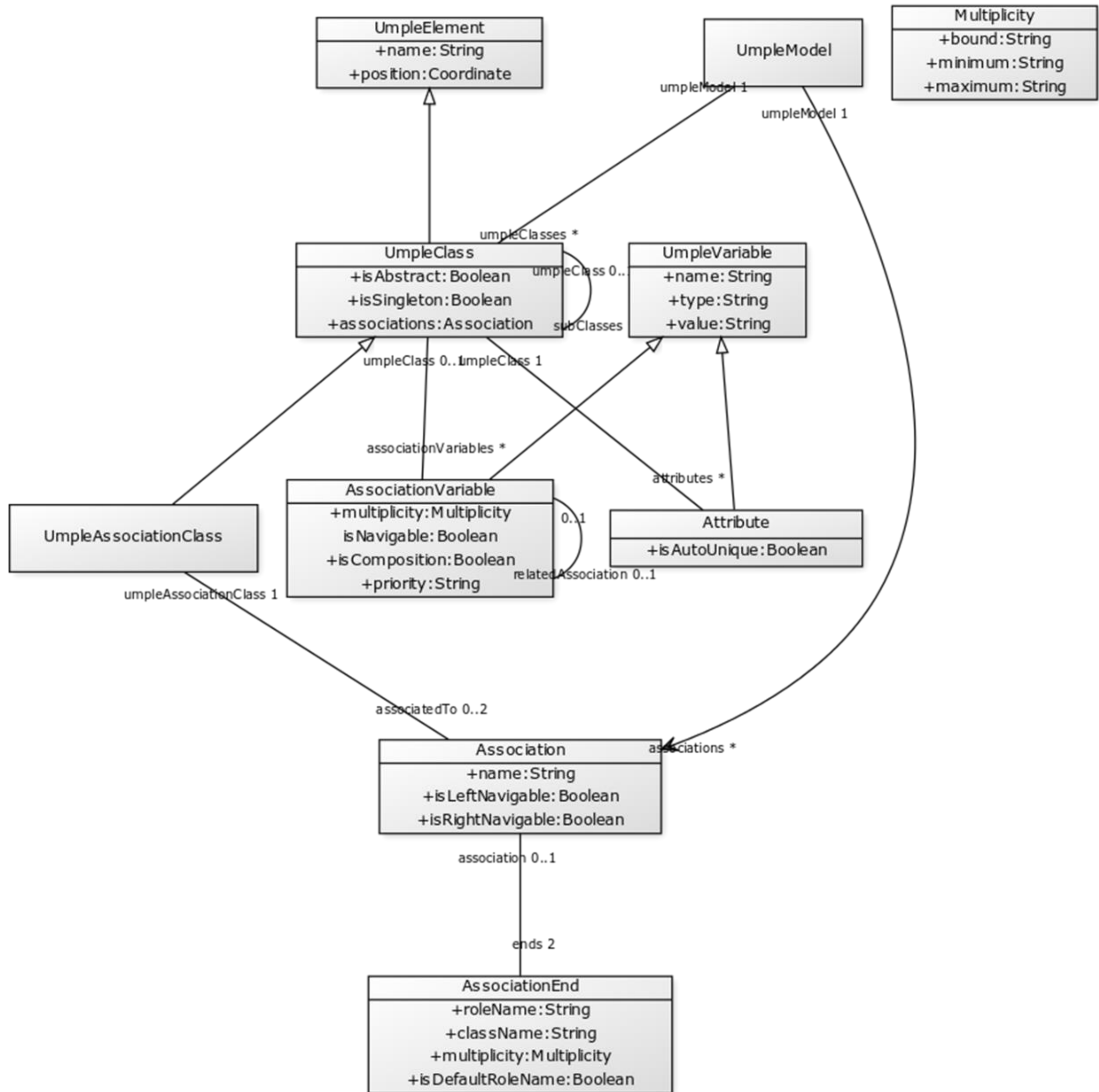


FIGURE 7. PARTIAL METAMODEL OF UMPLE FOR CLASS MODELS

Figure 7 is a *partial* metamodel of Umple extracted for the generation of Alloy specifications for class models as required by our work. It describes the elements we considered relevant to our implementation and the relationship among them. The corresponding textual representation of the model presented in Figure 7 is presented in Listing 13. A *complete* visual representation of the metamodel can be obtained from [78].

```
1   class UmpleModel  {
2     1 -<@> * UmpleClass;
3     1 -> * Association;
4   }
5   class UmpleElement {
6     name;
7     Coordinate position;
8   }
9   class UmpleClass {
10    isA UmpleElement;
11    0..1 -- * UmpleClass subClasses;
12    1 -- * Attribute;
13    0..1 -- * AssociationVariable;
14
15    //attribute definitions
16    Boolean isAbstract;
17    Boolean isSingleton;
18    Association [] associations;
19  }
20
21  class Association  {
22    0..1 -- 2 AssociationEnd ends;
23
24    //attribute definitions
25    name;
26    Boolean isLeftNavigable;
27    Boolean isRightNavigable;
28  }
29
30  class UmpleAssociationClass
31  {
32    isA UmpleClass;
33    1 -- 0..2 Association associatedTo;
34  }
35
36  class AssociationEnd
37  {
38    //attribute definitions
39    roleName;
40    className;
41    Multiplicity multiplicity;
42    Boolean isDefaultRoleName;
43  }
50  class UmpleVariable
51  {
52    //attribute definitions
53    name;
54    type;
55    value;
56  }
57
58  class Attribute
59  {
60    isA UmpleVariable;
61    Boolean isAutoUnique;
62  }
63
64  class AssociationVariable
65  {
66    isA UmpleVariable;
67    Multiplicity multiplicity;
68    immutable Boolean isNavigable;
69    Boolean isComposition = false;
70    String priority = "";
71
72    0..1 self relatedAssociation;
73  }
74
75  class Multiplicity
76  {
77    //used when minimum=maximum;
78    lazy bound;
79    lazy minimum;
80    lazy maximum;
81
82    key { bound, minimum, maximum }
83  }
```

LISTING 13. TEXTUAL REPRESENTATION OF THE PARTIAL METAMODEL OF UMPLE.

The *UIR of class models*, conforming to the metamodel of Umple (i.e., Figure 7 and Listing 13) are transformed based on its modeling elements. As noted in the architecture (see **Section 3.1**), these elements include associations, attributes, classes, and inheritance hierarchy. We rely on the *syntactic checker* developed for Umple to guarantee that the UIR is free of syntactic errors.

```
1    class AlloyObject {
2     abstract;
3    }
4    class Function {
5     beginEndRoleName;
6     targetMult;
7     targetClassName;
8    }
9    class AlloyObject { }
10   class Fact {
11    isA AlloyObject;
12
13    factName;
14    firstClassName;
15    secondClassName;
16    rName1;
17    rName2;
18   }
19   class NoExtendedFact {
20    isA Fact;
21   }
22
23   class NoSelfRelationFact {
24    isA Fact;
25   }
26
27   class BidirectionFact {
28    isA Fact;
29   }
30
31   class GenHierarchyFact {
32    isA Fact;
33   }
34
35   class AssociationFact {
36    isA Fact;
37    fMult;
38    sMult;
39   }
44   class Statement {
45    name;
46   }
47   class OpenStatement {
48    isA Statement;
49    packageName;
50    className;
51    after constructor {
52     name = "open";
53    }
54   }
55   class AlloyModel {
56    isA AlloyObject;
57
56    1 -> * Signature;
57    1 -> * Fact;
58    1 -> * Statement;
59    name;
60    modelNamespace;
61   }
62   ------------------------------------------------
63
64   class Signature {
65    isA AlloyObject;
66    name;
67    extendsName;
68    Boolean isBounded;
69    Boolean isAbstract;
70    Boolean isSingleton;
71    1 -> * Function function;
     }
```

LISTING 14. TEXTUAL REPRESENTATION OF THE PARTIAL METAMODEL OF ALLOY

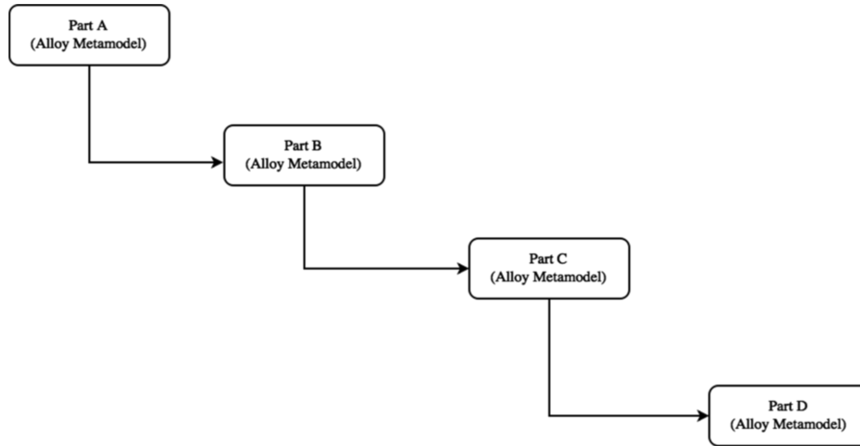


FIGURE 8. AN OVERVIEW OF THE ALLOY METAMODEL

In Figure 8, we present an overview of the Alloy metamodel we implemented for the purpose of code generation. Due to its size, we have divided this into 4 different parts labelled A-D. Figure 9 is a visual representation of the metamodel corresponding to Part A; while C corresponds to the diagrams indicated as label B-D. The arrow-head indicates the direction and position of the corresponding parts. By bringing these parts together, we will realize a unified metamodel.

AlloyObject
+printObject()
+convertEndsToAssociation()
+addStartDottedLine()
+addEndDottedLine()

AlloyModel
+name: String
+modelNamespace: String
+AlloyModel()
-genAssociationFact()
-hasStringAttribute()
-namespace()
-checkAssocitaionBtw()
-analyzeClassForH()
-genSignatures()
-genReflexiveRelations()
+getImportStatements()
+printImportStatements()
-genOtherConstraints()
-analyzeAssociation()
-compareRoleNames()
+comment()

alloyModel 1
alloyModel 1
alloyModel 1
signatures *
statements *

Signature
+name:String
+extendsName: String
+isBounded:boolean
+isAbstract:boolean
+isSingleton:boolean
+Signature()
-getExtendsClass()
-instantiateDataMembers()
-reComposeAssociation()
-comment()
-addExtension()
-funPart()
+printSignature()
-compute()
-print()

Statement
+getName()
+print()

signature 1
function *

Function
+beginEndRoleName:String
+targetMult:String
+targetClassName:String
+Function()
+Function()
+Function()
+printFunction()

OpenStatement
+packageName:String
+className:String
+print()

NumericBoundFact
+className:String
+rName: String
+mult:Multiplicity
+index:Integer
+NumericBoundFact()
-initializeExtraVar()
-analyzeEndMult()
-declaration()
-temp()
-content()
-comment()
+print()

SymmetricFact
+SymmetricFact()
-declaration()
-comment()
-content()
+print()

FIGURE 9. PART A - ALLOY METAMODEL

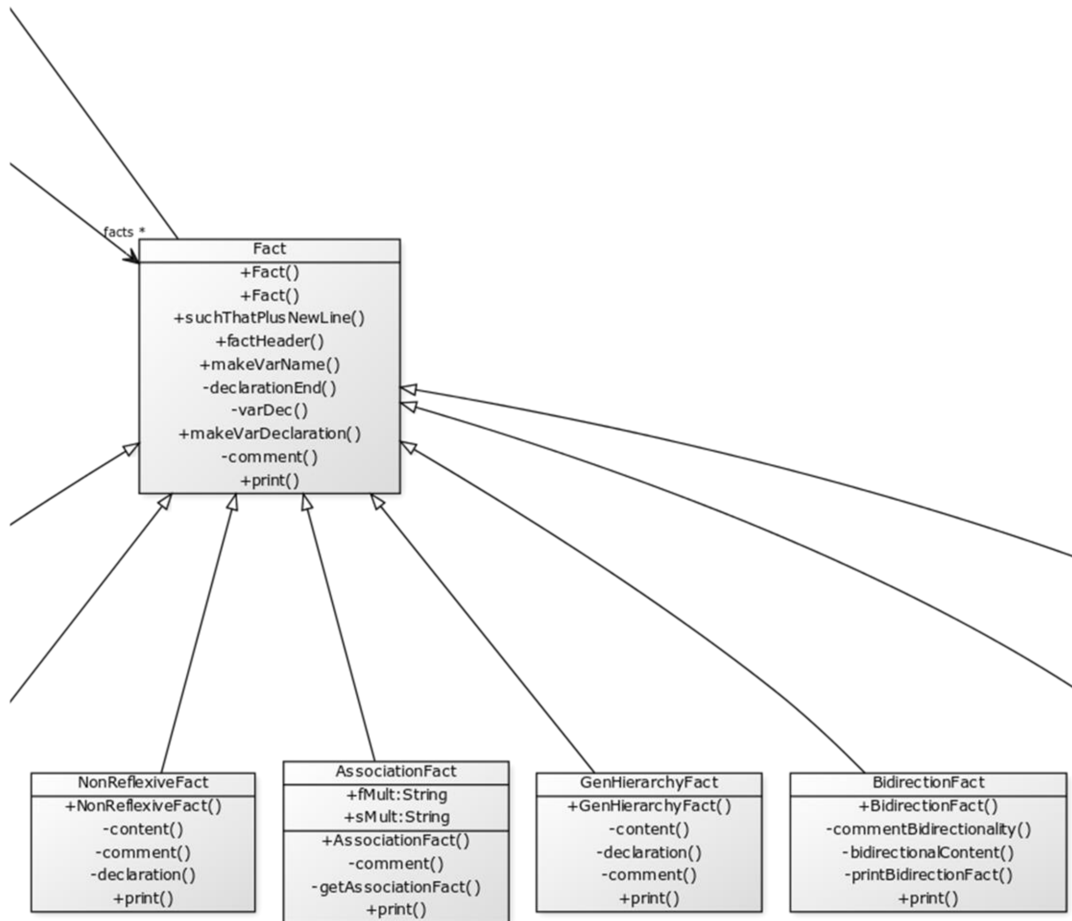


FIGURE 10. PART B - ALLOY METAMODEL

The AIR (see Figure 9 **-** Figure 12) is a collection of core elements (e.g. signatures, facts, predicates, functions, commands) of the metamodel. The main elements in the metamodel based on our work are signatures and facts. We considered these elements important because signatures model classes and facts model constraints among classes. In particular, we map associations to classes and facts. But for every kind of association, there is a different type of fact set generated.

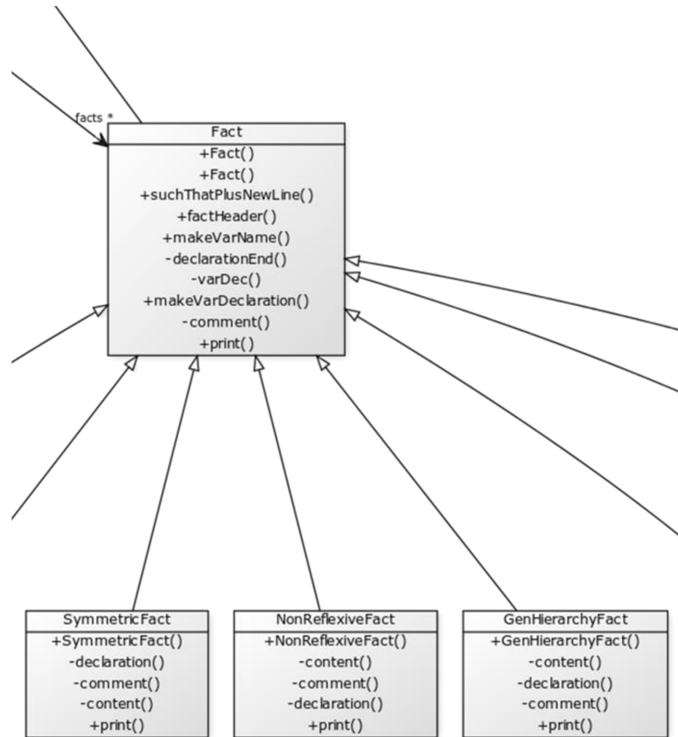


FIGURE 11. PART C - ALLOY METAMODEL

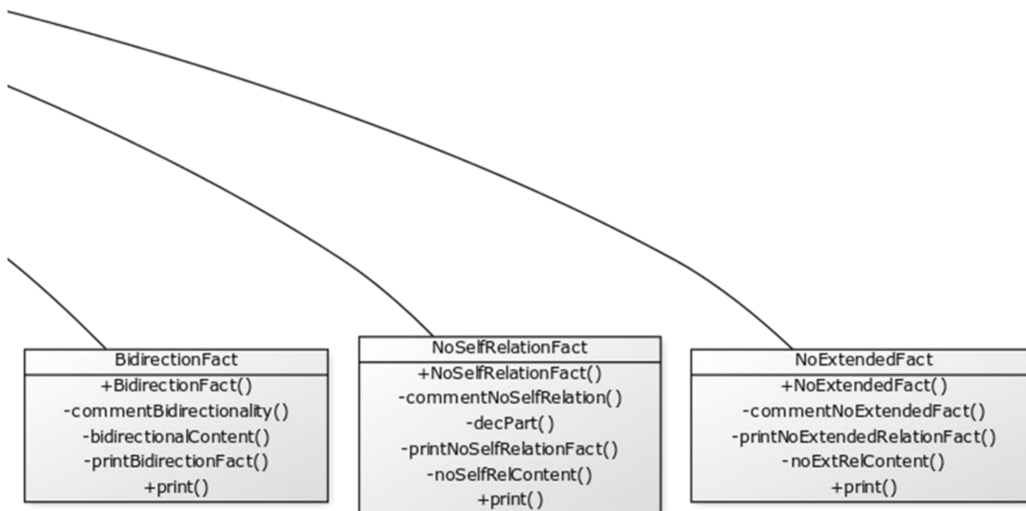


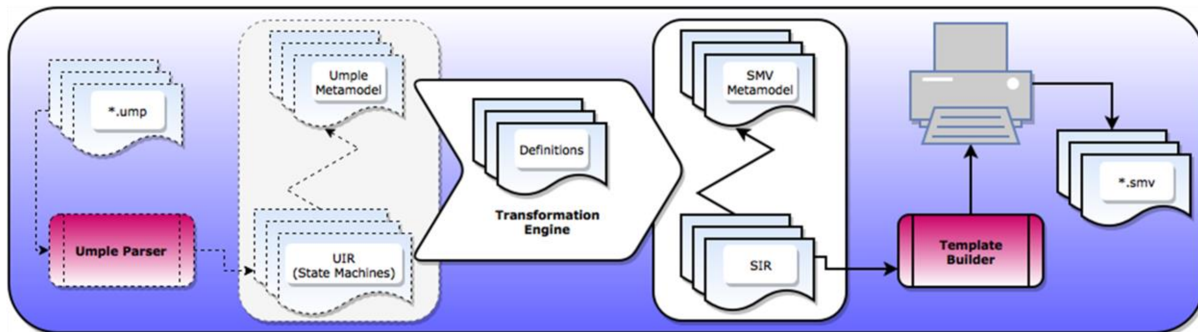
FIGURE 12. PART D - ALLOY METAMODEL

Other elements (e.g. predicates, functions, and commands) are also key to our work, but are not automatically generated. They represent the properties that the analyst wants to verify and validate. Currently, the user or analyst is required to specify these elements to obtain any value from the approach.

3.3 SMV Translator

This section presents the process of transforming Umple's state machine to SMV, the input language of the nuXmv model checker. In particular, we discuss our approach to translating **.ump* to **.smv*.

Our approach combines **M2M** and **M2T** transformations such that a transformation is made from Umple's Internal Representation (UIR) with the focus on state machines to SMV's Internal Representation (SIR). Then the SIR is further converted to text via templating and written into file. The overall architecture is given in Figure 13.



KEY: ------------ EXISTING IMPLEMENTATION ———— OUR IMPLEMENTATION

FIGURE 13. TRANSFORMATION ARCHITECTURE FOR NUXMV CODE GENERATION

To realize a consistent *M2M transformation*, we extracted a partial metamodel of Umple's state machines (see Figure 14) and created a partial metamodel for SMV. A corresponding textual representation of the metamodel (i.e., Figure 14) is presented in Listing 15. The complete visual representation of Umple's metamodel can be obtained from [79].

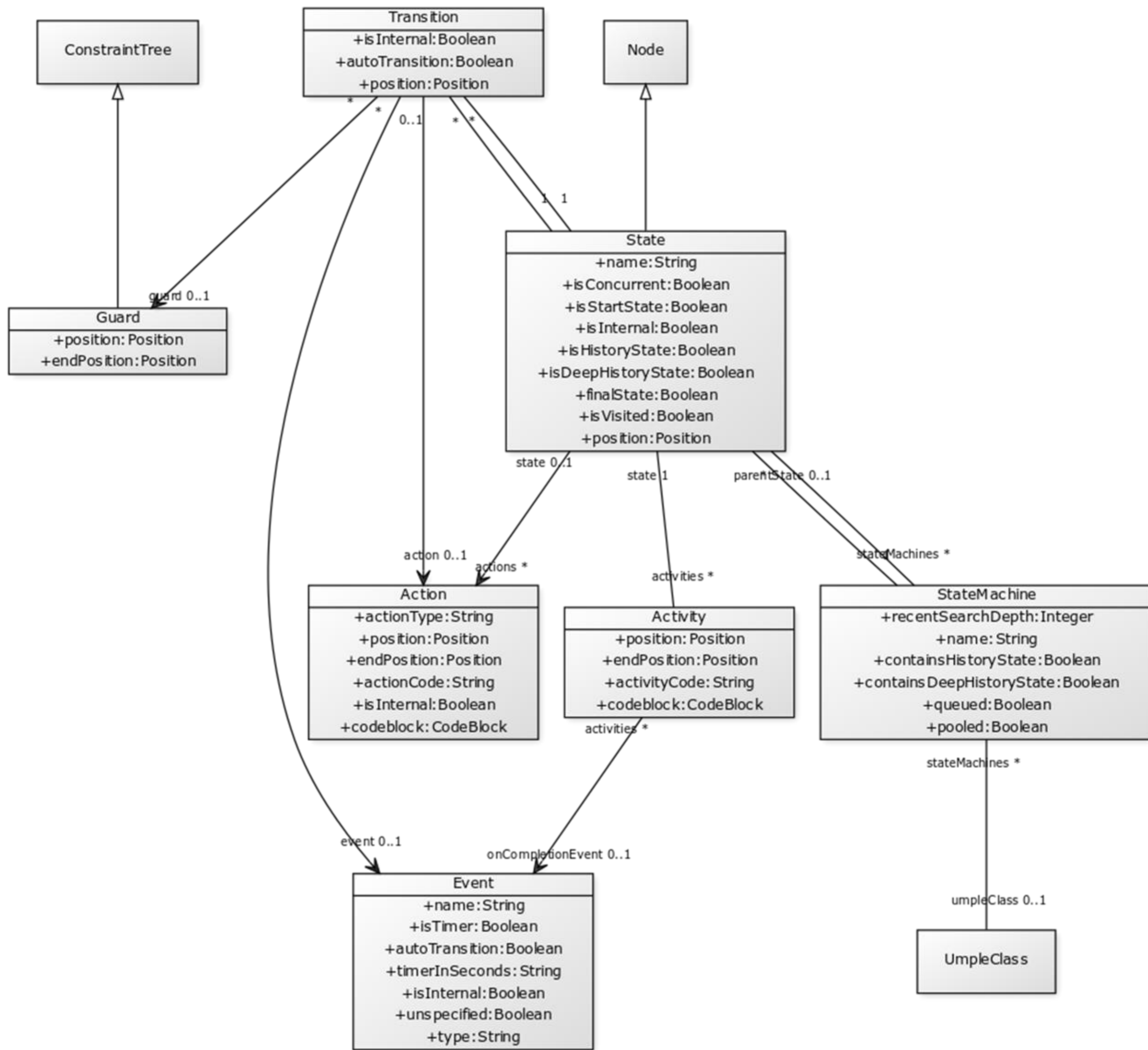


FIGURE 14. UMPLE'S METAMODEL FOR STATE MACHINE DIAGRAMS

```
1   external UmpleClass { … }
2
3   class StateMachine {
4     Integer recentSearchDepth;
5     String name;
6     Boolean containsHistoryState;
7     Boolean containsDeepHistoryState;
8     * -- 0..1 UmpleClass;
9     * -- 0..1 State parentState;
10    key { parentState, name }
11    Boolean queued = false;
12    Boolean pooled = false;
13  }
14
15  class State {
16    isA Node;
17    name;
18    Boolean isConcurrent;
19    1 -- * Activity;
20    0..1 -> * Action;
21    * -- 1 StateMachine;
22    Boolean isStartState;
23    Boolean isInternal;
24    Boolean finalState;
25  }
26
27  class Activity {
28    * -> 0..1 Event
29      onCompletionEvent;
30  }
31
32  class Transition {
33    Boolean isInternal;
34    Boolean autoTransition;
35    * -> 0..1 Event;
36    * -- 1 State fromState;
37    * nextTransition -- 1 State
38      nextState;
39    * -> 0..1 Guard;
40    0..1 -> 0..1 Action;
41    key { fromState, event,
42      nextState, guard, action }
43  }
44  class Action {
45    lazy actionType;
46    String actionCode;
47    Boolean isInternal = false;
48
49    key { actionType, actionCode,
50      position }
51  }
52
53  class Event {
54    name;
55    Boolean isTimer = false;
56    Boolean autoTransition = false;
57    type = "Boolean";
56    key { name }
57  }
58
59  class Guard { isA ConstraintTree; }
60
61  class ConstraintTree  {
62    ConstraintVariable root;
63    ConstraintOperator requestor;
64    Boolean shouldDisplayBrackets;
65    Boolean displayNegation;
66    Boolean displayBrackets;
67    Integer numberOfElements;
68    key { root }
69  }
70
71  external Node { … }
```

LISTING 15. PARTIAL METAMODEL OF STATE MACHINE IN UMPLE

```
1    trait MyUtility { ... }
2    external StateTableEntry { ... }
3    class NuSMVModule {
4     isA MyUtility;
5     String identifier;
6     String [] parameters;
7     1 -> 1 ModuleBody;
8    }
9    class CounterExampleTable {
10     isA MyUtility;
11     1 -> 1 ModuleElement requirement;
12     1 -> * CounterExampleColumn;
13     String [] rowLabels;
14     String sourceMachine;
15     String sourceClass;
16   }
17   class ColumnEntry {
18     isA StateTableEntry;
19     lazy Boolean isDerived;
20     String value;
21   }
22   class CounterExampleColumn {
23     isA MyUtility;
24     Integer index;
25     1 -> * ColumnEntry;
26     String [] stateValues;
27     String header;
28   }
29   class ModuleBody {
30     1 -> 1..* ModuleElement;
31   }
32   class ModuleElement {
33     lazy String header;
34   }
35   class VarDeclaration {
36     isA ModuleElement;
37     1 -> 1..* VariableSpecifier;
38   }
39   class VariableSpecifier {
40     isA MyUtility;
41     String identifier;
42     String [] typeSpecifier;
43     lazy Boolean isBracketed;
44     lazy String typeName;
45   }
46   class DefineBody {
47     String identifier;
48     1 -> 1 BasicExpression;
49   }
50   class NextExpression {
51     isA BasicExpression;
52     1 -> 1 BasicExpression;
53   }
54   class IVarDeclaration {
56     isA ModuleElement;
57     1 -> 1..* VariableSpecifier;
58   }
59   class DefineDeclaration {
60     isA ModuleElement;
61     1 -> 1..* DefineBody;
62   }
63   class BasicExpression {
64    0..1 parent -- 0..2 BasicExpression
65      children;
66    identifier;
67    lazy Boolean bracketed;
68    lazy Boolean displayNegation;
69    operator { AND, OR, XOR, XNOR,
70      IMPLY, IFF, EQ, NE, LT, GT, LE,
71      GE, NULL, NOT, MOD, PLUS, MUL,
72      MINUS, DIV };
73   }
74   class CTLSpecification {
75     isA ModuleElement;
76     lazy Boolean displayNegation;
77     1 -> 1 CTLExpression;
78   }
79   class InvarExpression {
80     isA BasicExpression;
81     invarOperator { next };
82     lazy Boolean qualified;
83   }
84
85   class CTLExpression {
86     isA BasicExpression;
87     ctlOperator { EG, EX, EF, AG, AX,
88      AF, E, A, U };
89     lazy Boolean qualified
90   }
91   class AssignConstraint {
92     isA ModuleElement;
93     1 -> 1..* Assign;
94   }
95
96   class Assign {
97     String identifier;
98     1 -> 1 BasicExpression;
99   }
100
101  class SimpleAssign { isA Assign; }
102  class InitAssign { isA Assign; }
103  class NextAssign { isA Assign; }
104
105  class CaseExpression {
106    isA BasicExpression;
107    1 -> 1..* CaseStatement;
108  }
109  ...
```

LISTING 16. TEXTUAL REPRESENTATION OF THE NUXMV METAMODEL

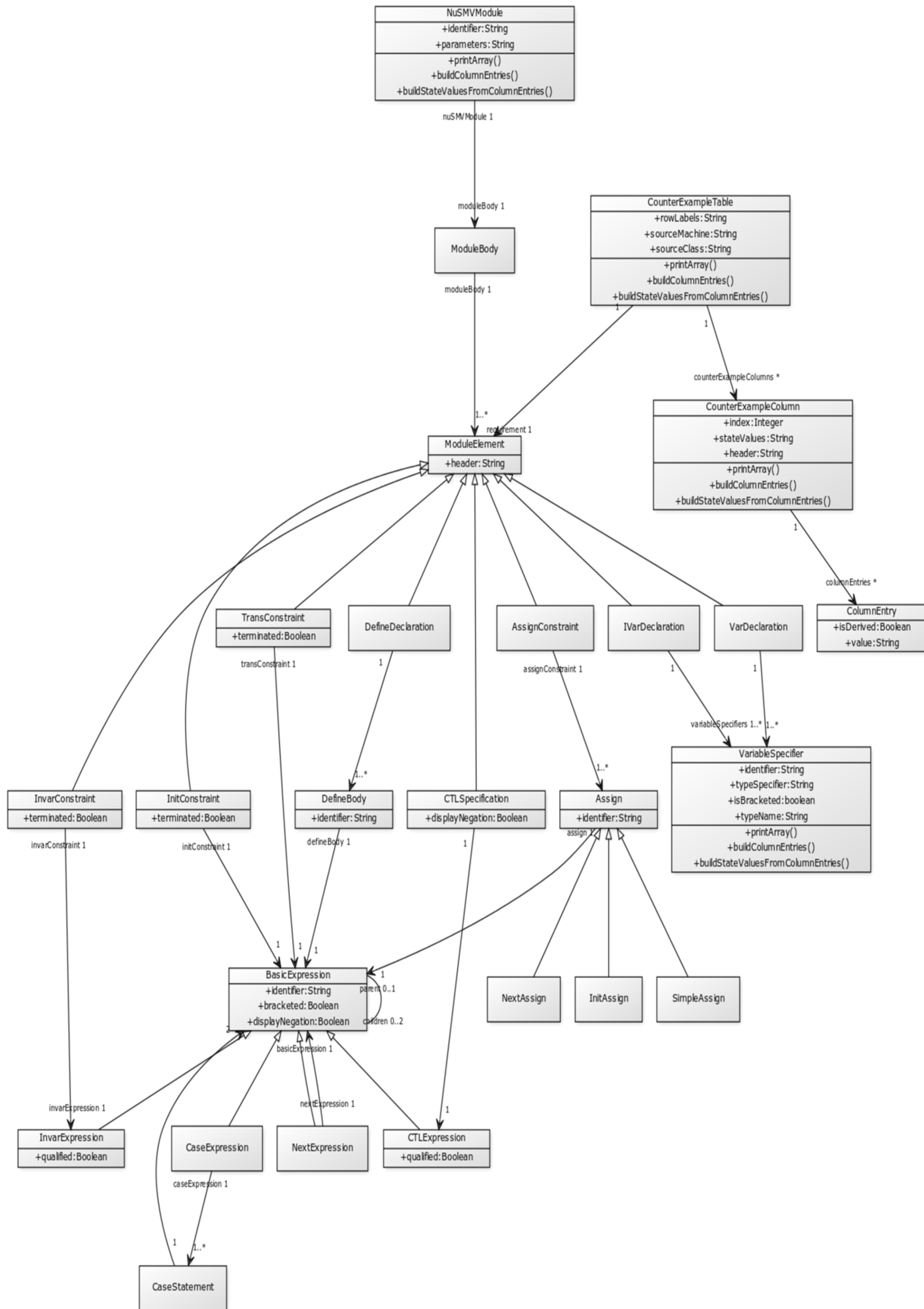


FIGURE 15. VISUAL REPRESENTATION OF NUXMV METAMODEL

The metamodel presented in Figure 15 is for the purpose of generating an SMV model during model transformation from its corresponding state machine expressed in Umple. It is incomplete when compared to the grammar of the SMV language provided in [69]. But we consider this subset to be sufficient for our work. The corresponding textual representation of the metamodel, expressed in Umple is provided in Listing 16.

3.4 Summary

In this chapter, we discussed our approach to engineering the transformations from Umple to Alloy and Umple to SMV. We have developed an architecture that guided our work from the starting point (i.e., design phase) to the code generation phase. We presented a detailed discussion of the activities involved at each phase.

For the translations of Umple to Alloy and Umple to SMV, we presented an architecture which shows the internal components of the translator at an abstract level. Then we proceed to the presentation of the metamodels of Alloy and Umple. Readers should note that the metamodels presented are partial: they represent the subset relevant to our work. We presented them textually and diagrammatically.

We defer other discussions on the transformation steps to various chapters dedicated to each modeling constructs. In particular, we will discuss the transformations of class models in **Chapter 4**. Similarly, we will discuss the steps for the transformations of state machines in **Chapter 7**.

4 Transformation of Class Models

In this chapter, we present the transformation of Umple class models for the purpose of analysis. Umple class models include: *classes*, *attributes*, and *associations*. These modeling components will be formally specified in **Alloy's** input language.

We will focus our discussions on the following concepts: attribute and multiplicity mappings, and object-oriented design patterns. We map *String* and *Integer* attributes directly to their equivalent data types in Alloy. However, other data types not directly supported by Alloy are mapped based on Table 8 and additional constraints required to model semantics such as Date, Time, etc. will be added in the future. We acknowledge that there is indeed some loss of semantics but this is in accordance with Alloy small scope hypothesis (see [80]).

The multiplicities are mapped directly to their equivalent in **Alloy**. Our approach to formally specify *associations* of Umple systems is based on object-oriented design patterns. We first introduce a set of constraints with modelling examples for ease of understanding. Finally, we present the generated code for each modeling example considered.

TABLE 8. ATTRIBUTE MAPPING FROM UMPLE TO ALLOY

Umple	**Alloy**
Integer, *Double*, *Float*	*Int*
Date, *Time*, *String*	*String*
Boolean	*Boolean*
ObjectName	ObjectName

4.1 Attributes

Umple allows users to declare variables of various types. These include: *Integer*, *Float*, *Double*, *Boolean*, *Date*, *Time*, *String*, as well as the names of other classes. However, **Alloy** only supports the declaration of variables of integer and string types. To ensure all data types are properly handled, Table 8 presents mappings between **Alloy** and **Umple** data types.

4.2 Multiplicity

In this section, we present a mapping of the UML multiplicity expressed in **Umple** to its equivalent in **Alloy**. These include: *mandatory-one* (i.e., *1..**), *one* (i.e., *1* or *1..1*), *many* (i.e., *n* or *n..m*), *any-number* (i.e., *** or *0..**), and *optional* (i.e., *0..1*).

Table 9 presents the mapping of Umple's multiplicity to Alloy. We acknowledge that semantics information is incomplete based on this table, however we have defined a set of constraints (e.g., Listing 23) to ensure this information is preserved, even after transformation.

TABLE 9. MULTIPLICITY MAPPINGS FROM UMPLE TO ALLOY

Umple	Alloy
1, 1..1	*one*
1..*, n, n..m	*some*
, 0..	*set*
0..1	*lone*

Where:

$$\forall_{\{m,n\} \subseteq \mathbb{N}} \cdot n > 1 \land m \geq n$$

4.3 Transformation Phases (Umple to Alloy)

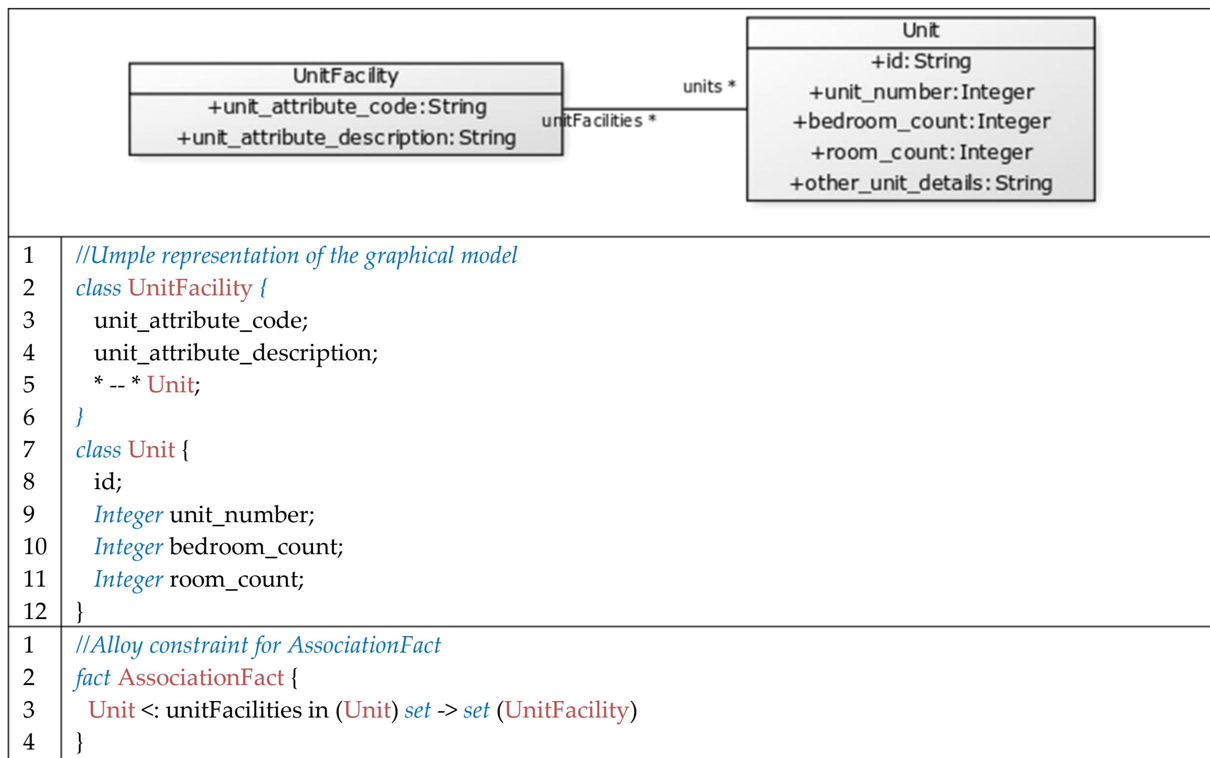
This section presents an overview of various phases of transformation for the generation of **.als* files from an Umple model as implemented in this work. Recall that the UIR is a product of the

Umple Compilation Unit which existed before our work, so we skip transformations in this context from our discussion.

We discuss the phase involved in the process of creating an **AIR** from the **UIR** (M2M transformation). To avoid unnecessary details, we make the discussion as abstract as we can. In particular, we skip discussions of the mappings and templates but present the textual result of the overall transformations and the visual representation of the input model. Interested readers may check [81] for further details. We divide this phase into steps based on the artifact being transformed (or produced). These are explained as follow:

Step 1 – Generation of AssociationFact

An AssociationFact defines a constraint based on the association ends for every association in a given model. It specifies multiplicities on unique association ends. In particular, AssociationFact constraints are not generated for reflexive associations with exactly the same multiplicities at each end. We regard this as redundant because the same number of instances will be needed per time for the purpose of analysis. But for different ends, it is necessary to enforce the number of instances to be created at any given time.

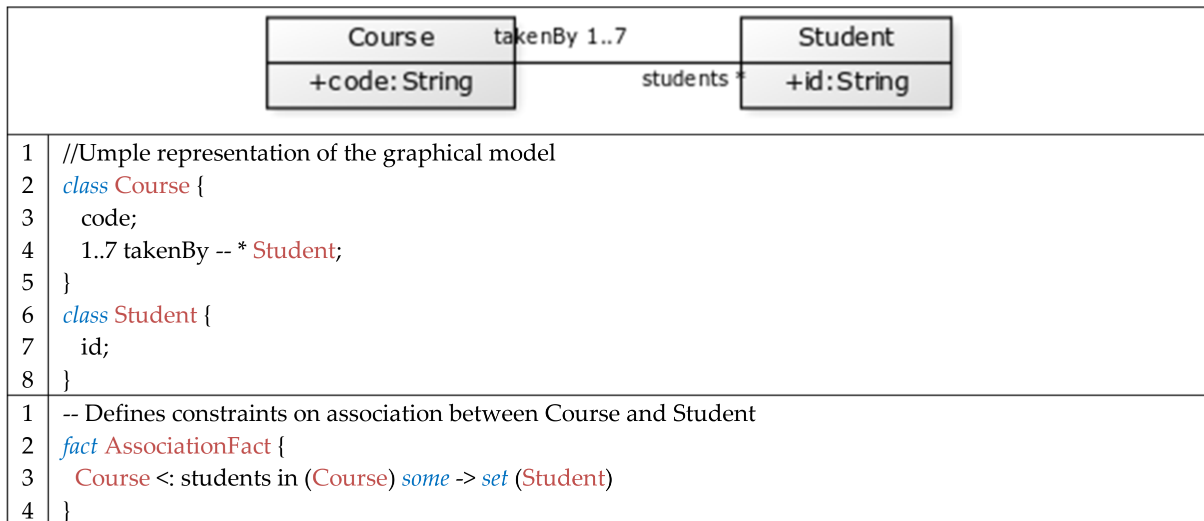



```
1   //Umple representation of the graphical model
2   class UnitFacility {
3     unit_attribute_code;
4     unit_attribute_description;
5     * -- * Unit;
6   }
7   class Unit {
8     id;
9     Integer unit_number;
10    Integer bedroom_count;
11    Integer room_count;
12  }
```

```
1   //Alloy constraint for AssociationFact
2   fact AssociationFact {
3    Unit <: unitFacilities in (Unit) set -> set (UnitFacility)
4   }
```

LISTING 17. EXAMPLE OF ASSOCIATIONFACT CONSTRAINT (NON-NUMERIC)

A typical example of this constraint is presented in Listing 17. The constraint implies that for every Unit there is any number of UnitFacility and vice-versa.

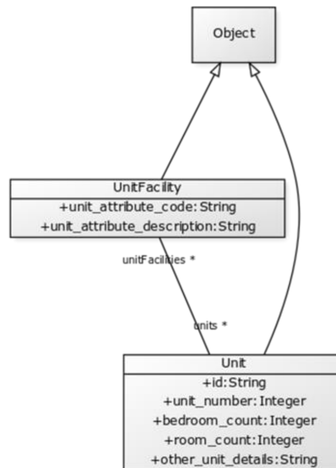



```
1  //Umple representation of the graphical model
2  class Course {
3    code;
4    1..7 takenBy -- * Student;
5  }
6  class Student {
7    id;
8  }
```

```
1  -- Defines constraints on association between Course and Student
2  fact AssociationFact {
3   Course <: students in (Course) some -> set (Student)
4  }
```

LISTING 18. EXAMPLE OF ASSOCIATIONFACT CONSTRAINT (NUMERIC)

When associations are numerically bounded, AssociationFact are constructed with *some* multiplicity. An example is presented in Listing 18 for an example of Student-Course association.




```
1   //Umple representation of the graphical model
2   class Object {}
3   class UnitFacility {
4     unit_attribute_code;
5     unit_attribute_description;
6     * -- * Unit;
7    isA Object;
8   }
9   class Unit {
10    id;
11    Integer unit_number;
12    Integer bedroom_count;
13    Integer room_count;
14    isA Object;
15  }
1   -- Defines a signature for class Object
2   abstract sig Object { }
3
4   -- Defines a signature for class UnitFacility
5   sig UnitFacility extends Object {
6    units : set Unit,
7    unit_attribute_code : String,
8    unit_attribute_description : String
9   }
10
11  -- Defines a signature for class Unit
12  sig Unit extends Object {
13   unitFacilities : set UnitFacility,
14   id : String,
15   unit_number : Int,
16   bedroom_count : Int,
17   room_count : Int,
18   other_unit_details : String
19  }
```

LISTING 19. ALLOY SIGNATURE FROM CLASSES

Step 2 – Generation of Signature

A Signature corresponds to an UmpleClass according to our implementation. For each class in the UIR, a corresponding Signature is created. The name of the UmpleClass becomes the name of the Signature being created. In Listing 19, classes Object, UnitFacility and Unit become signatures on lines 2, 5 and 12 respectively.

For an UmpleClass with a *symmetric-reflexive* association, a NonReflexiveFact and a SymmetricFact are composed to enforce *non-self-exclusiveness* and *symmetricity* respectively on the Signature.

The necessary and sufficient constraints for an UmpleClass with an *asymmetric* association include a TransitiveClosureFact and NoExtendedRelation.

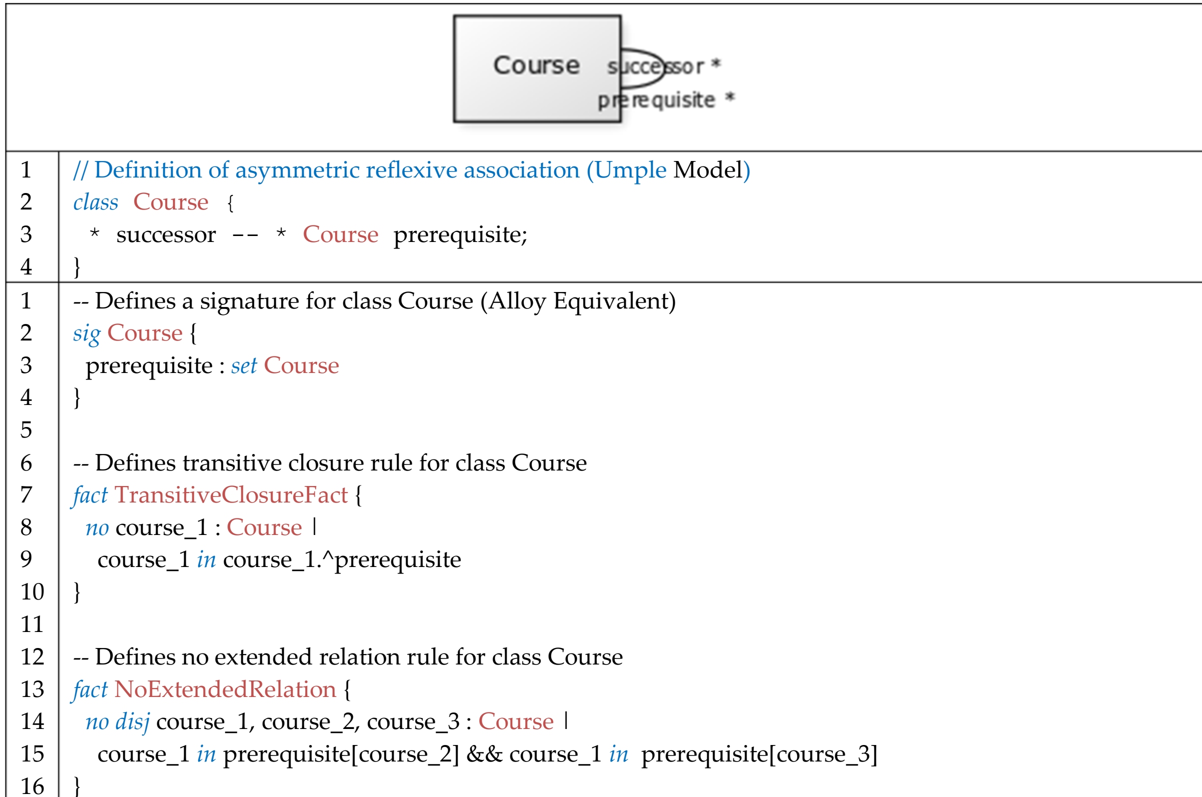
When UmpleClass may also be a sub-class of another UmpleClass (i.e., superclass) or it may be abstract. For the former, its associated extendsName is set to the name of its superclass. But for the latter, its "*isAbstract*" attribute is set to *true*.

An example of a sub-class is on line 5 of Listing 19. It defines UnitFacility as a subclass of Object. Similarly, line 2 defines Object as an abstract Signature.

Suppose an UmpleClass has attributes, a Field is constructed for each of them. For example, on lines 14-18 various fields were created for attributes of class Unit. These are of types *String* and *Integer*. A Field is also constructed for every association end involving the UmpleClass. On lines 6, 13 fields are constructed for association ends with role names units and unitFacilities.

Step 2.1 – Generation of NonReflexiveFact

The NonReflexiveFact defines a constraint to enforce that reflexiveness is not bi-directional. In particular, a *parent instance* of an UmpleClass with a reflexive association is prohibited from becoming a *child instance* of its child by this constraint.




```
1  // Definition of asymmetric reflexive association (Umple Model)
2  class Course {
3    * successor -- * Course prerequisite;
4  }
```

```
1  -- Defines a signature for class Course (Alloy Equivalent)
2  sig Course {
3    prerequisite : set Course
4  }
5
6  -- Defines transitive closure rule for class Course
7  fact TransitiveClosureFact {
8    no course_1 : Course |
9      course_1 in course_1.^prerequisite
10 }
11
12 -- Defines no extended relation rule for class Course
13 fact NoExtendedRelation {
14   no disj course_1, course_2, course_3 : Course |
15     course_1 in prerequisite[course_2] && course_1 in  prerequisite[course_3]
16 }
```

LISTING 20. ASYMMETRIC ASSOCIATION EXAMPLE

Step 2.2 – Generation of SymmetricFact

The SymmetricFact models symmetricity on the concerned Signature such that whenever two instances of a class, say course1 (see line 14 of Listing 21) is symmetrically related to another instance, say course2, both being instances of class Course then a reference link is defined from course1 to course2 and vice-versa (see lines 13-17 of Listing 21).

Step 2.3 – Generation of TransitiveClosureFact

The TransitiveClosureFact defines a transitive closure property on the concerned Signature. In particular, it enforces the model such that *descendant-ancestor relationship* is maintained throughout an object's life cycle. For example, on lines 7-10 of the Alloy section of Listing 20 a constraint that defines this relationship for Course is presented.

Course
isMutuallyExclusiveWith *

```
// Definition of asymmetric reflexive association (Umple Model)
class  Course  {
  *  self isMutuallyExclusiveWith;
}
```

```
-- Defines a signature for class Course (Alloy Equivalent)
sig Course {

  isMutuallyExclusiveWith : set Course
}

-- Defines non-reflexive rule for class Course
fact NonReflexiveFact {
  no course_1 : Course |
    course_1 in isMutuallyExclusiveWith[course_1]
}

-- Defines symmetric rule for class Course
fact SymmetricFact {
  all course_1, course_2 : Course |
    course_1 in isMutuallyExclusiveWith[course_2]
      <=> course_2 in isMutuallyExclusiveWith[course_1]
}
```

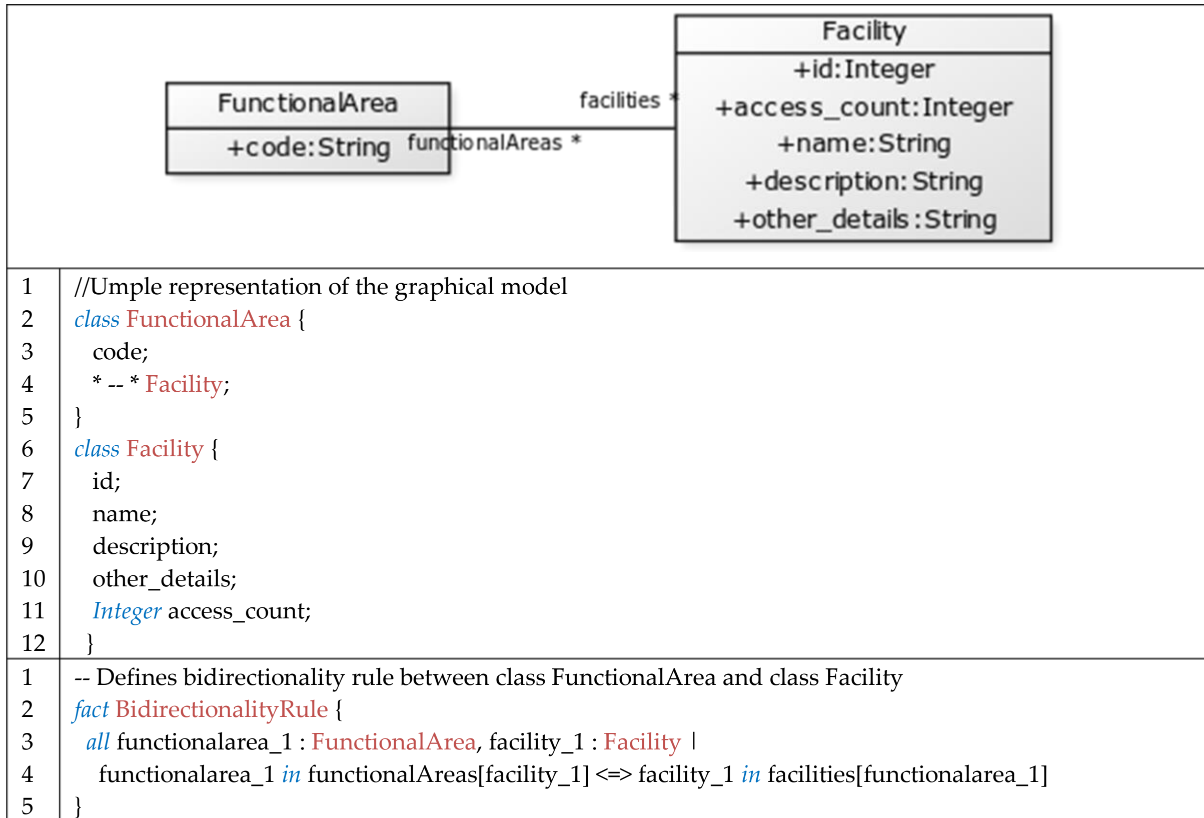
LISTING 21. SYMMETRIC ASSOCIATION EXAMPLE

Step 2.4 - Generation of NoExtendedRelation

The NoExtendedRelation defines a constraint on an asymmetric association such that for disjoint instances A, B and C of the type X, whenever instance C is genetically related to A there is no instance C that is genetically related to B. For example, on lines 13-16 of Listing 20, a constraint of this form is created for Course (an asymmetrically-related class).

Step 3 – Generation of BidirectionalityFact

We generate BidirectionalityFact for every bi-directional association in the model. It is defined such that instances of the participating signatures of an association have reference links from one end to another and vice-versa. The reference link is created based on the role names. We present an example of a bi-directionality rule on lines 3-5 in Listing 22.



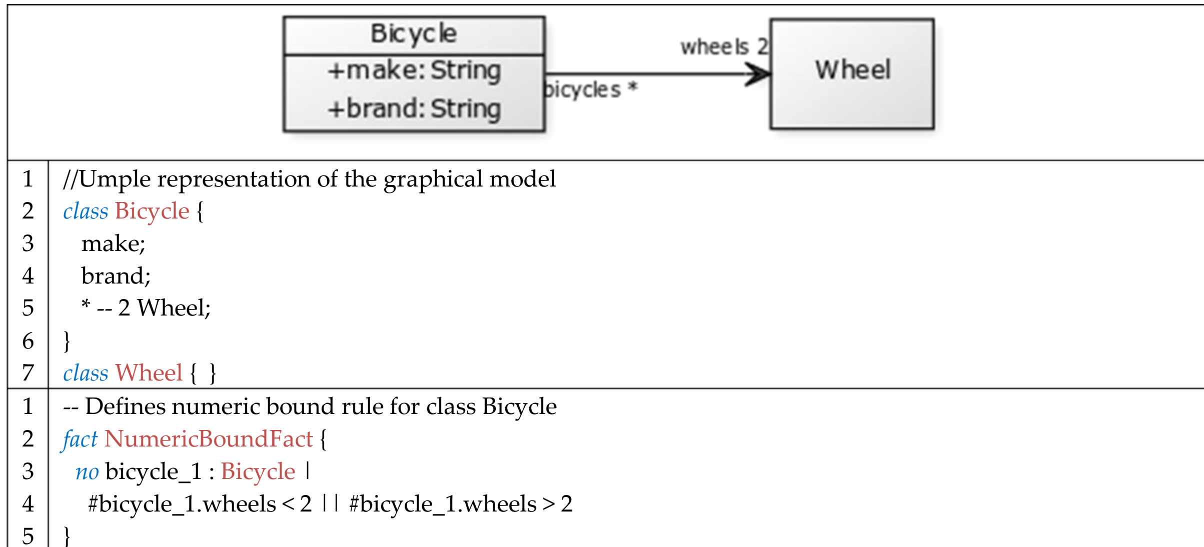

```
1   //Umple representation of the graphical model
2   class FunctionalArea {
3     code;
4     * -- * Facility;
5   }
6   class Facility {
7     id;
8     name;
9     description;
10    other_details;
11    Integer access_count;
12   }
```

```
1   -- Defines bidirectionality rule between class FunctionalArea and class Facility
2   fact BidirectionalityRule {
3    all functionalarea_1 : FunctionalArea, facility_1 : Facility |
4     functionalarea_1 in functionalAreas[facility_1] <=> facility_1 in facilities[functionalarea_1]
5   }
```

LISTING 22. AN EXAMPLE OF BI-DIRECTIONALITY FACT

Step 4: Generation of NumericBoundFact

We generate NumericBoundFact for associations whose ends have numeric multiplicities. The purpose of the constraint is to ensure that the values specified in the multiplicities are maintained whenever instances are created during analysis. The numeric values in this context exclude "*1*" or "*0..1*" because such values have equivalent constructs in Alloy. We present the Bicycle-Wheel example in Listing 23 that shows the constraint generated whenever the multiplicity is an exact numerical value.

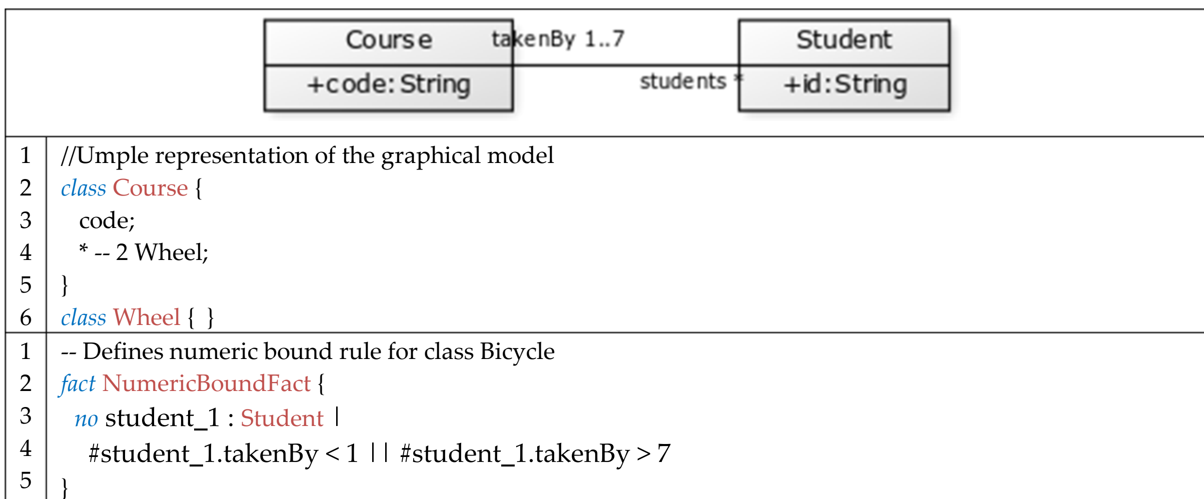



```
1  //Umple representation of the graphical model
2  class Bicycle {
3    make;
4    brand;
5    * -- 2 Wheel;
6  }
7  class Wheel { }
```

```
1  -- Defines numeric bound rule for class Bicycle
2  fact NumericBoundFact {
3   no bicycle_1 : Bicycle |
4    #bicycle_1.wheels < 2 || #bicycle_1.wheels > 2
5  }
```

LISTING 23. AN EXAMPLE OF NUMERICBOUNDFACT (EXACT VALUE)

On the other hand, a similar constraint is generated for associations involving inexact numeric bounds. However, the values at the ends become the bounds in such cases. A typical example is presented in Listing 24.



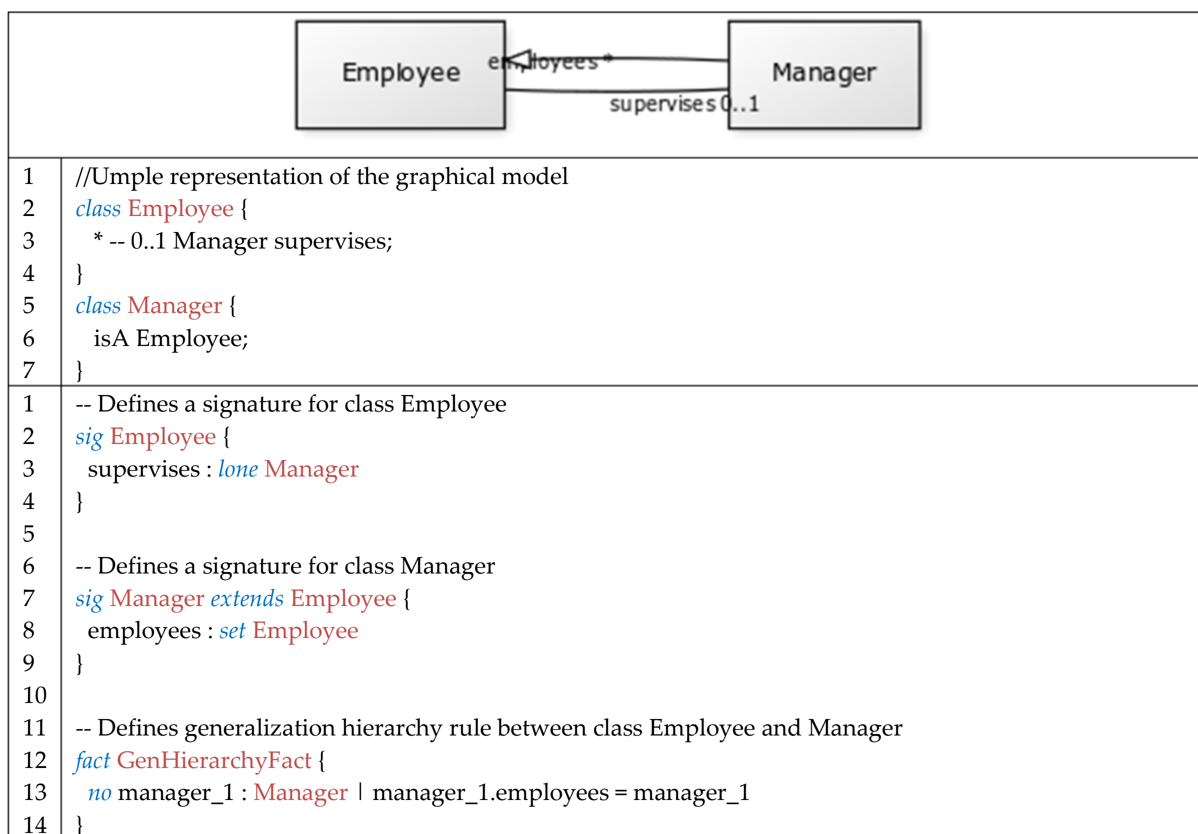

```
1  //Umple representation of the graphical model
2  class Course {
3    code;
4    * -- 2 Wheel;
5  }
6  class Wheel { }
```

```
1  -- Defines numeric bound rule for class Bicycle
2  fact NumericBoundFact {
3   no student_1 : Student |
4    #student_1.takenBy < 1 || #student_1.takenBy > 7
5  }
```

LISTING 24. AN EXAMPLE OF NUMERICBOUNDFACT (RANGE VALUE)

Step 5: Generation of GenHierarchyFact

The GenHierarchyFact is generated for a set of instances that are in a hierarchical relationship involving superior and subordinate. The common pattern for this in a class diagram involves two classes that have both a generalization and an association. According to [82], participating objects in a general hierarchy can have zero or more objects above them in the hierarchy (known as *superior*) and zero or more objects below them in the hierarchy (known as *subordinates*). This is also known, in a more limited form, as the "Composite" pattern [83]. For example, the Employee-Manager relationship presented in Listing 25 and the associated constraint on lines 12-14. It enforces that no manager manages him/herself. In other words, the set of employees managed by a manager excludes the manager.




```
1  //Umple representation of the graphical model
2  class Employee {
3    * -- 0..1 Manager supervises;
4  }
5  class Manager {
6    isA Employee;
7  }
```

```
1  -- Defines a signature for class Employee
2  sig Employee {
3   supervises : lone Manager
4  }
5
6  -- Defines a signature for class Manager
7  sig Manager extends Employee {
8   employees : set Employee
9  }
10
11 -- Defines generalization hierarchy rule between class Employee and Manager
12 fact GenHierarchyFact {
13   no manager_1 : Manager | manager_1.employees = manager_1
14 }
```

LISTING 25. THE GENERAL HIERARCHY CONSTRAINT

4.4 Object-Oriented Design Patterns

We consider the formal specification of the following object-oriented design patterns in this section. A detailed discussion on some software design patterns can be obtained from [72], [82].

P1: *Abstract class pattern.*

P2: *Singleton pattern.*

P3: *Class hierarchy pattern.*

P4: *Bi-directional association pattern.*

P5: *Unidirectional association pattern.*

P6: *General hierarchy pattern.*

P7: *Asymmetric-reflexive association pattern*

P8: *Symmetric-reflexive association pattern.*

P9: *UML qualified association pattern.*

4.4.1 Abstract class pattern (P1)

The following example illustrates the formal specification of an abstract class Car expressed in Umple and its equivalent notation in Alloy.

Umple

```
class Car { abstract; }
```

Alloy

```
abstract sig Car { }
```

4.4.2 Singleton class pattern (P2)

In the following example, we present a mapping of a singleton class pattern from Umple to Alloy. We illustrated this with a Company class in a management system of an organization.

Umple

```
class Company { singleton; }
```

Alloy

```
one sig Company { }
```

4.4.3 Class hierarchy pattern (P3)

The following example demonstrates the formal representation of class hierarchy patterns in an object-oriented system. It models SUV as a kind of Car.

Umple

```
class Car { abstract; }
class SUV { isA Car;  }
```

Alloy

```
abstract sig Car { }
sig SUV extends Car { }
```

4.4.4 Bidirectional association pattern (P4)

We demonstrate the usage of bi-directional association pattern with the following example. The example models the relationship between FunctionalArea and Facility. It is extracted from a model of AccessControl found in [56].

Umple

```
class FunctionalArea { }
association { * FunctionalArea -- * Facility; }
class Facility { }

//     OR

class FunctionalArea { }
class Facility {
 * facilities -- * FunctionalArea functionalAreas;
}
```

Alloy

```
sig FunctionalArea {
     facility : set Facility
}
sig  Facility {
    functionalAreas : set FunctionalArea
}
fact AssociationFact {
   Facility <: functionalAreas in (Facility) set -> set (FunctionalArea)
}
fact BidirectionalityRule {
  all functionalarea : FunctionalArea, facility : Facility |
   functionalarea in functionalAreas[facility]   <=> facility in facilities[functionalarea]
}
```

4.4.5 Unidirectional association pattern (P5)

We demonstrate the usage of the *unidirectional association pattern* with the following example. The example models the relationship between FacilityType and Facility. The source code in Umple is obtained from [56].

Umple

```
class FacilityType { }
association {
  0..1 FacilityType <- * Facility;
}
class Facility { }

//                    OR

class FacilityType { }
class Facility {
  * facilities -> 0..1 FacilityType facilityType;
}
```

Alloy

```
sig FacilityType { }
sig  Facility {
    facilityType : lone FacilityType
}

fact AssociationFact {
   Facility <: facilityType in (Facility) set -> lone (FacilityType)
}
```

4.4.6 General hierarchy pattern (P6)

The general hierarchy pattern models hierarchical relationships among a set of objects involving superior and subordinate. According to [82], participating objects in such hierarchy can have zero or more objects above them in the hierarchy (known as *superior*) or zero or more objects below them in the hierarchy (known as *subordinate*). The following example illustrates this scenario. Note that this pattern is also known in [83] as Composite.

Umple

```
class Employee { }
class Manager {
  isA Employee;
  irreflexive 0..1 supervises -- * Employee employees;
}
```

Alloy

```
sig Employee {
  supervises : lone Manager
}
sig Manager extends Employee {
  employees : set Employee
}

fact Association-Fact {
  Employee <: supervises in (Employee) set -> lone (Manager)
}

fact Bi-directionality {
  all manager : Manager, employee : Employee |
    manager in supervises[employee] <=> employee in employees[manager]
}

fact No-Self-Loop {
  no manager : Manager |
   manager in  manager.employees
}
```

4.4.7 Asymmetric-reflexive association pattern (P7)

The following example defines our approach to formally specify asymmetric-reflexive association. It models parent-child relationship where the parent and child are instances of person. In spite of the fact that both ends of the association are the same, the role names at the ends are distinct.

The Umple modeling construct does not explicitly require the association to be acyclic, but by creating a cycle in this model would be an error; i.e. we cannot allow a child to become the parent of his/her own parent.

To enforce this, we analyze the association to discover such pattern and automatically generate a constraint to forbid such a cyclic relationship.

Umple

```
class Person {
  0..2 parent -- * Person children;
}
//                    OR
class Person { }
association {
  0..2 Person parent -- * Person children;
}
```

Alloy

```
sig Person {
  parents : set Person
}

fact NoCyclicRelation {
  no person : Person |
    person in person.^parents
}

fact NumericBounds {
  no person : Person |
    #person.parents  < 0 && #person.parents > 2
}
```

4.4.8 Symmetric-reflexive association pattern (P8)

This pattern only differs from asymmetric association pattern because the ends are semantically indifferent. We demonstrate its usage with the following example. The example models a mutually exclusive set of courses in a university.

Umple

```
class Course {
  * self isMutuallyExclusiveWith;
}
```

Alloy

```
sig Course {
 isMutuallyExclusiveWith : set Course
}

fact No-Self-Loop {
  no course : Course |
   course in isMutuallyExclusiveWith[course]
}

fact Symmetric-Relation {
  all course1, course2 : Course |
   course1 in isMutuallyExclusiveWith[course2] <=> course2 in isMutuallyExclusiveWith[course1]
}
```

4.5 Summary of Class Model Specification

In this chapter, we presented the formal specification of key features of UML class models expressed in Umple for the purpose of analysis. We achieved this by presenting the Alloy specifications of UML multiplicities, associations, and attributes.

The Alloy language provides mapping of multiplicity supported by UML and Umple, although not all details of multiplicities are permitted in Alloy. For the multiplicity constraints in UML, we provided a mapping to their corresponding equivalent in Alloy.

We remark that the Alloy language provides mapping only for integer, boolean and string data types. Hence our approach maps UML attributes facilitated by Umple to integer and string domains.

Finally, we mapped UML/Umple associations using pattern-based approaches. We began by discussing constraints particular to patterns under consideration. Our discussion illustrates applications of constraints in modeling contexts. We discussed object-oriented design patterns

with modeling examples. The examples were presented in Umple and Alloy. Our approach reads Umple models and generates semantically equivalent Alloy models.

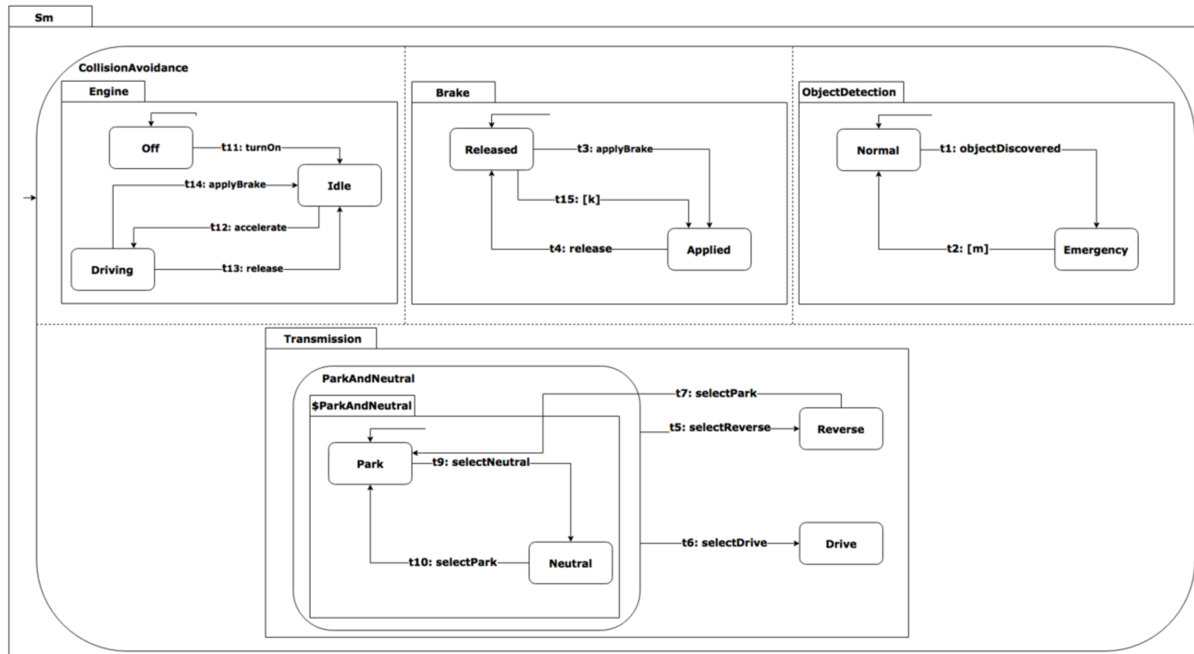
5 Formalization of State Machines

In this chapter, we present a novel approach to formally encode state machine diagrams for the purpose of formal analysis by model checking. Our approach is based on representing simple and hierarchical state machines (including orthogonal regions) in Umple and automatically transforming them to SMV [35], the input language of the nuXmv model checker. We cleanly separate concerns but systematically integrate components of hierarchical systems as opposed to the traditional flattening approach.

We enable states (i.e. simple and composite) and disable sub-state machines (parallel and non-parallel) by means of transitions. This approach is an extension to Badreddin et. al's [67] work on code generation strategy for composite state machines. In particular, their work transformed Umple's hierarchical state machines internally to an equivalent set (or a collection) of simple state machines.

Contributions:

We translated these simple state machines into SMV as opposed to Java, the ultimate focus of Badreddin et. al's [67] work. We compute transitions set to enable and disable states and sub-state machines in various cases (including region-crossing) and formally analyze these sets for the sake of consistency.



$$k := warningRadius \leq 3 \;\&\&\; v(\$ObjectDetection, s^{i}) = Emergency$$

$$m := warningRadius \leq 3$$

FIGURE 16. COLLISION AVOIDANCE SYSTEM

```
1   class Example {
2    Integer warningRadius;
3    sm {
4     CollisionAvoidance {
5      ObjectDetection {
6        Normal {
7         objectDiscovered -> Emergency; //t1
8        }
9        Emergency {
10          [warningRadius <= 3] -> Applied; //t2
11       } }
12      ||
13      Brake {
14        Released { applyBrake -> Applied;//t3 }
15        Applied { release -> Released; //t4 }
16     }
17      ||
18      Transmission {
19       [warningRadius <= 3 && ]
20      ParkAndNeutral {
21        selectReverse -> Reverse; //t5
22        selectDrive -> Drive; //t6
23        Park { selectneutral -> Neutral; //t9 }
24        Neutral { selectPark -> Park;//t10 }
25      }
26       Reverse { selectPark -> Park; //t7 }
27       Drive { selectNeutral -> Neutral;//t8 }
28      }
29      ||
30      Engine {
31       Off {  turnOn -> Idle; //t11 }
32       Idle { accelerate -> Driving; //t12 }
33       Driving {
34         release -> Idle; //t13
35         applyBrake -> Idle; //t14
36       }
37      } } } }
```

LISTING 26. TEXTUAL REPRESENTATION OF COLLISION AVOIDANCE SYSTEM

We present (in Figure 16) a "CollisionAvoidance" SSUA (state machine system under analysis) for the purpose of illustration. We will use this example to describe concepts of state machines.

5.1 Formal Semantics of Umple's State Machines

To facilitate the discussion of the concepts involved in our approach to formalizing state machines, a formal description of the syntax and semantics of Umple becomes critical. We introduce the notion of SSUA as a collection of state machines that forms the system to be analyzed. In particular, we define an SSUA as follows:

Definition 9. State Machine System Under Analysis (SSUA) in Umple

An SSUA in Umple is a 6-tuple $\langle V_{SSUA}, S_{SSUA}, M_{SSUA}, m^0_{SSUA}, R_{SSUA}, L_{SSUA} \rangle$ *such that:*

- V_{SSUA} *is a finite set of attributes of the SSUA;*
- S_{SSUA} *is the universal set of states of the SSUA;*
- M_{SSUA} *is the universal set of state machines (including the root machine and sub-state machines) forming the SSUA (see definition of state machine in Definition 10);*
- $m^0_{SSUA} \in M_{SSUA}$ *is the root state machine of the SSUA;*
- R_{SSUA} *is the universal set of transitions of the SSUA; and*
- L_{SSUA} *is a finite set of labels of the SSUA.*

Let V_{SSUA} be a set of pairs (n_v, t_v) where n_v is a name and $t_v \in \mathbb{T}$ is a type of a given variable $v \in V_{SSUA}$, such that $\mathbb{T} = \{$ Integer, Boolean, Float … $\}$. For example, consider the SSUA (i.e., collision avoidance system) presented in Figure 16 (i.e., Listing 26); the set V_{SSUA} is a singleton set with the following element:

$$V_{SSUA} = \{\ (\text{warningRadius, Integer})\ \}$$

The elements of S_{SSUA} according to Figure 16 is given as follows:

$$S_{SSUA} = \{\text{ CollisionAvoidance, ObjectDetection, Normal, Emergency, Applied, Brake, Released, Transmission, ParkAndNeutral, Reverse, Drive, Park, Neutral, Engine, Off, Idle, Driving }\}$$

A state of an SSUA may be simple, non-orthogonal composite or orthogonal.

Given any state $s \in \mathrm{S_{SSUA}}$, we define a mapping function $\beta: s \rightarrow \mathbb{N}$ to map a state, s to its number of sub machines.

$$\beta(s) = \begin{cases} 0; if\ s\ is\ a\ simple\ state \\ 1; if\ s\ is\ a\ nonorthogonal\ composite\ state \\ n; if\ s\ is\ an\ orthogonal\ composite\ state\ (n > 1) \end{cases} \quad (1)$$

For example, $\beta(\text{Applied}) = 0$, $\beta(\text{ParkAndNeutral}) = 1$, and $\beta(\text{CollisionAvoidance}) = 4$. In essence, "Applied" is a simple state, "ParkAndNeutral" is a non-orthogonal composite state, and "CollisionAvoidance" is an orthogonal composite state.

To simplify the naming of state machines, we introduce operator "$" such that given a non-orthogonal composite state, say A, "$A" is the corresponding state machine of "A". For example, a fully-qualified name of a state machine corresponding to non-orthogonal composite state "ParkAndNeutral" is "SmCollisionAvoidanceTransmissionParkAndNeutral". However, we consider this too long, thus "$ParkAndNeutral" is equivalent to the fully qualified name.M_{SSUA} based on Figure 16 is given as follows:

$$M_{SSUA} = \{\text{ sm, \$Engine, \$ObjectDetection, \$Brake, \$Transmission, \$ParkAndNeutral }\}$$

An SSUA may be simple or hierarchical. This is dependent on the size of M_{SSUA} or the number of composite states of the SSUA. Particularly, an SSUA with no composite state is considered simple. It implies that the size of universal set of state machine is one (i.e., the root state machine). Therefore, for a simple SSUA, the following must hold:

$$(\forall_{s \in S_{SSUA}} \cdot \beta(s) = 0) \rightarrow |M_{SSUA}| = 1$$

And for a hierarchical SSUA, there is at least a state in the universal set that has an embedded state machine. This implies that there are at least two state machines in the universal set of state machines (including the root state machine and others). In particular,

$$(\exists_{s \in S_{SSUA}} \cdot \beta(s) > 0) \rightarrow |M_{SSUA}| > 1$$

Let R_{SSUA} be a set of triples (a, b, c) such that $\forall t = (a_1, b_1, c_1)$, $a_1 \in S_{SSUA}$ and $c_1 \in S_{SSUA}$ are the source and next states of transition t respectively and $b_1 \in L_{SSUA}$.

Then R_{SSUA} is given as a collection of the following expressions for Figure 16:

(Normal, objectDiscovered, Emergency)
(Emergency, [m], Normal)
(Released, applyBrake, Applied)
(Applied, release, Released)
(ParkAndNeutral, selectReverse, Reverse)
(ParkAndNeutral, selectDrive, Drive)
(Reverse, selectPark, Park)
(Park, selectNeutral, Neutral)
(Neutral, selectPark, Park)
(Off, turnOn, Park)
(Idle, accelerate, Driving)
(Driving, release, Idle)
(Park, applyBrake, Idle)
(Release, [k], Applied)

Then L_{SSUA} is the set of the following labels:

L_{SSUA} = { turnOn, applyBrake, accelerate, release, objectDiscovered, selectPark, selectReverse, selectDrive, selectNeutral, [k], [m] }

Definition 10. State Machine in Umple

A state machine of an SSUA in Umple, A (i.e., $A \in M_{SSUA}$) is a 7-tuple $\langle n_A, S_A, l_A, s_A^0, U_{SA}, E_A, O_A, R_A \rangle$;

Where:

- n_A is the name of the state machine;

- S_A is a finite set of top-level states (excluding sub-states) and ($S_A \subseteq S_{SSUA}$);
- $s_A^0 \in S_A$ is the initial state;
- U_{SA} is a universe of states embedded in state machine A (i.e., top-level states and their sub-states).
- l_A is a finite set of labels ($l_A \subseteq L_{SSUA}$) enclosed in A;
- $E_A \subseteq S_{SSUA} \times l_A \times U_{SA}$ defines a set of embedded transitions of A (i.e., $E_A \subseteq R_{SSUA}$); and
- $O_A \subseteq U_{SA} \times l_A \times S_{SSUA}$ defines a set of originating transitions of A (i.e., $O_A \subseteq R_{SSUA}$).

By '*sub-state*' of a state machine, we mean a state whose ancestor is a top-level state of the state machine under consideration irrespective of its depth. The single top-level states of "sm" (see Listing 26) is "*CollisionAvoidance*" and its indirect sub-states are "*Normal*", "*Park*", "*Emergency*" and so on.

We consider G_l as the universal set of guards, E_l as the universal set of events without parameters for the SSUA, and A_l as the universal set of actions of the SSUA such that $l_A \subseteq G_l \times E_l \times A_l$.

Similarly, we consider R_A as the union of sets E_A (see Definition 9) and O_A (see Definition 9). In particular,

$$R_A = E_A \cup O_A$$

Let $\gamma: l_A \rightarrow E_l$ such that $\gamma((g, e, a)) = e$. We also define $L: R_A \rightarrow l_A$ such that $L((s, l, s')) = l$.

Given any state machine M, we define a mapping function $\alpha: M \rightarrow$ { simple, hierarchical } to map a state machine, M to its kind. We express this formally in (2).

$$\alpha(M) = \begin{cases} simple; if \ \forall_{s \in S_M} \beta(s) = 0 \\ hierarchical; if \ \exists_{s \in S_M} \beta(s) \geq 1 \end{cases} \quad (2)$$

For example, “sm” is a hierarchical state machine because there is a top-level state “*CollisionAvoidance*” such that $\beta(CollisionAvoidance) = 4$.

We introduce a mapping function $\rho : M \rightarrow U_{SA}$ such that,

$$\rho(M) = \begin{cases} \bot; if\ M\ is\ a\ root\ machine \\ s; if\ M\ is\ a\ nonroot\ machine\ (where\ s \in U_{SA}) \end{cases} \tag{3}$$

Where:

- s is the parent state of state machine M.
- Z is a root state machine iff $\rho(Z)$ is undefined (denoted with symbol $\bot$).

M is a sub-state machine if and only if there exists a state machine L such that S_L being a set of states of state machine L, $\exists_{k \in S_L}$, $\rho(M) = k$ and $M \sqsubseteq L$ as defined in Section 5.1.1.

5.1.1 Relationships Between State Machines

To facilitate the specification of hierarchical structures, we introduce a pair $\langle \sqsubset, M_{SSUA} \rangle$, such that U_M is the universal set of all state machines (including the root) forming the SSUA and $\sqsubset$ defines a partial-order relation on U_M. $\langle \overleftarrow{\sqsubseteq}, M_{SSUA} \rangle$ defines a reflexive closure on the universal set of state machines. We define a transitive closure relation $\overleftarrow{\sqsubset}$ on the universal set of state machines as a pair $\langle \overleftarrow{\sqsubset}, M_{SSUA} \rangle$. $\langle \sqsubset_{||}, M_{SSUA} \rangle$ defines the parallelism relationship on sub-state machines in the universal set of state machines. The following gives a detailed description of the operators.

- For every pair $\langle I, L \rangle$, such that $L \in U_M$ and $I \in U_M$ expression $I \sqsubset L$ specifies L as an immediate ancestor (or a parent) state machine of I.
- For every pair $\langle I, L \rangle$, such that $L \in U_M$ and $I \in U_M$, expression $I \overleftarrow{\sqsubseteq} L$ implies that L is a member of the set of ancestors of I including I itself.
- For every pair $\langle I, L \rangle$, such that $I \in U_M$ and $L \in U_M$, expression $I \overleftarrow{\sqsubset} L$ implies that L is a member of the set of ancestrally related (i.e., ancestor) state machines to I.

- For every pair $\langle I, L \rangle$, such that $I \in U_M$ and $L \in U_M$, expression $I \sqsubset_{||} L$ implies that L is a member of a set of parallel (or orthogonal) state machines of I. In particular, $I \sqsubset_{||} K$ iff $\exists_{y \in S_H} \cdot \rho(I) = \rho(K) = y, I \sqsubset H, L \sqsubset H$, and $K \in L$.

To facilitate comprehensions of various relationships between state machines, let us consider the relationships between the state machines of Figure 16. This defines the universal set of sub-state machines forming the SSUA (including the root state machine “Sm”).

M_{SSUA} = { Sm, $ParkAndNeutral, $Engine, $ObjectDetection, $Transmission, $Brake }

Then for a binary relation $\sqsubset$, the following statement is valid:

If

A = { $Engine, $Brake, $Transmission, $ObjectDetection } and $a \in A$

Then

$a \sqsubset$ Sm

Similarly, for Figure 16, the following statement is valid.

$ParkAndNeutral $\sqsubset$ $Transmission

For binary relation $\overleftarrow{\sqsubseteq}$, the following holds:

If

A = { Sm, $Transmission, $ParkAndNeutral } and $a \in A$

Then

$ParkAndNeutral $\overleftarrow{\sqsubseteq}\, a$

For binary relation $\overleftarrow{\sqsubset}$ the following statement is valid.

If

A = { Sm, $Transmission } and $a \in A$

Then

$ParkAndNeutral $\overleftarrow{\sqsubset}\, a$

For binary relation $\sqsubset_{||}$ the following statement is valid.

If

A = { $Engine, $Brake, $Transmission, $ObjectDetection }
and $\{a, b\} \subseteq A$ and $a \neq b$

Then

$a \sqsubset_{||} b$ and $b \sqsubset_{||} a$

5.1.2 Relationships Between States

To facilitate the specification of relationships between states, we define a pair $\langle \lhd, S_{SSUA} \rangle$, such that $\lhd$ defines a partial-order relation on the universe of states of the SSUA. $\langle \overleftarrow{\unlhd}, S_{SSUA} \rangle$ defines a reflexive closure on the universe of states of the SSUA. We define a transitive closure relation $\overleftarrow{\lhd}$ on the universe of states of the SSUA as a pair $\langle \overleftarrow{\lhd}, S_{SSUA} \rangle$. The following gives a detailed description of the operators.

- For every pair $\langle j, s \rangle$, such that $j \in S_{SSUA}$ and $s \in S_{SSUA}$ expression $j \lhd s$ implies that s is an immediate ancestor (also known as parent) state of j.
- For every pair $\langle j, s \rangle$, such that $j \in S_{SSUA}$ and $s \in S_{SSUA}$, expression $j \overleftarrow{\unlhd} s$ implies that s is an element of the set of ancestors of j including j itself.
- For every pair $\langle j, s \rangle$, such that $j \in S_{SSUA}$ and $s \in S_{SSUA}$, expression $j \overleftarrow{\lhd} s$ implies that s is an element of the set of ancestrally related (i.e., ancestor) states to j.

To facilitate comprehensions of various relationships between states, let us consider the relationships between the states of Figure 16.

> U_s = { CollisionAvoidance, ParkAndNeutral, Park, Neutral, Drive, Reverse, Off, Idle, Driving, Normal, Emergency, Released, Applied }

Then, for binary relation $\lhd$, the following holds on Figure 16:

> *If*
>
> A = { ParkAndNeutral, Off, Idle, Driving, Released, Applied, Normal, Emergency, Drive, Reverse } and $a \in A$
>
> *Then*
>
> $a \lhd$ CollisionAvoidance

Similarly,

> *If*
>
> A = { Park, Neutral } and $a \in A$
>
> *Then*
>
> $a \lhd$ ParkAndNeutral

For binary relation $\unlhd$, the following holds:

If

A = { CollisionAvoidance, ParkAndNeutral, Park } and $a \in A$

Then

Park $\overleftarrow{\trianglelefteq}$ a

For a binary relation $\overleftarrow{\triangleleft}$, the following holds on Figure 16:

If

A = { CollisionAvoidance, ParkAndNeutral } and $a \in A$

Then

Park $\overleftarrow{\triangleleft}$ a

To facilitate discussions of the underlying concepts, we present the following definitions.

Definition 11. Source State (or from state)

Given a transition t such that t = (a, l, b) then "a" and its descendants are the source states of t. We present a formal definition F(t), a set of source states of transition t in (4).

$$\text{F}(t) = \{ s \,/\, \exists_t \cdot t = (a, _, _) \wedge s \overleftarrow{\trianglelefteq} a \} \quad (4)$$

Definition 12. Destination State (or next state)

Given a transition t such that t = (a, l, b) then "b" is a destination state of t. We present a formal definition of X(t), the destination state of transition in (5).

$$X(t) = b \leftrightarrow t = (_, _, b) \quad (5)$$

Definition 13. In-transition of State

Given a state x such that $x \in U_{SUA}$, the set of in-transitions of x, denoted as $\Delta(x)$ is a collection of all transitions whose next state is x. The formal definition of this statement is given in (6).

$$\Delta(x) = \{ t \mid x = X(t) \} \quad (6)$$

Figure 17 is a symbolic example presented to discuss the from transitions, next transitions and set of in-transitions of states (i.e. simple and composite). Readers should note that we omit submachines corresponding to regions in the example.

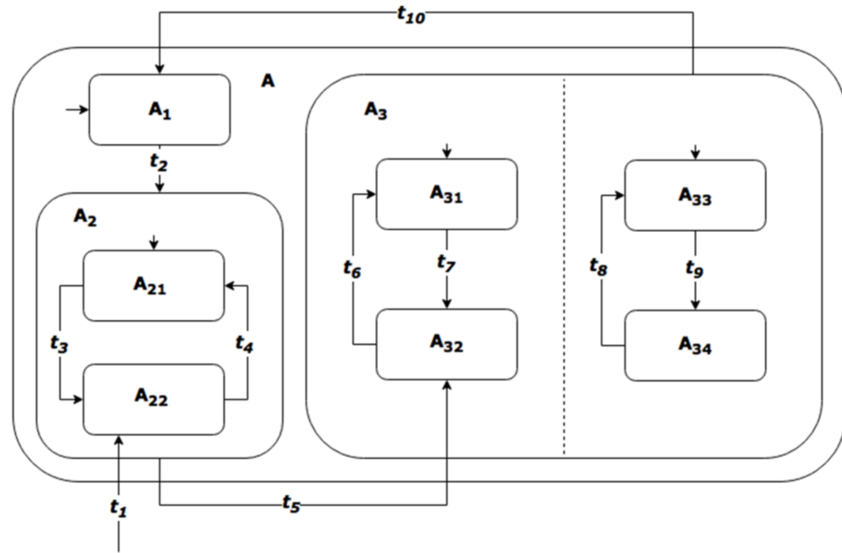


FIGURE 17. SYMBOLIC EXAMPLE TO ILLUSTRATE FROM, NEXT AND IN-TRANSITIONS OF SIMPLE AND COMPOSITE STATES

In Figure 17, the set of transitions (say K) is defined based on their identities as follows:

$$K = \{t_i | \ 1 \leq i \leq 10\}$$

Let G be the set of states in Figure 17 are given as follows:

$$\mathrm{G} = \{A, A_1, A_2, A_{21}, A_{22}, A_3, A_{31}, A_{32}, A_{33}, A_{34}\}$$

such that G is bounded above by A according to relation $\overleftarrow{\trianglelefteq}$

Thus, we can say A is the least upper bound (LUB) or supremum of set G. We can express this in a partially-ordered manner.

$$A_1 \overleftarrow{\trianglelefteq} A$$

$$A_{21} \overleftarrow{\trianglelefteq} A_2 \overleftarrow{\trianglelefteq} A$$

$$A_{22} \overleftarrow{\trianglelefteq} A_2 \overleftarrow{\trianglelefteq} A$$

$$A_{31} \overleftarrow{\trianglelefteq} A_3 \overleftarrow{\trianglelefteq} A$$

$$A_{32} \overset{\leftarrow}{\trianglelefteq} A_3 \overset{\leftarrow}{\trianglelefteq} A$$

$$A_{33} \overset{\leftarrow}{\trianglelefteq} A_3 \overset{\leftarrow}{\trianglelefteq} A$$

$$A_{34} \overset{\leftarrow}{\trianglelefteq} A_3 \overset{\leftarrow}{\trianglelefteq} A$$

We begin the illustration with "from transitions" of a composite state. A "from transition" of a composite state is logically from all its sub-states. The transition is enabled whenever control is transferred to sub-states of the composite state, any controlling guard is satisfied and associated event is received. For example, consider state A_2 of Figure 17. We can refer to t_5 as a from-transition of A_2 since it is of the following form:

$$t_5 = (A_2, _, _)$$

In the same vein, t_5 is a "from-transition" of all descendant states of A_2, given that

$$X = \{A_{21}, A_{22}\} \text{ since } \forall_{x \in X} \cdot x \overset{\leftarrow}{\trianglelefteq} A_2$$

The same principle applies to A_3 and its descendants with respect to t_{10}. The following expression becomes true.

$$t_{10} = (A_3, _, _) \longrightarrow \mathrm{F}(t_{10}) = \{A_3, A_{31}, A_{32}, A_{33}, A_{34}\}$$

In addition to the inherited "from-transitions" of a simple state, a set of transitions originating from the state is included in the set of its "from-transitions". For example, the set of "from-transitions" of A_{34} is given as follows:

$$A_{34} \overset{\leftarrow}{\triangleleft} A \wedge t_8 = (A_{34}, _, _) \wedge\ t_{10} = (A_3, _, _) \longrightarrow A_{34} \in \mathrm{F}(t_{10}) \wedge A_{34} \in \mathrm{F}(t_8)$$

Hence, the set of "from-transitions" of A_{34} is $\{t_8, t_{10}\}$.

As opposed to the method of computing the source(s) of transitions to include the source (*or from*) state and its descendants, the destination (*or next*) of a transition is its next state but not the descendants of the next state. For example, let us consider state A_2 and transition t_2 in Figure 17. Although, A_2 is a composite state but its descendant's states are not in the destination state of t_2.

$$\text{Since } t_2 = (_,_, A_2), \text{ even though } \forall_{x \in \{A_{21},\ A_{22}\}} \cdot x \trianglelefteq A_2$$

The set of "in-transitions" is the set of all transitions into any given state of the SSUA. Hence, the *in-transitions* of $g \in G$ (where G is defined above) are denoted as:

$$\Delta(A) = \emptyset; \Delta(A_1) = \{t_{10}\}$$

$$\Delta(A_2) = \{\ t_2\ \}; \Delta(A_{21}) = \{\ t_4\ \}; \Delta(A_{22}) = \{t_1, t_3\}$$

$$\Delta(A_3) = \emptyset; \Delta(A_{31}) = \{t_6\}; \Delta(A_{32}) = \{t_7\}; \Delta(A_{33}) = \{t_8\}; \Delta(A_{34}) = \{t_9\}$$

Definition 14. Embedded Transitions of State

Given a composite state, say z such that $z \in U_s$. The set of embedded-transitions of z, denoted as $\nabla(z)$ is the set of transitions whose next state is a descendant of z. A formal definition is given in (7).

$$\nabla(z) = \bigcup_{y \trianglelefteq z} \Delta(y) \tag{7}$$

For example, let us consider some states presented in Figure 17 for the purpose of illustrating embedded transitions of various kinds of state.

$$\nabla(A_3) = \{\ t_5, t_6, t_7, t_8, t_9\ \}$$

$$\nabla(A_2) = \{t_1, t_3, t_4\}$$

$$\nabla(A_1) = \emptyset$$

$$\nabla(A) = \{\ t_i \mid 1 \leq i \leq 10\ \}$$

5.2 Transitions in Umple

Umple facilitates the representations of various kinds of transitions as modeling constructs. These includes basic, guarded, auto, reflexive, high-level transitions, and and-cross transitions. A transition may be associated with a guard statement and/or a controlling event. In Figure 18, we

present a visual representation of the relationship between these transitions as we discuss them in this section.

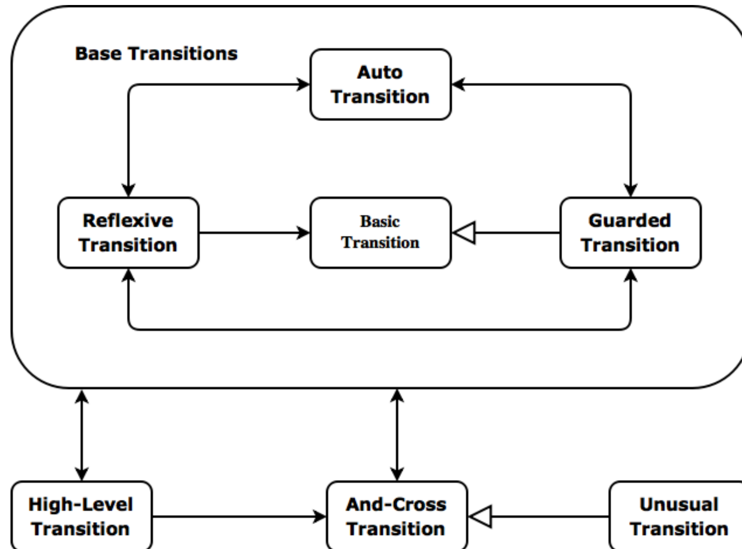


FIGURE 18. VISUAL REPRESENTATION OF RELATIONSHIP BETWEEN TRANSITIONS IN UMPLE

The triangular head *arrows* represent the *isA (generalization)* relationship and *solid arrows* represent the *may-be* relationship. For example, a special kind of *and-cross transition* is what we refer to as an *unusual transition*. We will define these transitions later (see Definitions 16 & 17). Similarly, a *guarded transition* extends a *basic transition* with a *guard statement*. Other specific relationships between transitions are discussed in their respective sections.

5.2.1 Base Transitions

The notion of transition as facilitated in Umple logically maps a source state, an optional event, an optional guard, and a target state. We consider a transition as basic when it maps source state(s) to destination state(s) but with a controlling event. The semantics of auto transitions imply that the transition occurs automatically at the moment control is transferred to its source state or after the execution of an activity whenever the source state has an activity. Auto transitions do not have a controlling event.

A guarded transition is a special kind of basic transition attached with a guard statement that controls its execution. A transition is reflexive when its source state(s) is/are the same as its destination state(s).

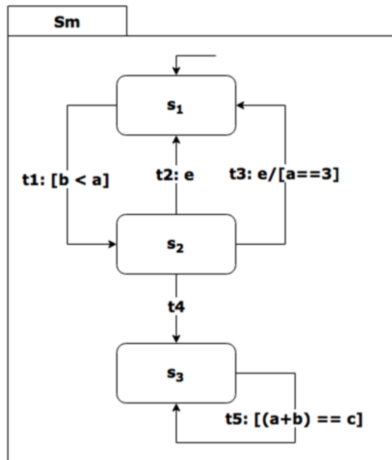


FIGURE 19. VISUAL REPRESENTATION OF BASE TRANSITIONS

We illustrate base transitions by example in Figure 19. Transition t_1 is an auto transition with a guard statement; while t_4 is an auto transition without a controlling guard statement. An example of a basic transition is t_2; while t_3 is an example of basic transition but with a controlling guard statement. Transition t_5 is an example of a reflexive transition with a controlling guard statement.

5.2.2 High-Level Transitions

Umple facilitates the specification of high-level transitions. We therefore present the formalization of the notions of high-level transitions in this section.

Definition 15. High-Level Transition

Transition t of an SSUA is a high-level transition if the source state of t is a composite state. A formal definition of $H(s)$, the set of high-level transitions of a state s is given as follows.

Given s, such that $\beta(s) \geq 1$, then $H(s) = \{ t \mid t = (s, _, _) \}$

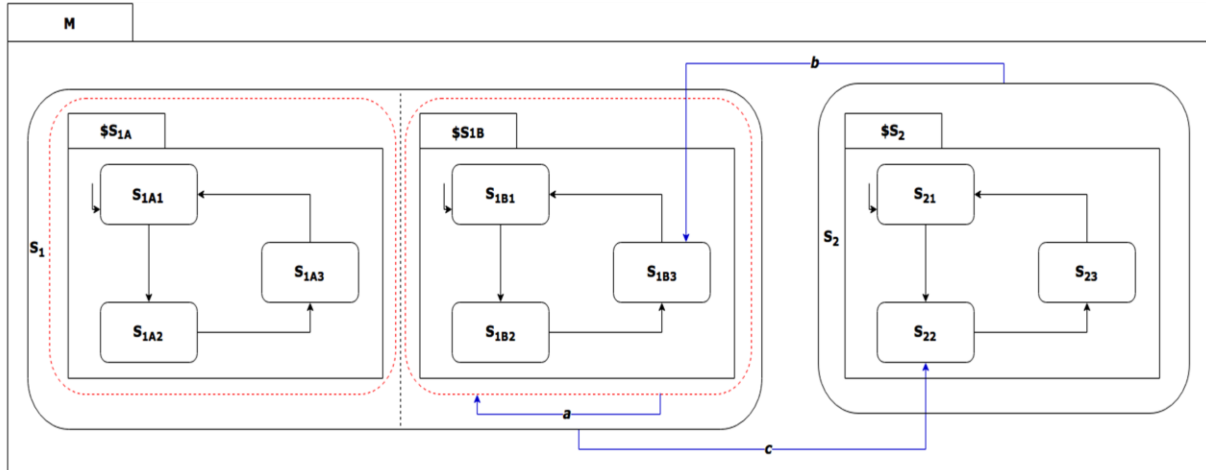


FIGURE 20. EXAMPLES OF HIGH-LEVEL TRANSITIONS

In Figure 20, transitions a, b, c are *high-level* transitions. Transition a will execute whenever a state:

$$x \in \{S_{1B1}, S_{1B2}, S_{1B3}\}$$

is active and its controlling guard evaluates to 'true' and event is received.

Transition b will execute whenever a state:

$$y \in \{S_{21}, S_{22}, S_{23}\}$$

is active and its controlling guard evaluates to 'true' and event is received. Transition c will be executed whenever a state:

$$z \in \{S_{1A1}, S_{1A2}, S_{1A3}, S_{1B1}, S_{1B2}, S_{1B3}\}$$

is active and its controlling guard evaluates to 'true' and event is received.

5.2.3 And-Cross Transitions

According to the OMG specification for UML [85], an and-cross transition is a transition from one region to another in the same immediate enclosing composite state. In other words, transitions

whose sources and destinations states are located in distinct parallel regions of orthogonal states [61].

Definition 16. And-Cross Transitions

A transition t from a region to another region in the same enclosing composite state is referred to as and-cross transition. A formal definition of the set of and-cross transitions of a given state s is presented as follows.

$$\text{Given } s \text{ such that } \beta(s) \geq 2, \text{ then}$$

$$a(s) = \{ t \mid t = (a, _, b) \text{ such that } \exists_{M,N}\ s = \rho(M) = \rho(N),\ M \sqsubset_{||} N,\ \ a \in U_{SM} \wedge b \in U_{SN}, \}$$

For illustration, we present a model with and-cross transitions in Figure 21. Transitions a, b, c are and-cross transitions but b is a special kind of and-cross transition.

According to Faghih and Day [86], b is referred to as an *unusual transition.* Unusual *transitions* are and-cross transitions whose sources and destinations are sub-states of an orthogonal region.

Definition 17. Unusual Transitions

An and-cross transition t is an unusual transition if the source(s) and the destination states of t are embedded states of an orthogonal region. We express the set $\mathfrak{u}(s)$ of unusual transition of a parallel state as:

$$\mathfrak{u}(s) = \{ t \mid \exists_{s'} \cdot t \in a(s') \wedge s' \overleftarrow{\lhd} s \} \quad (8)$$

$$\text{where } \beta(s') \geq 2 \text{ and } \beta(s) \geq 2$$

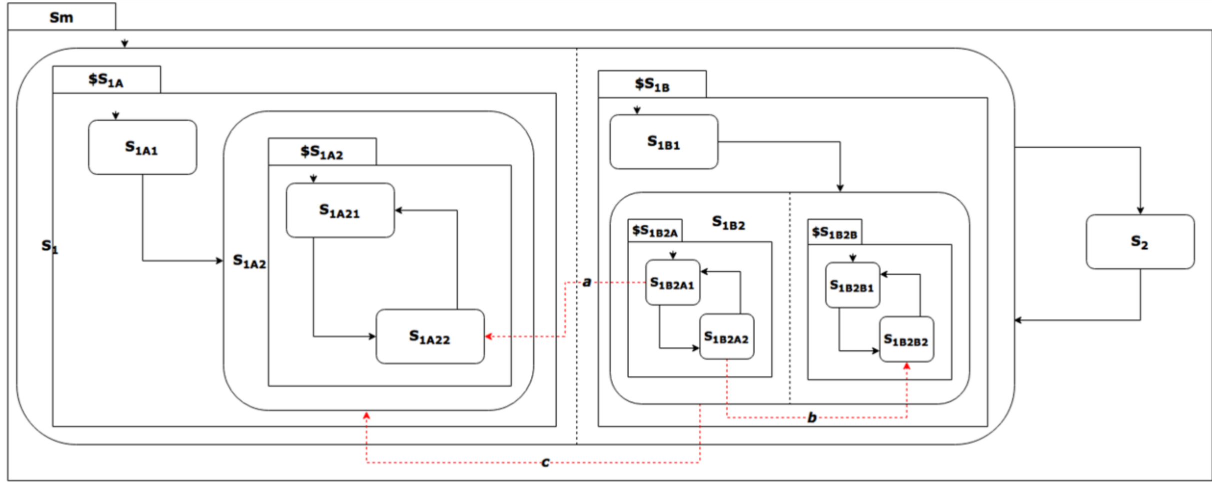


FIGURE 21. MANIFESTATIONS OF AND-CROSS TRANSITIONS

5.3 Managing States of Umple State Machines

A state in Umple will remain inactive unless control is transferred to it or it is activated by default. There is considerable complexity in the analysis of which states are enabled when transitions occur in complex state machines. We present the formalization of the process of enabling states of a state machine expressed in Umple. Particularly, we are interested in simple and composite states (i.e. orthogonal and non-orthogonal). We formally represent the enabling set of transitions for state s as $\varphi(s)$.

Definition 18. Enabling Transitions of a State

All transitions whose execution enable (or activate) a state are the enabling transitions of the state. The state becomes active at the step the transition completely executes. We define the set of enabling transitions of state x, denoted as $\varphi(x)$ as:

$$\varphi(x) = \Delta(x) \cup \nabla(x) \tag{9}$$

Definition 19. Step of Execution

A step S of an SSUA is a triple $\langle C_{S^0},\ \tau_S, C_{S^n}\rangle$, such that C_{S^0} and C_{S^n} are initial and final global configurations (see Definition 1) and τ_S is a set of executing transitions taking the SSUA from C_{S^0} to C_{S^n}.

We consider τ_S as a set but not a sequence of transitions because in a concurrent state machine, more than a transition can execute simultaneously. This may result in inconsistency but we rely on our method (see Section 6.1) to discover nondeterminism to resolve these issues, since inconsistency problem can be reduced to nondeterminism.

Definition 20. Micro-Step of Execution

Given a step, $S = \langle C_{S^{k-1}},\ \tau_S, C_{S^k} \rangle \cdot k \geq 1$ *then S is a micro-step whenever:*

a) $\#\tau_S \leq 1$ *(i.e., the cardinality of set* τ_S*), given that* $t \in \tau_S$ *is neither a high-level nor an and-cross transition; or*
b) $C_{S^{k-1}} \rightarrow C_{S^k}$ *occurs by default activation.*

Definition 21. Macro-Step of Execution

Given a step, $S = \langle C_{S^0},\ \tau_S, C_{S^n} \rangle \cdot n \geq 1$ *then S is a macro-step whenever:*

a) $\#\tau_S \geq 1$*;*

b) $\exists_{S^0,\ S^1 \ldots S^n}$ *such that:*

$$S^0 = \langle C_{S^0},\ \tau_{S^0}, C_{S^1} \rangle$$

$$S^n = \langle C_{S^{n-1}},\ \tau_{S^{n-1}}, C_{S^n} \rangle$$

with $\tau_S = \bigcup_{i\,=\,0..n} \{\tau_{S^i}\}$*; and*

c) $S^0 \ldots S^n$ *are micro steps.*

5.3.1 Execution Semantics of Umple

In this section, we will discuss the execution semantics of Umple language for the purpose of evaluating condition(s) and action(s) over a macro-step.

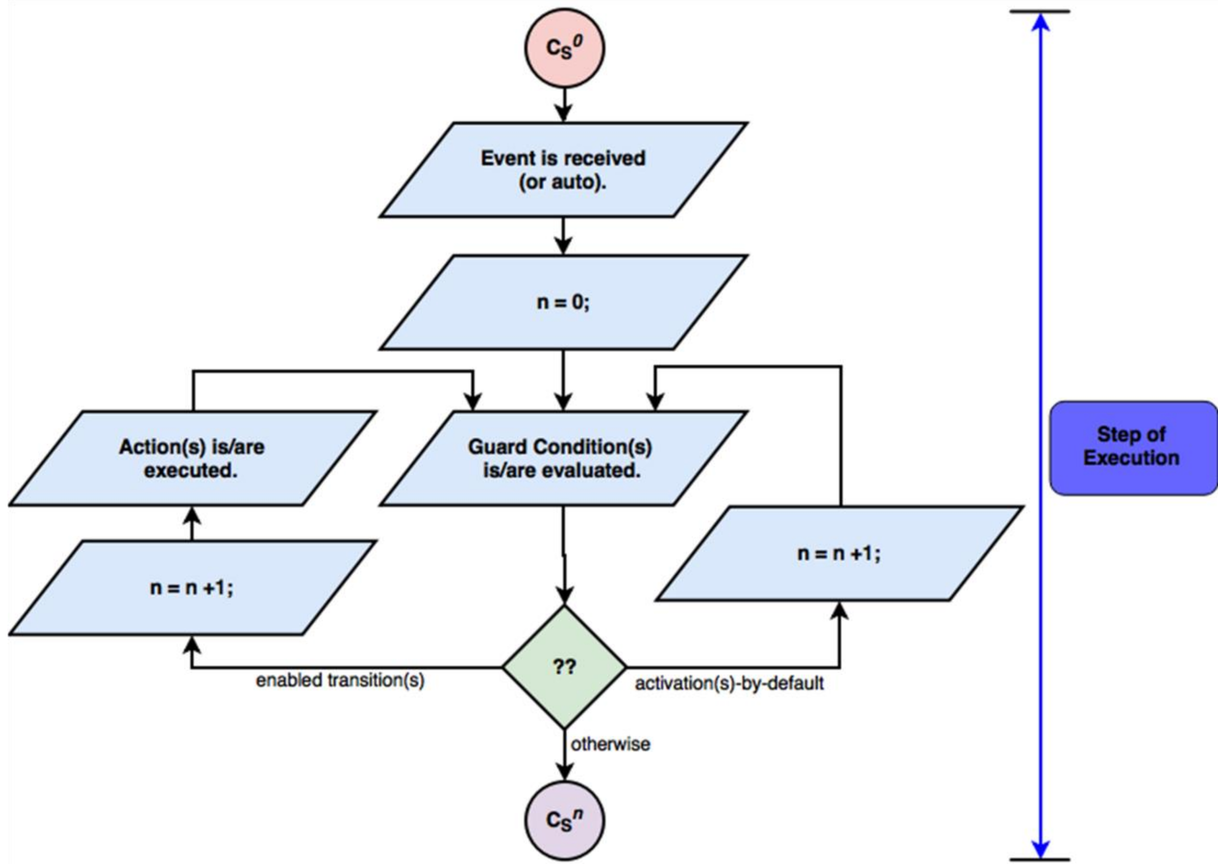


FIGURE 22. FLOW OF EXECUTION IN UMPLE

In Figure 22, the flowchart presented illustrates execution semantics of action(s) and guard condition(s) as they are being evaluated over a macro-step in our work. We consider C_{S^0} and C_{S^n} as two consecutive stable configurations of the SSUA such that n is the step index. An SSUA is in a "stable configuration" whenever it is ready to accept an external trigger (e.g., event). The symbol "**??**" implies a decision point during execution such that after condition evaluation, either a transition is enabled or there is activation-by-default. If none of these is the case, the system proceeds to a stable configuration which may be new if $n > 0$, or remain unchanged whenever $n = 0$. This remains unchanged whenever there is neither transition enabled as a result of the trigger received nor activation by default.

For example, while the SSUA is in a stable configuration C_{S^0} suppose an external trigger (i.e., event) is observed, all the guard condition(s) of transitions with the trigger received are evaluated. If the evaluation leads to an enabled transition, then the step index is incremented by "1" and the associated action(s) are executed. As a consequence of the action(s) executed at the just concluded phase (if any), a set of transitions may be enabled which in-turn leads to further executions and increments in step indexes. This continues in a run-to-completion manner as defined in the UML semantics. A similar run occurs for activation-by-default.

5.3.2 Enabling Simple States

The process of computing the set of transitions that enable a simple state is straightforward. All the *in-transitions* of the state under consideration is sufficient to enable it because its embedded set of transitions is empty. From equation (9) we have:

$$\varphi(x) = \Delta(x);$$

$$\text{for a simple state } x \text{ (or } \beta(x) = 0\text{) since } \nabla(x) = \emptyset.$$

For example, let us consider enabling transitions set of simple states in Figure 16. These are as follows:

{Off, Idle, Driving, Park, Neutral, Drive, Reverse, Released, Applied, Emergency and Normal}

$$\varphi(\text{Off}) = \emptyset, \varphi(\text{Idle}) = \{ t_{11}, t_{13} \}, \varphi(\text{Driving}) = \{ t_{12} \}, \varphi(\text{Park}) = \{ t_7, t_{10} \}$$

$$\varphi(\text{Neutral}) = \{ t_9 \}, \varphi(\text{Drive}) = \{ t_6 \}, \varphi(\text{Reverse}) = \{ t_5 \}, \varphi(\text{Released}) = \{ t_4 \}$$

$$\varphi(\text{Applied}) = \{ t_3, t_{15} \}, \varphi(\text{Emergency}) = \{ t_1 \}, \varphi(\text{Normal}) = \{ t_2 \}$$

5.3.3 Enabling Composite States

Unlike simple states, the transition set to enable composite states (i.e. orthogonal and non-orthogonal) is not straightforward. The set of in-transitions for a composite state is insufficient. Hence, to enable a composite state, the set of sufficient transitions is a union of its in-transitions and embedded transitions. From equation (9) we have:

$$\varphi(x) = \Delta(x) \cup \nabla(x) \quad (10)$$

whenever $\beta(x) > 0$ (or x is a composite state) since $\nabla(x)$ may not necessarily be empty.

In particular, $\nabla(x)$ will be empty whenever x is a composite state with sub-machine(s) with one state (i.e., initial) with an empty set of in-transitions. A typical example of this case is presented in Figure 23.

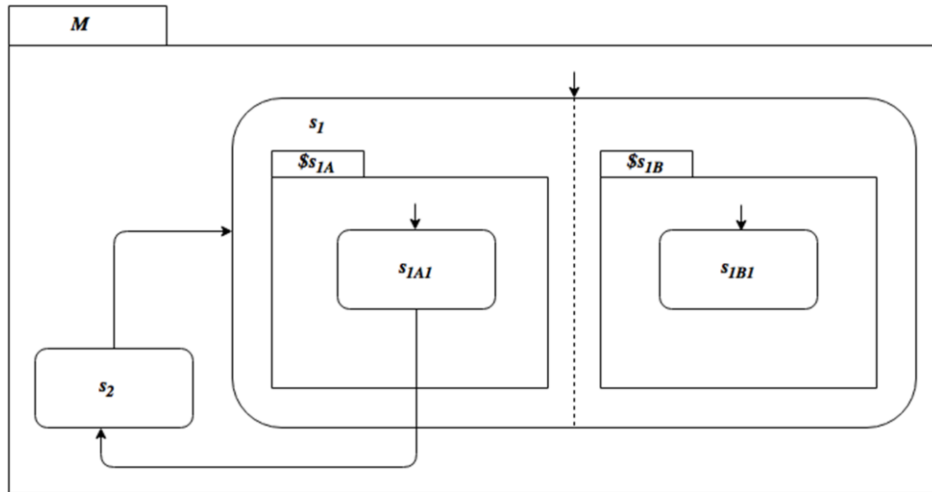


FIGURE 23. COMPOSITE STATE WITH EMPTY SET OF EMBEDDED TRANSITIONS

Furthermore, we illustrate other cases of enabling transitions of composite states with "*CollisionAvoidance*" and "*ParkAndNeutral*" in Figure 16. "*CollisionAvoidance*" is a typical example of orthogonal composite state and "*ParkAndNeutral*" is an example of non-orthogonal composite state.

Let us take "ParkAndNeutral" for the purpose of illustrating the computation of the set of enabling transitions of a non-orthogonal composite state.

$$\varphi(ParkAndNeutral) = \Delta(ParkAndNeutral) \cup \nabla(ParkAndNeutral)$$

But,

$$\Delta(\text{ ParkAndNeutral }) = \emptyset$$

We compute the set of embedded states of “ParkAndNeutral” as follows:

$$\nabla(\text{ ParkAndNeutral }) \bigcup_{x \trianglelefteq X} \Delta(\text{ x }); \text{where: } X = \{\text{“Park”, “Neutral”}\}$$

$$\nabla(\text{ ParkAndNeutral }) = \Delta(\text{ Park }) \cup \Delta(\text{ Neutral })$$

$$\Delta(\text{Park}) = \{t_7, t_{10}\}; \text{ and } \Delta(\text{ Neutral }) = \{ t_9 \}$$

$$\nabla(\text{ ParkAndNeutral }) = \{ t_7, t_9, t_{10}\}$$

$$\varphi(\text{ ParkAndNeutral }) = \emptyset \cup \{ \text{t}_7, \text{t}_9, \text{t}_{10} \} = \{ \text{t}_7, \text{t}_9, \text{t}_{10} \}$$

On the other hand, let us consider “CollisionAvoidance” to illustrate the case of an orthogonal composite state.

$$\varphi(\text{ CollisionAvoidance }) = \Delta(\text{ CollisionAvoidance }) \cup \nabla(\text{ CollisionAvoidance })$$

Such that:

$$\Delta(\text{ CollisionAvoidance }) = \emptyset$$

$$\nabla(\text{ CollisionAvoidance }) = \bigcup_{x \trianglelefteq X} \Delta(\text{ x })$$

Where:

$$X = \{\text{Off, Idle, Driving, Released, Applied, Normal, Emergency, ParkAndNeutral, Reverse, Drive}\}$$

Since “ParkandNeutral” is a composite state, we recursively compute its enabling transition set in the same manner as “CollisionAvoidance”.

If Y = {Park, Neutral} such that $Y \trianglelefteq$ ParkAndNeutral then $x \in X \cup Y$. Therefore, we have the enabling transition set for CollisionAvoidance example is given as:

$$\varphi(\text{ CollisionAvoidance }) = \{ t_i \mid 1 \leq i \leq 15 \}$$

Where:

$$\Delta(\text{ Off }) = \emptyset$$

$$\Delta(\text{ Idle }) = \{ t_{11}, t_{13}, t_{14}\}$$

$$\Delta(\text{ Driving }) = \{ t_{12} \}$$

$$\Delta(\text{ Released }) = \{ t_4 \}$$

$$\Delta(\text{ Applied }) = \{ t_3, t_4 \}$$

$$\Delta(\text{ Normal }) = \{ t_2 \}$$

$$\Delta(\text{ Emergency }) = \{ t_1 \}$$

$$\Delta(\text{ Park }) = \{ t_7, t_{10} \}$$

$$\Delta(\text{ ParkAndNeutral }) = \emptyset$$

$$\Delta(\text{ Neutral }) = \{ t_9 \}$$

$$\Delta(\text{ Reverse }) = \{ t_5 \}$$

$$\Delta(\text{ Drive }) = \{ t_6 \}$$

5.3.4 Default Activation

The semantics of state machines requires the activation of start states by default. In particular, a start state (i.e., simple or composite) is enabled in a *micro-step i* whenever its parent state becomes activated (see Definition 20) in the previous *micro-step i-1* [85].

Definition 22. Active State of a State Machine

Given a state machine M and its top-level state s *such that* $s \in S_M$ *then* s *is an active state of M at micro-step (or step) 'i' if control is transferred to it or its sub-state. By "transfer of control", we mean a transition leading to the state or its sub-state executes.*

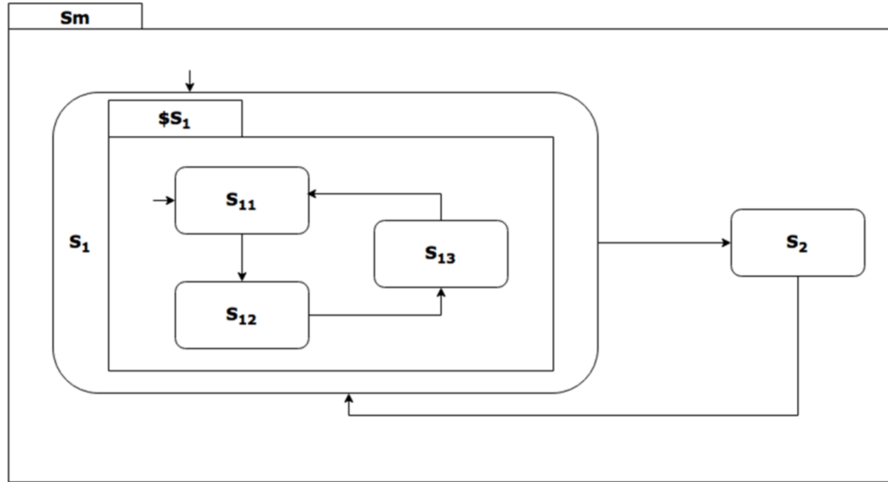


FIGURE 24. ACTIVATION-BY-DEFAULT FOR SIMPLE AND NON-ORTHOGONAL COMPOSITE STATE

The global configuration maps state machine M to state s at step s^i. We denote this as $\upsilon(\$A, s^i)$ such that "υ" is an environment operator which returns the active state of "$A" at step '$i$'. In addition to the set of transitions that enable states (i.e., simple and composite), states of a state machine can be enabled by default. We broadly discuss *activation-by-default* in two categories. These include: root state machine (i.e., $\rho(\ M\ ) = \bot$) and sub-state (or non-root) machines (i.e. $\rho(M) \neq \bot$); where M corresponds to a state machine.

In Figure 24, we present a symbolic state machine diagram to illustrate *activation-by-default*. It is a hierarchical system where Sm and $\$S_1$ are root and non-root state machines respectively. An arrow into a state without a source state indicates the start state of the machine under consideration.

The relationship between S_1 and S_{11} is expressible as $S_{11} \lhd S_1$. We skip the labels on transitions for the sake of brevity.

Recall (see Section 2.1.11) that Umple adds a "*null*" to the states of every sub-state machine (i.e. non-root state machines). This allows the disabling of such machine whenever control is yet to be transferred to its state and when control is transferred out of its state.

The states of these machines are given as follow:

$$S_{Sm} = \{S_1, S_2\}$$

$$S_{\$S_1} = \{S_{11}, S_{12}, S_{13}, null\}$$

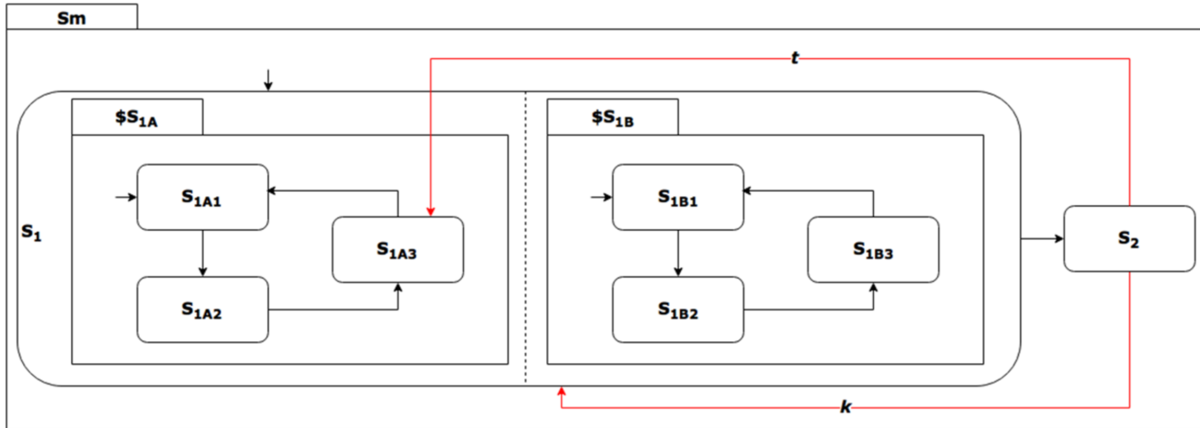


FIGURE 25. ACTIVATION-BY-DEFAULT FOR PARALLEL STATES

Recall that $\upsilon(\$A, s^i)$ returns the active state of "$A" at step s^i. Hence, at the initial state (i.e., step s^0); the active state of Sm is $\upsilon(Sm, s^0) = S_1$ and the active state of $\$S_1$ is $\upsilon(\$S_1, s^0) = null$. In particular, the *global configurations* (see Definition 1) of the SSUA (i.e., Figure 24) at steps s^0, s^1, are given as follows:

$$\langle\langle Sm, \$S_1\rangle, \langle S_1, null\rangle, s^0, \langle\langle_\rangle, \langle_\rangle\rangle\rangle; \langle\langle Sm, \$S_1\rangle, \langle S_1, S_{11}\rangle, s^1, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

Hence, we will like to note that the initial state of the root state machine is activated at step s^0 and remain active throughout the entire system life cycle. Similarly, for non-root state machine (or sub-state machine), the initial state is activated in a step immediately after the step its parent state is activated (e.g. s^i). For instance, if the parent state of a sub-state machine becomes active at step s^i, the sub-state machine becomes active at step s^{i+1}.

This is not limited to non-orthogonal cases. The start states of parallel sub-state machines (or regions) are activated by default whenever control is transferred to their parent states. For example, states S_{1A1} and S_{1B1} of Figure 25 will be activated by default if transition k executes at step s^1 of the overall execution of the *SSUA*.

The possible global configurations at steps s^0 and s^1 of the SSUA given in Figure 25 are as follows:

$$\langle\langle Sm, \$S_{1A}, \$S_{1B}\rangle, \langle S_1, null, null\rangle, s^0, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

$$\langle\langle Sm, \$S_{1A}, \$S_{1B}\rangle, \langle S_1, S_{1A1}, S_{1B1}\rangle, s^1, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

At any other step s^i, suppose we have configuration:

$$\langle\langle Sm, \$S_{1A}, \$S_{1B}\rangle, \langle S_2, null, null\rangle, s^i, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

In the next steps s^{i+1} and s^{i+2}, after the execution of transition t the system will assume the configuration given as follows:

$$\langle\langle Sm, \$S_{1A}, \$S_{1B}\rangle, \langle S_1, S_{1A3}, null\rangle, s^{i+1}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

$$\langle\langle Sm, \$S_{1A}, \$S_{1B}\rangle, \langle S_1, S_{1A3}, S_{1B1}\rangle, s^{i+2}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

The variations between configurations above are based on the fact that we separated concerns but delegate responsibilities to the parent state for the purpose of default activation. In particular, the target state (S_{1A3}) becomes active as well as its parent state (i.e. S_1) at the same step (i.e. s^{i+1}) because transition t is an embedded transition of S_1 and an in-transition of S_{1A3}.

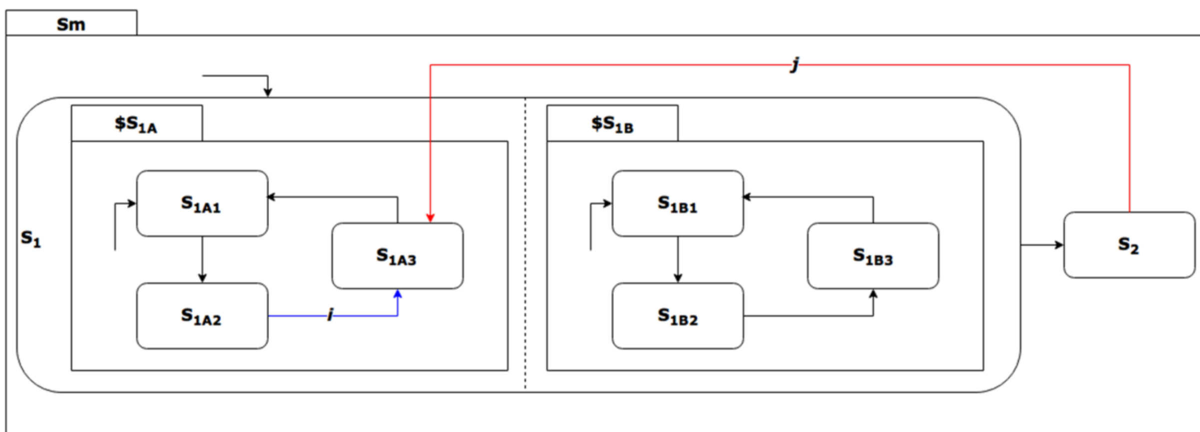


FIGURE 26. DEMONSTRATING COMPLEXITY OF TRANSITIONS INTO CONCURRENT STATE

The execution of a pair of transitions $\langle i, j \rangle$ in Figure 26 into state S_{1A3} affect the global configuration of the SSUA differently whenever the controlling event or guard differs on the transitions. The execution of transition i will not change the global configuration with respect to $\$S_{1B}$ but the execution of j will change the configuration of the SSUA with respect to $\$S_{1B}$. This is due to the sources of these transitions. The source of transition j is external while the source of transition i is internal. From another point of view, while a transition into an orthogonal state initializes all the regions of the orthogonal state, a transition into a state of a region of an orthogonal state initializes every other region of the state except the region of the target state. The target region is set to the target state; while other regions of the orthogonal states are set to their initial states. Thus, the complexity introduced by the semantics of transitions into regions of an orthogonal composite state raises a question of interest at this point is:

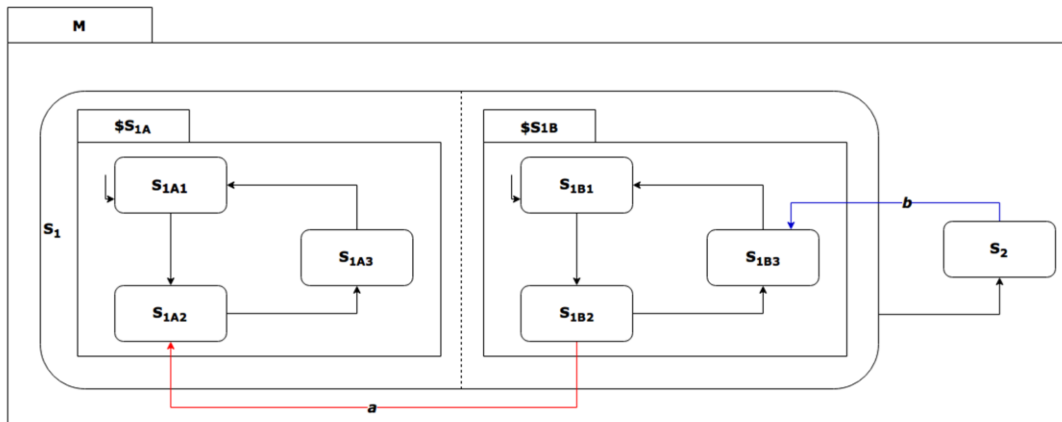


FIGURE 27. ACTIVATION-BY-DEFAULT FOR NON-ORTHOGONAL COMPOSITE STATE

Research Question 2

What sufficient constraint will activate an initial state of a sub-state machine by default or initialize a region?

Proposition 1. Sufficient Constraint for Activation-By-Default

Given that state s is the parent state of a sub-state machine R, and that s contains a state of state machine M such that:

$$\rho(R) = s,\ s \in S_M \text{ and } R \sqsubset M$$

R is initialized to its initial state in the next step (i.e., s^j) if and only if at s^{j-1}, R was inactive and its parent state was active. A formal description is given in expression (1).

$$\langle\langle \boldsymbol{M}, \boldsymbol{R}\rangle, \langle \boldsymbol{s}, \boldsymbol{null}\rangle, \boldsymbol{s}^{\boldsymbol{j-1}}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle \rightarrow \langle\langle \boldsymbol{M}, \boldsymbol{R}\rangle, \langle \boldsymbol{s}, \boldsymbol{s}_{\boldsymbol{R}}^{\boldsymbol{0}}\rangle, \boldsymbol{s}^{\boldsymbol{j}}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle \qquad \text{Exp. (1)}$$

Discussion

We consider two broad cases to discuss satisfiability of the constraint proposed in expression (1) for the purpose of default activation. These include orthogonal and non-orthogonal regions.

For a non-orthogonal composite state, the only set of transitions that can activate a sub-state machine by default are *in-transition* of its parent state (see Figure 27). Similarly, if the parent state is the initial state of the root machine of the SSUA then the parent state is initialized at the beginning of execution. Hence, if activation occurs at step s^j then it is guaranteed at step s^{j-1}, the LHS of expression (1) holds for these scenarios. That is, the parent state is enabled and the region is inactive.

The same principle we applied to the non-orthogonal composite state is applicable to orthogonal composite states. However, all the regions enclosed in an orthogonal composite state are activated in a step s^j after the parent state is enabled at step s^{j-1}. It becomes more complex to handle when a transition is an *in-transition* of a state in a region.

Two kinds of transitions in this category introduce such complexity. Transitions such as b and and-cross transition such as a are examples in this category (see Figure 27). They are enabling transitions of S_1 (i.e. $\{a, b\} \subseteq \varphi(S_1)$). Hence, whenever they execute at any step s^j, it is certain that S_1 becomes enabled. In addition, parallel regions to the region of the target state will be inactive at this step. For example, when b executes, S_{1B3} and S_1 becomes active at the same step because b is an in-transition of S_{1B3} and an embedded transition of S_1. At this point, it is guaranteed that \$$S_{1A}$ is not enabled; so the left-hand-side (LHS) of expression (1) holds.

However, since the parent state of $\$S_{1A}$ (i.e., S_1) is activated because of the execution of transition b, if $\$S_{1A}$ relies on the activeness of S_1 then it is guaranteed that $\$S_{1A}$ will become active at the immediate next step. Thus, the *right-hand-side* (RHS) of expression (1) holds because of the LHS.

The semantics of a (i.e., and-cross transition) introduces more complexity because its source and destination states are in parallel regions of an orthogonal state. Following Harel's state chart semantics for and-cross transitions [61][87], we are mandated to exit and re-enter the enclosing orthogonal state. Then, regions of the orthogonal state except the target are reinitialized to their initial states; while the target region is set to the target state. By exiting and re-entering the orthogonal state reduces the problem to the case demonstrated by b; thus, LHS and RHS of expression (1) also hold for regions of the state except the target.

5.4 Formulating the Disabling Transitions Set for Sub-State Machines

We have noted that a state (or sub-state) machine (or region) is enabled whenever one of its top-level states is active. A top-level state is enabled whenever control is transferred to it or whenever control is transferred to any of its descendants (they become active). We begin by giving a formal definition of a set of enabling transitions of any given state machine (or sub-state machine or region). This rule applies not only to sub-state machines but also the root state machine of the SSUA. We also present our method to formally disable (or deactivate) sub-state machines (or regions) of an SSUA. Specifically, we compute the set of necessary transitions to disable a sub-state machine irrespective of the underlying complexity inherent to the semantics of some complex transitions (e.g. and-cross, unusual transition, etc.).

In the following subsections, our goal is to derive a consistent set of transitions for disabling any given sub-state machine. By "consistent set", we meant a set of transitions whose execution truly disables the sub-state machine (or region) under consideration. For readability's sake, we present a generalized formula for computing this set of transitions but defer discussion on its derivation to the following subsections.

Given that:

$$(\forall_{A}, \exists_{B} \text{ such that } A \sqsubset B)$$

then,

$$\mathrm{D(A)} = (\mathrm{R_B} \backslash \nabla(\rho(\mathrm{A}))) \cup \mathrm{DH(A)} \cup \mathscr{b}(\mathrm{A}, \rho(\mathrm{A})) \tag{11}$$

where:

$$\mathrm{DH(A)} = (\ (\bigcup\nolimits_{\rho(\mathrm{A}) \overleftarrow{\trianglelefteq} \mathrm{k}} \mathrm{H(k)}) \backslash \mathrm{IH(A)}\ )$$

$$\mathrm{IH(A)} = \{\ \mathrm{t} \mid \exists_{\mathrm{k}} \cdot \mathrm{t} \in \bigcup\nolimits_{\rho(\mathrm{A}) \overleftarrow{\trianglelefteq} \mathrm{k}} \mathrm{H(k)} \text{ and } \mathrm{t} = (_, _, \mathrm{s}') \text{ with } \mathrm{s}' \overleftarrow{\ntrianglelefteq} \rho(\mathrm{A})\ \}$$

$$\mathscr{b}(\mathrm{A}, \rho(\mathrm{A})) = \{\ \mathrm{t} \mid \mathrm{t} \in \mathscr{a}(\rho(\mathrm{A})) \wedge \exists_{s'} \cdot \mathrm{t} = (_, _, s') \text{ with } s' \notin \mathrm{U_{SA}}\}$$

We discuss the derivation of sets DH(A) and IH(A) in subsection 5.4.2 and $\mathscr{b}$(A, ρ(A)) in subsection 5.4.3.2.

5.4.1 Enabling Transitions of a State Machine

Recall that a state machine is enabled by any set of transitions that enable its top-level states because top-level states are enabled at the same step their descendants' states are enabled. Hence, the set of enabling transitions of a state machine A, denoted as E_A can be defined as follows:

$$\mathrm{E_A} = \bigcup\nolimits_{x \overleftarrow{\trianglelefteq} y} \varphi(\ \mathrm{x}\ ) \cdot \forall_{\mathrm{y} \in \mathrm{S_A}}$$

5.4.2 Disabling Non-Parallel Sub-State Machine

In this section, we will discuss the process of disabling (deactivating) non-parallel sub-state machines. Since a sub-state machine is enabled by the set of in-transitions and embedded transitions of its top-level states, it is reasonable to conclude that the set of disabling transitions of the region is the set difference of the universal set of transitions and the enabling transitions of the region. However, since the SSUA may be extremely large, scalability becomes an important property of our formulations. In the worst case, the cost of computing the set difference is $O(m \cdot n)$, where n is the number of enabling transitions of the region, m is the number of transition in the universal set of transitions of the SSUA and $m \geq n$. For clarity, the complexity is derivable from the cost of comparing elements of the two sets. This raises an important question:

Research Question 3

What minimal set of transitions of the SSUA that would need to be triggered to disable a sub-state machine (or region) of an SSUA?

Proposition 2. Sufficient subset of transitions of the SSUA for disabling a non-parallel sub-state machine

Given state machines A and B of a SSUA, such that $A \sqsubset B$ (i.e., B is the immediate ancestor or parent of A). A minimal set of transitions that if triggered would disable sub-state machine A, denoted as $D(A)$ is given as follows:

Given that

$$(\forall_{\mathrm{A}}, \exists_{\mathrm{B}} \text{ such that } \mathrm{A} \sqsubset \mathrm{B} \text{ and } \beta(\rho(\mathrm{A})) = 1)$$

then

$$\mathrm{D(A)} = (\ (\mathrm{R_B} \cup \mathrm{DH(A)}) \backslash \mathrm{E_A}\ ) \tag{12}$$

where:

$$\mathrm{DH(A)} = (\ (\textstyle\bigcup_{\rho(\mathrm{A}) \overleftarrow{\trianglelefteq} \mathrm{k}} \mathrm{H(k)}) \backslash \mathrm{IH(A)}\ ) \tag{13}$$

$$\mathrm{IH(A)} = \{\ \mathrm{t} \mid \exists_{\mathrm{k}} \cdot \mathrm{t} \in \textstyle\bigcup_{\rho(\mathrm{A}) \overleftarrow{\trianglelefteq} \mathrm{k}} \mathrm{H(k)} \text{ and } \mathrm{t} = (_, _, \mathrm{s'}) \text{ with } \mathrm{s'} \overleftarrow{\ntrianglelefteq} \rho(\mathrm{A})\ \} \tag{14}$$

(Illustration.)

For illustration and discussion purposes, we present an SSUA with features relevant to the concept under discussion in Figure 28. It also illustrates a case of potential inconsistency that may result from activation-by-default and high-level transitions. Dashed-round boxes indicate steps of executions and dotted lines indicate step links from the active object (i.e., state or transition) to its corresponding step it becomes active. For example, at step s^2, states D (i.e., by executing t_3) and C_1 (i.e., by default) may become active when t_3 and C were active at step s^1.

The relationships between the state machines of the SSUA presented in Figure 28 are expressed as the following sequences:

$$\$\mathrm{C} \sqsubset \$\mathrm{B};\ \$\mathrm{B} \sqsubset \$\mathrm{A}$$

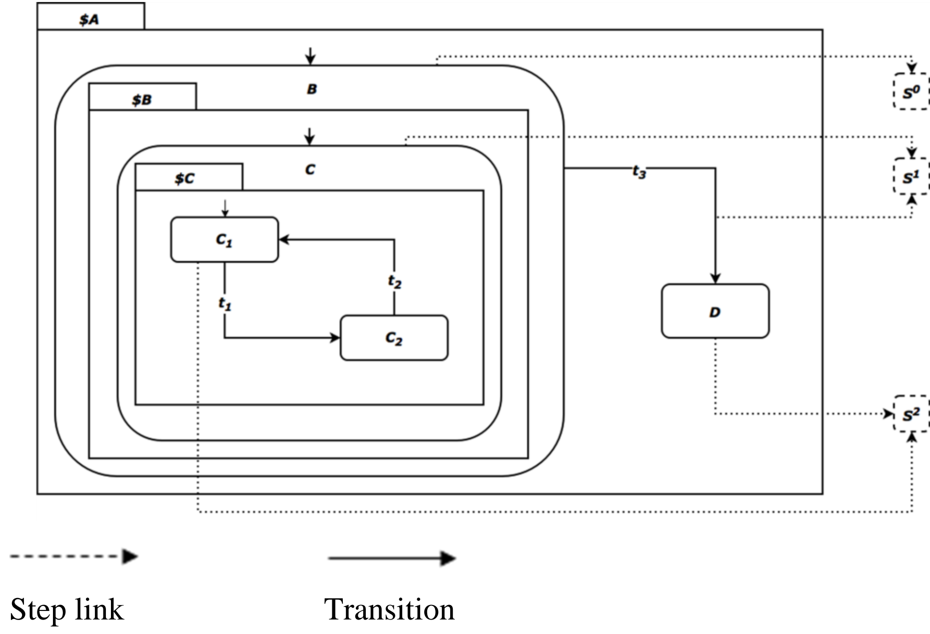


FIGURE 28. ILLUSTRATING INCONSISTENCIES RESULTING FROM ACTIVATION-BY-DEFAULT AND HIGH-LEVEL TRANSITIONS

Let us consider, the disabling transition set for sub-state machine $\$C$. This set can be described as follows:

$$\mathrm{D}(\$\mathrm{C}) = \left(\mathrm{R}_{\$\mathrm{B}} \cup \mathrm{DH}(\$\mathrm{C})\right) \setminus \mathrm{E}_{\$\mathrm{C}} = \{\, \mathrm{t}_3 \,\}$$

since

$$\$\mathrm{C} \sqsubset \$\mathrm{B},\ \beta(\rho(\$\mathrm{C})) = 1, \mathrm{R}_{\$\mathrm{B}} = \{\, \mathrm{t}_1, \mathrm{t}_2\};\ \mathrm{DH}(\$\mathrm{C}) = \{\, \mathrm{t}_3\}\ \text{ and } \mathrm{E}_{\$\mathrm{C}} = \{\, \mathrm{t}_1, \mathrm{t}_2\}$$

(End of illustration)

Discussion

In the following, we present a generalized discussion on the notion of disabling transitions.

For any given sub-state machine A, $\mathrm{E_A}$ defines its set of enabling transitions. In particular, it is a set of enabling transitions of its top-level states. These are transitions into its top-level states and their descendants. We express this as the following:

$$\mathrm{E_A} = \bigcup_{\mathrm{x} \trianglelefteq \mathrm{y}} \varphi(\mathrm{x}),\ \forall y \in \mathrm{S_A}$$

Given that there is a state machine B such that $A \sqsubset B$. In this case, the enabling transitions of A will be the set embedded transitions of $\rho(A)$ which is a subset of enabling transitions of B.

$$E_A \subseteq \varphi(\rho(A)) \subseteq E_B \subseteq R_B$$

In the same vein, given that $s = \rho(A)$, all transitions originating from any ancestor of s (including those originating from "s") are considered as high-level transitions of sub-state machine "A". These transitions will disable sub-state machine "A" whenever they execute. Therefore, the following defines a consistent set of transitions that will disable sub-state machine "A".

$$D(A) = \{\ t \mid t \in ((\ R_B \cup (\bigcup_{\rho(A) \overleftarrow{\lhd} k} H(k)))\backslash E_A) \text{ with } A \sqsubset B\ \}$$

By definition (see Definition 10), R_B is a union of the set of transitions originating from top-level states and sub-states of state machine B. Given that $A \sqsubset B$ implies A being the descendant of state machine B. Therefore, $E_A \subseteq E_B$ and the set difference between R_B and E_A, denoted as $(R_B \backslash E_A)$ becomes consistent.

The expression $\bigcup_{\rho(A) \overleftarrow{\lhd} k} H(k)$ denotes the set of all high-level transitions originating from the ancestors of $\rho(A)$. However, the only high-level transitions not included in the set are those originating from $\rho(A)$ but this is included in R_B since $\rho(A)$ is a state of B and such transition is included in O_B (a subset of R_B). Therefore, there is no intersection between this set and the enabling transitions of A.

Another problem arises whenever the destination of a high-level transition of the sub-state machine under consideration is a state (or sub-state) of the machine. In this case, the transition is by default an enabling transition of the sub-state machine. Therefore, inconsistency sets in since an enabling transition is blindly being considered as a disabling transition of the machine.

To resolve this, we introduce the notion of ignorable high-level transitions for any given sub-state machine A (denoted as IH(A)). We present its formal definition as follows:

$$IH(A) = \{\ t \mid \exists_k \cdot t \in \bigcup_{\rho(A) \overleftarrow{\unlhd} k} H(k) \text{ and } t = (_, _, s') \text{ with } s' \overleftarrow{\ntriangleleft} \rho(A)\ \}$$

Therefore, the set of ignorable high-level transitions should be eliminated from the set of disabling transitions of A. Consequently, the set of considerable high-level transitions for any sub-state machine A (denoted as DH(A)) is given as follows:

$$DH(A) = ((\bigcup_{\rho(A) \leqq k} H(k)) \backslash IH(A))$$

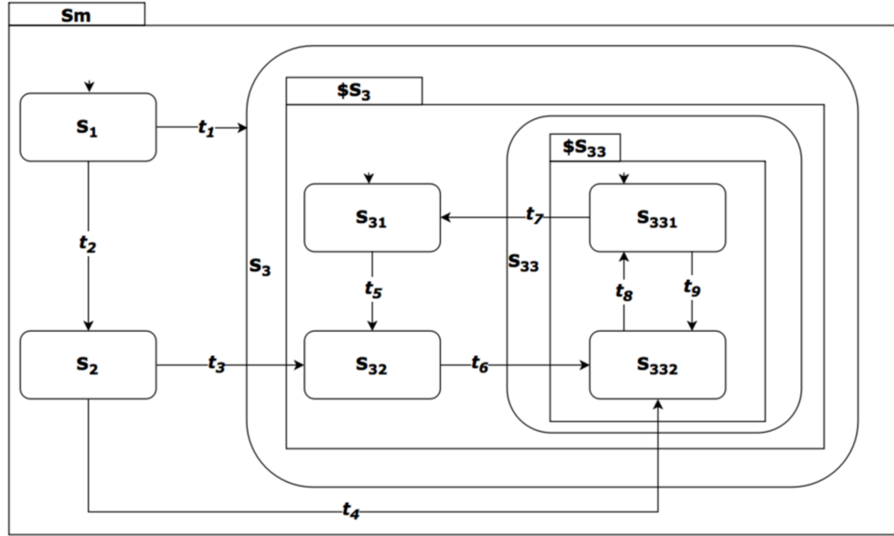


FIGURE 29. DEMONSTRATING DISABLING TRANSITIONS FOR NON-PARALLEL STATE MACHINE

(Illustration)

In Figure 29, we present another SSUA to illustrate how the disabling set of transitions are computed when the high-level transitions set is empty. The relationship between the state machines of the SSUA and other related expressions as presented in Figure 29 are expressed as follows:

$$\$S_{33} \sqsubset \$S_3; \$S_3 \sqsubset Sm$$

$$R_{Sm} = \{t_i \mid 1 \leq i \leq 9\};$$

$$R_{\$S3} = \{t_j \mid 3 \leq i \leq 9\}; \text{ and}$$

$$R_{\$S33} = \{t_4, t_6, t_7, t_8, t_9\}$$

$$E_{\$S33} = \{t_4, t_6, t_8, t_9\};$$

$$E_{\$S3} = \{t_3, t_4, t_5, t_6, t_7, t_8, t_9\};$$

$$DH(\$S_{33}) = DH(\$S_3) = \emptyset$$

First, let us consider the set of disabling transitions of $\$S_{33}$. Since DH($\$S_{33}$) = $\emptyset$, then we have the following:

$$D(\$S33) = (R_{\$S33} \cup DH(\$S_{33})) \setminus E_{\$S33} = R_{\$S3} \setminus E_{\$S33} = \{t_3, t_5, t_7\}$$

Similarly, the set of disabling transitions for $\$S_3$ will be the following since DH($\$S_3$).

$$D(\$\$S_3) = (R_{Sm} \cup DH(\$S_3)) \setminus E_{\$S3} = \{t_1, t_2\}$$

(End of illustration.)

However, the semantics of orthogonal states complicates the computation of disabling transitions for a parallel region. The complexity is introduced following the semantics that transitions originating from any state of parallel regions must execute simultaneously whenever they receive same trigger and the guard conditions are satisfied. We consider the case of parallel regions in the next section.

5.4.3 Disabling Parallel Sub-State Machine (or Region)

In this section, we will focus on our method for computing the set of disabling transitions of a parallel region of an SSUA. Parallelism introduces multiple forms of complexity. A special case of complexity is introduced by the notion of and-crossing which demands special treatment.

5.4.3.1 Parallel Sub-State Machine without And-Cross Transitions

The semantics of an orthogonal region demands that enabling transitions of its top-level states executes independently of other regions of the same enclosing state. The adoption of a set of transitions outside the area enclosed by the region to disable parallel region can introduce inconsistencies. These inconsistencies are possible because the set contains embedded transitions of other parallel regions which may execute concurrently according to the semantics of orthogonal regions.

For example, let us consider the SSUA presented in Figure 30 to illustrate inconsistencies resulting from adopting the same approach as non-parallel sub-state machines. The dashed lines indicate

transitions that can execute simultaneously; while solid lines transitions that execute independently. The elements of the following set of transitions can execute simultaneously:

$$\{ t_1, t_2 \}; \{ t_4, t_9 \}; \text{and } \{ t_5, t_7 \}$$

However, to compute disabling transitions of “$A” using the formula presented in equation (11) the disabling transitions of “$A” is as follows:

$$D(\$A) = \{ t_i \mid 4 \leq i \leq 9 \}$$

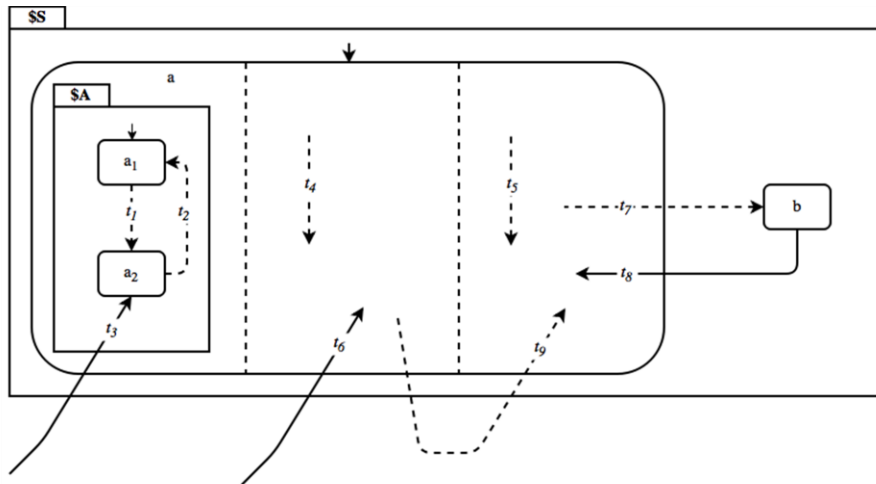


FIGURE 30. SYMBOLIC EXAMPLE TO ILLUSTRATE INCONSISTENCIES WITH PARALLELISM

The implication of the set D($A) is that whenever any of its elements executes, $A must be disabled. However, since the semantics of paralslel regions allows simultaneous execution of parallel transitions, the execution of the resulting set will result in inconsistency. Thus, we ask the following question to address this complexity:

Research Question 4

Is there a consistent and minimal set of transitions to disable a region yet maintain consistency of other regions within the same enclosing state? Particularly, we are interested in the smallest subset of the universal set of

transitions whose execution will disable a parallel region without introducing any form of inconsistencies to other regions within the same enclosing state.

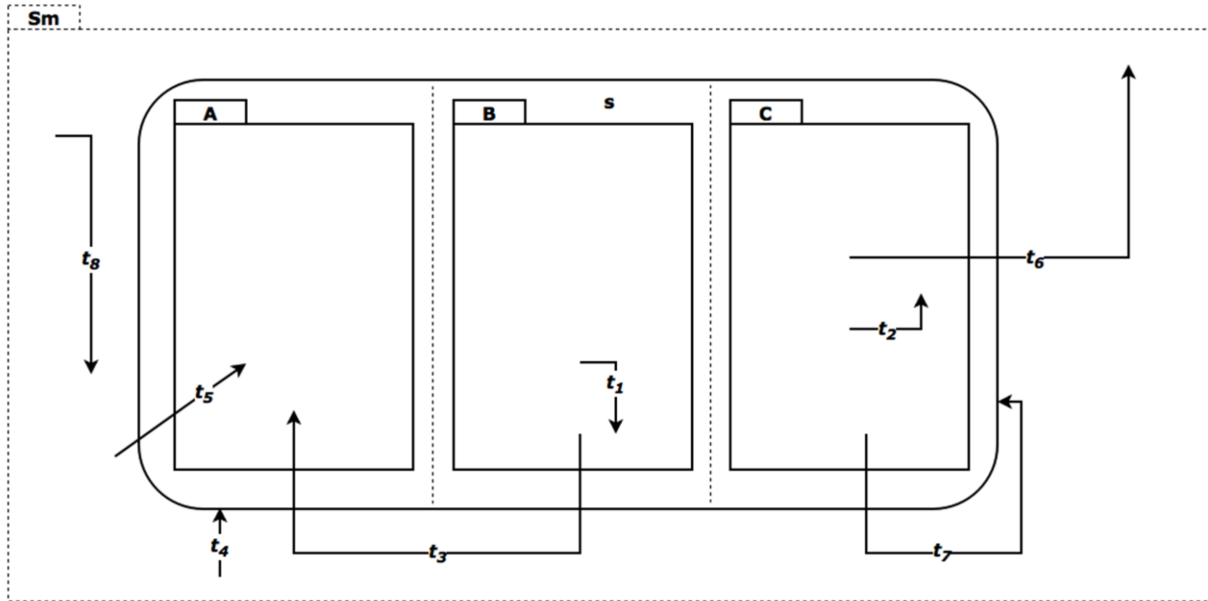


FIGURE 31. ANALYZING MINIMAL SUBSET TO DISABLE ORTHOGONAL REGION

To facilitate the discussion, we present Figure 31 to demonstrate how we compute minimal subset for disabling an orthogonal region yet maintaining consistencies of other region(s) of the same enclosing states. Transitions $\mathrm{t}_1, \mathrm{t}_2, \mathrm{t}_3, \mathrm{t}_6$ and t_7 can execute in parallel based on the semantics governing orthogonal transitions [85]. t_3 is an and-cross transition with different semantics in terms of the resulting configuration after its execution. In particular, its execution interferes with other regions; thus, we defer the discussion of its management to another section.

But for t_1 and t_2 execution is possible simultaneously without interfering with other regions of the enclosing state. Thus, we can say that for a given region the embedded transitions of its parent state (i.e. the union of its enabling transitions and other regions) must not be included in the set of transitions to disable the region. A parent state of a region A is the same for any parallel region B to region A.

However, whenever transitions t_6 and t_7 execute it is expected that regions A, B and C are disabled because control is expected to be transferred out of state s. Although, t_7 re-enters s but via *activation-by-default*. In particular, t_7 would disable the enclosed region then re-enter to initialize them. Therefore, in our method we consider t_7 as a disabling transition of all the regions of the state under consideration.

Semantically, we note that for the sake of consistency, whenever there are parallel regions B and C then the intersection of disabling transitions of B and the enabling transitions of C must be empty. We illustrate this formally as follows:

$$(B \sqsubseteq_{||} C \rightarrow (D(B) \cap E_C = \emptyset))$$

However, we are seeking the set of disabling transitions for any parallel region "B" (denoted as D(B)) such that the execution of enabling transitions of other parallel regions to "B" will not disable it. A simple and straightforward set are those outside the boundary of the enclosing state of B. This implies:

$$\text{Given that } s = \rho(B) \text{ then } D(B) = R_{SSUA} \backslash \nabla(s)$$

Therefore, we can say $D(B) = R_{SSUA} \backslash \nabla(s)$ is a consistent and sufficient set to disable region B without interfering with the enabling transitions of parallel region C since $E_C \subseteq \nabla(s)$. Therefore, the following holds for D(B):

$$(B \sqsubseteq_{||} C \wedge E_C \subseteq \nabla(s) \wedge D(B) = R_{SSUA} \backslash \nabla(s)) \rightarrow (D(B) \cap E_C = \emptyset)$$

Similarly, the following holds for D(C) since $s = \rho(C)$ and $E_B \subseteq \nabla(s)$:

$$(B \sqsubseteq_{||} C \wedge E_B \subseteq \nabla(s) \wedge D(C) = R_{SSUA} \backslash \nabla(s)) \rightarrow (D(C) \cap E_B = \emptyset)$$

Since the disabling transitions of "B" do not interfere with those of "C" and vice-versa, it is reasonable to say to generalize for all parallel regions (say A, B, and C), that the set of disabling transitions are the same and never interferes with enabling transitions of one another.

Given that

$$(\forall_{A,B,C} \text{ such that } (A \sqsubseteq_{||} B \sqsubseteq_{||} C) \text{ and } \rho(A) = \rho(B) = \rho(C))$$

then

$$A \sqsubseteq_{||} B \sqsubseteq_{||} C \rightarrow D(A) = R_{SSUA} \setminus \nabla(\rho(A))$$
$$\wedge\ D(B) = R_{SSUA} \setminus \nabla(\rho(B))$$
$$\wedge\ D(C) = R_{SSUA} \setminus \nabla(\rho(C))$$

(Illustration)

We illustrate this as follows using Figure 31:

$$R_{Sm} = \{\ t_i \mid 1 \leq i \leq 8\ \}$$
$$\nabla(s) = \{\ t_1, t_2, t_3, t_5\ \}$$
$$(\ R_{Sm} \setminus \nabla(s)\ ) = \{\ t_4, t_6, t_7, t_8\ \}$$

$E_C = \{\ t_2\}$ (i.e., the enabling transitions of “C”)

Therefore, if

$$D(B) = (\ R_{Sm} \setminus \nabla(s)\ ) = \{\ t_4, t_6, t_7, t_8\ \}$$

then it becomes obvious that:

$$D(B) \cap E_C = \emptyset$$

In a similar manner, for parallel region “A” the disabling transitions of “B” do not interfere with “A”.

$$D(B) \cap E_A = \emptyset$$
$$\text{since } E_A = \{\ t_3, t_5\ \}$$

(End of illustration.)

Recall that from Research Question 4, just as we as we dealt with the case of disabling non-parallel sub-state machines, we are also interested in the minimal set of transitions that is sufficient to disable the region. Therefore, in a similar manner to the referenced case (i.e., disabling non-parallel sub-state machine), suppose “A” is the immediate ancestor of “B” and “C”, then the following is valid:

$$(\forall_{B,C}, \exists_A \text{ such that } (B \sqsubseteq_{||} C) \sqsubset A) \rightarrow (\exists_{s \in S_A} \cdot\ s = \rho(B) \wedge\ s = \rho(C)) \text{ and } \nabla(s) \subseteq R_A$$

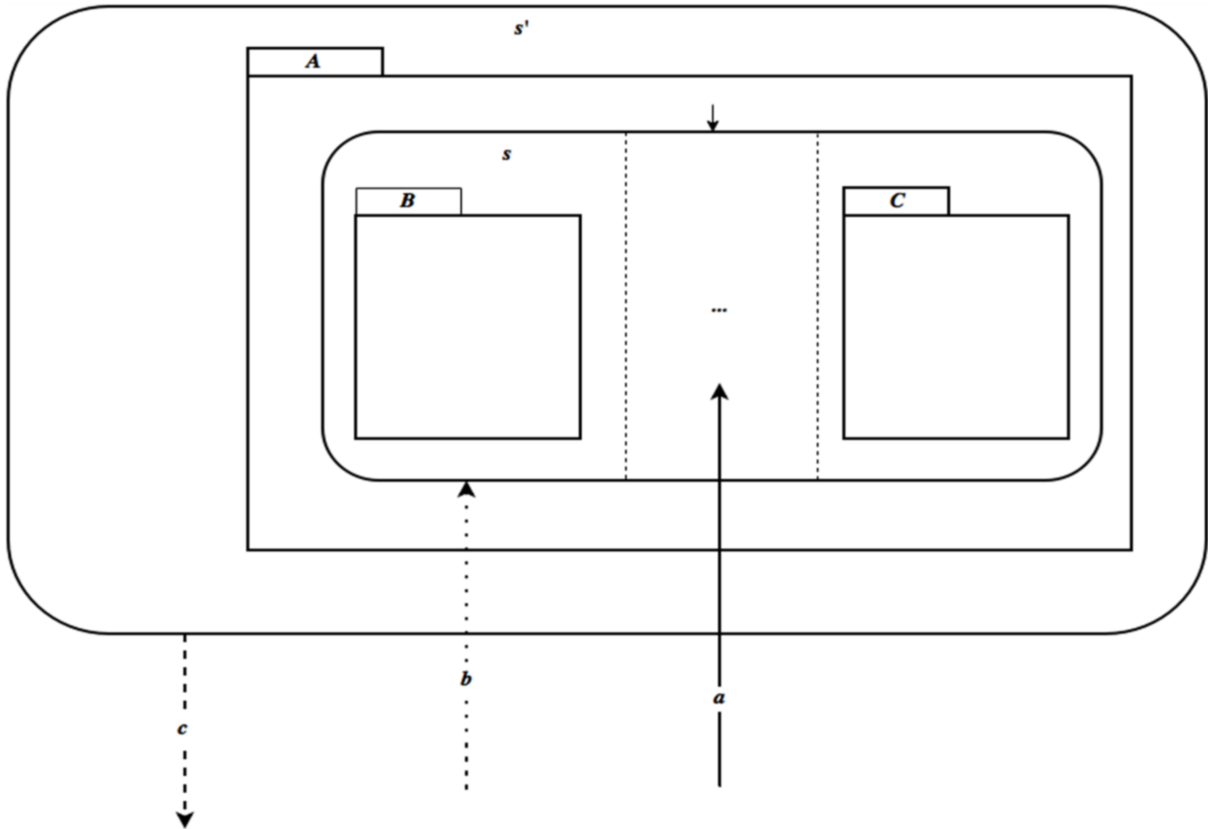
Since $\nabla(s)$ is a subset of R_A, then R_A can substitute R_{SSUA} without jeopardizing the semantics yet maintain consistency of executable transitions between the parallel regions. Then, the number of transitions is reduced to a minimal set and D(B) can be written as follows:

Given that

$$(\forall_{B,C,D}\ \exists_A \text{ such that } (B \sqsubseteq_{||} C \sqsubseteq_{||} D) \sqsubset A \text{ and } \rho(B) = \rho(C) = \rho(D))$$

then

$$B \sqsubseteq_{||} C \sqsubseteq_{||} D \rightarrow D(B) = R_A \setminus \nabla(\rho(B))$$
$$\wedge\ D(C) = ((R_A \setminus \nabla(\rho(C)))$$
$$\wedge\ D(D) = ((R_A \setminus \nabla(\rho(D)))$$



$$a = \nabla(s); b = \Delta(s); c = DH(s')$$

FIGURE 32. ANALYZING A DISABLING TRANSITION SET FOR AN ORTHOGONAL REGION

In Figure 32, we present an example to illustrate the set of transitions to disable an orthogonal region for the sake of clarity. We highlight the following concerning Figure 32 for purpose of analysis.

$$\nabla(s) \cup \Delta(s) \subseteq R_A$$

Besides disabling transitions, we consider high-level transitions of ancestor states of the enclosing state of the regions under consideration in a similar manner as established in Section 5.4.2. In particular, all high-level transitions of parent states of these parallel regions as well as those of its ancestors should be added to the set of disabling transitions established above.

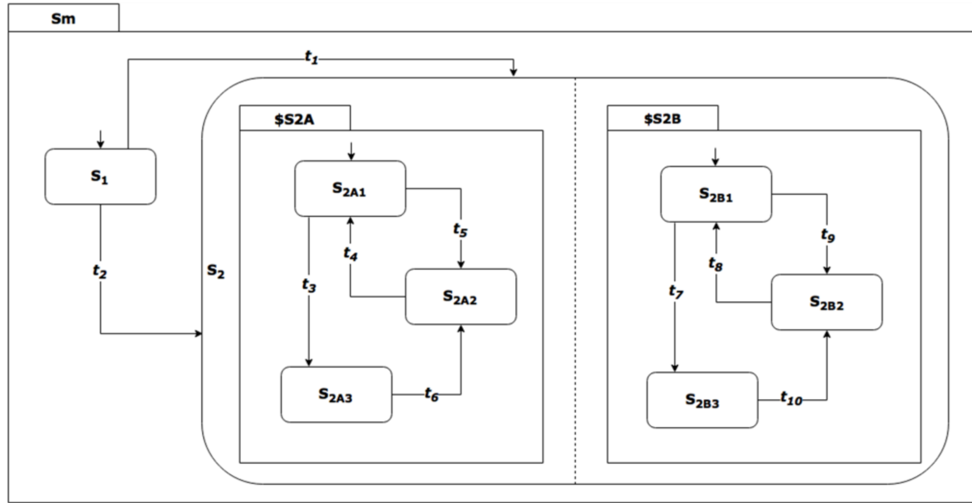


FIGURE 33. DISABLING REGIONS OF ORTHOGONAL STATES

However, those with destinations targeting a sub-state of a given region should be ignored for the target region but not for other regions since their activations will be done by default. Consequently, the final set of disabling transitions is region-specific and expressible as follows:

Given that

$$(\forall_{B,C,D}\ \exists_A \text{ such that } (B \sqsubseteq_{||} C \sqsubseteq_{||} D) \sqsubset A \text{ and } \rho(B) = \rho(C) = \rho(D))$$

then

$$B \sqsubseteq_{||} C \sqsubseteq_{||} D \rightarrow (D(B) = ((R_A \setminus \nabla(\rho(B))) \cup DH(B))$$
$$\wedge D(C) = ((R_A \setminus \nabla(\rho(C))) \cup DH(C))$$
$$\wedge D(D) = ((R_A \setminus \nabla(\rho(D))) \cup DH(D))) \quad (15)$$

Figure 33 is another example of an SSUA with parallel regions to demonstrate the process of disabling orthogonal regions without and-crossing. The following expresses the relationship between sub-state machines (or regions) of the SSUA.

(Illustration)

We illustrate this set by a means of example below:

Given that

$$(\$S2A \sqsubseteq_{||} \$S2B) \sqsubset Sm \text{ such that } \rho(\$S2A) = \rho(\$S2B) = s_2$$

then

$$D(\$S2A) = (R_{Sm} \setminus \nabla(\rho(\$S2A))) \cup DH(\$S2A) = \{ t_1, t_2 \}$$

since

$$DH(\$S2A) = \emptyset;\ R_{Sm} = \{ t_i \mid 1 \leq i \leq 10 \} \text{ and } \nabla(\rho(\$S2A)) = \{ t_j \mid 3 \leq j \leq 10 \}$$

Similarly,

$$D(\$S2B) = (R_{Sm} \setminus \nabla(\rho(\$S2B))) \cup DH(\$S2B) = \{ t_1, t_2 \}$$

since

$$DH(\$S2B) = \emptyset; \text{ and } \nabla(\rho(\$S2B)) = \nabla(\rho(\$S2A)) \text{ since } \$S2A \sqsubseteq_{||} \$S2B$$

(End of illustration.)

5.4.3.2 Parallel Sub-State Machine with And-Cross Transitions

As earlier noted, the semantics of and-cross transitions introduces some complexity. To deal with this complexity, we need some clarifications to the semantics. The following define the steps involved in realizing semantics of executing an and-cross transition.

Step 1 - Exit enclosing orthogonal state and re-enter it.
Step 2 - Target region is initialized to the target state of the transition.
Step 3 - All regions except the target are re-initialized to their start (or initial) states.

We modeled this semantics by computing the set of and-cross transitions of the enclosing state.

Given an and-cross transition *t* from region *A* to *B* and that there are regions *C* and *D* parallel to *A* and *B*.

Rule 1

Add *t* to the set of disabling transitions (as computable from equation 15) of all non-targeted regions of *t* (i.e., *A*, *D* and *C*) but the target region of *t* (i.e., *B*) since *t* is also an enabling transition of *B* (i.e., $t \in E_B$). Therefore, the disabling transitions of *A*, *C* and *D* now includes *t*.

Rule 2

For *B* (i.e., the target region of *t*) apply equation (15) for the computation of the disabling set of transitions. In particular, the disabling transition set for the target region of an and-cross transition does not change.

These rules are applied repeatedly to every and-cross transition for the enclosing state under consideration until there is no more and-cross transition.

To define the disabling transition set, we first define and-cross whose target is not the region under consideration as follows:

Definition 23. And-cross transition with a target region other than the region under consideration

Given an and-cross transition, t of state s (i.e., $t \in \mathcal{a}(s)$) (therefore, there are regions A and B such that $A \sqsubseteq_{||} B$ with $s = \rho(A) = \rho(B)$), we say t is an and-cross transition whose target is different from region A (i.e., region under consideration) and the corresponding set of such transitions is denoted as $\mathcal{b}(A, s)$. Its formal representation is as follows:

$$\mathcal{b}(\mathrm{A}, \mathrm{s}) = \{ \mathrm{t} \mid \mathrm{t} \in \mathcal{a}(\mathrm{s}) \wedge \exists_{s'} \cdot \mathrm{t} = (_, _, s') \text{ with } s' \notin \mathrm{U}_{\mathrm{SA}} \} \quad (16)$$

And the set of disabling transitions of a parallel sub-state machine (or region) whenever the enclosing state has and-cross transition(s) is given as follows:

Given that

$$(\forall_{B,C,D}, \exists_{A} \text{ such that } (B \sqsubseteq_{||} C \sqsubseteq_{||} D) \sqsubset A \text{ and } \rho(B) = \rho(C) = \rho(D))$$

then

$$\begin{aligned} B \sqsubseteq_{||} C \sqsubseteq_{||} D \rightarrow (\ & D(B) = ((R_A \setminus \nabla(\rho(B))) \ \cup DH(B) \cup \mathcal{b}(B, \rho(B))) \\ & \wedge D(C) = ((R_A \setminus \nabla(\rho(C))) \ \cup DH(C) \cup \mathcal{b}(C, \rho(C))) \\ & \wedge D(D) = ((R_A \setminus \nabla(\rho(D))) \ \cup DH(D) \cup \mathcal{b}(D, \rho(D))) \ ) \end{aligned} \qquad (17)$$

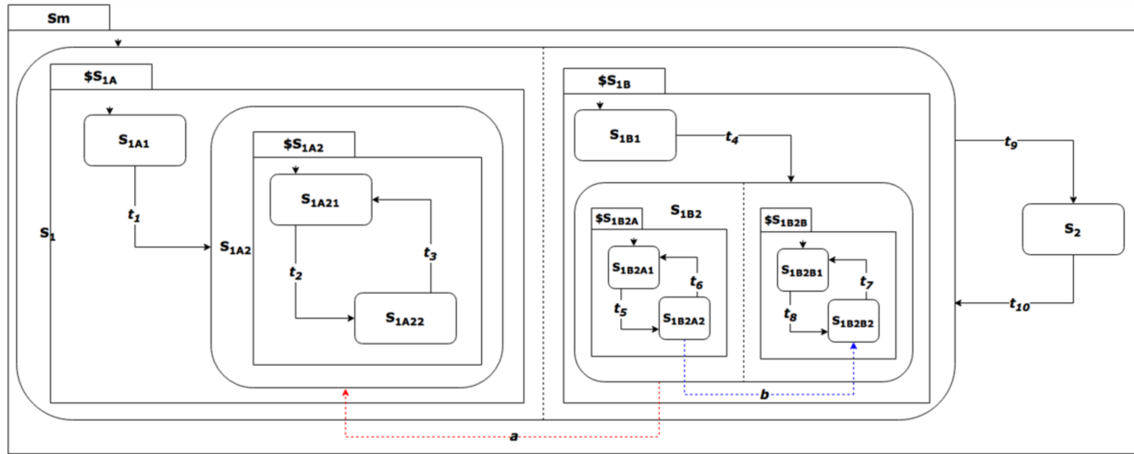


FIGURE 34. AND-CROSS TRANSITION EXAMPLES

In Figure 34, we present basic and-cross transition (label *a*) and an unusual transition (label *b*) to help explain the process of disabling a region in the presence such transitions. We discuss the steps to computing the disabling transitions of the containing regions of transitions with labels *a* and *b* in Figure 34. For orthogonal composite state s_1 of Figure 34, the top-level regions are $\$S_{1A}$ and $\$S_{1B}$; while the sources and destination states of transition associated with label *a* are in S_{1B} and S_{1A} respectively. The transition labelled 'a' is an and-cross transition of s_1 but the transition labelled 'b' is not an and-cross transition of s_1.

$$(\$S1A \sqsubseteq_{||} \$S1B) \sqsubset Sm$$

$$R_{Sm} = \{ t_i \mid 1 \leq i \leq 10 \} \cup \{ a, b \}$$

$$\nabla(\rho(\$S1A)) = \nabla(\rho(\$S1B) = \{ t_j \mid 1 \leq j \leq 8 \} \cup \{a, b\}$$

$$a(\rho(\$S1A)) = \{ a \}$$

Since "a" targets \$S1A, $b(\$S1A, \rho(\$S1A)) = \emptyset$

$$DH(\$S1A) = DH(\$S1B) = \emptyset$$

Thus,

$$D(\$S1A) = (R_{Sm} \setminus \nabla(\rho(\$S1A))) \cup DH(\$S1A) \cup b(\$S1A, \rho(\$S1A)) = \{ t_9, t_{10} \}$$

But for \$S1B, we have the following:

$$D(\$S1B) = (R_{Sm} \setminus \nabla(\rho(\$S1B)) \cup DH(\$S1A)\ ) \cup b(\$S1B, \rho(\$S1B)) = \{ t_9, t_{10}, a \}$$

Now, let us consider transition 'b', the unusual transition of s_1 but and-cross transition of s_{1B2}. We address this as an and-cross transition of s_{1B2}. We apply the same principle we applied to 'a' as above.

Given that:

$$(\$S1B2A \sqsubseteq_{||} \$S1B2B) \sqsubset \$S1B$$

$$R_{\$S1B} = \{ t_i \mid 4 \leq i \leq 8 \} \cup \{b\}$$

$$\nabla(\rho(\$S1B2A)) = \nabla(\rho(\$S1B2B) = \{ t_j \mid 5 \leq j \leq 8 \} \cup \{b\}$$

$$a(\rho(\$S1B2A)) = \{ b \}$$

Since "b" targets \$S1B2B, $b(\$S1B2A, \rho(\$S1B2A)) = \{b\}$

$$DH(\$S1B2A) = DH(\$S1B2B) = \{t_9\}$$

Then,

$$D(\$S1B2A) = (R_{\$S1B} \setminus \nabla(\rho(\$S1B2A))) \cup DH(\$S1B2A) \cup b(\$S1B2A, \rho(\$S1B2A)) = \{t_4, t_9, b\}$$

But for $\$S_{1B2B}$, we have the following:

$$D(\$S1B2B) = (R_{\$S1B} \setminus \nabla(\rho(\$S1B2B))) \cup DH(\$S1B2B) \cup b(\$S1B2B, \rho(\$S1B2B)) = \{t_4, t_9\}$$

$$\text{since } b(\$S1B2B, \rho(\$S1B2B)) = \emptyset$$

5.4.4 Generalizing Disabling Transitions for Sub-State Machines

In this section, we focus on the generalization of the set of disabling transitions for sub-state machines irrespective of the underlying nature of complexity (e.g., parallelism or and-cross transitions). Particularly, we want to harmonize equations [12, 15 and 17] for ease of understanding.

In Figure 35, we represent the relationship between some sets of transitions necessary for our generalizations formally in the form of Venn diagram for clarity sake.

Given that A is any sub-state machine and B is its immediate ancestor,
where:

- E_A is the set of enabling transitions of A;
- DH(A) is the disabling high-level transitions for A;
- $\nabla(\rho(A))$ is the set of embedded transitions of A;
- $b(A,\rho(A))$ is the set of and-cross transitions of $\rho(A)$ with destinations not in A; and
- R_B is the set of transitions of B (i.e., the union of E_B and O_B).

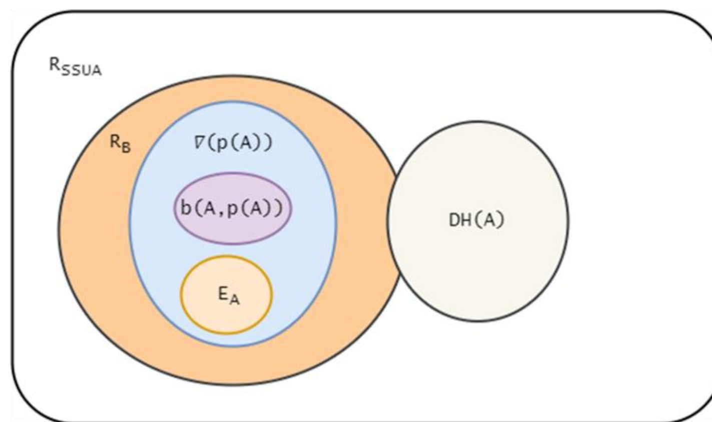


FIGURE 35. A VENN DIAGRAM TO ILLUSTRATE THE RELATIONSHIPS BETWEEN SETS OF TRANSITIONS NECESSARY FOR DISABLING ANY REGION A WHOSE IMMEDIATE ANCESTOR IS B

First, we would like to express equation (12) like equations [15, 17] to allow (or ease) generalization. Recall that equations [12, 15] are expressible as follow:

$$D(A) = ((R_B \cup DH(A)) \setminus E_A)$$

From Figure 35, it is obvious that:

$$E_A \subseteq R_B \wedge E_A \cap DH(A) = \emptyset$$

Therefore, D(A) can be written as follows without jeopardizing the semantics:

$$D(A) = (R_B \setminus E_A) \cup DH(A) \quad (18)$$

However, whenever $\beta(\rho(A)) = 1$ the set of enabling transitions of *A* is the same as the set of its embedded transitions since *A* will be the only sub-state machine for the parent state. Thus, its embedded transitions are the transitions enabling the top-level states of *A*.

$$\beta(\rho(A)) = 1 \rightarrow E_A = \nabla(\rho(A))$$

Therefore, equation (18) can be re-written as equation (19) without jeopardizing semantics:

$$D(A) = (R_B \setminus \nabla(\rho(A))) \cup DH(A) \quad (19)$$

But since and-cross transitions are a notion relevant to parallel region, the set of and-cross transitions relevant to disabling *A* will be empty. That is:

$$\mathscr{b}(A, \rho(A)) = \emptyset$$

Hence, the general formulation for computing disabling transitions set is expressed for any given sub-state machine *A* where *B* is the ancestor A, as follows:

Given that:

$$(\forall_A, \exists_B \text{ such that } A \sqsubset B)$$

then,

$$D(A) = (R_B \setminus \nabla(\rho(A))) \cup DH(A) \cup \mathscr{b}(A, \rho(A))$$

We highlight the following to simplify and clarify issues relevant to the formulation:

1. Computing disabling transitions for sub-state machines (i.e., parallel and non-parallel) is the same whenever there are no and-cross transitions because $\mathscr{b}$(A, ρ(A)) for sub-state machine A will be empty and both expressions can be reduced to the same (see equations 15 and 18) in every instance.

2. For a sub-state machine A when there is an and-cross transition of its enclosing state for which the target is A, the rule for computing (1) is applicable because such and-cross transition will also be considered as its enabling transition. Hence, the set remains unchanged.

3. For a sub-state machine A when there is an and-cross transition of its enclosing state for which the target is not A, the rule for computing (1) is applicable but we added such and-cross transition to the disabling transitions set because such and-cross transition will be considered a disabling transition of the sub-state machine.

5.5 Summary of State Machines Formalization

In this chapter, we discussed our approach to formalizing state machines expressed in Umple. We began by establishing the basis of this research and the contribution it offers. This work implements the encoding proposed by Badreddin et. al [67] for the purpose of formal reasoning of temporal properties on state machines. Unusual transitions demand systematic encoding due to their underlying complexities. Consequently, approaches like Faghih and Day [86] are limited in managing it. Our approach resolves this issue without extra computational cost. Furthermore, we formally describe the SSUA. We gave a description of Umple state machine and the relationships between the states and sub-state machines forming the SSUA.

We answered three fundamental research questions. These are: What sufficient constraint will activate an initial state of a sub-state machine by default or initialize a region? What subset of transitions of the SSUA is sufficient for the computation of disabling transitions of a region of an SSUA? Is there a consistent and minimal set of transitions to disable a region yet maintain consistency of other regions of the same enclosing state? By tackling these questions, we could identify key issues to ensuring consistency and correctness. By providing answers to these questions, we could justify our decisions.

These led to some definitions that shaped our focus and sound strategies to compute disabling and enabling transitions set for states and sub-state machines. Despite the complexities introduced by parallel regions and and-cross transitions, we were able to generalize our approach over the wide range of possible regions (or sub-state machines).

6 Quality Assurance of Umple's SSUAs

In this chapter, we present an aspect of our work that focuses on quality assurance of state machine systems under analysis (i.e., SSUA). Quality assurance involves ensuring that an SSUA is free of bugs. The problem of complete quality assurance of software systems or state machine systems is beyond what a tool can address because many of desirable properties are domain-specific. Therefore, we focus on some domain-independent properties whose corresponding specifications can be generated and analyzed automatically. The subset of domain-independent properties we cover includes automation of the process of: a) discovering non-determinism and b) certifying that states of an SSUA are reachable.

Nondeterminism refers to a situation whereby the simultaneous execution of a pair of transitions of an SSUA results in uncertainty about which configuration the state machine will subsequently be in. Although, specifying a nondeterministic situation is a technique often adopted by requirement engineers for the purpose of understanding a system under development (SUD), it may result in safety consequences if shipped mistakenly in the actual implementation. Thus, discovering these situations and considering them as flaws becomes critical. However, the increasing complexity of software systems makes such discovery infeasible when performed manually [88].

By "reachability of states", we mean the process of determining whether there is at least one path to a given state of the SSUA from its initial configuration. In particular, we automatically generate specifications for these states such that they can be fed into the model checker to discover flaws resulting from unreachable states (whenever possible).

6.1 Discovering Non-Determinism

Our goal in this aspect of the thesis is to certify that Umple state machines systems are free of non-determinism. We report our method to systematically compute a minimal set of potentially conflicting pairs of transitions and formally specify invariants based on these transitions.

To facilitate understanding, we present two major categories of nondeterminism fundamental to Umple. These include same-source and region-cross *versus* parallel transitions. We consider various kinds of state machine transitions (including those with overlapping guard conditions).

6.1.1 Same-Source Transitions

In this category, we will discuss pairable transitions whose sources are physically and logically the same. A physically same-source pair of transitions involves transitions originating from the same source states; while a logically same-source pair of transitions involves embedded transition *versus* high-level transition.

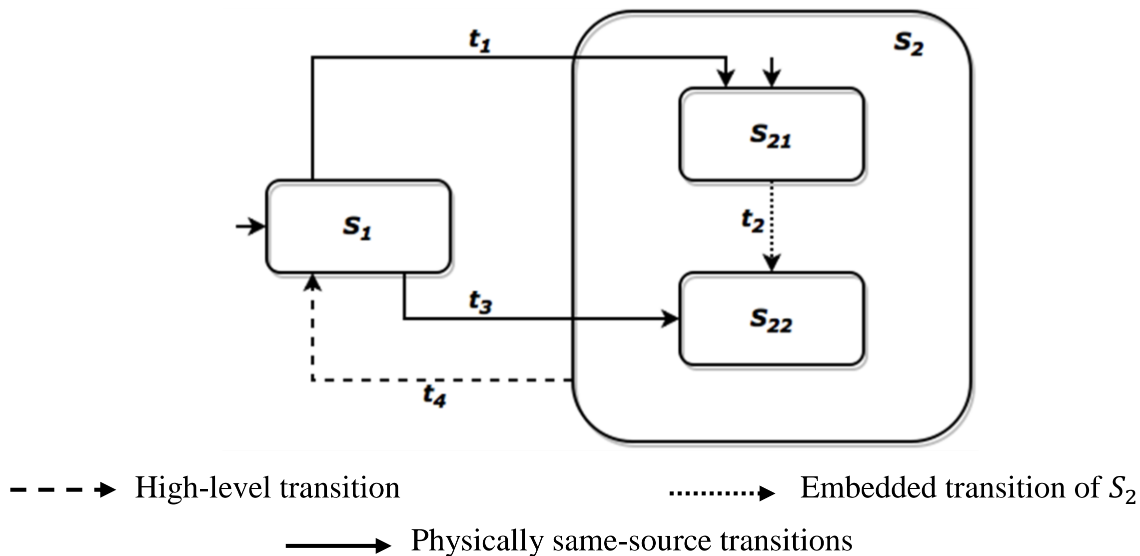


FIGURE 36. ABSTRACT EXAMPLE TO DEMONSTRATE SAME-SOURCE PAIRS OF TRANSITIONS

In Figure 36, we present a modeling example to illustrate same-source pairs. Transitions $\{t_i|1 \leq i \leq 4\}$ may be interpreted as overlapping (i.e., having same trigger and/or overlapping guard statement). We categorize these pairs as *physically same-source* and *high-level transition versus embedded transition*. By “physically same-source pairs”, we mean transitions originating from the same source states.

For example, transitions t_1 and t_3 originate from vertex s_1. We refer to these transitions as physically same-source transitions. Therefore, whenever the triggering event on these transitions are the same, a pair of the transitions is considered potentially conflicting.

Another pair of transitions in this category combines a high-level transition and embedded transition of a composite state. Semantically, these transitions originate from the same state (i.e., a sub-state of the composite state). A pair of transitions t_2 and t_4 are potentially conflicting if the triggering event on the transitions are the same.

6.1.2 Region-Cross Transitions

In this category, we will discuss pairable transitions whose sources are embedded states of an orthogonal composite state. The notion of region-cross involves not only and-cross transitions but also outgoing transitions of the host orthogonal state. To facilitate discussion, we introduce the notions of parallel transitions of an orthogonal state.

Definition 24. Parallel Transitions of an Orthogonal State

A transition t_1 is parallel to transition t_2 if their sources are different regions of the same enclosing state. A formal description of parallel transitions is given as follows:

Given that there is $t_i = (a, _, _)$, and states x, s such that $x \overleftarrow{\lhd} s$ and $\beta(s) > 1$. Then the set of parallel transitions t_j with respect to t_i within s denoted as $\mathfrak{p}(s, t_i)$ is expressed as:

$$\mathfrak{p}(s, t_i) = \{t_j \mid t_j = (b, _, _) \land \exists_{x,y} \cdot a \overleftarrow{\unlhd} x \land b \overleftarrow{\unlhd} y\} \quad (20)$$

whenever

$$x \in S_A \land \ y \in S_B, A \sqsubseteq_{||} B \land s = \rho(A) = \rho(B)$$

We illustrate region-cross transition cases with an abstract example presented in Figure 37. This example models various kinds of transition relevant to our discussion. State “s” is an orthogonal composite state with regions “s1” and “s2”. Transition “t4” is an outgoing transition of “s”. Any transition $x \in$ { t2, t3, t4, t5 } is parallel to any other transition $y \in$ { t6, t7 }. Transition “t5” is an and-cross transition of “s”.

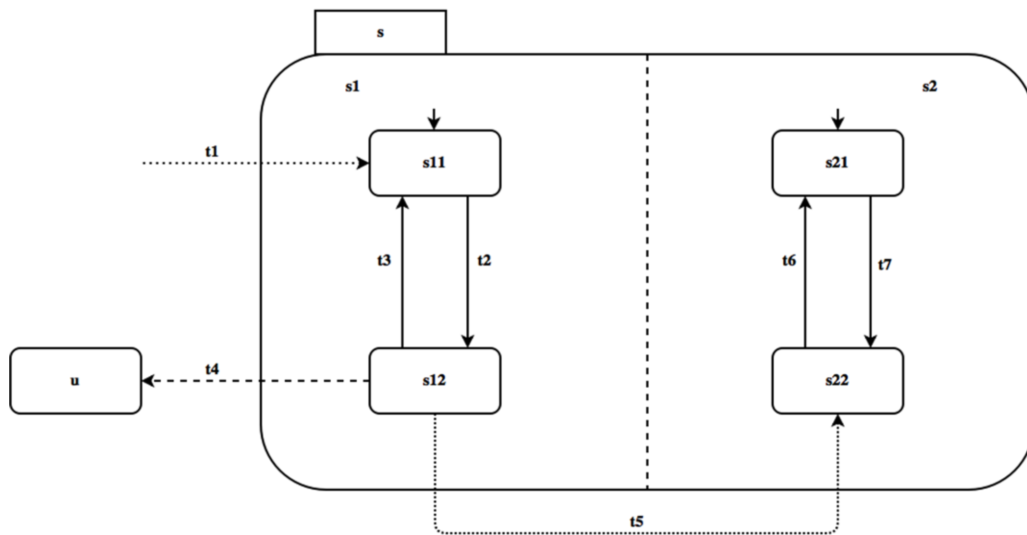


FIGURE 37. ABSTRACT EXAMPLE TO DEMONSTRATE REGION-CROSS CASES

The three major categories of conflicting pairs we consider here are and-cross transition versus parallel transition; outgoing transition versus parallel transition; and multiple and-cross transitions between regions of the same enclosing state.

Type 1 – And-Cross plus Parallel Transitions Pair

Transitions in this category may result in non-determinism if they are triggered simultaneously thereby resulting in undesired configurations. For example, let us consider pair (t5, t6) as potentially conflicting transitions and a case of and-cross versus parallel transitions. We express a possible scenario leading to conflict as follows:

At s^i:	Transition "t5" and "t6" execute.
At s^{i+1}:	States "s21" and "s22" become activated.
At s^{i+2}:	State "s11" becomes activated by default (i.e., region "s1" is re-initialized by and-crossing).

A non-deterministic situation occurs at step $i + 1$ when region "s2" is expected to be in two distinct states simultaneously.

Type 2 – Outgoing Transition plus Parallel Transition

In the same vein, a pair of transitions (t4, t7) are likely to result in nondeterminism as the pair is an example of outgoing versus parallel transition pair. We illustrate this situation as follows:

At s^i: Transition "t4" and "t7" execute.
At s^{i+1}: States "*u*" and "s22" become activated.
At s^{i+2}: Regions "s1" and "s2" become disabled.

As can be seen, the simultaneous activation of states "*u*" and "s22" constitute non-determinism because state diagram semantics forbids an SSUA from activating distinct OR-states simultaneously. Hence, the occurrence of step $i + 1$ should be avoided.

Type 3 – Multiple and-cross in same enclosing state

Multiple and-cross transitions in the same enclosing state introduce a deeper level of complexities. In particular, we are aware of the following problems:

1. Nondeterministic configurations; and
2. Undesirable configurations.

To illustrate these problems, we present a symbolic example in Figure 38. Orthogonal state "*s*" is the enclosing state for regions "*a*", "*b*" and "*c*". Transition "*w"* is an enabling transition of state "*s*" and those in dotted arrow are the and-cross transitions (i.e., "*x*", "*y*" and "*z*") for state "*s*". We introduce transitions $\{ t_i \mid 1 \leq i \leq 5 \}$ to demonstrate issues highlighted above and their corresponding scenarios.

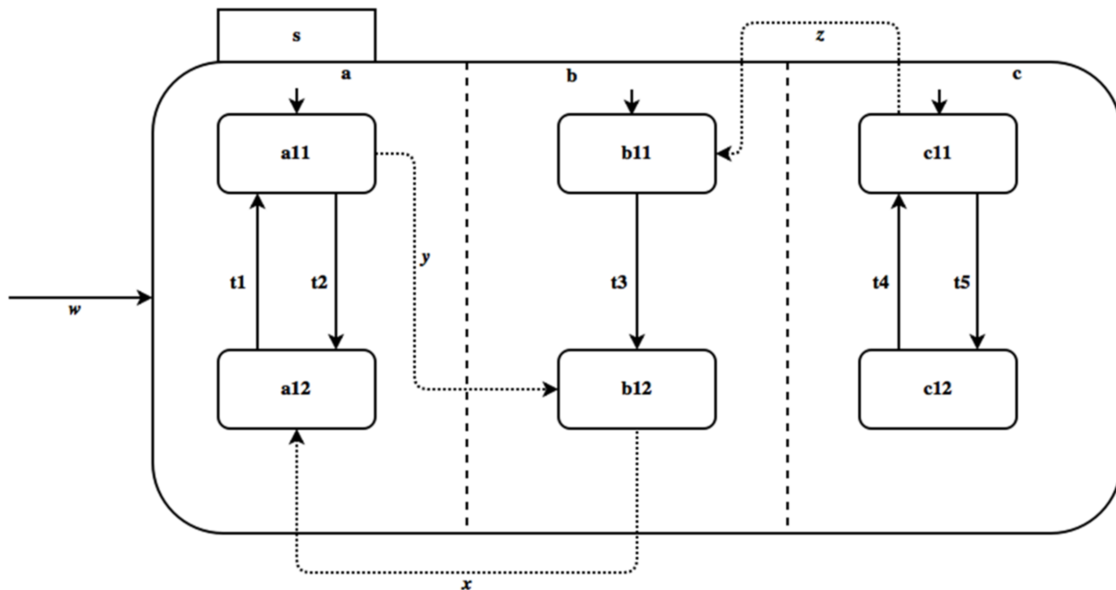


FIGURE 38. DEMONSTRATING INCONSISTENCIES WITH MULTIPLE AND-CROSS TRANSITIONS IN THE SAME ENCLOSING STATE

For the purpose of simplicty, we assume transitions *x, y* and *z* have similar triggers and that their controlling guard statements can evaluate to true at the same step (i.e., simultaneously). Hence, the simultaneous executions of the transitions are possible. Suppose transition "*w*" fires at step *i*, then the following configurations will happen in a *run-to-completion* [85], [89] manner. Starting from step *i+3*, the SSUA will not be stable until step *i+5*.

At s^i:	Transition "*w"* executes.
At s^{i+1}:	State "*s*" becomes activated.
At s^{i+2}:	States "a11", "b11", and "c11" become activated by default.
At s^{i+3}:	Transitions "*y*" and "*z*" fire simultaneously.
At s^{i+4}:	States "b11" and "b12" become activated (non-deterministic case); while regions "*a*" and "*c*" are disabled.
At s^{i+5}:	States "a11" and "c11" become activated by default.

As can be observed, at step i+4 the SSUA becomes inconsistent since region "*b*" wants to assume states "b11" and "b12" simultaneously. We consider this case as non-deterministic; thus, transitions leading to this configuration should be prevented.

We illustrate another scenario, to allow us to formulate a generalized approach to deal with inconsistencies related to and-cross transitions. Let us consider the possibility of executing transitions "*x*" and "*y*" simultaneously.

At s^i:	Transition "*w"* executes.
At s^{i+1}:	State "*s*" becomes activated.
At s^{i+2}:	States "a11", "b11", and "c11" become activated by default.
At s^{i+3}:	Suppose transition "t3" fires.
At s^{i+4}:	State "b12" becomes activated (i.e., region "*b*" is set to state "b12").
At s^{i+5}:	Transitions "*x*" and "*y*" fire simultaneously.
At s^{i+6}:	States "a12" and "b12" become activated; while region "*c*" is disabled.
At s^{i+7}:	Regions "*a*", "*b*" and "*c*" become disabled. **Reason:** by executing "*x*" regions "*b*" and "*c*" are disabled and by executing "*y*" regions "*a*" and "*c*" are disabled.

This results in an undesirable configuration at step *i+7*. A similar scenario can occur whenever transitions "*x*" and "*z*" execute simultaneously. For the case of non-determinism, potentially conflicting pairs of transitions must be examined from a dynamic point of view and prevented whenever they are actual candidate (i.e., *true positives*) and preserved whenever they are *false positives*.

Generally speaking, this case (i.e., Type 3) can only be prevented if it is not the case that two and-cross transitions of the same enclosing state can fire simultaneously. In particular, it is important to note that simultaneous execution of pairs of and-cross transitions in the same enclosing state result in an undesirable configuration whenever their target states reside in different regions. Similarly, multiple and-cross transitions of the same enclosing state with destinations in the same region results in non-determinism. We note that these instances are bugs at any instance and must

be disallowed. Therefore, we forbid simultaneous execution of two or more and-cross transitions in the same enclosing state. To facilitate the representation of this scenario formally, we define *enabledness* of a transition of an SSUA as follows:

Definition 25. Enabledness of transitions of an SSUA

The enabledness of a set of transitions of an SSUA is a triple $\langle x, y, z \rangle$ *such that x is a finite set of transitions, y is a step at which the transitions in x are enabled and z is the configuration resulting from the execution of transitions in x. The following defines enabledness of a set of transitions of an SSUA at step k.*

Given that configuration c $= \langle \langle _ \rangle, \langle s_1, s_2, \ldots s_m \rangle, s^{k+1}, \langle \langle _ \rangle, \langle _ \rangle \rangle \rangle$*, then the enabledness of transitions* t_i *such that* $1 \leq i \leq n$ *whose executions led to c is as follows:*

$$\langle \langle t_1, t_2, \ldots t_n \rangle, k, c \rangle$$

We express the constraint that *no two conflicting executable transitions within the same enclosing state* is allowed formally as follows:

Given that "s" is an orthogonal composite state (i.e., $\beta(s) \geq 2$*)*

$$\forall_{t_i, t_j} \cdot \{t_i, t_j\} \subseteq a(s) \rightarrow \nexists_k \cdot \langle \langle t_i, t_j \rangle, k, c \rangle$$

Where: "k" is a step of execution and "c" being undesired (or non-deterministic) configuration.

6.1.3 Cases of Nondeterminism

In this section, we summarize the cases of nondeterminism. From Sections 6.1 and 6.2 two broad cases (i.e., same-source and region-cross) of nondeterminism were presented. These cases were further subdivided into other cases. We summarize these cases and associate labels with each case for ease of reference in Table 10.

TABLE 10. CASES OF NONDETERMINISM

Same-Source Transitions	**Region-Cross Transitions**
Physically same-source pairs (C1.1)	And-Cross transition *versus* Parallel transition pairs (C2.1)
High-level transition *versus* Embedded transition pairs (C1.2)	Outgoing transition *versus* Parallel transition pairs (C2.2)
	And-Cross versus And-Cross transitions pair in the same enclosing state (C2.3).

6.1.4 False Cases of Nondeterminism

The presence of infinite state variables in guards made us generalize our method to computing conflicting pairs of transitions. In particular, not all cases of conflicting pairs of transitions reported in the preceding sections are actual cases. We refer to these cases as *false positives.*

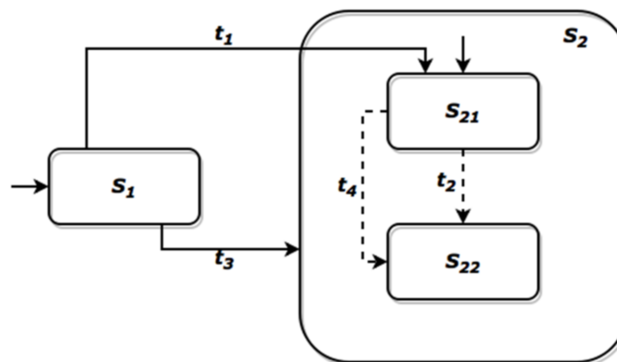


FIGURE 39. CASES OF FALSE POSITIVES

A *false positive* is a pair of transitions conflicting statically but dynamically conflict-free. The latter means that in the global state-space there is no situation where both transitions execute simultaneously or their execution can be overridden by priority semantics [85]. We present an example of transitions resulting in various cases of *false positives* in Figure 39.

We identify various cases of false positives as follows: a) sources and destinations are the same (e.g., pair (t_2, t_4) in Figure 39); b) controlling guards are non-overlapping; and c) controlling events differ.

For case (b) we introduce an operator $\mathcal{v}: l \rightarrow G$ such that l is a label of a transition and G is a set of guard statements of the SSUA. Therefore, case (b) implies that evaluations of the controlling guards of potentially conflicting pairs of transitions differ.

Let us assume the following:

$$t_4 = (S_{21}, l_1, S_{22}) \wedge l_1 = (g_1, e_1, _) \rightarrow \mathcal{v}(l_1) = g_1 \quad (21)$$

$$t_2 = (S_{21}, l_2, S_{22}) \wedge l_2 = (g_2, e_1, _) \rightarrow \mathcal{v}(l_2) = g_2 \quad (22)$$

Therefore, we introduce a non-overlapping relation $a \ncong b \rightarrow$ { true, false } such that {a, b} $\subseteq G$ and G is a set of guard statements of the SSUA. Expression $a \ncong b$ is "true" whenever *a* and *b* do not overlap and "false" otherwise. For example, a pair of transitions (t_2, t_4) defined in (21) and (21) respectively are non-overlapping if:

$$\mathcal{v}(l_1) \ncong \mathcal{v}(l_2)$$

Recall that $\gamma: l \rightarrow E$ such that E is the set of events of the SSUA. Therefore, we say for a pair of transitions c such that l_1 and l_2 are as defined in equations (21) and (22) respectively has differing control events if and only if:

$$\gamma(l_1) = e_1 \wedge \gamma(l_2) = e_2 \wedge e_1 \neq e_2$$

6.2 The Match-Making Algorithm

In this section, we overview a method of computing a set of potentially conflicting pairs of transitions and provide an algorithmic solution (i.e., match-maker) to mechanically achieve the underlying tasks. The match-making algorithm consist of the following steps:

(a.) Compute transition pairs for cases: C1.1, C1.2, C2.1, C2.2 and C2.3;

(b.) Eliminate duplicate pairs (e.g., $(t_2, t_4) \equiv (t_4, t_2)$);

(c.) Eliminate pairs with same sources and destinations; and

(d.) Eliminate pairs with different control events.

```
1   NAME: matchMaker
2   INPUT: m^0_SSUA = <n_I, S_I, l_I, s^0_I, U_SI, E_I> //Input is a root machine of an SSUA
3   OUTPUT: L := Set<Map<Transition>,<Transition>> //Potentially conflicting pairs
4   FORALL s ∈ S_I //Explorations of top-level states of root machines begins
5    FORALL x ∈ 𝔓(s) //computes the set of embedded states of s
6      explore( x, L ) //explores each embedded state of s and add potentially conflicting pairs (see Listing 28)
7    END-FORALL
8    explore( s, L ) //explores state s and potentially conflicting pairs (see Listing 28)
9   END-FORALL
10  filter( L ) //removes redundant pairs (see Listing 29)
11  RETURN L
```

LISTING 27. THE MATCH-MAKING ALGORITHM

We present the match-making algorithm (i.e., Listing 27) for the construction of potentially conflicting pairs of transitions for the SSUA (simple or hierarchical). To ease the specification of the match-maker, we define the set of embedded states (i.e., sub-states) of a given composite state s as $\mathfrak{P}(s)$ such that:

$$\mathfrak{P}(s) = \{ x \mid x \trianglelefteq s \} \quad (23)$$

$$\text{given that } \beta(s) \geq 1$$

Similarly, we introduce operator $\mathfrak{m}$ which maps a top-level state to its corresponding state machine. In particular,

$$\mathfrak{m} : s \rightarrow M \text{ such that } \mathfrak{m}(s) = M$$

whenever s is a top-level state of state machine M

```
1   NAME: explore
2   INPUT: state : State,
3          L : Set<Map<Transition>, <Transition>>
4   outgoingEdges = ∅
5   H = 𝔪( state )
6   IF β( state ) == 0 //Case of simple states
7      FORALL t ∈ E_H
8          IF state ∈ F( t )
9              outgoingEdges ∪ { t }
10         END-IF
11     END-FORALL
12     IF | outgoingEdges | > 1
13         match( outgoingEdges, outgoingEdges, L )
14     END-IF
15  END-IF
16  IF  β( state ) > 0  //Case of composite states
17     highLevelEdges = ∅
18     embeddedEdges = ∇( state )
19     FORALL t ∈ E_H
20       IF state ∈ F( t )
21           highLevelEdges ∪ { t }
22        END-IF
23     END-FORALL
24     IF | highLevelEdges | > 0 AND | embeddedEdges | > 0
25         match( highLevelEdges, embeddedEdges, L )
26     END-IF
27     IF  β( state ) > 1  //Case of orthogonal composite states
28        andCrossEdges = α( state )
29        FORALL t ∈ andCrossEdges
30          IF | 𝔭( state, t ) | > 0   // 𝔭( state, t ) is a set of parallel transitions to t
31            match( highLevelEdges, embeddedEdges, L )
32         END-IF
33        END-FORALL
34        IF | andCrossEdges | > 1
35          match( andCrossEdges, andCrossEdges, L )
36        END-IF
37     END-IF
38  END-IF
```

LISTING 28. THE EXPLORATION ALGORITHM

The input is a root state machine of an SSUA expressed in Umple and the output is a finite set of pairs of potentially conflicting transitions given as a set of maps between elements of the domain

and that of the range. The algorithm systematically explores all the top-level states of the root state machine.

The exploration involves the extraction of embedded states of each top-level state (see lines 4-9 of Listing 27) and computing relevant pairs of potentially conflicting transitions. To simplify the algorithm, we present algorithm *explore*(…) in Listing 28. Its goal is to explore a given state and add potentially conflicting pairs resulting from the state to the set of potentially conflicting transition pairs. To realize this, we categorize the states as simple and composite (i.e., orthogonal and non-orthogonal).

The *match*(A, B, L) procedure takes two sets (i.e., domain – "A" and range – "B") as input and matches their elements to compute their Cartesian product. The resulting product is then added to set "L" (i.e., the set of potentially conflicting pairs of transitions). We present the details of matching procedure by example below (i.e., Table 11 and Table 12).

A matching is made between elements of pair (t_i, t_j); whenever t_i is an element in the range and t_j is an element in the domain such that: $i + 1 \leq j \leq |domain|$.

For example, let us consider sets A and B such that:

$$A = \{ t_1, t_2, t_3 \} \text{ and } B = \{ t_4, t_5, t_6 \}$$

By matching sets A and B (i.e., *match*(A, B, _)), assuming there is no matched pair with the same destinations, we will generate pairs presented in Table 11(*a*).

On the other hand, suppose we have the following as the definitions of transitions (where t_5 and t_3 have the same destination states):

$$t_5 = (_, _ x) \land t_3 = (_, _, x) \text{ and } t_1 = (_, _ y) \land t_6 = (_, _, y)$$

The resulting potentially conflicting pairs for sets A and B is presented in Table 11(*b*). We denote the pairs with the same destination states as "*". A pair marked with "*" is being considered redundant and so must be eliminated from the result.

TABLE 11. SYMBOLIC EXAMPLE TO ILLUSTRATE MATCHING BETWEEN DIFFERENT SETS

(a)

B \ A	t_1	t_2	t_3
t_4	(t_4, t_1)	(t_4, t_2)	(t_4, t_3)
t_5	(t_5, t_1)	(t_5, t_2)	(t_5, t_3)
t_6	(t_6, t_1)	(t_6, t_2)	(t_6, t_3)

(b)

B \ A	t_1	t_2	t_3
t_4	(t_4, t_1)	(t_4, t_2)	(t_4, t_3)
t_5	(t_5, t_1)	(t_5, t_2)	*
t_6	*	(t_6, t_2)	(t_6, t_3)

TABLE 12. SYMBOLIC EXAMPLE TO ILLUSTRATE MATCHING AND FILTERING

(a)

A \ A	t_1	t_2	t_3
t_1	-	(t_1, t_2)	(t_1, t_3)
t_2	-	-	(t_2, t_3)
t_3	-	-	-

(b)

A \ A	t_1	t_2	t_3
t_1	-	(t_1, t_2)	*
t_2	-	-	(t_2, t_3)
t_3	-	-	-

However, suppose A is matched with itself (i.e., *match*(A, A, _)), we will obtain Table 12(*a*) where duplicate matches are denoted by “-”.

Similarly, suppose $t_1 = (_, _ z)$ and $t_3 = (_, _, z)$, we will obtain Table 12(b). Since pairing elements in pair (t_i, t_i) will be redundant, we label such entry with "-"; while "*" is as defined previously.

We also filter every entry of the form: $\langle t_i, t_j \rangle \equiv \langle t_j, t_i \rangle$ to eliminate duplicates and pairs with the same destinations. In particular, pair $\langle t_i, t_j \rangle$ is added to the set of potentially conflicting transitions if and only if:

$$t_i = (_, _, s_1) \text{ and } t_j = (_, _, s_2) \text{ such that } s_1 \neq s_2$$

```
1   NAME: filter
2   INPUT: H : Set< Map< Transition >, < Transition > >
3   FORALL x ∈ H
4       // Recall that:          L : (s1, l, s2 ) -> l (see Section 5.1)
5       IF ( x[0] == x[1] )  OR ( γ( L( x[0] ) ) ≠ γ(L( x[1] ) ) ) OR ( X( x[0] ) = X( x[1] ) )
6          H = H – x  // (such that H – x implies the removal of element x from set H)
7     END-IF
8   END-FORALL
9   i = 0, size = length( H )
10  WHILE i  < size - 1
11        j = i + 1
12        WHILE j < size
13            IF ( H[i][0] = H[j][1] ) AND ( H[i][1] = H[j][0] )
14                H = H – H[j]
15        END-IF
16            j = j + 1
17     END-WHILE
18        i = i + 1
19  END-WHILE
```

LISTING 29. THE FILTERING ALGORITHM

We realize filtering of redundant entries systematically by a means of an algorithm in Listing 29. Recall that on line 10 of Listing 27 we applied algorithm *filter*(…) to remove redundant entries in the input set of pairable and potentially conflicting transitions. Since the resulting set of pairs are statically computed and may contain some false positives as defined earlier (pairs that dynamically would never result in non-determinism); it is infeasible to eliminate these statically.

Particularly, the analysis of guard statements to determine overlap demands sophisticated approaches (e.g., theorem proving) whenever they involve infinite-state variables (i.e., integer and

real). To deal with this issue, we transform the SSUAs to nuXmv using our translator (see Chapter 7) and construct an invariant (see Exp. 2) to constrain the overall model for every element of the resulting set of pairs of potentially conflicting transitions. The nuXmv model corresponding to the SUA becomes the model and the invariant becomes a property. These are used as inputs to the nuXmv model checker for the purpose of our analysis (including finite- and infinite-state domains).

6.2.1 Specifying Invariance for Potentially Conflicting Pairs

To facilitate the representation of invariance specifications formally for the purpose of analyzing the SSUA for non-determinism, we generalize the formal specifications of invariance for the pairs of potentially conflicting transitions as follows:

Given that transitions t_i and t_j are of the following forms:

$$t_i = (_,\ _,\ s_i) \text{ and } t_j = (_,\ _,\ s_j)$$

we say whenever t_i and t_j are enabled at step k and executed successfully then

$$\langle\langle t_i, t_j\rangle, k, c\rangle \qquad \textit{Exp. (2)}$$

$$\textit{where } c = \langle\langle_\rangle, \langle \dots s_i, \dots s_j, \dots \rangle, s^{k+1}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

In particular, expression (2) states that if transitions t_i and t_j are enabled at step, k and execute accordingly, then at step $k + 1$, the destination states of these transitions are state values of their corresponding state machines in the *global configuration* (i.e., s_i and s_j in c).

A violation of any invariant implies the presence of non-determinism or otherwise, a false positive. The requirements engineer can analyze a counterexample (or execution trace) resulting from the violation of a constraint for the purpose of diagnosis.

6.2.2 2-Bit Counter Case Study

To facilitate readers' understanding, we illustrate the match-making algorithm with a simplified version of the 2-bit counting machine [90] as a case study. We chose this system because it is characterized with simple and composite (i.e., orthogonal and non-orthogonal) states, as well as and-cross transition. Particularly, we intend to be more specific so that readers can have a detailed

understanding of the processes involved in computing the set of potentially conflicting transitions. Besides, we provide a detailed description of the problem in question as it is relevant to the discussion of other concepts facilitated by our research later in this thesis.

```
1   class TwoBitCounter {
2    Boolean done = false;
3    Boolean tk1 = false;
4    Integer count = 0;
5    BitCounter {
6     Counter {
7       Bit1 {
8         Bit11{ tk0 ->/{ count++;} Bit12;} //t1
9         Bit12 { tk0 ->/{ tk1 = true; count++;}   Bit11; } //t2
10      }
11      ||
12      Bit2 {
13        Bit21 { [tk1 == true]  -> Bit22; } //t3
14        Bit22 { [tk1 == true]  ->/{ done = true; } Max;} //t4
15      }
16      ||
17     Status {
18       Counting {
19         Zero { [ even_odd(count) == 0]   -> Odd; } //t6
20         Odd { [ even_odd(count) == 1]   -> Even; } //t7
21         Even { [even_odd(count) == 0]    -> Odd; } //t8
22       }
23       Max { reset -> /{count = 0; }  Counting; } //t5
24    } } }
25    public int even_odd( int x ) {
26     return x % 2 == 0 ? 1 : 0;
27    }
28  }
```

LISTING 30. UMPLE REPRESENTATION OF 2-BIT COUNTER SYSTEM

6.2.2.1 The Specification of 2-Bit Counter

A 2-bit counter (i.e., ripple counter) combines two flip-flops whereby a transformation of the output of a flip-flop generates an input to the other. Esmaeilsabzali et al.'s design [90] suggested that states Bit_1 and Bit_2 represents the least- and most-significant bits respectively. *Status* is a parallel region to Bit_1 and Bit_2. It monitors the processes of the *Counter* to determine whether it

is in the process of *counting* or has already counted four ticks and should be *reset*. For Bit_1, possible state values are Bit_{11} and Bit_{12} with Bit_{11} as its initial state. In the same vein, the possible states of Bit_2 are Bit_{21} and Bit_{22} with Bit_{21} as its initial state. Finally, Status can assume states *Counting* and *Max* where *Counting* is its initial state. *Counting* keeps the record of the status to determine whether it is odd or even.

Whenever event tk_0 corresponding to clock ticks is received, the counter increments by one. After an even number of ticks, Bit_1 sends event tk_1 thereby instructing Bit_2 to change its state. Similarly, after counting four ticks, the Counter generates event "*done*". Consequently, the Counter is automatically reset by changing Bit_1 and Bit_2 to their initial states; while Status assumes state Max.

To realize the desired configuration where Bit_1 and Bit_2 assumes their initial states and Status becomes Max, *and-cross transition* becomes the most relevant solution due to the abstraction it offers. In Listing 30, we present a textual representation of 2-Bit Counter system with and-cross transition in Umple. A discussion of the design decision resulting in it is presented in Section 6.2.2.2.

6.2.2.2 Designing the 2-Bit Counter System

The corresponding Umple code for the system being discussed is presented in Listing 30. We implemented it within a class (i.e., TwoBitCounter – see line 1) such that the SSUA becomes an element of the class which is represented by its root state machine (i.e., BitCounter – see lines 5-24).

The root machine has a single top-level state – Counter (see lines 6-24) which also serves as its initial state. Counter is modeled as an orthogonal composite state whose regions Bit_1 (see lines 7-10), Bit_2 (see lines 12-15) and Status (see lines 13-24). By default, Umple initializes regions to "null" state since they become activated only when control is transferred to their parent state or its sub-state(s). Therefore, adding "null" to the set of states of a region. Change of states for each region is model by transitions such that whenever a transition is enabled and execute, the state of the corresponding region changes to the value of the target state of the transition. A transition

become enabled whenever its source state is activated (see Section 5.1), event is received and its controlling guard evaluates to *true*.

We represent generated events (i.e., tk_1 and *done*) as Boolean attributes (see lines 2, 3) of the containing class and initialize them to *false* (indicating such event is yet to be generated). Each of them is generated whenever a transition leading to its generation executes. To represent this semantics, we added an action (e.g., line 9) whose goal is to set its value to *true* whenever the leading transition executes.

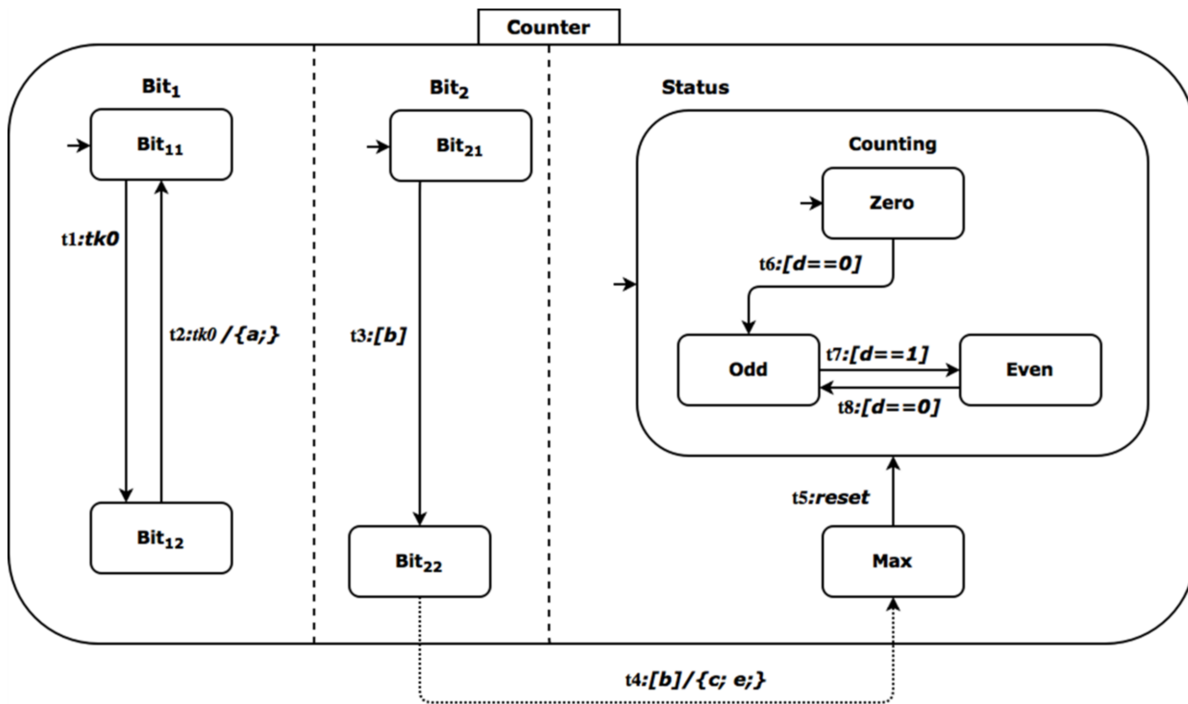


FIGURE 40. 2-BIT COUNTER DESIGNED BASED ON AND-CROSS TRANSITION

To represent the arrival of a generated event for consumption (i.e., to enable a transition), a corresponding guard statement is constructed to test whether the corresponding attribute evaluates to *true* (see line 14).

In Figure 40, we present a graphical representation of the solution to the model of state *Counter* based on and-cross transition (i.e., *t4*). We denote additional transitions with dotted arrow in the corresponding graphical representations of the system. For ease of representation of *Counter*

pictorially for the solutions, we denote relevant guards and action statements symbolically as follow:

$$a := tk_1 = true$$
$$b := tk_1 == true$$
$$c := done = true$$
$$d := even_odd(\ count\ )$$
$$e := count = 0$$

6.2.2.3 Decomposing the 2-Bit Counter Example

To facilitate the demonstration, first we represent the system as an SSUA so as to give readers insight to its constituents. For the SSUA, the following defines its constituents:

$$V_{SSUA} = \{\ (done, Boolean), (tk1, Boolean), (count, Integer)\ \}$$
$$S_{SSUA} = \{\ Counter, Bit11, Bit12, Bit21, Bit22, Counting, Zero, Odd, Even, Max\ \}$$
$$M_{SSUA} = \{\ BitCounter, \$Bit1, \$Bit2, \$Status, \$Counting\ \}$$
$$m^0_{SSUA} = BitCounter\ (i.e., the\ root\ state\ machine)$$
$$L_{SSUA} = \{\ tk0, tk0/\{a;\}, [b], [b]/\{\ c; e;\ \}, reset, [d == 0], [d == 1]\ \}$$

We define R_{SSUA} for the Bit Counter system as follow:

$$t_1 = (Bit11, tk0, Bit12)$$
$$t_2 = (Bit12, tk0/\{a;\}, Bit11)$$
$$t_3 = (Bit21, autoTransition[b], Bit22)$$
$$t_4 = (Bit22, autoTransition[b]/\{c; e;\}, Max)$$
$$t_5 = (Max, reset, Counting)$$
$$t_6 = (Zero, autoTransition[d == 0], Odd)$$
$$t_7 = (Odd, autoTransition[d == 1], Even)$$
$$t_8 = (Even, autoTransition[d == 0], Odd)$$

In the same vein, we present specific information with respect to each state machine of the SSUA as follow:

Let "A" denote the state machine corresponding to BitCounter (i.e., the root state machine of the SSUA), then we have the following:

$$n_A = \text{\$BitCounter}; s_A^0 = \text{Counter}; S_A = \{ \text{ Counter } \}$$
$$U_{SA} = \text{Counter, Bit11, Bit12, Bit21, Bit22, Counting, Zero, Odd, Even, Max } \}$$
$$l_A = L_{SSUA}; E_A = \{ \ t_i \mid 1 \leq i \leq 8 \ \}$$

Let "B" denote the state machine corresponding to region Bit1 of the SSUA, then we have the following:

$$n_B = \text{\$Bit1}; s_B^0 = \text{Bit11}; S_B = \{ \text{ Bit11, Bit12 } \}; U_{SB} = \{ \text{ Bit11, Bit12 } \}$$
$$l_B = \{ \text{ tk0 } \}; E_B = \{ t_1, t_2 \}$$

Let "C" denote the state machine corresponding to region "Bit2" of the SSUA, then we have the following:

$$n_C = \text{\$Bit2}; s_C^0 = \text{Bit21}; S_C = \{ \text{ Bit21, Bit22 } \}; U_{SC} = \{ \text{ Bit21, Bit22 } \}$$
$$l_C = \{ \text{ [b], [b]/\{ c; e; \} } \}; E_C = \{ t_3 \}$$

Let "D" denote the state machine corresponding to region "Status" of the SSUA, then we have the following:

$$n_D = \text{\$Status}; s_D^0 = \text{Counting}; S_D = \{ \text{ Counting, Max } \};$$
$$U_{SD} = \{ \text{ Counting, Zero, Odd, Even, Max } \}$$
$$l_D = \{ \text{ [d == 0], [d == 1], [b]/\{ c; e; \}, reset } \}; E_D = \{ t_i \mid 4 \leq i \leq 8 \}$$

Let "E" denote the state machine corresponding to region "Counting" of the SSUA, then we have the following:

$$n_E = \text{\$Status}; s_E^0 = \text{Counting}; S_E = \{ \text{ Zero, Even, Odd } \};$$
$$U_{SE} = \{ \text{ Zero, Even, Odd } \}$$
$$l_E = \{ \text{ [d == 0], [d == 1] } \}; E_E = \{ t_i \mid 6 \leq i \leq 8 \}$$

Now, the input to the match-maker algorithm for this case study is "BitCounter" (i.e., the root state machine).

Input: I := $<n_A, S_A, l_A, s_A^0, U_{SA},\ E_A>$ //(Recall that A denotes "BitCounter")

Output: L := Set<Map<Transition>, <Transition>> // this is initially empty

L = { }

Then, we explore all the top-level states systematically one after the other. For the SSUA in question, S_A is the set of top-level states. To achieve this, we do the following for each of the top-level states of the SSUA:

1. Explore embedded states of each top-level state;
2. Explore the top-level state whose embedded states were explored in (1);
3. Statically eliminate false positives; and
4. Generate invariance specification for the resulting set of potentially conflicting transitions.

6.2.3 Applying the Match-Maker Algorithm on 2-Bit Counter Machine

Given the SSUA in our example (i.e., Figure 40), the set of top-level states is as follows:

$$S_A = \{ \text{Counter} \};$$

Step 1 - Explore Embedded States of Counter

We execute **Step 1** as follows:

$$\mathfrak{P}(\text{Counter}) = \{ \text{Bit11, Bit12, Bit21, Bit22, Counting, Zero, Odd, Even, Max} \}; \text{ and}$$

$$\text{embeddedStates} = \mathfrak{P}(\text{Counter})$$

Let us assume the elements of $\mathfrak{P}$(Counter) are indexed as follows to facilitate discussion:

k	*0*	*1*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
state	*Bit11*	*Bit12*	*Bit21*	*Bit22*	*Counting*	*Zero*	*Odd*	*Even*	*Max*

$$\text{outgoingEdge} = \{\}$$

At k = 0,

$$k = 0 \rightarrow (\text{state} = \text{Bit11} \wedge \mathfrak{m}(\text{state}) = \$\text{Bit1} \wedge \text{H} = \$\text{Bit1})$$

$$\beta(\text{Bit11}) = 0 \text{ and } E_H = \{ t_1, t_2 \}$$

$$(\forall \text{ t} \in \{ t_1, t_2 \} \cdot \text{F(t)} = \text{Bit11}) \rightarrow \text{outgoingEdges} \cup \text{t}$$
$$\text{F}(t_1) = \text{Bit11 but F}(t_2) \neq \text{Bit11}$$
$$\text{outgoingEdges} = \{ t_1 \}$$
$$\text{since} \mid \text{outgoingEdges} \mid = 1, \text{we skip the matching}$$

Similarly, for all states at index *k*, such that *k* = { 1, 3, 4, 6, 7, 8, 9 } in this example (i.e., Figure 40) the matching will be skipped because they all have only one outgoing transition and they are simple states. Hence, we focus on Counting, a composite state for the application of the algorithm.

At k = 5,

$$\text{highLevelEdges} = \{\}$$
$$\text{embeddedEdges} = \{\}$$
$$\text{k} = 5 \rightarrow (\text{state} = \text{Counting} \wedge \mathfrak{m}(\text{state}) = \$\text{Status} \wedge \text{H} = \$\text{Status})$$
$$\beta(\text{Counting}) = 1 \text{ and } E_H = \{ t_i \mid 4 \leq i \leq 8 \}$$
$$(\nexists \text{ t} \in \{ t_i \mid 4 \leq i \leq 8 \} \cdot \text{F(t)} = \text{Counting}) \rightarrow \text{highLevelEdges} = \{\}$$
$$\mathfrak{P}(\text{Counting}) = \{ \text{Zero, Odd, Even} \}$$

Hence, we extract all transitions whose source state is a sub-state of Counting as follows:

$$\forall \text{ t} \in \{ t_i \mid 4 \leq i \leq 8 \}$$
$$(\forall \text{ s} \in \{ \text{Zero, Odd, Even} \}, \exists \text{ t} \in \{ t_6, t_7, t_8 \} \cdot \text{F(t)} = \text{s}) \rightarrow \text{embeddedEdges} \cup \text{t}$$
$$\text{embeddedEdges} = \{ t_6, t_7, t_8 \}$$

But since | highLevelEdges | = 0, although | embeddedEdges | = 3, therefore no matching happens. At the end of step 1, readers should note that:

$$\text{L} = \{ \}$$

Step 2 - Exploring Counter

We then explore state “Counter”, since its embedded states were explored in **Step 1**. It can be observed that Counter is an orthogonal composite state.

$$\beta(\text{Counter}) > 0$$

Thus, we have the following:

$$\text{highLevelEdges} = \{\}$$
$$\text{embeddedEdges} = \{\}$$
$$(\text{state} = \text{Counter} \wedge \mathfrak{m}(\text{state}) = \text{BitCounter} \wedge \text{H} = \text{BitCounter})$$
$$\beta(\text{Counter}) = 3 \text{ and } E_H = \{\, t_i \mid 1 \leq i \leq 8 \,\}$$
$$(\nexists\, \text{t} \in \{\, t_i \mid 1 \leq i \leq 8 \,\} \cdot \text{F(t)} = \text{Counter}) \rightarrow \text{highLevelEdges} = \{\}$$
$$\mathfrak{P}(\text{Counter}) = \{\, \text{Bit11, Bit12, Bit21, Bit22, Counting, Zero, Odd, Even, Max} \,\}$$

We extract all transitions whose source state is a sub-state of Counter as follows:

$$\forall\, \text{t} \in \{\, t_i \mid 1 \leq i \leq 8 \,\}$$
$$(\,\forall\, \text{s} \in \mathfrak{P}(\text{Counter}), \exists\, t \in \{\, t_i \mid 1 \leq i \leq 8 \,\} \cdot \text{F(t)} = \text{s}\,) \rightarrow \text{embeddedEdges} \cup \text{t}$$
$$\text{embeddedEdges} = \{\, t_i \mid 1 \leq i \leq 8 \,\}$$

But since | highLevelEdges | = 0, although | embeddedEdges | = 8, therefore no matching happens. At the end of this phase, readers should note that:

$$\text{L} = \{\ \}$$

Recall that β(Counter) = 3 which implies that Counter is an orthogonal composite state. As a result of this, we further the exploration as follows:

$$a(\text{Counter}) = \{\, t_4 \,\}$$
$$\text{paralelTransitions} = \mathfrak{p}(\,\text{Counter}, t_4\,) = \{\, t_1, t_2, t_5, t_6, t_7, t_8 \,\}$$
$$\text{match}(\,\{t_4\}, \text{paralellTransitions, L}\,)$$

Therefore, at this point we have the following:

$$\text{L} = \{\, (t_4, t_1), (t_4, t_2), (t_4, t_5), (t_4, t_6), (t_4, t_7), (t_4, t_8) \,\}$$

We further, the pairing by matching the set of and-cross transitions to prevent simultaneous execution of its elements.

$$\text{match}(\,\{t_4\}, \{t_4\}, \text{L}\,)$$

At this point set L is updated with pairs of and-cross transition of the same enclosing state. Thus, we have the following:

$$\text{L} = \{ (t_4, t_1), (t_4, t_2), (t_4, t_5), (t_4, t_6), (t_4, t_7), (t_4, t_8), (t_4, t_4) \}$$

Step 3 – Statically Eliminate False Positives

We apply filter on set L, the resulting set from **Step 2**. This process involves the removal of pairs with the same element; different triggers; and the same source and destination states.

Step 3.1 – Eliminate pairs with the same elements

We first eliminate pairs with the same element. In this case, (t_4, t_4) is eliminated from set L such that the resulting set is:

$$\text{L} = \{ (t_4, t_1), (t_4, t_2), (t_4, t_5), (t_4, t_6), (t_4, t_7), (t_4, t_8) \}$$

Step 3.2 – Eliminate pairs with different triggers

Given that:

$$\forall \text{ t} \in \{ t_3, t_4, t_6, t_7, t_8 \} \cdot t = (_, l_i, _), \gamma(l_i) = \text{autoTransition}$$

This implies that these transitions may fire simultaneously. Thus, every pair in L involving these transitions must be preserved for the purpose of dynamic analysis since they are candidates of non-determinism. For example, pairs (t_4, t_6), (t_4, t_7) and (t_4, t_8) are elements of L in this category and thus must be preserved.

On the other hand, the following is the case under consideration:

$$\nexists \{t_i, t_j\} \subseteq \{ t_1, t_2, t_5 \} \cdot t_i = (_, l_\text{i}, _) \text{ and } t_j = (_, l_j, _)$$
$$\text{where } \gamma(l_i) = \gamma(l_j)$$

Therefore, transition pairs involving t ∈ { t_1, t_2, t_5 } will never execute simultaneously and should be eliminated from the resulting pairs. After this exercise, the resulting set for Figure 40 is given as:

$$L = \{ (t_4, t_6), (t_4, t_7), (t_4, t_8) \}$$

Step 3.3 – Eliminate pairs with same source and destination states

Considering the participating transitions of the remaining pairs in L (from Step 3.2), there is no transition with the same destination states.

$$\nexists \{t_i, t_j\} \subseteq \{ t_4, t_6, t_7, t_8 \} \cdot X(t_i) = X(t_j)$$

Therefore, the final set L is the same as the resulting set from Step 3.2. For each of the pairs in L, an invariance specification is composed for the purpose of eliminating false positives by dynamic analysis.

Step 4 – Generating invariance specifications for resulting set

Recall that the definitions of the given transitions are given below.

$$t_4 = (\text{Bit22}, \text{autoTransition}[\text{b}]/\{\text{c; e;}\}, \text{Max})$$
$$t_6 = (\text{Zero}, \text{autoTransition}[\text{d} == 0], \text{Odd})$$
$$t_7 = (\text{Odd}, \text{autoTransition}[\text{d} == 1], \text{Even})$$
$$t_8 = (\text{Even}, \text{autoTransition}[\text{d} == 0], \text{Odd})$$

Therefore, given that the resulting set of potentially conflicting transitions is L, the invariance specifications are given as follows:

$$L = \{ (t_4, t_6), (t_4, t_7), (t_4, t_8) \}$$

If (t_4, t_6) execute at step *k*, then at step k + 1 “Max” and “Odd” must be activated.

$$\langle\langle t_4, t_6\rangle, k, a\rangle$$
$$\text{where a} = \langle\langle_\rangle, \langle \ldots Max, \ldots Odd, \ldots\rangle, s^{k+1}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

Similarly, for pairs (t_4, t_7) and (t_4, t_8) a successful execution of the participating transitions at step k will yield the following configurations at step $k+1$ respectively.

$$\langle\langle t_4, t_7\rangle, k, b\rangle$$

$$\text{where b} = \langle\langle_\rangle, \langle \ldots Max, \ldots \text{Even}, \ldots\rangle, s^{k+1}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

and

$$\langle\langle t_4, t_8\rangle, k, c\rangle$$

$$\text{where c} = \langle\langle_\rangle, \langle \ldots Max, \ldots Odd, \ldots\rangle, s^{k+1}, \langle\langle_\rangle, \langle_\rangle\rangle\rangle$$

6.3 Reachability Analysis of States of an SSUA

By "reachability of states", we mean the process of determining whether there is at least a path to a given state of the SSUA from the initial configuration of the SSUA. A developer may desire a situation where a specific configuration of the system is reachable; however, this is dependent on the domain of interest. In this aspect, our goal is to ensure that modellers develop systems whose states are reachable independently. This property is domain-independent; hence we consider it important to mechanize the process of discovering such flaws in the SSUAs.

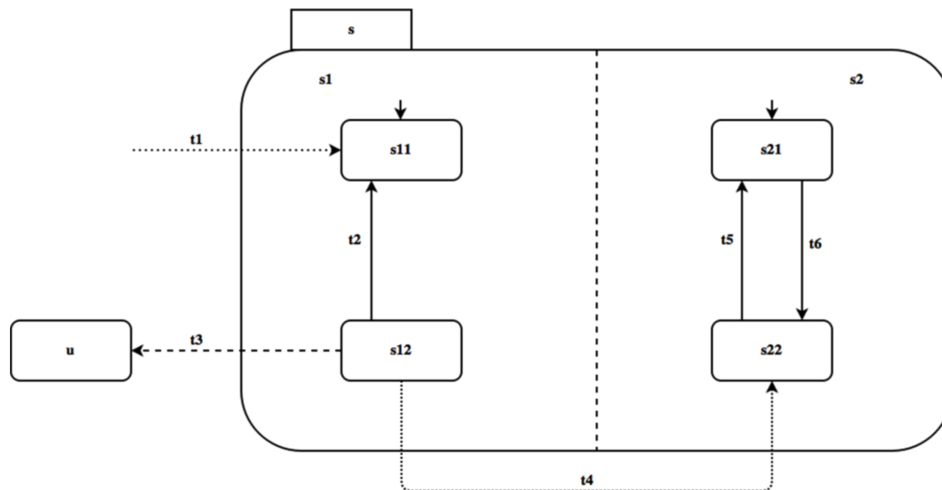


FIGURE 41. SYMBOLIC EXAMPLE TO ILLUSTRATE UNREACHABILITY OF STATES

Let us consider the design of an SSUA presented in Figure 41 to illustrate the situation of interest. It models an orthogonal state with both reachable and unreachable states. For example, states like "s11", "s21" and "s22" are reachable; while states "s12" and "u" are unreachable. A good design requires that each state of the SSUA be reachable independently. In particular, there must be at least a path from the initial configuration of this SSUA to the states of the SSUA (including "s12" and "u").

To systematically uncover these flaws, a configuration of the form presented in expression (3) is constructed in the form of a specification for every state of the SSUA.

$$\langle\langle_\rangle,\langle_\rangle,s^0,\langle\langle_\rangle,\langle_\rangle\rangle\rangle \rightarrow \cdots \langle\langle_\rangle,\langle \ldots b \ldots \rangle,s^k,\langle\langle_\rangle,\langle_\rangle\rangle\rangle \qquad \text{Exp. (3)}$$

$$\forall_b \cdot b \in S_{SSUA} \wedge k \in \mathbb{N}$$

This implies that there is a future step, k from the initial configuration of the SSUA where b becomes an active state (i.e., receives control) of the global configuration of the SSUA. Therefore, at initial step (i.e., s^0) the only active state is the initial state of the root state machine of the SSUA. We express this formally in expression (4).

$$\langle\langle_\rangle,\langle_\rangle,s^0,\langle\langle_\rangle,\langle_\rangle\rangle\rangle \rightarrow \cdots \langle\langle_\rangle,\langle \ldots c \ldots \rangle,s^0,\langle\langle_\rangle,\langle_\rangle\rangle\rangle \qquad \text{Exp. (4)}$$

$$\text{Where: c} = s_M^0 \wedge M = m_{SSUA}^0$$

6.3.1 Limitations of this approach

While the approach is deployable both on simple and hierarchical SSUAs at large, we consider it important to specify the main limitation of the method we developed for the reader's sake. Readers should recall that our goal is to provide a fully automatic method to discover reachability of states. Hence, we chose model checking for its automated capability to analyze large-scale systems.

To specify constraints to be executed on the model (i.e., hierarchical and simple SSUAs) by model checking demands a more expressible logic like computational tree logic (i.e., CTL) since the hierarchy is to be explored exhaustively. However, the technology we adopted (i.e., nuXmv [45]) does not facilitate the analysis of CTL specifications whenever the SSUA is characterized with infinite domains (i.e., integer and real). Although Vakili and Day [91] showed the possibility of analyzing these models automatically with Z3 [92] an SMT-based solver, we defer its exploration

to the future. Therefore, we limit our work to a subset of SSUAs that are not characterized with infinite domains.

6.4 Improving Qualitative Analysis via And-Cross Transition

In this aspect of our work, we focus on the benefits derivable from and-cross transitions as a means of realizing a more qualitative abstract design. By and-cross transitions, we mean transitions whose source and destination states are located in distinct parallel regions of the same enclosing orthogonal state. Various methods of increasing abstraction have been studied, including hierarchical state machines. The state hierarchy approach reduces the number of transitions and the number of states required to represent a system.

Depending on the encoding strategy used for model-checking, the global state-space can grow exponentially with an increase in the number of transitions, and fewer transitions result in a more compact state-space. This is the case with the encoding proposed by Faghih and Day (i.e., BSML2SMV [86]). Another tactic that can significantly reduce the number of transitions required for encoding certain systems is the use of and-cross transition.

6.4.1 Motivation for And-Cross Transition

And-cross transitions are those whose source and destination states are located in distinct parallel regions of the same enclosing orthogonal state. Harel's original state-chart semantics (see [87], [1]) for and-cross transitions involve re-initializing every parallel region to its start state, while setting the target region (i.e., host machine of the destination of the and-cross transition) to the specified destination state. This is illustrated with transition "t4" in Figure 40.

The ability to specify and-cross transitions has, however, been dropped from recent model-driven engineering (MDE) tools for reasons that include limited use-in-practice, unmanageable underlying complexity (e.g., Mueller [93]), and availability of alternative modeling solutions. This is exemplified by the removal of and-cross transitions support from the Object Management Group's (OMG) specifications for the Unified Modeling Language (UML), version 2.5 [8].

An alternative modeling solution to an and-cross transition is simply to add all the needed transitions to reinitialize each parallel region. But there is a price to pay for this, in terms of an

increased number of transitions. Therefore, the effort demanded to perform model-checking on such an alternative model increases with respect to the number of regions of the immediate enclosing state. The effort is constant whenever an and-cross transition is adopted.

Furthermore, since the central goal of MDE is to provide as high a level of abstraction as possible, we deem abstraction offered by this mechanism important. Therefore, in our previous work (i.e., Adesina et al. [94]), we advocated the use of and-cross transition for modeling state machine diagrams when abstraction relevant to its usage is demanded. We also argued that the underlying complexity can be managed adequately.

In this thesis, we go further and argue that and-cross transitions do have a place in the suite of modeling capabilities for state machines, and that their internal complexity can be managed and formally specified. In fact, our results with a sample scenario show that the use of and-cross transitions drastically reduces the number of transitions required to model the scenario in question without jeopardizing semantics.

6.4.2 Modeling Solutions to the 2-Bit Counter Problem (see Section 6.2.2)

In this section, we present various modeling solutions to designing orthogonal composite state – *Counter* (see Figure 40). We are aware of five modeling solutions (including the usage of and-cross transitions. However, since our goal is to explore benefits of adopting and-cross transitions, we derive other solutions from the one we designed with and-cross transitions. In particular, we transform the system (i.e., Figure 40) we designed based on an and-cross transition to its equivalent using alternative modeling approaches.

In Section 6.2.2, we present the problem of the 2-Bit Counter machine. This involves its design and corresponding model using and-cross transition (i.e., Figure 40). As can be seen visually, the additional transition required to model the system is *one* and-cross transition. We refer to the alternative modeling solutions, that avoid the use of the and-cross transition as **A1**, **A2**, **A3** and **A4**.

6.4.2.1 Alternative Solution 1 – A1

The first alternative method will require in each region that is neither the source nor the destination of the and-cross transition, a transition from each non-initial but top-level state(s) to the region's initial state. For the destination region of and-cross transition (to be substituted), an additional transition is needed from each state that is not the destination of the and-cross transition to the destination state of the and-cross transition.

One additional transition is also needed for the region of the source state of and-cross transition to the initial state of its region. We applied these rules to Figure 40 and the resulting system is presented in Figure 42. As can be seen, *three* additional transitions are required as opposed to *one* and-cross transition.

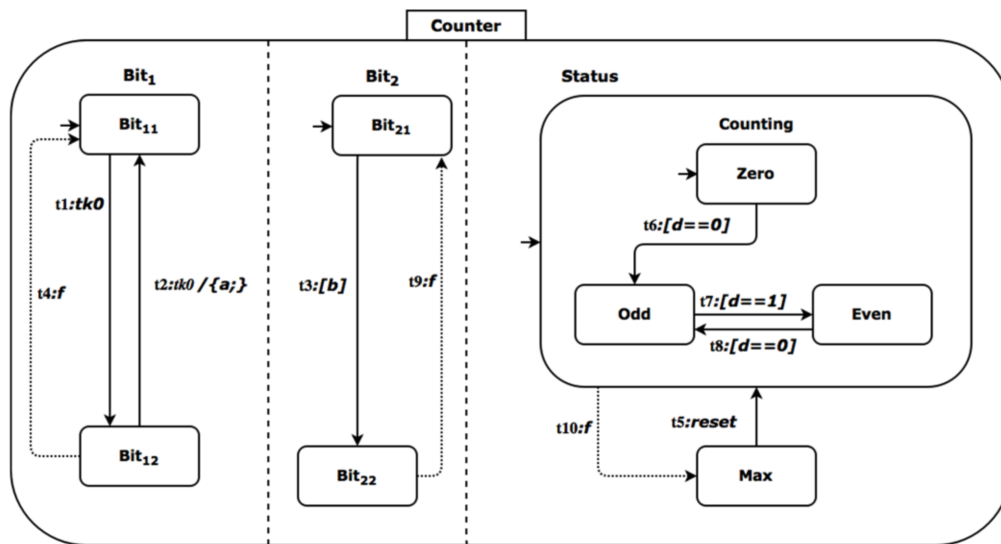


FIGURE 42. 2-BIT COUNTER DESIGNED BASED ON A1

From Figure 42, it can be seen that Bit1 is the only region which is neither the source nor destination of *t4*. Therefore, there is an additional transition originating from Bit12 (i.e., non-initial state of Bit1) to Bit11, the initial state of Bit1. Similarly, the target region of the and-cross transition is Status. For this region, an additional transition is required to originate from each of its the top-level states (except the destination state of the and-cross transition) to the destination state

of the and-cross transition. In this example, a transition will originate from Counting to Max. Finally, a transition will originate from the source of the and-cross transition (i.e., Bit22) to the initial state of its region (i.e., Bit21).

For these additional transitions, a Boolean statement is *and-ed* with existing guard statement of the transition (where applicable). Otherwise, it becomes the guard for additional transitions. To illustrate this by example, we introduce an operator $\mathcal{v}: r \rightarrow s$ mapping region r to its current state s such that $\mathcal{v}(r) = s$. Considering the and-cross transition (i.e., *t4* of Figure 40), the label for each additional transition will be reformulated as follows (based on t4):

$$\text{f} := [\, \text{g} \wedge \text{b} \,] \,/\{\, \text{c; e;} \,\} \tag{24}$$

where:

$$\text{g} := \mathcal{v}(Bit_2) == Bit_{22}$$

In general, the number of additional transitions (denoted as A_T) for method **A1** as follows:

$$A_T = \sum_{i=1}^{n-1} \left| S_{M_i} \right| - n + 2 \tag{25}$$

Where: M_i denotes sub-state machine at index i and the state machine of the source mode is at index n.

For example, n = 3 for Counter; thus, there are M_1 and M_2 which correspond to regions Bit1 and Status respectively; while M_3 represents region Bit2.

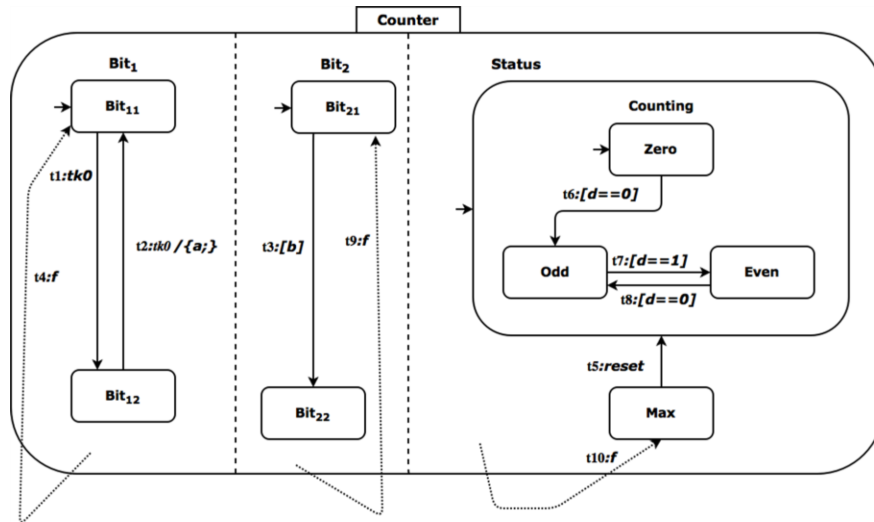


FIGURE 43. 2-BIT COUNTER DESIGN BASED ON A2

6.4.2.2 Alternative Solution 2 – A2

Another alternative method, we refer to as **A2** is based on the abstraction accessible via hierarchy. In particular, it requires *one* additional transition from each non-targeted region to its initial state. It will also require *one* additional transition from the target region to the destination state of the and-cross transition to be substituted. Consequently, there will be *n* transitions where *n* is the number of regions of the enclosing state.

These transitions will be controlled by a statement of the form "*g*" that we discussed in Section 6.4.2.1. For example, if we apply **A2** to Figure 40, the guard will be the same as "*f*" (see equation 24). Figure 43 is the result obtained after transforming Figure 40 with method **A2** to illustrate the solution it offers.

As can be seen, transitions (with dotted arrows) originate from the regions (but not specific states). By this, we mean these transitions originate from any of the states enclosed within its source region. This can be specified in Umple, however, we are not aware of UML construct that facilitates its representation. For this example, *three* additional transitions are required. *Bit1* and *Bit2* are the non-targeted regions of the and-cross transition (see *t4* in Figure 40). Thus, the added

transitions have *Bit11* and *Bit21* as their destination states. However, for the target region where *Max* is the target state, the transition originating from the region has Max as its destination state.

Although the number of additional transitions required is exactly the same as **A1**, readers should note that in this example, the number coincides with the number of regions of *Counter*. **A1** depends on the number of top-level states of the regions being discussed. In general, the number of additional transitions with A2 is:

$$A_T = n \tag{26}$$

where n is the number of regions of the enclosing state

6.4.2.3 Alternative Solution 3 – A3

The third alternative solution requires the addition of a transition to each leaf node (i.e., simple state) of the regions of the enclosing orthogonal composite state except the regions where the source and target states of the and-cross transition in question (or to be substituted) belong. For each of these of these transitions, the destination state will be the initial state of the enclosing region. In essence, such leaf nodes must not be initial states of their respective regions as this will result in redundancy.

However, for the target region of the and-cross transition, a transition is added to its leaf nodes (including its initial state, when the target state of the and-cross transition is not the initial state of the region). The destinations of these additional transitions will be the destination state of the and-cross transition. But for the source region of the and-cross transition, we add one transition whose origin is the source state of the and-cross transition and its destination is the initial state of its region. The enabledness of each of the added transitions will be controlled by a condition of the form "*g*" (as discussed in Section 6.4.2.1). In Figure 44, we apply method **A3** to the 2-Bit Counter example (see Figure 40).

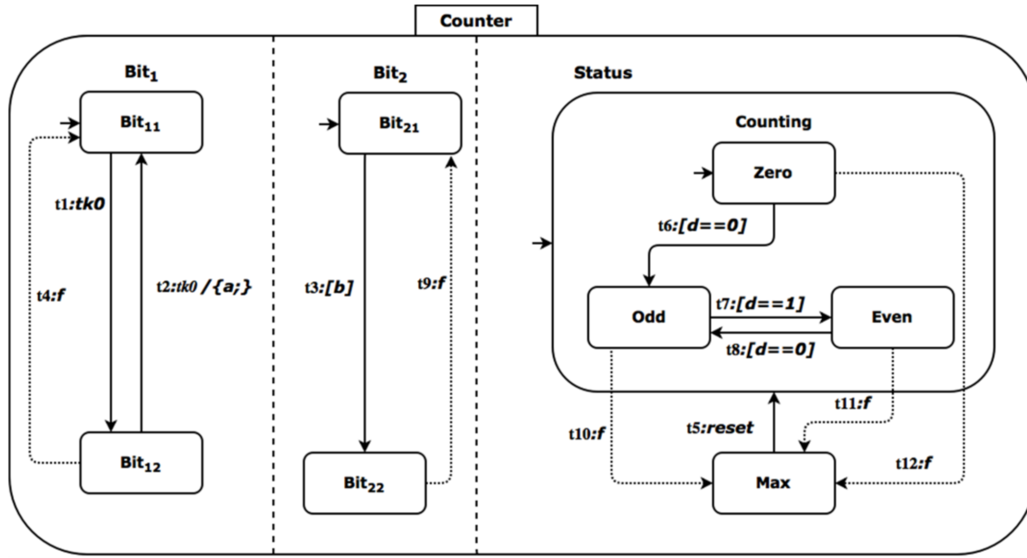


FIGURE 44. 2-BIT COUNTER DESIGN BASED ON A3

As can be observed from Figure 44, there are *five* additional transitions. Bit1 is neither the region of the source nor the region of target state of and-cross transition being substituted. Hence, there is a transition from *Bit12* (i.e., a non-initial leaf node of *Bit1*) to *Bit11* (i.e., initial state of *Bit1*). For *Status* (i.e., the region of *Max* – target state of and-cross transition), there is a transition from each of its leaf nodes to *Max*. Whereas for region *Bit2* (i.e., the region of *Bit22* – source state of and-cross transition), there is only *one* additional transition (i.e., *t9*) from *Bit22* to *Bit21* (i.e., the initial state of *Bit2*). Similar to **A1** and **A2**, in general, the number of additional transitions is as follows. Readers should note that some of the notation used here is defined later.

Let there be *n* regions in the enclosing orthogonal composite state such that regions other than the source and destination of the and-cross transition under consideration be R_i with $1 \leq i \leq n - 2$. Then, let there be *x* (i.e., a set of additional transitions), such that:

Given that l is the newly formulated label for additional transitions, then

$$x = \{\ t \mid \forall_{s \in S_{R_i}} \cdot \beta(s) = 0, t = (s, l, s^0_{R_i})\ \}$$

Similarly, let there be y (i.e., a set of additional transitions for the target region of and-cross transition t' being transformed) such that:

Given that R_{n-1} is the region corresponding to the destination of and-cross transition being transformed (i.e., t') and "l" being the newly formulated label for additional transitions, then

$$y = \{ t \mid \forall_{s \in S_{R_{n-1}}} \cdot \beta(s) = 0 \wedge \exists_d \cdot d = \mathrm{X}(t'), t = (s, l, d)) \}$$

Therefore, A_T denoting the number of additional transitions is expressible as follows:

$$A_T = |\,x\,| + |\,y\,| + 1 \quad (27)$$

We note that the cardinality of a set x is expressible as $|x|$ and "1" was added to indicate the additional transition from the source state of and-cross transition to the initial state of its region.

6.4.2.4 Alternative Solution 4 – A4

This method is based on the principle of flattening. In particular, the entire hierarchy is flattened and transitions added from possible combinations of states to the set of desired states. While this is a candidate solution, we ignore the resulting model of **A4** because it results in an exponential blow-up (both in state-space and transitions). Hence, we focus on those solutions with closely-related differences with our work that facilitate a reasonable level of abstraction.

6.5 Summary of Quality Assurance

In this chapter, we discussed our approach to guarantee that SSUAs comply with basic and domain-independent properties. These include: ensuring that SSUAs are free of non-determinism; ensuring that states of the SSUAs are reachable; and promoting abstraction via and-cross transitions.

We discussed various cases of non-determinism, including same-source transitions and region-cross transitions. We tackled complexities resulting from and-cross transitions. In particular, we were able to discover situations that may lead to inconsistencies as a result of and-cross transitions. We realize this by formulating an algorithm that systematically explores the hierarchy originating from the root state machine of an SSUA and produces a set of potentially conflicting transitions. We filtered false positives resulting from set statically but delegate complex cases to the model

checker for dynamic analysis. The algorithm was applied on a case study of a Bit Counter System (i.e., Figure 40) to demonstrate its applicability on a real system.

On the aspect of reachability analysis, we developed an approach that is applicable to analyze SSUAs for the purpose of discovering cases of unreachability of states. Our approach is limited to the subset of SSUAs that are not characterized with infinite domains. We will explore Z3 (proposed by Vakili and Day[91]) in the future to analyze systems with infinite domains for reachability of states.

We advocated for the support of and-cross transitions in developing systems because it offers a high level of abstraction that must not be replaced with the details offered by alternative approaches for the purpose of design and formal analysis. Given that the goal of MDE is to facilitate abstraction as a means of realizing a high-quality system, we believe and-cross transitions have a place in the suite of modeling capabilities for state machines and that its internal complexity can be managed and formally specified. To compare and-crossing and other modeling approaches that may substitute and-cross, we applied them on a bit counter. The results showed that and-cross transitions can reduce the number of transitions drastically when modeling SSUAs. Although, and-cross transition reduces the number of transitions required to model an SSUA. the complexity introduced by two or more of such transitions within the same enclosing state is not explored in this work. Thus, we defer this to the future.

We acknowledge that some other properties can be analyzed automatically but not covered in this chapter. These include race analysis, consistency and completeness checks, etc. We intend to explore the possibility of analyzing systems for these properties in the future.

7 Transformation of State Machine Models

In this chapter, we demonstrate our approach to mapping Umple constructs to their SMV equivalent by example. First, we give a formal overview of SMV models to facilitate reader's understanding. We also present a state machine example to illustrate transformations of various modeling constructs for state machine development.

Readers should note that the transformations presented here may be possible with other SMV constructs. However, we consider our choices of constructs to be simple, straightforward and sufficient to represent the semantics under considerations.

7.1 A Case Study

In this section, we introduce a case study to demonstrate the transformations of various elements of state machine model facilitated by Umple to their SMV equivalent as supported by our work.

The *transmission system* (or controller) embodies two other sub-systems (i.e. "ParkAndNeutral" and "Drive") and a "Reverse" mode. "ParkAndNeutral" as its name implies, embodies "Park" and "Neutral" modes. Our transmission sub-system is an extension of the "Car Transmission" state machine in [82], [95] by adding more abstraction to encapsulate "Park" and "Neutral" modes. The transmission system is initialized to "Park". In the "Drive" mode, there are other sub-modes which are: "First", "Second" and "Third". A transition relation is defined on these modes to realize a complete system.

The system was extracted from a collision avoidance system whose abstract representation was presented in [61]. The "*warningRadius*" indicates the distance in meters between the vehicle and an object. At *3 metres* distance, the vehicle is expected to halt; therefore, setting concerned sub-systems to the desired states, including transmission that is set to "Park". A visual representation of the system in Umple is given in Figure 45 and its textual representation is presented in Listing 31.

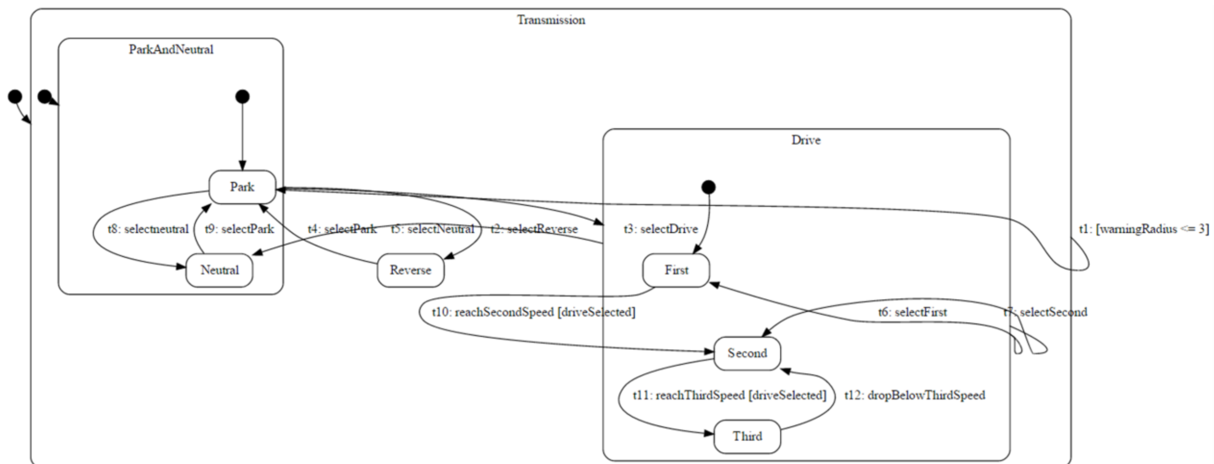


FIGURE 45. VISUAL REPRESENTATION OF CAR TRANSMISSION SUB-SYSTEM

```
1   class Example {
2    Integer warningRadius = 100;
3    Boolean driveSelected;
4     sm {
5      Transmission {
6        [warningRadius <= 3] -> Park; //t1
7        ParkAndNeutral {
8              selectReverse -> Reverse; //t2
9              selectDrive -> Drive;       //t3
10             Park { selectneutral -> Neutral; } //t8
11             Neutral { selectPark -> Park; } //t9
12       }
13       Reverse { selectPark -> Park; } //t4
14       Drive {
15         selectNeutral -> Neutral; //t5
16         selectFirst -> First; //t6
17         selectSecond -> Second; //t7
18         First { reachSecondSpeed[ driveSelected ] -> Second; } //t10
19         Second { reachThirdSpeed[ driveSelected ] -> Third; } //t11
20         Third { dropBelowThirdSpeed -> Second; } //t12
21       } } } }
```

LISTING 31. TEXTUAL REPRESENTATION OF CAR TRANSMISSION SUB-SYSTEM

7.2 Formal Overview of SMV

We present an overview of a subset of the syntax and semantics of SMV modules that is relevant to our work.

A module in SMV, say M, is a 5-tuple $\langle n_M, D_M, S_M, M_M, A_M \rangle$; such that n_M is the name, D_M is a finite set of dependency modules, S_M is a finite set of state variables, M_M is a finite set of macros, A_M is a finite set of constraints on state variables. The name of a module is a direct mapping from the name of the state machine under transformation.

The dependency set of a module M, defines a list of modules within the model that module M communicates with. These communications are carried out by accessing and manipulating enclosed state variables. A dependency set is empty for a module whenever there is no hierarchical relationship between itself and other modules. In our approach, examples of modules in this category are "main" and non-hierarchical root modules. However, the dependency set for modules generated from a hierarchical state machine (i.e. root and non-root) H, is not empty.

Suppose H is a root of a hierarchical system, we define its dependency set as the set of all modules corresponding to descendant state machines of H. For a non-root state machine H, the dependency set contains the immediate ancestor of the considered state and the root state machine. The immediate ancestor is relevant because every non-root module is initialized to 'null' but becomes activated whenever control is transferred to its parent state.

A state variable, v is a pair $\langle n_v, t_v \rangle$ such that n_v is the name and t_v is the type; where t_v is an element of the set: {boolean, integer, real, $\{v_1 \ldots v_n\}$} and $\{v_i | 1 \leq i \leq n \ \wedge n \geq 2\}$ defines a set of enumerated values as a type for the variable being declared. Each variable is constrained in the assign paragraph. The constraint defines the initial value and the next values.

A macro corresponds to reusable entities such as transitions, guards or stability constraints in our approach. In our case, we map corresponding expressions of these entities to a label whose evaluation is done at every *micro-step* during execution.

Actions are associated with transitions and the central focus is to manipulate variables, we model change of values of each variable by associating the corresponding transition. Similarly, the change

of state of module is associated with transitions. In particular, the value of variable "state" changes based on the transition that executes at each micro-step.

A module becomes active in a micro-step its sub-state is activated. This includes the micro-step at which its initial state receives control or activated by default. However, it is worthy to note that the module corresponding to the root state machine is enabled throughout the life cycle of the model under analysis.

In a similar manner to [43], we model dynamic environmental behavior such that events are assigned non-deterministically to variable "event" using the assign paragraph. We introduce a "null" event to indicate a stable state of a system. A system is stable *before* a step commences or *after* a step successfully executes. We realize stability by defining a macro "stable" which is a negation of logical disjunctions of cases when "event" is assigned an element of the event set recognized by the system.

7.3 Transformation of Attributes

In this section, we discuss the transformation of attributes for SMV specifications. A state machine in Umple may have attributes of types other than elements of the sets: {Boolean, Integer, Float, $\{v_1 \ldots v_n\}$}. But in our transformation, we deal with attributes of the specified types (i.e. primitives and enumerations). In particular, we ignore attributes of types 'String', 'Date' and so on because SMV only facilitates the representations of bounded and unbounded integer and real numbers; boolean and enumerations.

A declaration of the following form (Umple - LHS) is transformed to (SMV - RHS) accordingly. In particular, a declaration of primitive type is given as follows:

AttributeType variableName; => variableName : AttributeType;

For example, a declaration of unbounded integer attribute is given as follows:

Integer variableName; => variableName : *integer*;

The declarations of bounded integer and real variables are possible in SMV. In Umple, these variables are constrained. Let us consider a constrained attribute expressed in Umple as given below:

```
Integer age;
[age >= 18] [age <= 120] or [age >= 18 && age <= 120]
```

Its SMV equivalent is defined as range in the following:

```
age : 18..120;
```

An attribute may be declared as an enumeration. In this case, the attribute can be assigned a value from the set of enumerated values. For instance, an attribute declaring days of the week expressed in Umple as follows:

```
daysOfTheWeek { Monday, Tuesday, Wednesday, Thursday … };
```

The SMV equivalent of attribute "daysOfTheWeek" with the enumerated values is given as follows:

```
daysOfTheWeek : { Monday, Tuesday, Wednesday, Thursday … };
```

An attribute is associated with a value at every given step. SMV facilitates the assignments of default and non-default values at any step of execution to an attribute. This is achievable via the "Assign" paragraph. In particular, default values are assigned via the "init" construct and non-default values are assigned via the "case" construct.

For example, consider the following variable declaration in Umple:

```
Integer minAge = 18;
```

Its SMV equivalent is given as follows:

```
ASSIGN
  …
 init(minAge) := 18;
 …
```

To ease the discussion on assignment of values to attributes with respect to steps of execution, we defer this to a subsequent section.

7.4 Transformation of Events

A state machine in Umple may have triggers (events, with or without guards) that controls execution of transitions. In this section, we will focus our discussion on the transformations of events and discuss how we realize the desired semantics.

We declare a variable named 'event' that enumerates all the events that controls the behavior of the SUA. For example, consider the set of events for the system in Figure 45. The following is generated for SMV:

```
event : {
  selectDrive, selectNeutral, selectFirst, reachSecondSpeed,
  autoTransition, selectSecond, reachThirdSpeed, selectReverse, selectPark,
  dropBelowThirdSpeed, null
};
```

The variable "event" is defined as an enumeration of the set of events of the SSUA and "null". We introduce "null" to allow us to model nondeterministic behavior of arrivals of events without a predefined sequence from the environment. In particular, the SSUA is in a stable state whenever "event" is assigned "null" (i.e., after a completed step). Hence, any event (recognized by the SSUA) can be generated from the environment and assigned to variable "event". The initialization of "event" to "null" simulates that the system is initially stable before receiving any form of signal. We achieved this with the following sub-program:

```
ASSIGN
  init(event) := null;
  next(event) := case
    stable : { selectDrive, selectNeutral, selectFirst, reachSecondSpeed,
      autoTransition, selectSecond, reachThirdSpeed, selectReverse, selectPark,
      dropBelowThirdSpeed };
    TRUE : null;
  esac;
```

where:

```
DEFINE
    …
    stable := !( event = selectDrive | event = selectFirst
      | event = autoTransition | event = reachThirdSpeed | event = selectPark
      | event = selectNeutral | event = reachSecondSpeed | event = selectSecond
      | event = selectReverse | event = dropBelowThirdSpeed );
    …
```

7.5 Transformation of Transitions

In our approach, we associate a unique macro to each transition of the state machine. A transition expresses the relationship between states (i.e., source and target), its controlling trigger and an optional guard statement. A trigger is an event. The guard on a transition prevents (when "false") or allows (when "true") the trigger. A transition may also be termed "auto"; which implies that the

transition is automatically taken whenever its source state receives a control, unless prevented by a guard. A generalized representation of transition in Umple is presented in the following:

```
sourceState {
  e ∈ { ⊥, varName } [guardStatement]/{ actionStatement } → destinationState;
  …
}
```

Where:

⊥ denotes *undefined* and varName is any valid variable name.

Readers should note that in the generalized representation, whenever event is *undefined* it is termed "auto transition" in Umple. An equivalent of this general representation in SMV is given as follows:

t_i:= event = eventName & [targetMachine.]state = targetState [& g_j];
g_j:= guardStatement;

We denote […] as optionality. In particular, the target state machine is optional whenever it is the root state machine because all macros are defined in the root. Hence, resolving the scope is irrelevant but resolving the scope is necessary for non-root state machines. Similarly, for unguarded transitions the guard is unnecessary. Index i ranges between 1 and the total number of transitions of the SSUA; while index j ranges between 1 and the total number of unique guards of the SSUA.

We consider the following examples for the purpose of illustrating transformations of transitions from Umple to SMV.

Let us consider the auto transition with label t_1 in Figure 45. In Umple code, this is represented as follows:

```
Transmission {
  [warningRadius <= 3] → Park;
  …
}
```

This undergoes an internal transformation because it is an "auto transition". In particular, the controlling event is not explicitly specified. The result of the transformation is given as follows:

```
Transmission {
```

```
  _autoTransition_[warningRadius <= 3] → Park;
  …
}
```

However, to generate its equivalent in SMV, we compute unique identities (displayed visually in Figure 45) for the macros defining the transition and the guard expression on it. The following are the expressions:

t_1:= event = _autoTransition_ & state = Transmission & g_1;
g_1:= warningRadius <= 3;

For an unguarded transition, let us consider transition with identity t_9 in Figure 45. The Umple text corresponding to this transition is presented on line 11 of Listing 31. We generate the following for the transition:

t_9:= event = selectPark & $ParkAndNeutral.state = Neutral;

Another type of transition we treated differently is a high-level transition of concurrent states. In particular, for implementation purposes, concurrent sub-state machines must be wrapped in a wrapper state to allow the specification of high-level transitions. In Figure 20, we presented an SSUA with wrapper states (in dashed red).

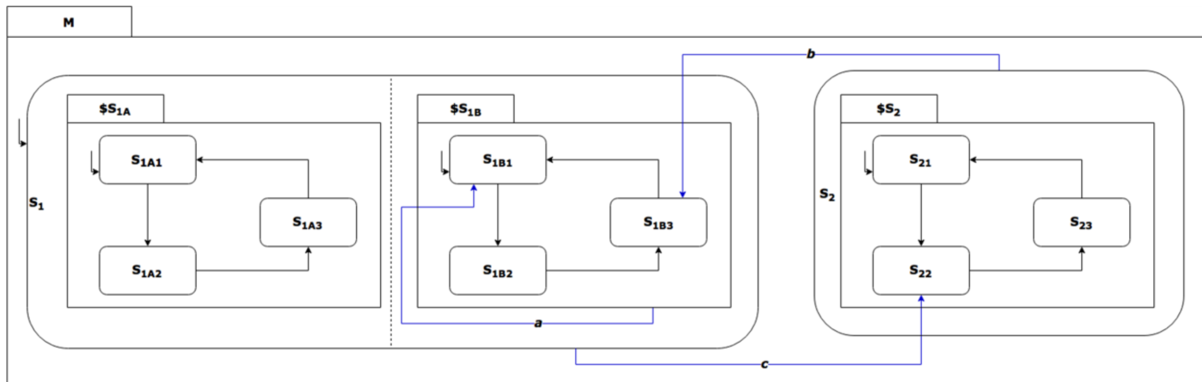


FIGURE 46. TRANSFORMED VERSION OF FIGURE 20

We note that implementation of a wrapper state contributes two states (i.e., *wrapper* and *null*) to the global state space and a sub-state machine. This situation will be exacerbated in any SSUA with a higher degree of concurrency. For the purpose of formal analysis, we consider the notion of

wrapper states irrelevant. Therefore, we removed the wrapper states and their containing sub-state machines without jeopardizing semantics to minimize the state-space explosion problem during analysis.

To manage this complexity, we transformed the SSUA such that the wrapper states and sub-state machines are removed. The high-level transitions of the wrapper states are systematically placed on the corresponding sub-state machines of the regions. In particular, transitions originating from the wrapper states now originate technically from the machine corresponding to the region and those into the wrapper states become in-transitions of the start state of the region under consideration. Hence, the resulting SSUA after our transformation to the SSUA in Figure 20 becomes Figure 46.

Recall that for a state machine, a "null" state is added to the set of states for the purpose of disabling the sub-state machine whenever control is yet to transferred to it or control is transferred out of its states. Particularly, the state variable of $S1B is assigned "null" whenever it is not in any of its states. Therefore, the set of states for sub-state machine $S1B is as follows:

```
state : { S1B1, S1B2, S1B3, null };
```

We transform the expression specifying the source state for the transition to an expression that checks if the state of the sub-state machine is not "null". For example, consider transition with label "a" in Figure 46.

t_k:= event = a & $S1B.state != null;

7.6 Transformation of States

In this section, we discuss the transformation of states from Umple to SMV. We note that states in Umple are simple or composite. To represent states in SMV, we adopt a *case statement* of a *case structure* within an *assign block* such that the logical disjunction of its enabling transitions becomes the LHS and the state itself becomes the RHS of the case statement.

In particular, a case structure is of the following form such that the LHS evaluates to a Boolean value (i.e., true or false):

```
LHS : RHS;
```

Recall that a state is enabled by its in-transitions and embedded transitions (see equation 10). But the set of embedded transitions for a simple state is empty since $\beta(x) = 0$. Otherwise (i.e., when $\beta(x) > 0$), the set may be non-empty.

Let us consider a simple state "Park" in Figure 45. Its enabling transition set is given as follows:

$$\varphi(\text{Park}) = \Delta(\text{Park}) = \{t_1, t_4, t_9\}$$

Therefore, the case statement corresponding to "Park" is given as follows:

t_1 | t_4 | t_9 : Park;

To illustrate the formation of case statement for a composite state, we consider "Drive" as an example. The in-transitions of "Drive" is $\{t_3\}$; while its set of embedded transitions is $\{t_6, t_7, t_{11}, t_{12}\}$.

Therefore, the case statement corresponding to "Drive" is given as follows:

t_3 | t_6 | t_7 | t_{11} | t_{12} : Drive;

7.7 Transformation of State Machines

The transformation of state machines unifies the building blocks presented in the prior sections of this chapter. We discuss our method to generate simple and hierarchical state machines as facilitated by this thesis.

7.7.1 Simple State Machines

We focus our discussion on the transformation of simple state machines in this section. By "simple state machine", we mean a SSUA such that:

$$\forall_{s \in S_{SSUA}} \cdot \beta(s) = 0;$$

Where: s is a *top-level state* of the machine.

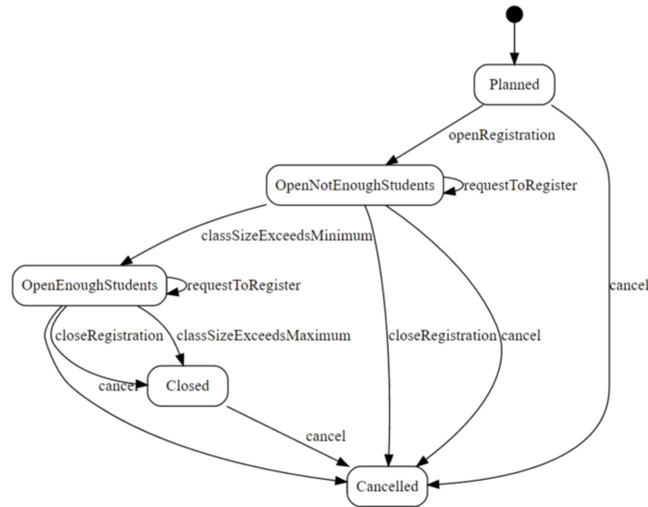


FIGURE 47. A FLAT VERSION OF COURSE SECTION STATE MACHINE

To demonstrate our approach to generating SMV models corresponding to simple state machines, we present an example of a state machine for course registration [56]. It defines the behavior of a system that allows students to enroll for courses. Although, the system can be modeled hierarchically, we consider its flat version, which is a simple state machine suitable for this illustration purposes.

```
1  MODULE CourseSectionStatus
2  VAR
3    state : { Planned, OpenNotEnoughStudents, OpenEnoughStudents, Cancelled,
4       Closed };
5    event : { cancel, classSizeExceedsMinimum, openRegistration,
6       requestToRegister, classSizeExceedsMaximum, closeRegistration, null };
7  DEFINE
8    stable :=  !( event = cancel | event = openRegistration |
9      event = classSizeExceedsMaximum | event = classSizeExceedsMinimum
10     | event = requestToRegister | event = closeRegistration );
11   t1 := event = cancel & state = Planned;
12   t2 := event = openRegistration & state = Planned;
13   t3 := event = requestToRegister & state = OpenNotEnoughStudents;
14   t4 := event = closeRegistration & state = OpenNotEnoughStudents;
15   t5 := event = cancel & state = OpenNotEnoughStudents;
16   t6 := event = classSizeExceedsMinimum & state = OpenNotEnoughStudents;
17   t7 := event = cancel & state = OpenEnoughStudents;
18   t8 := event = requestToRegister & state = OpenEnoughStudents;
19   t9 := event = closeRegistration & state = OpenEnoughStudents;
20   t10 := event = classSizeExceedsMaximum & state = OpenEnoughStudents;
21   t11 := event = cancel & state = Closed;
22 ASSIGN
23   init( state ) := Planned;
24   next( state ) := case
25    t2 | t3 : OpenNotEnoughStudents;
26    t6 | t8 : OpenEnoughStudents;
27    t4 | t7 | t1 | t5 | t11 : Cancelled;
28    t9 | t10 : Closed;
29    TRUE : state;
30   esac;
31 ASSIGN
32   init( event ) := null;
33   next( event ) := case
34    stable : { cancel, classSizeExceedsMinimum, openRegistration,
35     requestToRegister, classSizeExceedsMaximum, closeRegistration };
36    TRUE : null;
37   esac;
38
39 MODULE main
40 VAR
   courseSectionStatus : CourseSectionStatus;
```

LISTING 32. THE SMV EQUIVALENT OF COURSE REGISTRATION SYSTEM

Figure 47 is the visual representation of the course registration system. In accordance with our transformation, the equivalent transition system of the SSUA expressed in SMV is given in Listing 32.

We generate two (2) modules for the SSUA. One module (lines 1-37) corresponds to the root state machine of the SSUA; while the module (on lines 39-41) is the main module.

On lines 3, 4, we enumerate the states of the SSUA as values of variable "state" and lines 5, 6, we enumerate the states of the SSUA as values of variable "event". On lines 11-21, we define a set of macros associated with transitions of the SSUA. We define the stability macro on lines 8-10.

We model the assignment of states of the SSUA to variable "state" (see lines 22-30). On line 23, the initial state of the SSUA is set to "Planned". We model the assignment of events of the SSUA with an assign block on lines 31-37. The combination of this assign block and stability control models the non-deterministic behavior of the environment.

7.7.2 Hierarchical State Machines

In this section, we focus on the transformation of hierarchical state machines. Recall that by "hierarchical state machine", we mean an SSUA such that:

$$\exists_{s \in S_{SSUA}} \cdot \beta(s) \geq 1;$$

Where: s is a *top-level state* of the machine.

```
1   MODULE Sm (_Transmission,_ParkAndNeutral,_Drive)
2    VAR
3     state : { Transmission , null };
4     event : { selectDrive, selectNeutral, selectFirst, reachSecondSpeed,
5      _autotransition_, selectSecond, reachThirdSpeed, selectReverse,
6      selectPark , dropBelowThirdSpeed, null };
7     warningRadius : integer;
8     driveSelected : boolean;
9    DEFINE
10    stable := !( event = selectNeutral | event = reachSecondSpeed |
11     event = selectSecond | event = selectReverse | event = dropBelowThirdSpeed
12     | event = selectDrive | event = selectFirst | event = _autotransition_ |
13     event = reachThirdSpeed | event = selectPark );
14    t1 := event = _autotransition_ & state = Transmission & g1;
15    t2 := event = selectReverse & _Transmission.state = ParkAndNeutral;
16    t3 := event = selectDrive & _Transmission.state = ParkAndNeutral;
17    t4 := event = selectPark & _Transmission.state = Reverse;
18    t5 := event = selectNeutral & _Transmission.state = Drive;
19    t6 := event = selectFirst & _Transmission.state = Drive;
20    t7 := event = selectSecond & _Transmission.state = Drive;
21    t8 := event = selectNeutral & _ParkAndNeutral.state = Park;
22    t9 := event = selectPark & _ParkAndNeutral.state = Neutral;
23    t10 := event = reachSecondSpeed & _Drive.state = First & g2;
24    t11 := event = reachThirdSpeed & _Drive.state = Second & g2;
25    t12 := event = dropBelowThirdSpeed & _Drive.state = Third;
26    g1 := warningRadius <= 3;
27    g2 := driveSelected;
28   ASSIGN
29    init( state ) := Transmission;
30    next( state ) := case
31     t1 | t4 | t5 | t3 | t7 | t12 | t9 | t8 | t2 | t6 | t10 | t11: Transmission;
32     TRUE : state;
33    esac;
34   ASSIGN
35    init( event ) := null;
36    next( event ) := case
37     stable : {selectDrive, selectNeutral, selectFirst, reachSecondSpeed,
38      _autotransition_, selectSecond, reachThirdSpeed, selectReverse,
39      selectPark, dropBelowThirdSpeed };
40     TRUE : null;
41    esac;
42   ASSIGN
43    init( warningRadius ) := 100;
44   ASSIGN
45    init( driveSelected ) := FALSE;
```

LISTING 33. THE ROOT STATE MACHINE OF TRANSMISSION SUB-SYSTEM

The example presented in Figure 45 is a hierarchical state machine. It is structurally composed of a root and three (3) sub-state machines. These include: “Sm”, “$Transmission”, “$ParkAndNeutral”, and “$Drive”; where “Sm” is the root state machine. Therefore, for each state machine, we define a corresponding module to model the relations among its states.

In Listing 33, we present the root state machine of the SSUA (i.e., the transmission sub-system). This state machine encapsulates reusable entities for other sub-state machines of the SSUA. Particularly, we intend to separate concerns and minimize coupling. For example, transition macros (see lines 14-25), variables (or attributes – see lines 7, 8) and events (see lines 4-6) are reusable entities across the SSUA; therefore, we define them within the root module.

On line 1 (i.e., Listing 33) module “Sm” depends on “$Transmission”, “$Drive”, and “$ParkAndNeutral”. These dependencies refer to the list of modules within the SSUA that “Sm” communicates with. The communications are carried out by accessing and manipulating enclosed state variables. For a root state machine (e.g., “Sm”), the dependency set is composed from all its descendants sub-state machines. But for non-root state machines, the composition of dependency set differs. Let us consider a non-root state machine - “$Transmission”.

The “$Transmission” sub-state machine is an immediate descendant of the root state machine. Its corresponding SMV module is presented in Listing 34. For immediate descendants of the root state machine, the dependency set is a singleton with the root machine as its element (see line 1). The root state machine is necessary for the following reasons: (a) references to reusable entities, such as transitions are made within the module; (b) to determine the state at which control is transferred to its parent state (i.e., “Transmission” - a child state of the root) for the purpose of activation-by-default (see line 11). This is achieved by accessing “state” variable of the root module via the “dot” operator.

```
1  MODULE $Transmission ( _sm )
2   VAR
3    state : { ParkAndNeutral, Reverse, Drive, null };
4   ASSIGN
5    init( state ) := null;
6    next( state ) := case
7     _sm.t3 : null;
8     _sm.t9 | _sm.t8 | _sm.t1 | _sm.t4 | _sm.t5 : ParkAndNeutral;
9     _sm.t2 : Reverse;
10    _sm.t3 | _sm.t7 | _sm.t12 | _sm.t6 | _sm.t10 | _sm.t11 : Drive;
11    _sm.state = Transmission & state = null : ParkAndNeutral;
12     TRUE : state;
13   esac;
```

LISTING 34. TRANSMISSION MODULE

```
1  MODULE $ParkAndNeutral ( _sm, _smTransmission )
2   VAR
3    state : { Park, Neutral, null };
4   ASSIGN
5    init( state ) := null;
6    next( state ) := case
7     _sm.t3 | _sm.t7 | _sm.t11 | _sm.t2 | _sm.t6 | _sm.t10 | _sm.t12 : null;
8     _sm.t9 | _sm.t1 | _sm.t4 : Park;
9     _sm.t8 | _sm.t5 : Neutral;
10    _smTransmission.state = ParkAndNeutral & state = null : Park;
11     TRUE : state;
12   esac;
```

LISTING 35. PARKANDNEUTRAL MODULE

Another category of sub-state machines of interest is non-immediate sub-state machines of the root. For example, let us consider "$ParkAndNeutral" (i.e., Listing 35). The dependency set of a non-immediate sub-state machine of the root contains the root state machine and its immediate ancestor (see line 1). Transitions are accessible within the corresponding module of the state machine via the root state machine (see lines 7-9) and transfer of control to its parent state (i.e., for the purpose of activation-by-default) are done through the immediate ancestor (see line 10).

The state variable of "$ParkAndNeutral" is assigned "null" for the logical disjunction of elements of its disabling transitions; while its disabling transition set is given as follows:

$$D(\$ParkAndNeutral) = \{ t2, t3, t6, t7, t10, t11, t12 \}$$

In particular, whenever a transition in this set executes, the module is disabled by assigning "null" to its state variable.

```
1   MODULE ExampleSm_Machine
2    VAR
3     sm : Sm( transmission, parkAndNeutral, drive );
4     transmission : $Transmission( sm );
5     parkAndNeutral : $ParkAndNeutral( sm, transmission );
6     drive : $Drive( sm, transmission );
7
8   MODULE main
9    VAR
10    exampleSm_Machine : ExampleSm_Machine;
```

LISTING 36. WRAPPER AND MAIN MODULES FOR INSTANTIATION PURPOSES

To facilitate modularity and analysis of multiple SSUAs within an Umple file, we instantiated all the modules corresponding to the root and sub-state machines of an SSUA in a wrapper module. In particular, every SSUA has a corresponding wrapper module and each wrapper module is instantiated within a main module. For the SSUA under consideration, its wrapper and main modules are presented in Listing 36.

The complete SMV program (or model) of the SSUA presented here (i.e., Listing 31 or Figure 45) can be auto-generated from our translator.

7.8 Discovering Nondeterminism

In Section 6.1, we introduce our approach to discovering nondeterminism in the SSUA whenever simultaneous execution of any pair of potentially conflicting transitions results in nondeterminism.

We have deployed our match-making algorithm (i.e. Listing 27) on the example presented in Figure 45 (i.e., Listing 31) and found no potential threat of non-determinism in the model. A modeling example with some threats (including real cases) of nondeterminism discovered by our

algorithm can be obtained from [96]. We defer the discussion of the system and discovered threats of nondeterminism to Chapter 8 (i.e., Verification and Validation).

7.9 Reachability Analysis of the SSUA

We consider reachability analysis of SSUA as an integral part of our work because the presence of an unreachable state in an SSUA is regarded as a design flaw. Hence, it is important to discover such flaws and notify the analyst accordingly.

In this section, our goal is to present the resulting SMV specification generated based on the constraint given in expression (4) for the SSUA presented in Figure 45. We generate a CTL specification for every state of the SSUA. The following is a generalized form of the generated specification:

```
X  EF( parentMachineReference.state = stateName )
    Where:  X ∈ { CTLSPEC, SPEC }
```

```
1  CTLSPEC  EF( exampleSm_Machine.sm.state = Transmission )
2  CTLSPEC  EF( exampleSm_Machine.transmission.state = ParkAndNeutral )
3  CTLSPEC  EF( exampleSm_Machine.transmission.state = Reverse )
4  CTLSPEC  EF( exampleSm_Machine.transmission.state = Drive )
5  CTLSPEC  EF( exampleSm_Machine.parkAndNeutral.state = Park )
6  CTLSPEC  EF( exampleSm_Machine.parkAndNeutral.state = Neutral )
7  CTLSPEC  EF( exampleSm_Machine.drive.state = First )
8  CTLSPEC  EF( exampleSm_Machine.drive.state = Second )
9  CTLSPEC  EF( exampleSm_Machine.drive.state = Third )
```

LISTING 37. REACHABILITY SPECIFICATION FOR TRANSMISSION SYSTEM

Listing 37 is the code block corresponding to these constraints generated for the SSUA (i.e., Figure 45). This code block is placed in the main module of the SSUA.

7.10 Summary of State Machine Transformation

In this chapter, we presented our approach to encoding various Umple constructs relevant to the development of state machine systems (i.e., SSUA). In particular, for each construct in Umple, we present a corresponding program in SMV.

First, we overviewed formalization of SMV and discussed the transformations of simple modeling elements in Umple such as: attributes, transitions, states and events to their SMV equivalents. We dealt with state machines (i.e., simple and hierarchical) as modular entities and discussed their transformations.

Furthermore, we introduced the discovery of nondeterminism as a modeling flaw but deferred the discussion of code corresponding to invariant specifications of the pairs of potentially conflicting transitions to the next chapter. We also presented specifications generated for the determination of reachability of states of the SSUA.

8 Verification and Validation

This chapter focuses on various approaches we adopted for the purpose of verification and validation of our work. To verify our work, we perform simulations and rigorous test-driven development inherited from Umple.

With simulation, our goal is to determine necessary constraints required to model the behavior or structure under consideration. In particular, we are interested in systems that are neither under-constrained nor over-constrained.

We perform test-driven development (TDD), ensuring that our generator is free from bugs covered by a significant set of test cases that we developed and which cover the core Umple association, attribute and state machine semantics.

As a means of validating our work, we analyze some case studies to demonstrate applicability of our approach to real-world problems and illustrate its relevance in industry. The validation conducted in this research can be broadly categorized into that of dynamic and static properties. We validate dynamic properties ranging from discovering non-determinism, assessing the impact of abstraction via and-cross transitions, to asserting equivalence of similar systems. We also validate a system called ESeat, whose goal is to generate examination seating arrangements for students.

8.1 Verification

To ensure correctness of our work, we adopt simulation and test-driven development (TDD). While simulation helps prevent problems of under- and over-specification, TDD helps us develop code generators that produce working systems (i.e., syntax-error free).

8.1.1 Simulation

We simulate model examples and their requirements to verify that our approach finds violations of the requirements, and fails to find violations where none exist.

To avoid problems of *over-specification* (finding problems where none exist) *and under-specification* (failing to find problems), we have generated relevant constraints for each object-oriented design pattern considered in this thesis. More specifically, by “under-specification”, we mean lack of sufficient constraints; while by “over-specification”, we mean no instance of the model is generated after simulation of the model. To avoid this, we infer the necessary set of constraints for each pattern via simulation by leveraging the capabilities of Alloy and its analyzer for incremental specification and verification of software abstraction.

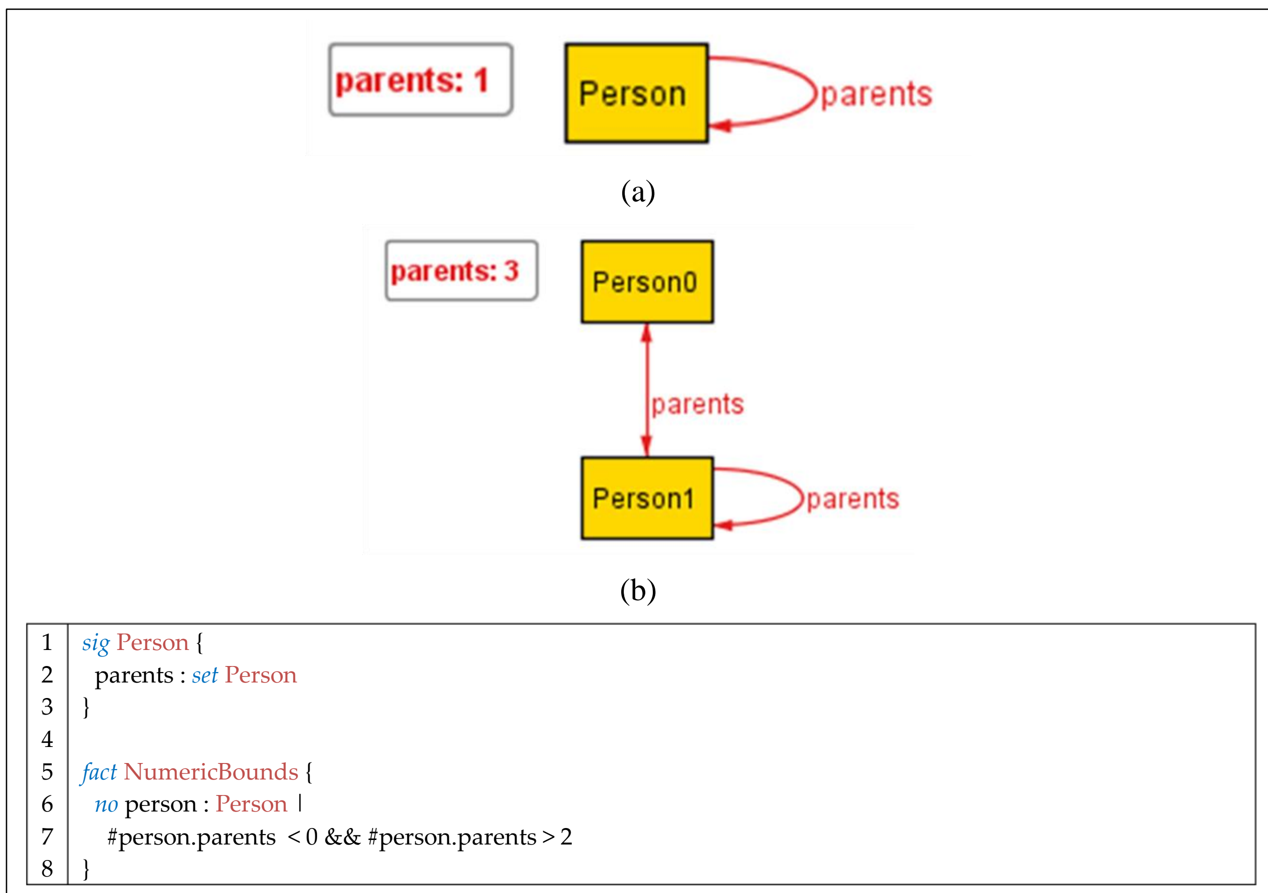


FIGURE 48. ILLUSTRATING UNDER-SPECIFICATION IN PARENT-CHILD ASSOCIATION

The use of simulation helped us prevent the problem of under-specification in the development of the set of constraints for a parent-child association. For example, consider the Alloy model presented in Figure 48 without the no-cycle constraint. In the example, we simulate the model using Alloy analyzer and it results are presented in Figure 48(a & b). In Figure 48(a), simulation

reveals that the model allows a person to be the parent of him/herself; while Figure 48(b) shows that a person can be a parent of his/her parent. Hence, we introduced the no-cycle constraint and result is presented in Figure 49.

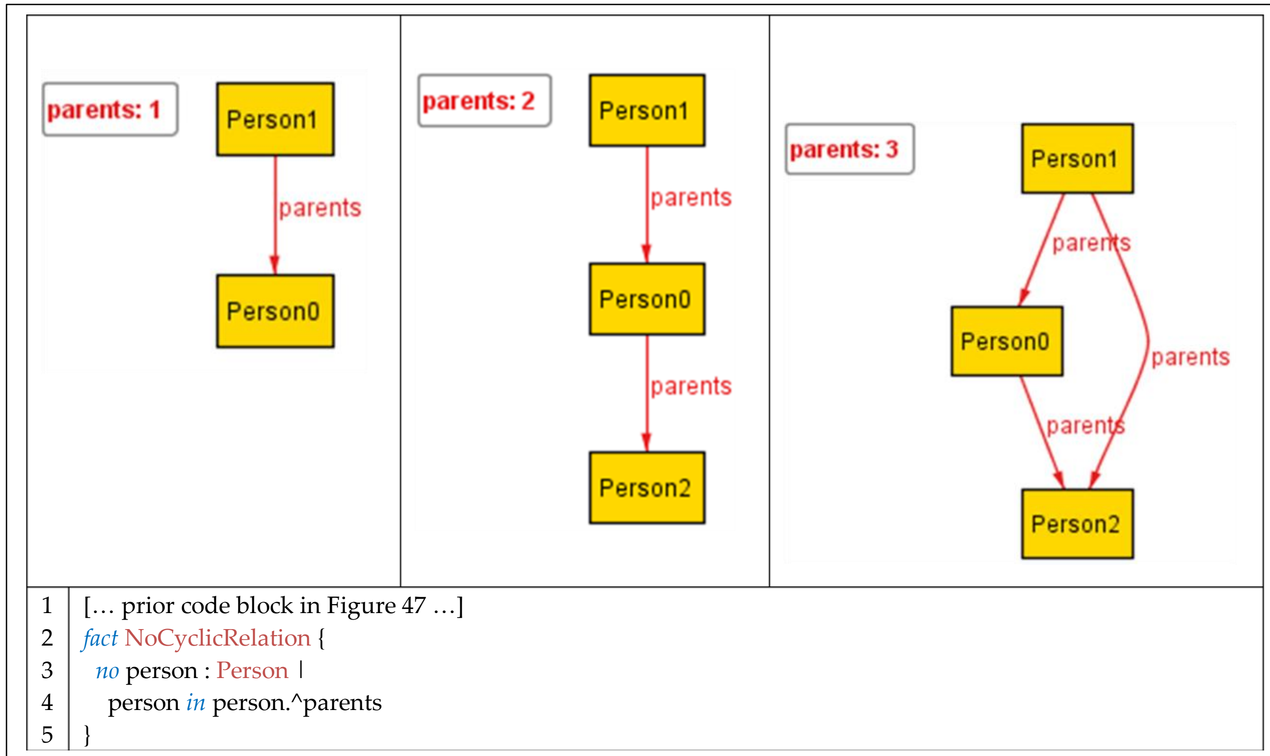


FIGURE 49. SIMULATION RESULT AFTER INTRODUCING NO-CYCLE CONSTRAINT

We applied a similar technique in the verification of SMV code with the nuXmv analysis engine. A general description of our steps to realizing a consistent and desired formal specifications is given in Figure 50. We began by specifying the pattern under consideration in the target language (i.e., SMV or Alloy). Then we specify known (or predetermined) constraints. The fusion of the pattern and constraints, expressed in the target language becomes an input to the simulator. For analysis of static properties (i.e., classes and associations), the Alloy analyzer is the target simulator. However, for the analysis of dynamic properties, the nuXmv analysis engine is the target simulator.

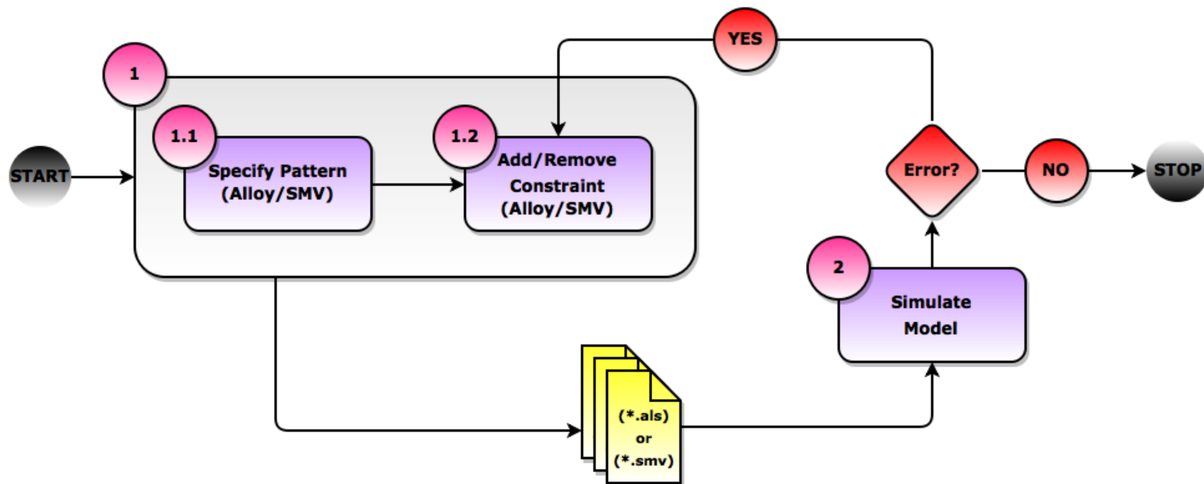


FIGURE 50. SIMULATION ARCHITECTURE

The resulting model (i.e., **.smv* or **.als*) is analyzed by the simulator and flaws (e.g., under- or over-specification) are discovered as discussed above during simulation. Whenever such flaws are discovered, the constraint(s) is/are modified and the model is re-simulated. This process continues iteratively until we are satisfied with the set of instances resulting from the exercise. The resulting model or specification (i.e., pattern and constraints) becomes the template to be implemented for code generators.

8.1.2 Test-Driven Development

To ensure our code generator behaves as expected, we adopted the rigorous test-driven development (i.e., TDD) principles facilitated by Umple to maintain its quality at all times.

Our tests compare *expected code* (i.e., the test case) with the *actual code* generated by our code generator. We begin by defining a simple test case for each Umple construct under consideration (CUC). Then, we modify the code generator to generate expected formal method code for the CUC. Every time Umple is built, the test engine compares the expected code with the actual code and flags any mismatch. We used this to guide our implementation for the generation of formal specifications of class diagrams and state machine models.

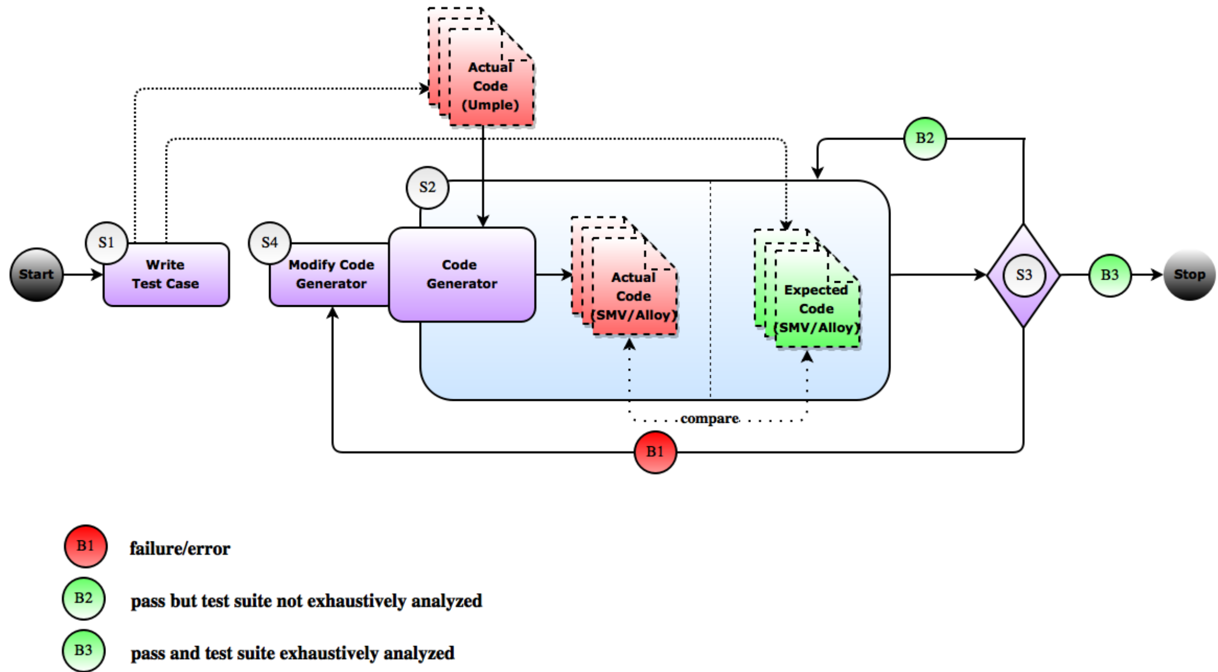


FIGURE 51. THE TEST-DRIVEN DEVELOPMENT (TDD) ARCHITECTURE

In Figure 51, we present our infrastructure for testing code generators (i.e., Umple2Alloy and Umple2SMV) to guarantee 100% pass rate of the test suite at all times Umple compiles itself. We have divided the stages of activities into seven (**7**). The following discusses each stage of activities and branches as presented in Figure 51.

S1: For each target language, we developed a test case for each construct. This includes an actual file with an Umple program and an actual file of expected constructs in the target language. The test case describes what is expected to be generated whenever our generator is invoked. To guarantee Umple2SMV generator behaves as expected we have developed 38 unique test cases (see [97]. Similarly, for Umple2Alloy translator, we have developed 20 unique test cases too (see [98]). The infrastructure allows addition of test cases to handle cases yet to be covered by our implementation.

S2: At this stage, the code generator is invoked to generate the actual code. Then the test oracle (i.e., [99] or [100]) is invoked to compare the actual code

with the expected code. The test result becomes the output to be analyzed at **S3**.

S3: At this stage the test result, output by the test oracle is analyzed to determine a failure or error or success. At the first time a test case is created and analyzed, the result is expected to be failure since the code generator is yet to be modified. At this point, **B1** is taken so that the code generator is modified (i.e., **S4**). Similarly, whenever the result is "error" (i.e., code generator fails to compile), **B1** is also taken to allow modifications to the generator. **B2** is taken whenever the test case under consideration passes but there is at least a test case in the test suite that is yet to be analyzed. However, whenever all the tests within the test suite passes then **B3** is taken to halt the exercise.

S4: At this stage, we modify the code generators to accommodate changes required to implement the feature whose test case was designed at **S1**. In particular, the necessary program is added to the generator so that it reflects the required changes to guarantee that a failing test passes.

8.2 Validation

To validate our approach, we have developed case studies in various domains with different objectives. The case studies consist of models and their respective correctness requirements (where applicable). We apply our system to generate the formal specifications of each system (i.e., SMV or Alloy). Finally, we will verify the correctness of each system against its requirements by invoking our back-end analysis engines and document analysis results where necessary.

8.2.1 Case Study 1 – Discovering Nondeterminism

Our first case study to validate our work is the home heating system of Lu et al. [66]. We use this to show how we can discover non-determinism. Our choice of system is based on the following criteria: it must be simple-to-understand and be characterized with finite and infinite domain variables, false positives and cases of non-determinism.

8.2.1.1 The Home Heating System

The system models the behaviour of house and furnace sub-systems executing concurrently. The house combines room and controller subsystems. There are three buttons: on, off, and reset. The controller coordinates all sub-systems via these buttons. The on and off buttons switch the entire system on and off. A system may be faulty or functional. A faulty system puts the controller in error mode. In this mode, whenever the user reset button is pressed the controller is turned off and the furnace is reset simultaneously.

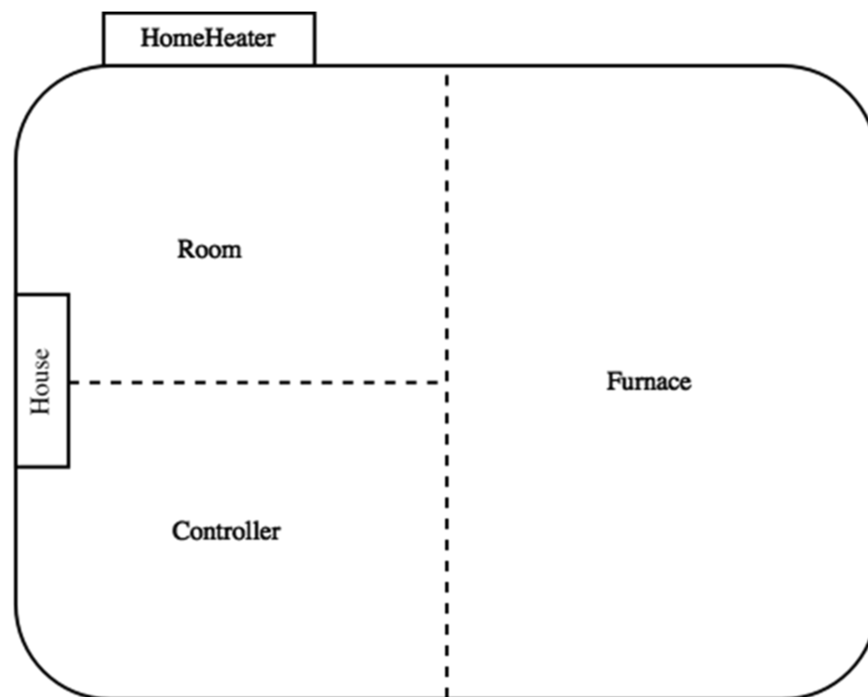


FIGURE 52. HIGH-LEVEL REPRESENTATION OF HOME HEATING SYSTEM

We model the hierarchical state machine of this system with Umple and translate it to SMV using our translator. The Umple version can be obtained from [96]; while its high-level representation is presented in Figure 52, the corresponding sub-systems are presented in Figure 53 - Figure 55. Transition and guard labels can be displayed by checking relevant options. We represent internally-generated events using Boolean variables, such that when the event is generated, the variable is assigned 'true'. Whenever the event is to be consumed by the system, we include it in

the guard of the concerned transition. The complete SMV code we used for this analysis is presented in Appendix 1. Readers should note that for auto transitions (in the visual representations below), we do not explicitly label the events. Similarly, transition labels are in italics (e.g., "*t1*"). A do-activity enforcing a delay of 1s is enclosed in every state of the system; while guards are evaluated in the order of indices on guard labels whenever more than a transition originates from a state. For example, "G2" is evaluated before "G3" is evaluated since their associated transitions both originate from "WaitForHeat".

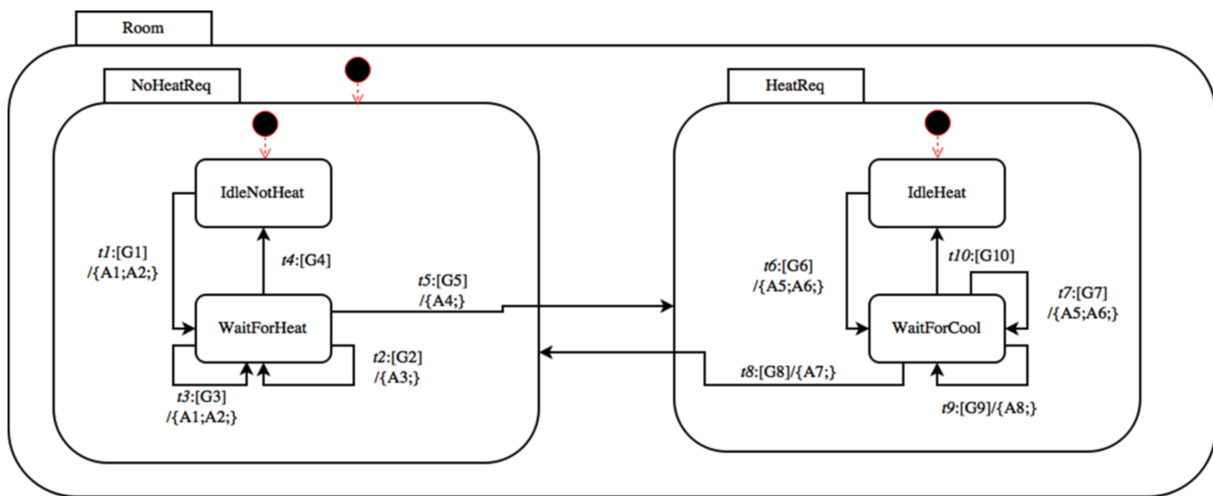


FIGURE 53. THE ROOM SUB-SYSTEM

Readers should note that the following symbols denote the corresponding guard statements (expressed in SMV) for Figure 53 - Figure 55:

G1 :- (setTemp - actualTemp) > 2
G2 :- waitedForWarm < WARMUPTIMER
G3 :- (valvePos != 2) & (waitedForWarm = WARMUPTIMER)
G4 :- (setTemp - actualTemp) <= 2
G5 :- (waitedForWarm = WARMUPTIMER) & (valvePos = 2) & ((setTemp - actualTemp) > 2)
G6 :- (actualTemp - setTemp) > 2
G7 :- (valvePos != 0) & (COOLDOWNTIMER = waitedForCool)
G8 :- (valvePos = 0) & (COOLDOWNTIMER = waitedForCool) & (actualTemp - setTemp) > 2
G9 :- waitedForCool < COOLDOWNTIMER
G10 :- (actualTemp - setTemp) <= 2
G11 :- requestHeat = TRUE
G12 :- requestHeat = FALSE

G13 :- furnaceRunning = TRUE
G14 :- furnaceReset = TRUE
G15 :- activate = TRUE
G16 :- deactivate = TRUE
G17 :- furnaceStartUpTime < FURNACETIMER
G18 :- furnaceStartUpTime = FURNACETIMER

where:

FURNACETIMER := 5
WARMUPTIMER := 5
COOLDOWNTIMER := 5

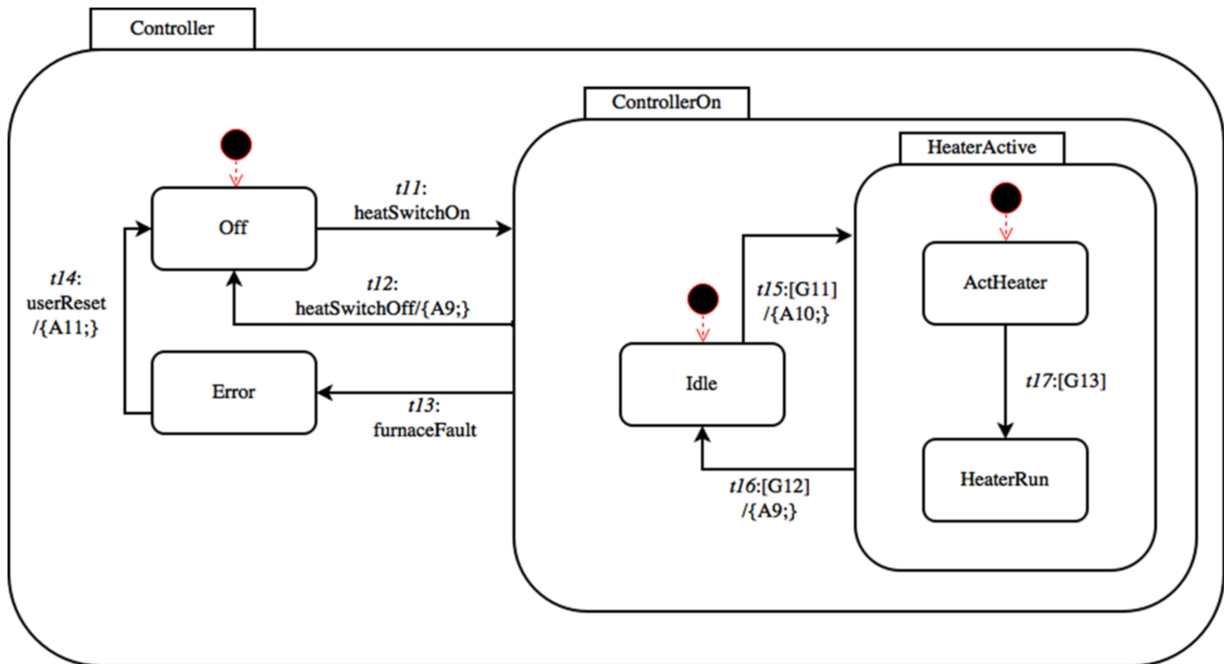


FIGURE 54. THE CONTROLLER SUB-SYSTEM

In the same vein, readers should also note that actions are denoted with the following labels:

A1 :- valvePos++
A2 :- waitedForWarm = 0
A3 :- waitedForWarm++
A4 :- requestHeat = TRUE
A5 :- valvePos--
A6 :- waitedForCool = 0
A7 :- requestHeat = FALSE
A8 :- waitedForCool++
A9 :- deactivate = TRUE
A10 :- activate = TRUE

A11 :- furnaceReset = TRUE
A12 :- furnaceStartUpTime = 0
A13 :- furnaceStartUpTime++
A14 :- furnaceRunning = TRUE

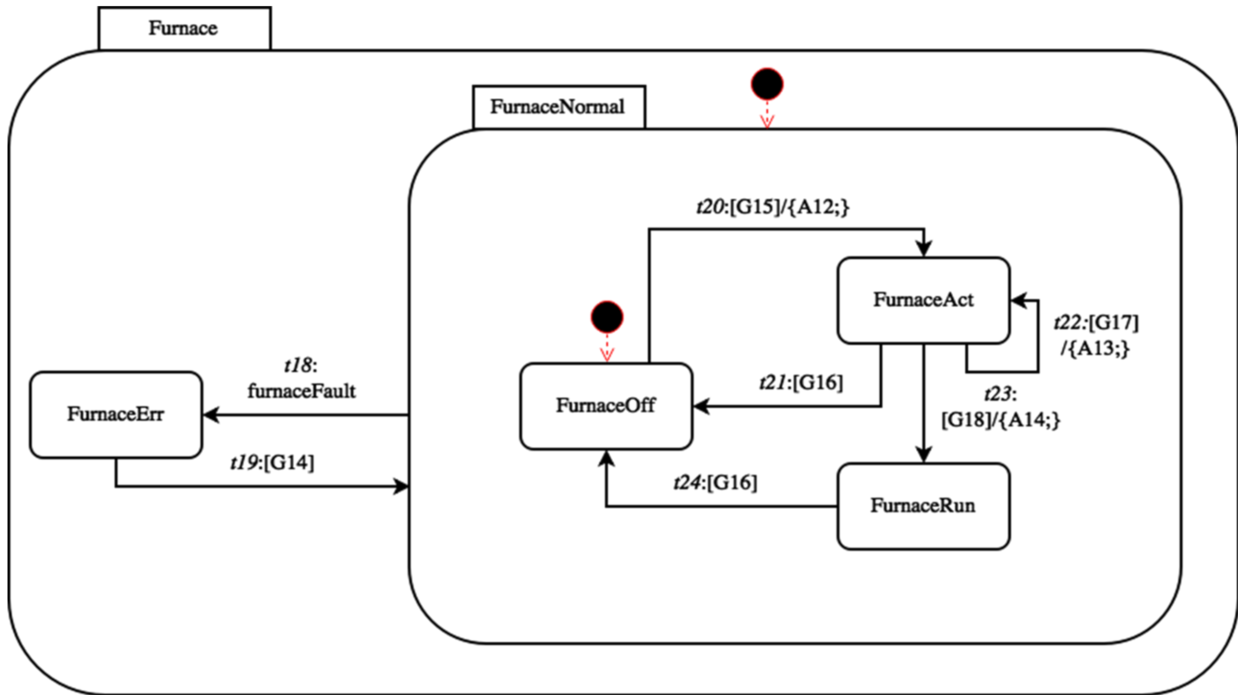


FIGURE 55. THE FURNACE SUB-SYSTEM

8.2.1.2 Analysis and Discussion of Results

We delegate our algorithm to compute the set of potentially conflicting transitions for the heating system. The universal set of transitions of the system are expressible as $\{t_j \mid 1 \leq j \leq 24\}$. For this analysis, we adopted the IC3 method implemented by the nuXmv model checker. Table 13 presents results (by model checking) of the potentially conflicting set of comparable transitions. These are real candidates of non-determinism found in the model by our algorithm. The error rate of the algorithm is 0% and the model checker dynamically discovers actual candidates of non-determinism.

Our approach to dealing with the problem is proof-by-contradiction. In particular, we specify an invariant constraint saying that the simultaneous execution of each pair of transitions is allowed within the SSUA. A violation of the constraint is non-determinism (i.e., a situation whereby the

pair cannot be executed simultaneously). Therefore, the “T” cases imply that the pair can execute simultaneously without resulting in non-determinism (i.e., *false positives*); while the “F” cases imply that a counterexample is generated whenever pair of transitions executes simultaneously (i.e., *true positives*).

TABLE 13. ANALYSIS RESULT FOR NON-DETERMINISM

PAIRS	RESULT	TYPE	Non-Deterministic
(t_2, t_4)	F	C1.1	YES
(t_2, t_5)	T	C1.1	NO
(t_3, t_4)	T	C1.1	NO
(t_3, t_5)	T	C1.1	NO
(t_4, t_5)	T	C1.1	NO
(t_7, t_8)	T	C1.1	NO
(t_7, t_{10})	T	C1.1	NO
(t_9, t_{10})	T	C1.1	NO
(t_8, t_9)	T	C1.1	NO
(t_8, t_{10})	T	C1.1	NO
(t_{16}, t_{17})	T	C1.2	NO
(t_{21}, t_{22})	T	C1.1	NO
(t_{21}, t_{23})	T	C1.1	NO
(t_{22}, t_{23})	T	C1.1	NO

Key: T - True, F-False, C1.1 - Physically Same-Source, C1.2 - High Level Vs. Embedded Transition

The following configuration led to the state of non-determinism (i.e., a situation whereby transitions t_2 and t_4 execute simultaneously) by enabling guards “G2” and “G4” in the SSUA. In Figure 53 transition t2 originates from “WaitForHeat” and targets “WaitForHeat” but is controlled by guard “G2”; while t4 originates from “WaitForHeat” and targets “IdleNotHeat” but is controlled by guard “G4”. The following defines the steps resulting in a nondeterministic situation.

```
HouseRoomNoHeatReq.state = waitForHeat
setTemp = 16
actualTemp = -45
warmUpTimer = 5
waitedForWarm = 0
g2:[waitedForWarm < warmUpTimer]
```

g4:[(setTemp – actualTemp ≤ 2)]

TABLE 14. SUMMARY OF RESULTS OF NON-DETERMINISM ANALYSIS

	Results (%)
Same-source pairs	100
Physically same-source pairs (C1.1)	93
High-level transition and embedded transition pairs (C1.2)	7
True positives	8
False positives	92

Table 14 summarizes the results obtained from the analysis. By deploying our algorithm on the case study, a set of potential cases of non-determinism was discovered statically. The entire set makes a hundred percent (i.e., 100%). The entire set falls in the category of same-source pairs; 93% of which is of type C1.1 while the rest are of type C1.2. By executing the system (i.e., dynamically analyzing the system), we observed that 8% of the cases discovered statically are true candidates of non-determinism (i.e., true positives) and 92% are false positives.

It can be inferred from the results that false positives dominate the results of the analysis. Hence, by eliminating transitions resulting in the *false positives* blindly may be too costly or impossible without destroying the overall architecture.

8.2.2 Case Study 2 – Abstraction via And-Cross Transitions

In this section, we present a modeling example with the goal of demonstrating the need (or an application) of *and-cross transitions* in the design of real-world systems. We present three alternative modeling solutions to the example model and the use of and-cross transitions to accomplish the desired semantics. To support our claim, we present results to compare a solution with an and-cross transition with other alternative modeling solutions (without and-cross transitions). Readers should note that there could be many other possibilities of modeling approaches to this system with equivalent semantics but we present these solutions for the sake of brevity.

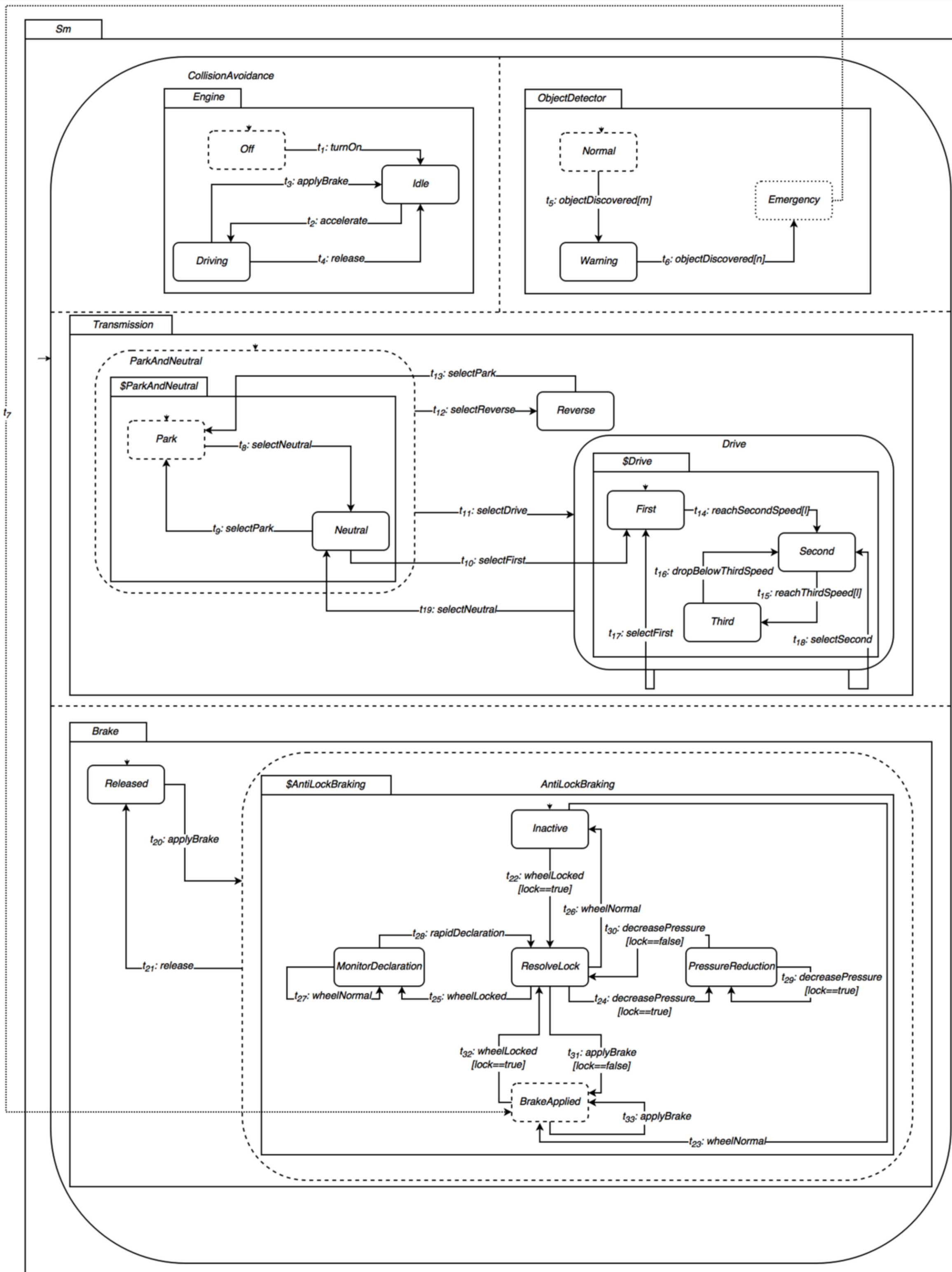


FIGURE 56. THE COLLISION AVOIDANCE SSUA[102]
L := DRIVESELECTED; M := WARNINGRADIUS == 10; N := WARNINGRADIUS <= 3

8.2.2.1 Collision Avoidance Feature

We consider the *collision avoidance feature* (CAF) of a vehicular system with a focus on the aspect that takes action autonomously without driver input (i.e., by braking). In particular, a typical use case relevant to our example may involve the activation of a collision detection system when a stationary object (e.g., vehicle, pedestrian, animal, etc.) is detected in a close-range (e.g., *warning radius – 3 meters*) and requires *maximum braking force*.

Collisions in this category may be *forward* or *rearward*. For example, a realistic automotive case is experienced whenever a threat is detected during parallel parking. Avoiding collision demands *issuing a warning and actuating the brake autonomously*. A collision avoidance system (e.g., Figure 56) is a collection of parallel sub-systems such as object detector, brake, transmission and engine sub-systems. Each sub-system models the relation between states of the system and control actions that represents its controller.

The *object detector sub-system* is equipped to periodically send signals to its environment so as to detect objects within its range of detection. If an object is discovered, it is classified according to its position and an input signal is generated from the environment indicating an object is discovered. The *brake sub-system* implements the ABS (i.e., *anti-lock braking sub-system*). It prevents the wheels from locking in an emergency stop. ABS applies optimum braking to each of the wheels by ensuring the vehicle is steerable and reducing braking distances on slippery surfaces (see Asim and Stephan [101] for ABS's specification).

The *transmission sub-system* (or controller) embodies two other sub-systems (i.e. "ParkAndNeutral" and "Drive") and a "Reverse" mode. "ParkAndNeutral" as its name implies, embodies "Park" and "Neutral" modes. Our transmission sub-system extends the "Car Transmission" state machine in [82], [95] by adding more abstraction to encapsulate "Park" and "Neutral" modes. The *engine sub-system* can assume any of the following modes: "Off", "Idle", and "Driving". A release trigger causes a driving engine to become idle but accelerate causes a change of state to "Driving" and so on. The details of our motivating example is expressed graphically in Figure 56 (can also be found at [102]).

And-cross semantics is relevant in this case because in an emergency situation, the brake is applied, object detector and transmission are reset to normal and park respectively, and engine is turned off (to avoid potential risk of fire, etc.). In essence, all parallel sub-machines of orthogonal state – "CollisionAvoidance" must be re-initialized except the brake whose state must be changed to a non-initial state - "BrakeApplied". We indicated this configuration with dashed and dotted boxes. The dotted box indicates the emergency state which leads to other states indicated in dashed boxes. In particular, emergency state is activated; while the states with dashed boxes (e.g., "BrakeApplied", "Park", etc.) become active before the completion of the overall step.

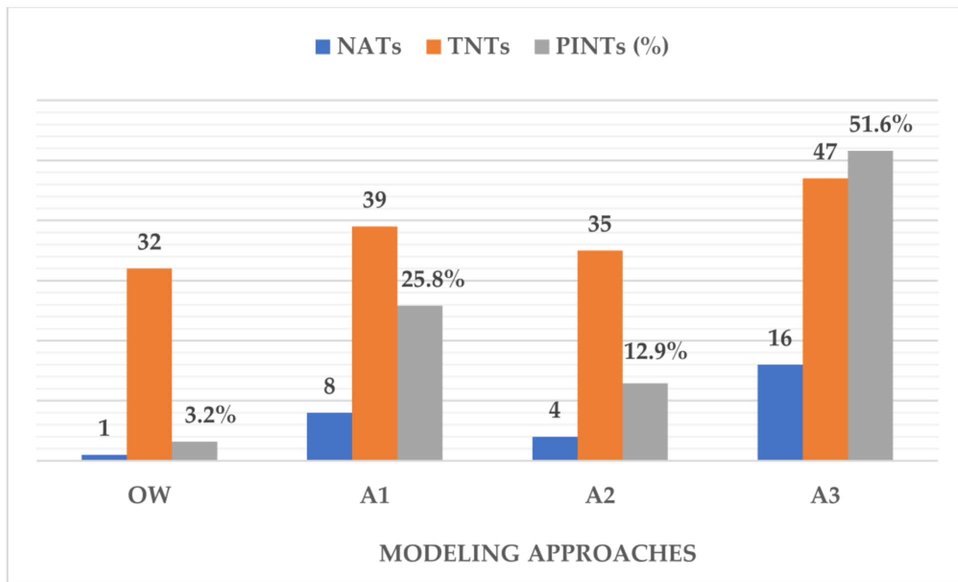


FIGURE 57. COMPARISON OF MODELLING SOLUTIONS (PER NUMBER OF TRANSITIONS) FOR THE COLLISION AVOIDANCE FEATURE

8.2.2.2 Comparing Various Solutions to Model the Collision Avoidance Feature

We have implemented this system using the approaches we proposed in Section 6.4. An implementation based on and-cross transition (denoted by **OW** - Our Work), A1, A2, and A3 can be obtained in UmpleOnline as [102]–[105] respectively. A preliminary result comparing solutions resulting from **OW**, **A1**, **A2** and **A3** with respect to the number of required transitions is summarized in Figure 57. It is intended to show the benefit of and-cross transitions in facilitating

abstraction. We ignore the resulting model of **A4** because it results in exponential blow-up (both in state-space and transitions) and for the sake of brevity.

In Figure 57, the following has corresponding interpretations:

PINTs – Percentage Increase in Number of Transitions
TNTs – Total Number of Transitions
NATs – Number of Additional Transitions

Readers should note that PINTs is calculated w.r.t. the base transitions of 31.

8.2.2.3 Analyzing Collision Avoidance Feature

In this section, we analyze the collision avoidance system to determine whether the model is free of modeling flaws (e.g. nondeterminism) and behaves as intended. In particular, we highlight the desired requirements of the SSUA in a more specific manner (i.e., after successive refinements realized by model checking) as follows:

R1: The collision avoidance feature (i.e., Figure 56) must be deterministic. That is, it must be free from every form of nondeterminism.

Formally:

$$\forall_{\{i,j\} \subseteq \mathbb{N}} \cdot i \neq j \wedge (t_i, t_j) \xrightarrow{\langle e_z, s^k \rangle} \langle\langle_\rangle, \langle \ldots s_i, \ldots s_j, \ldots \rangle, s^{k+1}, \langle\langle_\rangle, \langle_\rangle\rangle$$

OR

$$\text{AG ((} t_i \text{ \& } t_j \text{) -> AX (X(} t_i \text{) \& X(} t_j \text{)))}$$

R2: In an emergency situation, the brake is applied, object detector and transmission are reset to normal and park respectively, and engine is turned off (to avoid potential threats like fire).

Formally:

EF $ObjectDetector.state = "Emergency" and EX($AntiLockBraking.state = "BrakeApplied" and $ObjectDetector.state = "Normal" and $Engine.state = "Off" and $ParkAndNeutral.state = "Park"))

R3: A warning signal can be issued whenever an object (e.g., animal, pedestrian, stationary vehicle, etc.) is discovered at a distance of 10 meters and object detector is in "normal" mode.

Formally:

```
AG ( ( $ObjectDetector.state = "Normal" and warningRadius = 10 and
  event = "objectDiscovered" ) -> $ObjectDetector.state = "Warning" )
```

R4: For correctness purposes, there is no emergency situation unless an object has being discovered and warning radius reaches or drops below the threshold (i.e., 3 metres).

Formally:

```
A [ $ObjectDetector.state != "Emergency" U ( event = "objectDiscovered" -> warningRadius <= 3 )]
```

R5: For safety purposes, an operator must be warned prior to any emergency situation.

Formally:

```
AG ( $ObjectDetector.state = "Warning" -> EF $ObjectDetector.state = "Emergency" )
```

We are specifically interested in answering the following questions:

Q1. Given these desired behaviors of the feature under analysis (FUA), we would like to investigate whether the SSUA models (including alternative solutions) are semantically similar.

Q2. What impact does each approach have on increasing the number of potentially conflicting transitions?

Q3. What impact do these encodings have on the global state-space? In particular, are there differences in the number of reachable states of the global state-space or increase the state-space of the SSUA?

Q4. Does abstraction actually imply performance? In particular, is the level of abstraction in SSUA directly proportional to degree of performance (i.e., for execution time and memory usage).

We further our investigation by formalizing the CAF for each of the approaches in Umple and delegate our translator (i.e., Umple2SMV) to generate its equivalent systems in SMV, the input language of nuXmv [45] and NuSMV [84] model checker. The requirements are formalized in LTL [5] and CTL [5], [106] as the need arises. However, for this analysis, we adopted NuSMV

(version 2.6) running on Ubuntu 16.04 LTS machine with Intel Core i7 processor of 2.7 GHz and 8GB RAM as the model checker.

TABLE 15. ANALYSIS AND PERFORMANCE RESULTS FOR VARIOUS METHODS AND REQUIREMENTS

RQs	Method	State-space (*)	Reachable States (**)	MCET (s)	# of BDDs	MCMU (MB)	ARs	ATNP (s)	AMNP (MB)
R1	A1	2.34566	2.3	0.034	24353	9.84	TRUE	0.031	12.09
	A2	-	2.8	0.034	29277	10.01	-	0.032	10.44
	A3	-	2.5	0.032	25774	9.91	-	0.033	11.05
	OW	-	2.8	0.039	31547	10.08	-	0.021	8.49
R2	A1	-	2.3	0.031	27431	9.48	-	--	--
	A2	-	2.8	0.035	32865	9.68	-	--	--
	A3	-	2.5	0.036	28839	9.55	-	--	--
	OW	-	2.8	0.053	58587	10.53	FALSE	--	--
R3	A1	-	2.3	0.033	24933	9.42	TRUE	--	--
	A2	-	2.8	0.034	29876	9.58	-	--	--
	A3	-	2.5	0.035	26358	9.45	-	--	--
	OW	-	2.8	0.033	32177	9.65	-	--	--
R4	A1	-	2.3	0.031	24753	9.42	-	--	--
	A2	-	2.8	0.035	29666	9.58	-	--	--
	A3	-	2.5	0.029	26161	9.45	-	--	--
	OW	-	2.8	0.032	31935	9.65	-	--	--
R5	A1	-	2.3	0.034	26760	9.48		--	--
	A2	-	2.7	0.035	31738	9.64	-	--	--
	A3	-	2.5	0.034	28188	9.52	-	--	--
	OW	-	2.8	0.035	34071	9.72	-	--	--
R6 (IL)	A1	-	2.3	0.045	44199	9.85		--	--
	A2	-	2.7	0.047	40556	10.06	-	--	--
	A3	-	2.5	0.044	38624	9.92	-	--	--
	OW	-	2.8	0.057	61076	10.59	-	--	--

Key: (RQs – Requirements; * - E+08; ** - E+06; OW - Our Work; MCET – Model Checking Execution Time; MCMU – Model Checking Memory Utilization; ARs – Analysis Results; ATNP – Algorithm Time)

8.2.2.4 Discussion of Case Study 2

We analyzed the SSUA given the requirements established in the previous section (i.e., Section 6.2). For each of the encodings (A1, A2, A3, A4 and OW), we deployed our translator to generate the corresponding SSUA in SMV. Then, we fed the model (i.e., SMV representation of the SSUA) and its requirements (i.e., LTL and CTL representations) into the model checker. For memory

usage and execution times, results are computed as average of 6 successive runs (see Table 15) for each of the approaches.

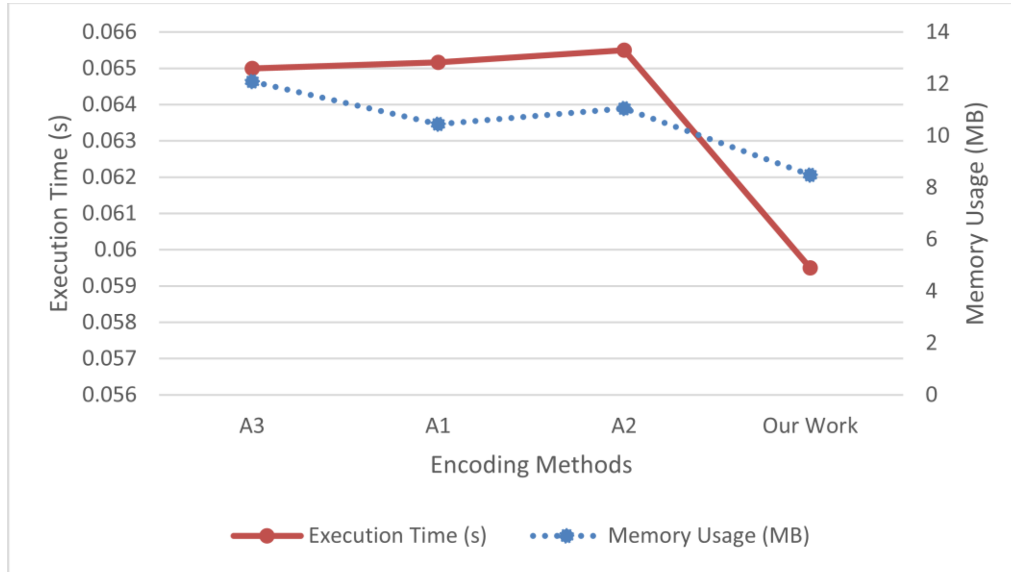


FIGURE 58. EXPERIMENTAL RESULTS FOR R1 VERSUS METHODS

where:

Memory Usage = MCMU+AMNP

Execution Time = MCET +ATNP

Requirement - R1

We deployed our algorithm (i.e., Listing 27) on each encoding to determine the set of potentially conflicting transitions. Then we analyzed the SSUAs w.r.t **R1**. The results obtained are presented in Figure 58.

For each encoding, a similar pair of transitions was discovered as potentially conflicting. The results obtained shows that the methods behave similarly for requirement R1. In particular, the models realized from these methods are deterministic. In Figure 58, we presented the results obtained graphically to explicate the benefit of the method we proposed (i.e., and-cross transition). We are interested in the total time taken to determine if the model under consideration is

deterministic or not. It can be seen that our method outperforms other methods both in memory consumption and execution time.

This is based on the fact that the greater the number of transitions, the longer our match-making algorithm (i.e., Listing 27) takes and higher the memory required to compute the set of pairs of potentially conflicting transitions. Therefore, we can conclude that the level of abstraction of a given model is inversely proportional to memory usage and execution time when certifying a model to be deterministic with our algorithm.

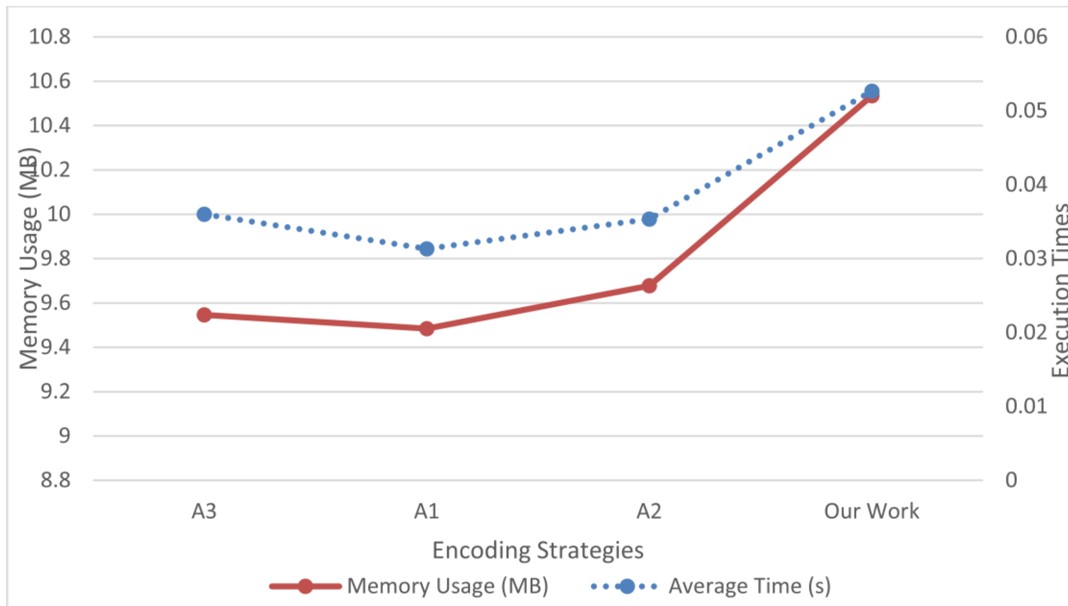


FIGURE 59. RESOURCE UTILIZATION FOR THE ANALYSIS OF R2

Requirement - R2

We analyzed the models with respect to **R2**. For all the alternative solutions, the SSUA conforms to R2. However, our approach produces a counterexample whose trace shows that there is no path fulfilling **R2** within the SSUA. This results from the fact that whenever an *and-cross transition*

executes, the target state is reached in the immediate step resulting from its execution. But the re-initializations of other parallel sub-state machines (or regions) are done at the next step.

This is a consequence of our encoding which activates a composite state at the same step as the step its embedded state becomes active. Hence, an orthogonal composite state becomes active in a step an and-cross transition from its enclosed region executes and re-initializes all other regions but the region of the target state in the next step. Resultantly, we have re-designed **R2** into **R6**.

> **R6:** In an emergency situation brake should be applied immediately then object detector, engine and transmission are reinitialized eventually. This is a refinement of **R2** with a focus on accommodating our semantics where initial states of a sub-state machine (including regions) are being activated automatically in the immediate step after its parent state is activated.
>
> Formally:

```
EF $ObjectDetector.state = "Emergency" ->
  EX( $AntiLockBraking.state = "BrakeApplied" and
    EF ( $ObjectDetector.state = "Normal" and
      $Engine.state = "Off" and $ParkAndNeutral.state = "Park" ) ) )
```

Although, our approach does not satisfy **R2**, but we consider the results obtained important to demonstrate that searching for a counterexample requires similar amount of resources as searching for an example. Thus, we present analysis result (see Table 15) of **R2** in Figure 59.

Requirements R3, R4, R5 and R6

We analyzed the encodings given that **R3** – **R6** are the requirements under consideration. It was observed that all encodings (including Our Work) fulfilled these requirements.

In particular, we verified the SSUAs constructed with the encodings under consideration against **R6** and observed they satisfy the requirement. We were able to prove that whenever an emergency arises, the brake is applied in the immediate step after the incidence. Then engine, transmission, and object detector are reset to their initial states in a step after the target state becomes active.

8.2.2.5 Answering Questions (Q1-Q4)

In this section, we present our answers to these questions based on the results we obtained from model checking the requirements established in Section 7.2.2.3 against the SMV representation of the SSUAs resulting from the encoding strategies adopted for this study.

> **Q1.** "Given these desired behaviors (i.e., **R1**-**R6**) of the feature under analysis (FUA), we would like to investigate whether the SSUA models presented in [102]–[105] are semantically similar."

For requirements **R1**, and **R3** – **R6** the resulting SSUAs behave identically (i.e., producing the same results for each requirement irrespective of the encoding strategy). Encodings **A1**-**A3** behave identically for requirements **R1**-**R6** because there is a direct link to the initial states of parallel regions. As highlighted (see discussion on **R2**), our encoding behaves differently with **R2** because we delegate to composite states the responsibility of its sub-state machine or its regions.

We consider R2 to be too strong and observe that it can be relaxed to accommodate our semantics yet fulfilling its objectives. This results in the formulation of R6 such that all encodings fulfill this requirement.

> **Q2.** "What impact does each approach have on the number of potentially conflicting transitions?"

The approaches yield the same result for potentially conflicting pairs. In particular, for the SSUA there is a pair of transitions whose sources and destinations are similar but the indexes may differ from one approach to another. The results obtained are as follows:

> **A1:** (t_{15}, t_{16}); **A2:** (t_{15}, t_{16}); **A3:** (t_{18}, t_{16}); and **OW:** (t_{14}, t_{15}).

> **Q3.** "What impact does these encodings have on the global state-space? In particular, are there differences in the number of reachable states of the global state-space or increase the state-space of the SSUA?"

We observed that each encoding maintains the same global state-space irrespective of the variations in the number of transitions in each method. In Figure 60, we present the results obtained graphically to ease understanding. This is based on our choice of encoding (i.e., usage of macros

for transitions). The results presented here validate the argument of Faghih and Day [86] that macros don't contribute states to the global state-space.

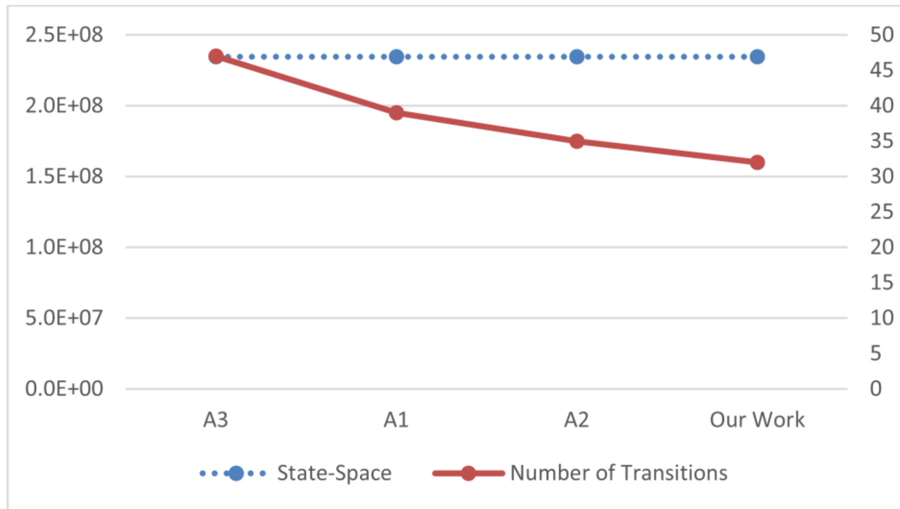


FIGURE 60. GRAPH OF STATE SPACE VS NUMBER OF TRANSITIONS

However, we also observed that the more abstract a model is, the greater the amount of reachable states of the entire state-space. In particular, the amount of reachable states of the entire state-space is inversely proportional to the number of transitions. Figure 61 is the visual representation of the results obtained from our study. To validate this conclusion, we present the results we obtained from studying the correlations between the number of transitions and amount of reachable states at ninety-nine percent confidence level is given as follows:

> No of Transitions Vs. Reachable States: **r** = -0.57 and **p-value** = 0.78

> **Q4.** "Does abstraction actually imply better performance during model checking? In particular, is the level of abstraction of an SSUA directly proportional to degree of performance (i.e., for execution time and memory usage)."

By conventional wisdom, the more abstract an SSUA is, the better we anticipate its performance. This appears to be *untrue* with the verification of requirements by model checking. In fact, the opposite is the case for memory usage, execution time, and number of BDDs. It seems to be

influenced by the reachable amount of states in the entire state-space. Therefore, we isolate the data presented in Table 16 from Table 15 to allow us draw accurate conclusions.

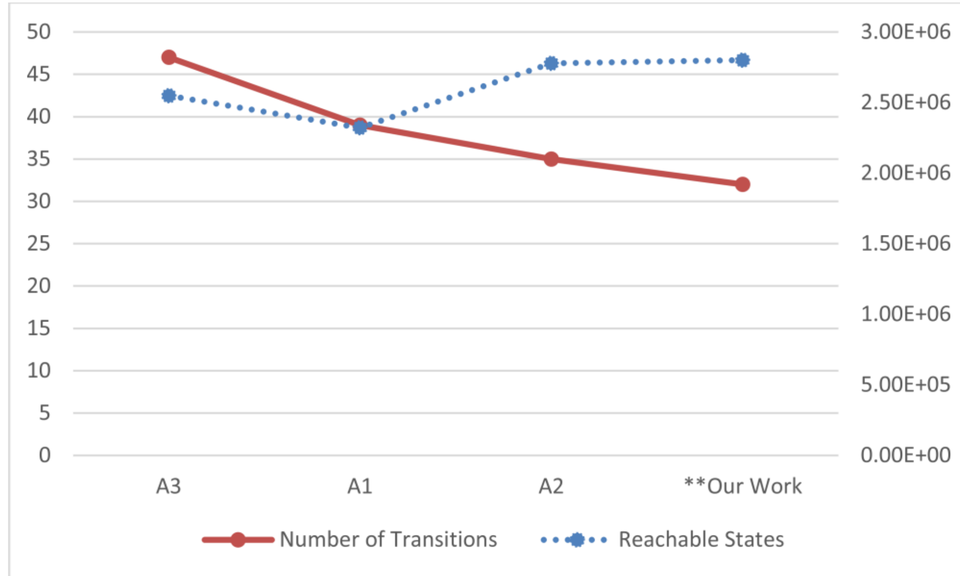


FIGURE 61. GRAPH OF NO. OF TRANSITIONS VS. REACHABLE NUMBER OF STATES

TABLE 16. SUMMARY OF ANALYSIS RESULTS
NB: AVERAGE TIME AND MEMORY ARE COMPUTED AS AVERAGES OF TIME AND MEMORY SPACE REQUIRED FOR VERIFYING THE REQUIREMENTS (I.E., **R1-R6**)

	Reachable States (E+06)	**Average Time (s)**	**Average Memory (MB)**	**Average # of BDDs**
A1	2.32	0.034	9.5	28738
A2	2.78	0.036	9.7	32330
A3	2.55	0.035	9.6	28991
OW	2.80	0.041	10.0	41566

We present in Figure 62 - Figure 64 the graphs visually representing the relationships between average execution times and memory usages for the encoding strategies. The graphs depict that

user-level abstraction does not necessarily imply good performance, both in memory and execution time.

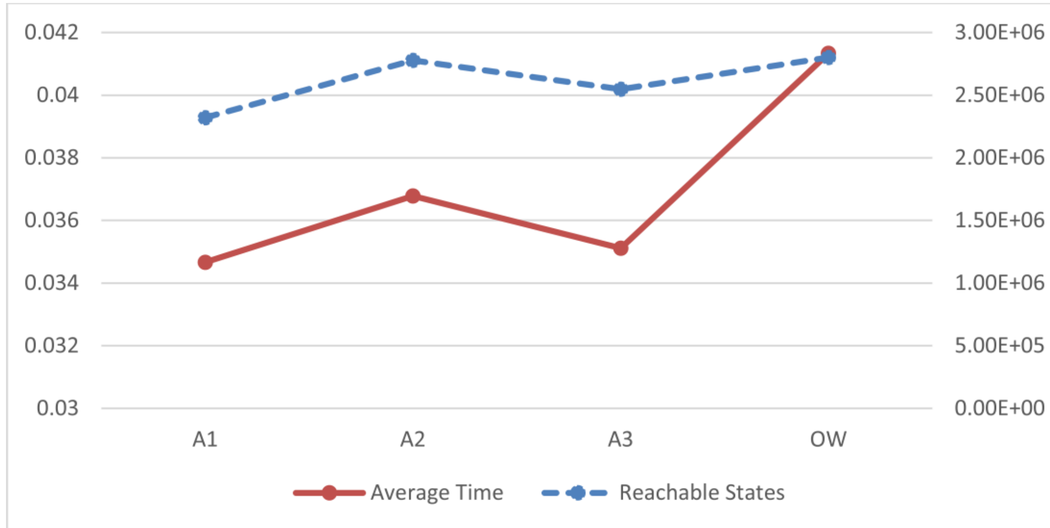


FIGURE 62. GRAPH OF AVERAGE TIME AND REACHABLE STATES OF THE SSUAS

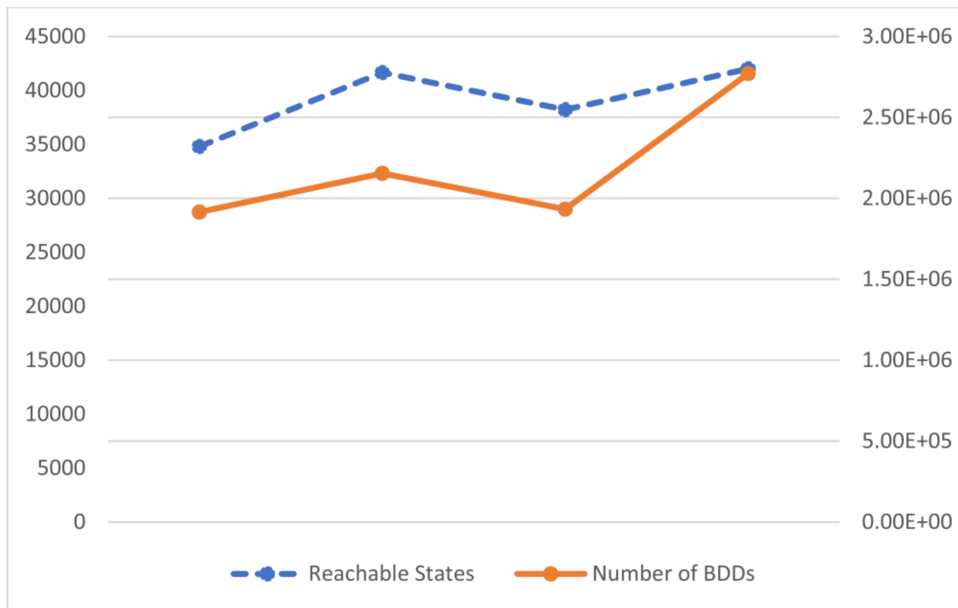


FIGURE 63. GRAPH OF AVERAGE NUMBER OF BDD NODES AND REACHABLE STATES OF THE SSUAS

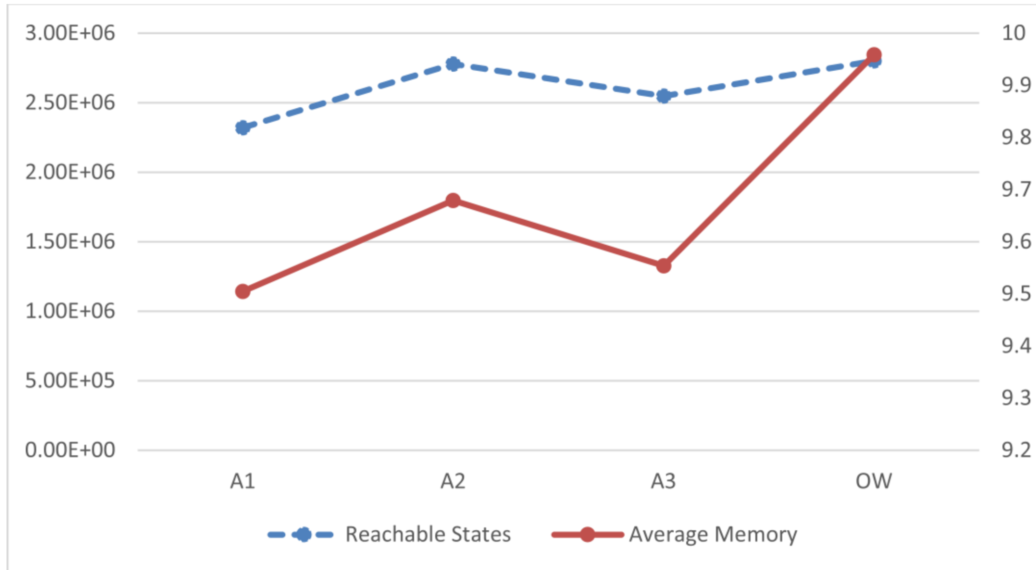


FIGURE 64. GRAPH OF AVERAGE MEMORY USAGE AND REACHABLE STATES OF THE SSUAS

In the same vein, it can be observed that performance is dependent on the number of reachable states of the global state-space. Particularly, we studied the correlations between execution times and reachable states as well as memory usage and reachable states with *ninety-nine percent confidence level* via the Pearson method. The following are the results obtained:

Execution Time Vs. Reachable States: ***r*** *= 0.77 and* ***p-value*** *= 0.23*
Memory Usage Vs. Reachable States: ***r*** *= 0.81 and* ***p-value*** *= 0.19*
Number of BDD Nodes Vs. Reachable States: ***r*** *= 0.74 and* ***p-value*** *= 0.26*

The results indicate strong correlations for memory usage and execution time w.r.t. number of reachable states. Consequently, execution time and memory usage tend to increase with an increase in the number of reachable states of the global state-space and vice-versa.

8.2.2.6 Concluding Remarks – Case Study 2

First, we can thus observe better abstraction as advocated by MDE with and-cross transitions. In particular, we observed a drastic reduction in the number of required transitions for encoding the SSUA with and-cross transitions as opposed other modeling approaches.

The results obtained is a reflection of the encoding strategies. For strategies A1, A2, and A3 there are direct transitions to the initial states of the non-targeted regions. Therefore, it requires at most a micro-step to realize the desired configuration unlike and-crossing that requires 2 micro-steps. In particular, a micro-step is required to activate the target state of the and-cross transition and the enclosing orthogonal composite state. Another step is required to activate the initial states of every parallel region of the enclosing state except the target region of and-cross transition.

We are aware that model checkers implement various forms of optimization to ensure large-scale models are analyzable. These optimizations might have influenced these results. Although, results show that user-level abstraction does not necessarily imply performance by model checking. However, it is worthy to note the following benefits of abstraction introduced by and-cross transitions:

> **B1.** Abstraction becomes important to us because it helps in realizing correctness when managed properly. By improving the level of abstraction, we tend to improve the quality of SSUAs being developed. This is the ultimate goal of model-driven engineering approaches.
>
> **B2.** With fewer transitions realized by the aid of and-cross transition, the time taken and memory required to certify a model to be deterministic is drastically reduced compared to other approaches (see Figure 57). This is influenced by the time and memory required to compute the set of pairs of potentially conflicting transitions.
>
> **B3.** Although we adopted macros for encoding transitions, thus the increase in the number of transitions has no impact on the global state-space irrespective of the encoding strategies. If we had considered BSML2SMV [86], the global state-space would have grown exponentially.
>
> **B4.** The effort or resources demanded to develop a similar model using alternative modeling solution increases with respect to the number of states (and/or regions) of the immediate enclosing state but is constant whenever and-cross transition is adopted.

In view of these, we consider managing the underlying complexities of the notion of and-crossing an important contribution of this research.

8.2.3 Case Study 3 – Asserting Observational Equivalence of Similar SSUAs

In this section, we determine the observational equivalence of two similar state machines by model checking. Cruz-Lemus et al. [107], [108] conducted an experimental study to investigate the influence of state hierarchy in understanding UML state machine diagrams.

They consider simple and hierarchical versions of Automatic Teller Machine and Xholon Watch as case studies for experimentation purposes. Categories of subjects for the experiment include software professionals and students (i.e., undergraduate and graduate) from various universities across the globe. The experiment setup involves various questions designed for subjects to be answered with respect to the versions of the state machines.

The experiment was well-defined and results were formally presented statistically. In fact, various threats to validity were identified and dealt with systematically. However, threats resulting from non-equivalence of the simple and hierarchical versions of the system were neither identified nor discussed. We suspect this might have influenced the overall result positively or negatively. Besides, we hope this experiment will help us validate (or clarify) the results we obtained while comparing abstraction with performance (Q4 of Case Study 2). In particular, we are inspired to conduct this experiment to establish that user-level abstraction does not necessarily impact performance.

Therefore, our goal is to formally (i.e., by model checking) determine if both hierarchical and simple versions of the SSUAs adopted for the experiments are semantically similar. But for the sake of brevity, we will focus our analysis only on the Xholon Watch. We realize this goal by carrying out the following steps:

Step 1: Translating the state machine diagrams presented in [107] to their equivalent Umple system;

Step 2: Transforming the Umple versions (i.e., simple and hierarchical) of this SSUA to its equivalent SMV model;

Step 3: Formally specify the questions for the participants on the SSUA in the dialect of temporal logic (i.e., CTL or LTL);

Step 4: Analyzing the SMV versions of the SSUA against the formal specifications of the questions using a back-end analysis engine like NuSMV or nuXmv (where necessary); and

Step 5: Comparing the analysis results of these requirements to draw a logical conclusion regarding the equivalence of the versions of the SSUA.

8.2.3.1 The Xholon Watch System

The Xholon watch models the internal behavior of a generic digital watch. It has four buttons labeled $\{s_i \mid 1 \leq i \leq 4\}$. It displays date or time by pressing these buttons in different sequence the configuration of the watch changes. The functionalities of the buttons are summarized as follow:

F1: s_1 manages the date and time;

F2: s_2 sets the alarm on or off;

F3: s_3 + normal is responsible for changes to Chronometer;

F4: s_3 + long controls update to date, time and alarm; and

F5: s_4 manages the light.

A hierarchical state machine corresponding to the watch as presented in [107] has been modeled in Umple and is publicly available at [56]. We present a formal representation of the questions as temporal logic (i.e., CTL and LTL) statements to allow us analyze them against the SMV specification of the SSUA via the model checker. While there are varieties of questions, ranging from modeling specific scenarios to the determination of conformity of model to specific properties, we are interested only in questions whose results are obtainable by model checking. The questions we considered are presented as follows with respect to their formal representations:

Q1: If we are in the state TIME and button s_1 is pressed twice we reach the state TIME again.

Formally:

```
G ( ( ${ Regular, Sm }.state = “time” -> event = s1 -> event = s1 ) -> ${ Regular, Sm }.state = “time” )
```

Q2: There is only one possible combination of buttons to set the alarm off.

To formulate this specification, we decide to get the series of events (or path) leading to the desired state (i.e., alarm off); then we request the model checker to find at least a path leading to the desired state other than the path obtained in the earlier search. This in-turn yields two (i.e., Q21 and Q22) sub-specifications which are formally presented as follows.

Formally:

```
Q21: !F ( ${ AlarmStatus, Sm }.state = "bothOff" )
```

By executing the SSUA with respect to Q21, we observed that from the initial configuration of the SSUA, there is a path to “bothOff” whenever the “time” is active and environment generates “s2”.

Thus, we can conclude that there is only one path in the SSUA if the consequence of generating “s2” at “time” always results in “bothOff”. We formulated Q22 based on this premise.

Formally:

```
Q22: AF ( ( ${ Regular, Sm }.state = "time" -> event = s2 )
          <-> AX ${ AlarmStatus, Sm }.state = "bothOff" )
```

Q3: The chronometer may be running while the date is displayed on the watch;

Formally:

```
F ( ${ Chronometer, Sm }.state = "lapRunning"  & ${ Regular, Sm }.state = "date" )
```

This resulted in false for both cases because there is no parent-child relationship between date and lap running. In particular, the validity of this requirement will result in inconsistencies or modeling flaw.

Q4: When updating the date and time we can increase and decrease the values by pressing several buttons.

To realize this specification, we divided it to four sub-specifications. First, we assume the current time and date as 08:11:00 and 17:12:2016 respectively. We then request the model to find a path leading to any previous time and date (i.e., checking the combination of buttons leading to decrease in date and time). Similarly, we request the model checker to find a path leading to any future

time and date (i.e., checking the combination of buttons leading to increase in date and time).

Q41: Given the current time 08:11:00, we can increase the values by pressing several buttons.

Formally:

F (Hour = 8 & Minute = 11 & Seconds = 0 -> F (Hour > 8 & Minute > 11 & Seconds > 0))

Q42: Given the current time 08:11:00, we can decrease the values by pressing several buttons.

Formally:

F (Hour = 8 & Minute = 11 & Seconds = 0 -> F (Hour < 8 & Minute < 11 & Seconds > 0))

Q43: Given the current date 17:12:2016, we can increase the values by pressing several buttons.

Formally:

EF (Day = 17 & Month = 11 & Year = 2016 -> EF (Day > 17 & Month < 11 & Year > 2016))

Q44: Given the current date 17:12:2016, we can decrease the values by pressing several buttons.

Formally:

EF (Day = 17 & Month = 11 & Year = 2016 -> EF (Day < 17 & Month < 11 & Year < 2016))

Q5: If button s_3 is pressed for 2 seconds while the date is being displayed, we get to the alarm update mode.

Formally:

F ((${ Regular, Sm }.state = “Date” -> event = s3during2Secs) -> $Sm.state = “AlarmUpdate”)

Q6: The order in which the alarm, date and time are updated is always the same;

We attempted to answer this question by contradiction. In particular, we assume that the alarm can be updated before date and time are updated and vice-versa. That is, there at least two orders for updating alarm, date and time.

We specified this as follows and obtain results mechanically by model checking.

Formally:

```
AG AF ( ${ Regular, Sm }.state  = "Update"
            -> ( ( ${ Sm, Update }.state = "Second"    or  ${ Sm, Update }.state = "Minute"
                or ${ Sm, Update }.state = "Hour" )    -> ( ${ Sm, Update }.state = "Month"
                or ${ Sm, Update }.state = "Day"    or  ${ Sm, Update }.state = "Year" ) ) )
```

Q7: While updating the alarm, date and time, the real time can be displayed at any moment by pressing one button.

Formally:

```
AG ( ( ${ Regular, Sm }.state  = "Update" or $Sm.state = "AlarmUpdate" )
                    -> ( ${ Regular, Sm }.state  = "Time" ) )
```

Q8: There is a specific button combination to change the day of the week we are in.

We tackle this by contradiction thus dividing the specification into two sub-specifications. Firstly, we assume there is no future state when the day of the week is being updated. This produces a trace of a path leading to "day" including all triggering events. Then, we formulated a specification stating that there is another path leading to "day" whose events combination differ from the path produced by the first sub-specification.

Q81: There is no future state where "day" of the week is being updated.

Formally:

```
!EF ( ${ Update, Sm }.state = "Day" )
```

Q82: The result of Q81 shows that there is a path from the initial configuration of the SSUA to "day" without the following combinations of events "s3during2Sec", "s3", "s1", "s1", "s1", and "s1".

Formally:

```
AG ( $Sm.Stable = TRUE and
   ! EX ( event = "s3during2Sec"  -> ( EX event = "s3" -> ( EX EX EX EX event = "s1" ) ) )
   -> ${ Update, Sm }.state = "Day" )
```

TABLE 17. ANALYSIS RESULTS FOR XHOLON WATCH STATE MACHINE DIAGRAMS

Req.	Simple State Machine of Xholon Watch				Hierarchical State Machine of Xholon Watch			
	Analysis Results	Time (s)	Memory (MB) E+07	# of BDDs E+05	Analysis Results	Time (s)	Memory (MB) E+08	# of BDDs E+06
Q1	F	5.4	5.77	6.77	F	322.4	0.75	0.72
Q21	F	6.1	5.69	8.96	F	467.2	1.08	1.50
Q22	T	2.5	5.69	6.19	T	141.6	0.85	0.88
Q3	F	5.6	5.72	6.49	F	314.1	0.75	0.80
Q41	T	2.7	5.43	1.27	T	151.1	0.69	0.65
Q42	T	2.8	5.42	1.26	T	148.4	0.69	0.65
Q43	T	2.7	5.70	7.18	T	147.9	0.78	1.22
Q44	T	2.7	5.70	6.39	T	144.5	0.71	1.38
Q5	F	6.4	5.69	5.83	F	110.5	0.59	0.65
Q6	T	3.1	5.69	6.19	T	117.4	0.62	0.99
Q7	F	3.2	5.69	6.23	F	124.0	0.62	0.99
Q81	F	3.2	5.69	6.32	F	109.9	0.62	0.66
Q82	T	3.1	5.69	6.19	T	119.7	0.62	0.95

8.2.3.2 Discussion of Results

In this section, we present the results obtained from the analysis of both simple and hierarchical state machines of the Xholon watch [107]. While some questions can be specified in temporal logics (i.e., LTL and CTL), some cannot be expressed. Similarly, we observed some similarities but few variations from the models. For example, some questions are expressible in the hierarchical state machine but not in the simple state machine. The following questions were ignored for reasons like: inability to express them as temporal properties and ambiguity in their construction (or statements).

IQ1: What would happen if the following sequence (i.e., s_3, s_1, s_2, s_1 and s_2) of buttons is pressed while the watch is displaying time?

IQ2: What would happen if the following sequence (i.e., s_1, s_1, s_2, s_1, s_1 and s_2) of buttons is pressed while the watch is displaying time?

IQ3: Whenever a button is pressed, there is a transition between states.

IQ4: Does the diagram model how the light of the watch works?

In Table 17, we present results obtained empirically comparing the simple and hierarchical versions of the Xholon Watch. Given the set of questions answered during the experiment, we are able to assert the equivalence of the SSUAs (i.e., simple and hierarchical versions). In particular, the requirements established from the questions produce the same results for both versions of the SSUA. This can be clearly seen from the columns corresponding to analysis results for hierarchical and simple versions of the SSUA (as presented in Table 17).

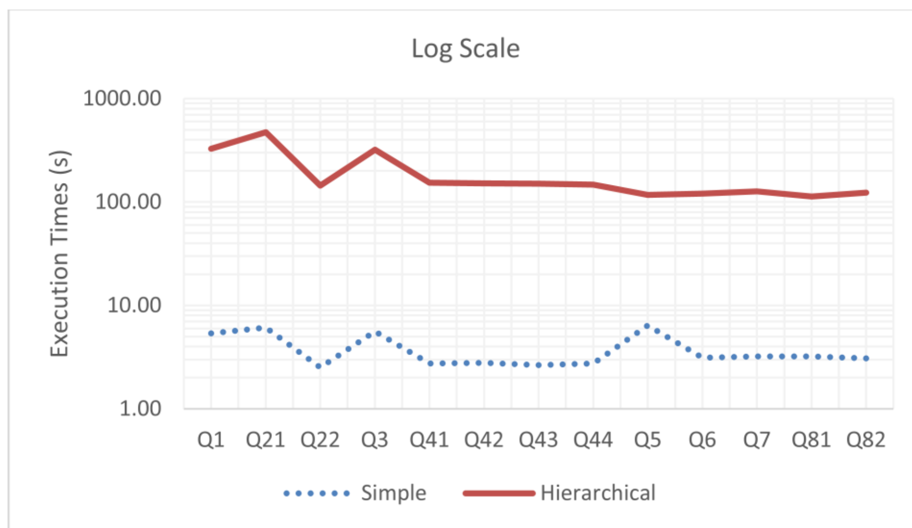


FIGURE 65. GRAPH OF EXECUTION TIMES FOR QUESTIONS FOR SIMPLE AND HIERARCHICAL SSUAS OF XHOLON WATCH

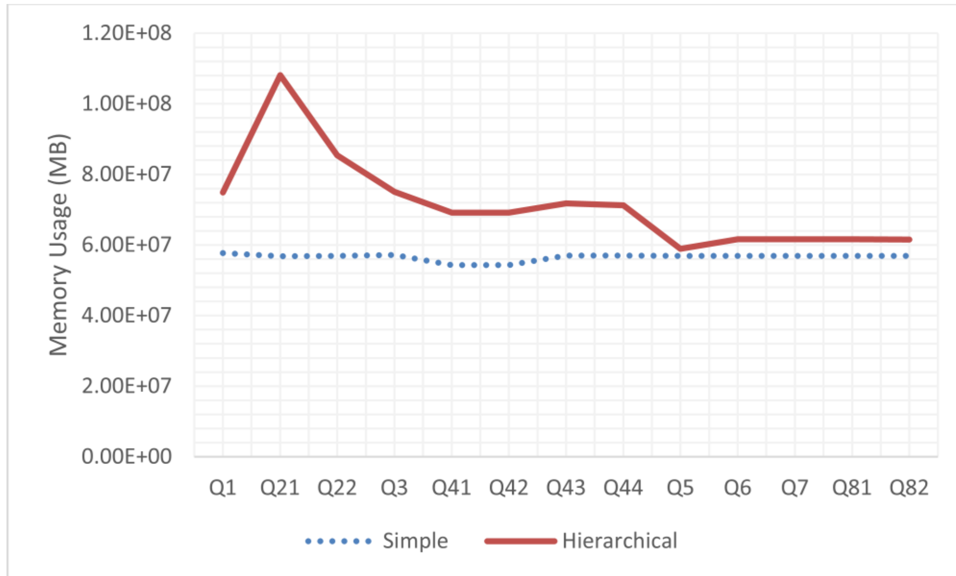


FIGURE 67. GRAPH OF MEMORY UTILIZATION FOR XHOLON WATCH SSUAS (I.E., SIMPLE AND HIERARCHICAL)

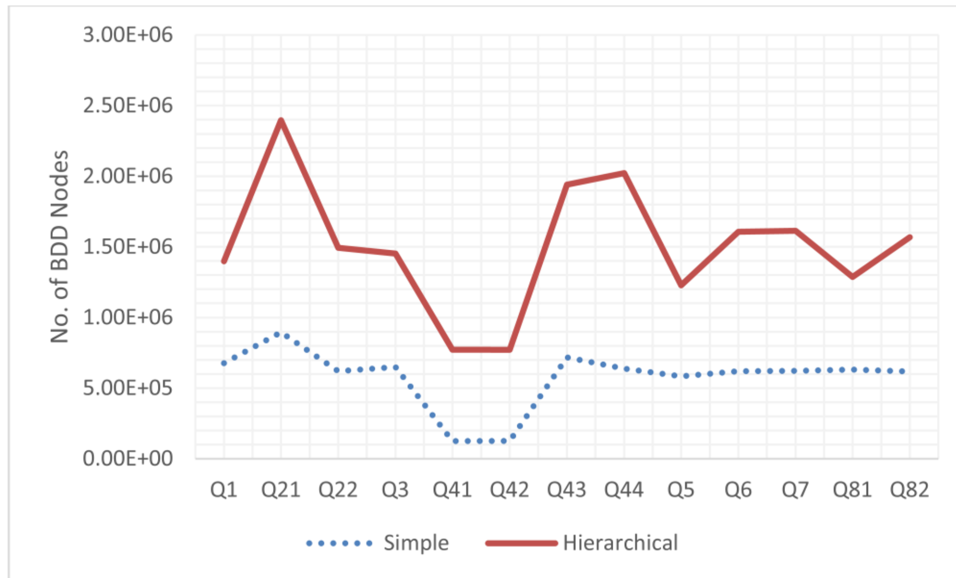


FIGURE 66. GRAPH OF REQUIRED NUMBER OF BDDS FOR SSUAS (I.E., SIMPLE AND HIERARCHICAL) OF XHOLON WATCH

In the same vein, we are interested in determining whether abstraction necessarily imply performance (i.e., execution times, memory usage and number of BDD nodes) during the process of analysis. To ease the comparison, we visualize the results obtained in Figure 65 - Figure 66.

It can be inferred that for all the parameters (i.e., memory, time and number of BDD nodes) and questions we considered in this experiment, the simple state machine outperforms its hierarchical version of the SSUA.

To further our investigation on the influence of abstraction on performance parameters, we consider **Case Study 3** important since it compares a non-hierarchical version of the Xholon watch with its hierarchical counterpart. This is a comparison with focus on different levels of abstraction. From **Case Study 2**, we observed that performance is a function of the reachable states of the global state-space. Therefore, we summarize the execution times, memory usage and number of BDDs that we presented in Table 17 as Table 18.

TABLE 18. SUMMARY OF ANALYSIS RESULTS FOR CASE STUDY 3

NB: AVERAGES OF TIME, MEMORY AND BDDS ARE COMPUTED AS AVERAGES OF RESULTS OBTAINED FOR THE VERIFICATIONS OF Q1 – Q82

	Reachable States (E+14)	Average Time (s)	Average Memory (MB – E+08)	Average # of BDDs (E+06)
Simple SM	1.7	4	0.6	0.6
Hierarchical SM	1.8	186	0.7	0.9

As can be inferred from Table 18, the simple version of the SSUA outperforms its hierarchical counterpart for execution time, memory usage and number of BDDs. We consider this as another indicator and a justification to our finding "*that user-level abstraction does not imply performance*".

8.2.3.3 Conclusion – Case Study 3

The focus of this investigation (or experiment) is to formally assert the equivalence of simple and hierarchical versions of the Xholon watch. Its goal is to bolster results obtained from the

experimental study conducted by Cruz-Lemus et al. [107], [108] and establish that user-level abstraction does not necessarily impact performance positively. After the experiments, we are able to reach the following conclusions:

> **C1:** Both simple and hierarchical versions of the Xholon watch behave similar semantically in the light of questions given to the participants (or designed for the experiments). We draw this conclusion by formally examining the SSUAs via model checking and obtaining the same results for each requirement.
>
> **C2:** Abstraction impacts performance (i.e., execution time, memory usage and number of BDD nodes) negatively. In particular, the simple version of the SSUA outperforms its hierarchical counterpart in all performance parameters even though their behavior is similar.
>
> **C3:** By comparing results obtained via analysis by model checking for both simple and hierarchical versions of the same SSUA, we were able to conclude that abstraction may impact performance of a model negatively in search of a solution to any given requirement.
>
> **C4:** We recommend that whenever possible, simple (or non-hierarchical) SMV equivalent of Umple state machines should be used for the purpose of analysis if performance parameters (e.g., memory usage, execution time and number of BDD nodes) are primary concerns.

8.2.4 Case Study 4 - Electronic Seating System

The electronic seating system (i.e., e-seat software) automates the process of assigning seats randomly to students for the purpose of examination. In this section, we deploy our work to validate the correctness of the **e-Seat** software. Doing this, we aim at validating: 1) our transformation engine from Umple to Alloy; 2) the structural correctness of the e-Seat system underlying model expressed as Umple class diagrams.

In Figure 68, we present the class model of the electronic seating arrangement system. It shows the classes and their attributes as well as associations between the classes of the model. Our transformation engine was applied to the system presented in Figure 68 to transform its Umple representation to its Alloy equivalent. We ran the generated Alloy specification of the e-Seat system on 100 objects and an instance was found in 69s.

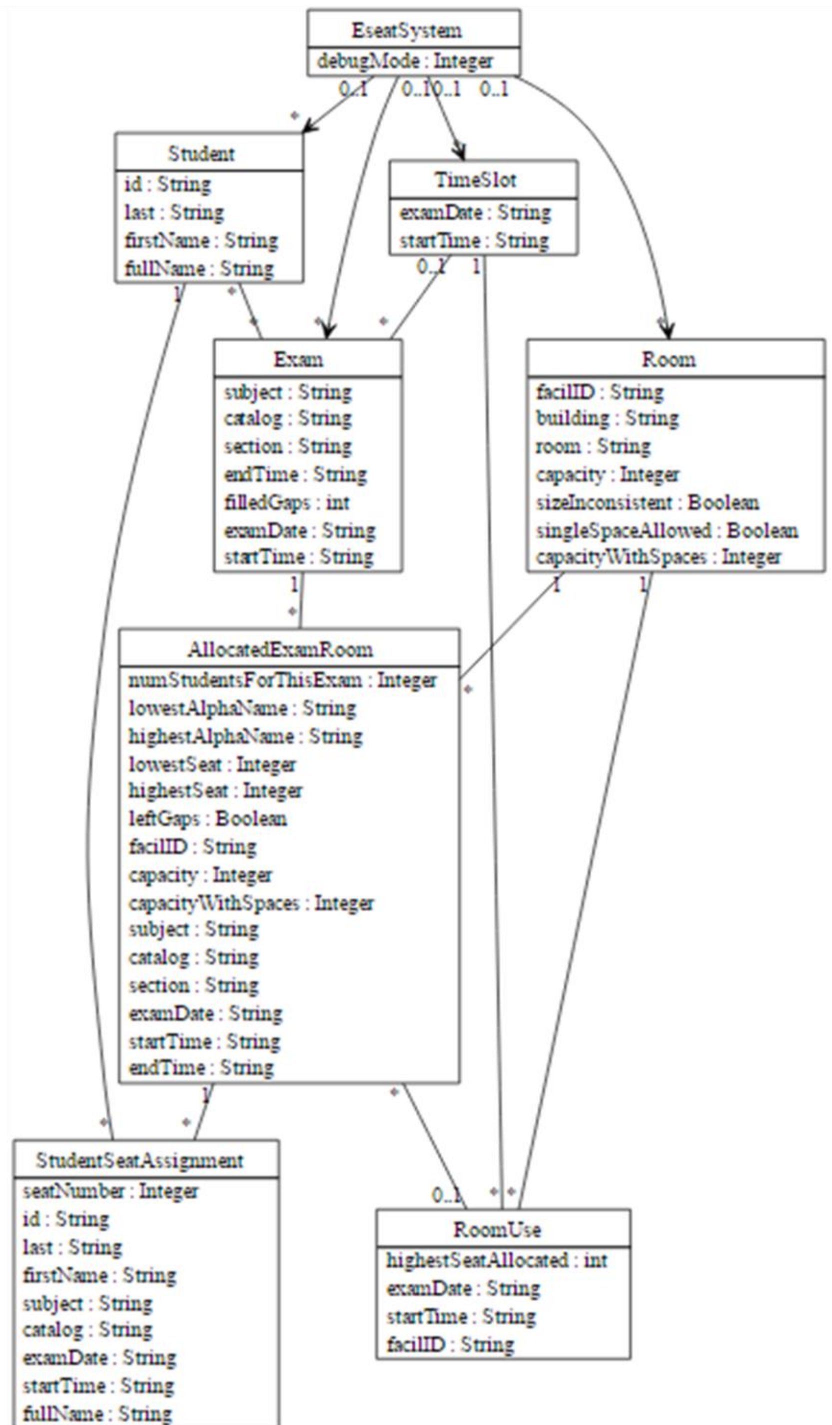


FIGURE 68. MODEL OF THE ELECTRONIC SEATING SYSTEM

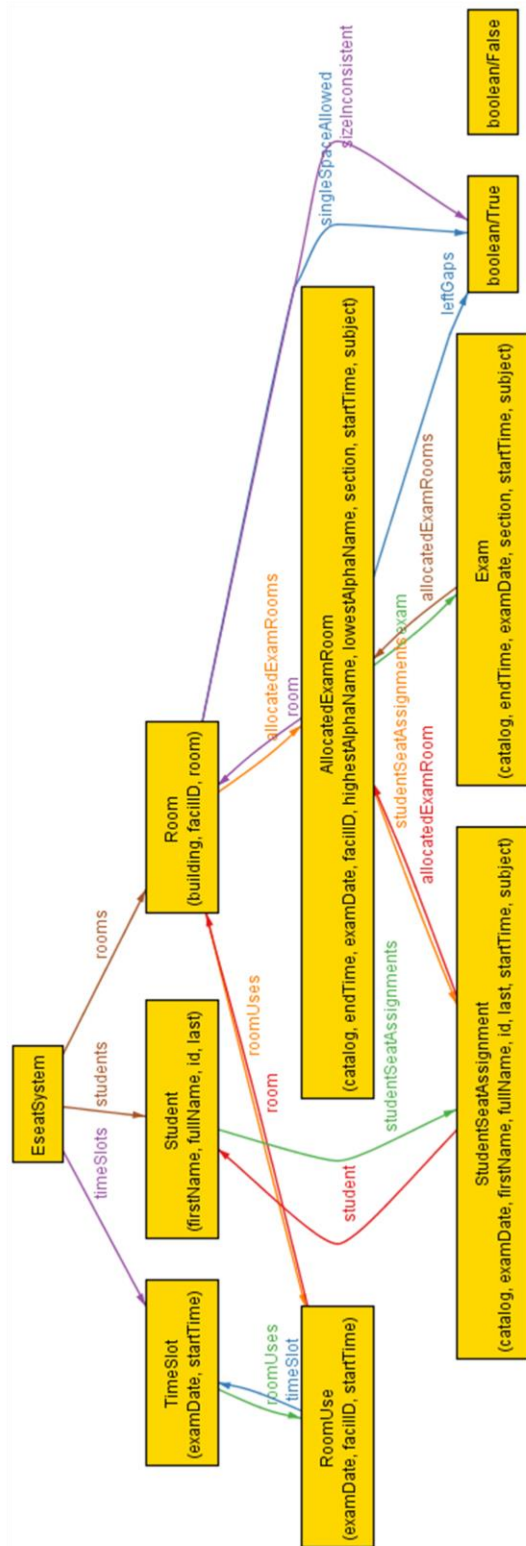


FIGURE 69. ALLOY EQUIVALENT OF THE ELECTRONIC SEATING SYSTEM

The generated CNF file has 3 million variables and 6 million clauses. Results show that the system is consistent. The instance obtained was projected over String and int and the visual representation of the electronic seating system that is being presented in Figure 69 (obtained via simulation).

8.2.4.1 Analyzing the e-Seat System

In this section, we discuss our approach to analyze the e-Seat system. Particularly, we present the basis of our analysis, then analyze the system and discuss the result. We are interested in answering the following questions:

> Q1. In the process of transformation from Umple to Alloy, are any of the structural constraints present in the Umple version not represented in the Alloy specification generated.

TABLE 19. ANALYSIS OF ASSOCIATIONS OF THE E-SEAT SYSTEM

	Student	StudentSeatAssignment	Exam	TimeSlot	Room	AllocatedExamRoom	RoomUse	**Umple**	**Alloy**
EseatSystem	U			U	U			T	T
Student		B						T	T
TimeSlot							B	T	T
Room						B	B	T	T
AllocatedExamRoom		B	B					T	T

Key: U – Unidirectional; B – Bidirectional; T - True

Answering Q1

In responding to Q1, there are multiple facets of analysis that are necessary. These include ensuring the following criteria as satisfied: 1) every class represented in Umple has a corresponding Alloy signature; 2) the association between classes (i.e., unidirectional or bidirectional); 3) the attributes (i.e., names and types); 4) the role names; and 5) the multiplicities in Umple are preserved in the generated Alloy specifications.

In Table 19, we present the analysis of associations of the e-Seat system by comparing modeling elements in Figure 68 and Figure 69. Recall that Figure 68 and Figure 69 are graphical representations of the textual version of models of the e-Seat system in Umple and Alloy respectively. Therefore, a careful examination of Table 19 is sufficient to certify that criteria 1 & 2 are satisfied since the results of comparison is the same for all mappings for Alloy and Umple.

Criteria 1

For criteria 1, the universe of classes (or signatures) along rows and columns are given as follows:

$$G = \{ \text{EseatSystem, Student, TimeSlot, Room, AllocatedExamRoom, StudentSeatAssignment, RoomUse, Exam} \}$$

Therefore, if G represents the set of classes in Umple (the source model) and K represents the set of signatures in Alloy. By analyzing the results obtained in Table 19 as well as critically examining the corresponding figures, it is obvious that:

$$G = K$$

Because for every class in Umple (see Figure 68), simulation shows a corresponding signature in Alloy (see Figure 69).

Criteria 2

For criteria 2, the associations between classes in Umple are preserved when translated into Alloy. The results in Table 16 shows this clearly since for every "T" in Umple, there is a corresponding "T" in Alloy.

To ensure criteria 3-5 are satisfied, we inspected both Umple source code and the generated code and present the comparative results in Table 20. We note that in mapping Umple to Alloy, we encountered some visualization challenges when Alloy's String type is being used. To resolve this, we modeled a signature – UString to represent String type in Alloy.

TABLE 20. COMPARISON BETWEEN UMPLE AND ALLOY OF E-SEAT SYSTEM

Class (or Signature) Name	Qualifier	Attributes			Associations			
		Name	Type (Umple)	Type (Alloy)	End Class (or Signature) Name	Umple End Multiplicities	Alloy Role Name	Alloy End Multiplicities
EseatSystem	S	debugMode	int	Int	Student	*	students	set
					Room	*	rooms	set
					TimeSlot	*	timeSlots	set
					Exam	*	exams	set
Room		room		UString	RoomUse	*	roomUses	set
		building		UString	AllocatedExamRoom	*	allocatedExamRooms	set
		facilId		UString				
		capacity	Integer	Int				
		sizeInconsistent	Boolean	Bool				
		singleSpaceAllowed	Boolean	Bool				
		capacityWithSpaces	Integer	Int				
Student		id		UString	Exam	*	exams	set
		last		UString	StudentSeatAssignment	*	studentsSeatAssignments	set
		firstName		UString				
		fullName		UString				
TimeSlot		examDate		UString	Exam	*	exams	set
		startTime		UString	RoomUse	*	roomUses	set
Exam		subject		UString	TimeSlot	0..1	timeSlot	lone
		catalog		UString	Student	*	students	set
		section		UString				
		endTime		UString				
		filledGaps	int	Int				
		examDate		UString				
		startTime		UString				
StudentSeatAssignment		id		UString	Student	1	student	one
		seatNumber	Integer	Int	AllocatedExamRoom	1	allocatedExamRoom	one
		last		UString				
		firstName		UString				
		subject		UString				
		catalog		UString				
		examDate		UString				
		startTime		UString				
		fullName		UString				
RoomUse		highestSeatAllocated	int	Int	Room	1	room	one
		examDate		UString	TimeSlot	1	timeSlot	one
		startTime		UString	AllocatedExamRoom	*	allocatedExamRooms	set

		facilID		UString				
AllocatedExamRoom		numStudentsForThisExam	Integer	Int	Room	1	room	one
		lowestAlphaName		UString	Exam	1	exam	one
		highestAlphaName		UString	StudentSeatAssignment	*	studentSeatAssignments	set
		lowestSeat	Integer	Int	RoomUse	0..1	roomUse	lone
		highestSeat	Integer	Int				
		leftGaps		Bool				
		facilId		UString				
		capacity	Integer	Int				
		capacityWithSpaces	Integer	Int				
		subject		UString				
		catalog		UString				
		section		Ustring				
		examDate		UString				
		startTime		UString				
		endTime		UString				

In Table 20, Umple's "Integer" and "int" types are mapped to "Int" in Alloy. Similarly, Umple defaults attributes without type to "String". Consequently, whenever the attribute type is omitted, it is represented internally as String type; thus, our translator maps such attributes to String type. We also note that in Umple, it is unnecessary to explicitly specify role names for association ends. However, based on the multiplicities, Umple automatically assign role names to association ends. For example, given an optional or one multiplicity for an end, a singular name is generated from the end class name.

Criteria 3-5

By critically examining Table 20, it can be inferred that these criteria are fulfilled. In particular, attribute names and types; role names; and multiplicities are preserved in the generated Alloy specifications.

In answering Q1, we observed that there are some structural constraints that were missing in the generated Alloy specifications. For example, the notion of "key" whose purpose is to represent unique attributes was lost in the transformation process. We observed this by specifying an assertion whose goal is to enforce uniqueness of some key attributes and check if the model currently fulfils the property.

For example, we check the following assertion with the goal of determining whether the model enforces that student identities are unique.

```
assert UniqueStudentID {
    no disj student1, student2 : Student | student1.id = student2.id
}
```

As a result of the execution, a counterexample (see Figure 70) was produced showing that the model is not constrained to enforce uniqueness of student identities. Particularly, Students 0-2 have the same identities.

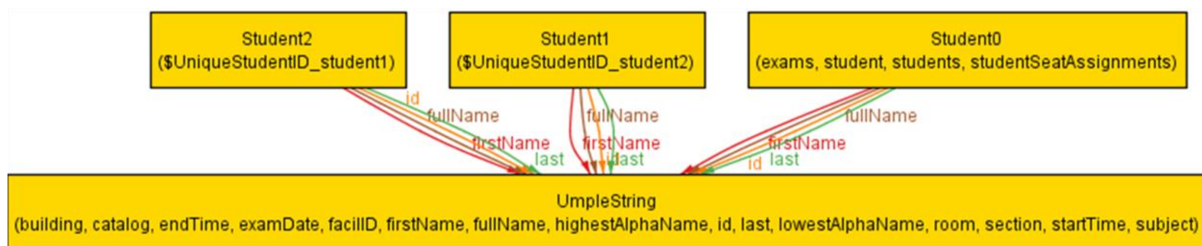


FIGURE 70. COUNTEREXAMPLE SHOWING MULTIPLE STUDENTS WITH THE SAME IDENTITIES

The generation of counterexample reveals a shortcoming of our translator, as it currently lacks the capability to generate structural constraints to enforce the semantics of key attributes. Therefore, we have added manually, the set of constraints required to enforce the semantics of key attributes for signatures: Student, Room, TimeSlot and Exam. We defer the automatic generation of these constraints to the future.

> Q2. Are there correctness properties that the system should fulfil? If there is any, does the system currently fulfil these properties?

By conventional wisdom, there are some correctness properties that the e-Seat system must fulfil besides those properties expressible as structural constraints. In the following, we present the list of those properties we consider important to certify that the system behaves correctly. Besides, we tackle the formulation using proof-by-contradiction. In particular, we specify a normal case as a predicate but negate result in search of counterexample that violate the specified constraint.

> **P1**: A student cannot be in two different rooms at the same time. Particularly, it is impossible for a student to be allocated into different rooms at the same time slot for the purpose of examination.

Alloy Assertion:

```
assert CheckP1 {
    no student : Student | p1[student]
}
pred p1[ student : Student ] {
    let t1 = student.allocatedExamRooms.roomUse.timeSlot,
        t2 = student.allocatedExamRooms.roomUse.timeSlot,
        r1 = t1.exams.allocatedExamRooms.room,
        r2 = t2.exams.allocatedExamRooms.room  | r2 = r1 => t1 != t2
}
```

P2: Any examination room cannot exceed its capacity. That is, on no occasion should an examination room be assigned a number of students that exceeds its official capacity. This can be considered an invariant that must be satisfied throughout the lifecycle of the system.

Alloy Assertion:

```
assert CheckP2 {
    no allocatedExamRoom : AllocatedExamRoom | p2[allocatedExamRoom]
}
pred p2[ r1 : AllocatedExamRoom ] {
    let x = #r1.studentSeatAssignments | x > r1.capacity
}
```

P3: For any allocated exam, there must be at least a student taking the examination. It is a waste of resources to allocate a room for an examination without student.

Alloy Assertion:

```
assert CheckP3 {
    all allocatedExamRoom : AllocatedExamRoom | p3[allocatedExamRoom]
}

pred p3[ r1 : AllocatedExamRoom ] {   let x = #r1.exam.students | x >= 1  }
```

Discussion of Analysis Results (from Q2)

We present the analysis results of the properties emanating from Q2. We will like to emphasize that to deal with issues related to scope and memory during analysis with Alloy (wherever necessary), we have simplified the model by reducing it to relevant signatures for the purpose of checking assertions. In particular, our approach to checking these assertions are based on the principle of compositional reasoning [109].

P1: We checked the assertion "checkP1" with varying scopes in search of possible cases of counterexample. The system could not find a counterexample indicating that the assertion may be valid.

P2: To check "P2", we identified that the relevant set of signatures for the analysis include: AllocatedExamRoom and StudentSeatAssignment. Therefore, we simplified the model by including these signatures and associated constraints (e.g., Bi-directionality rule). We deployed the Alloy analyzer to check the validity of the assertion.

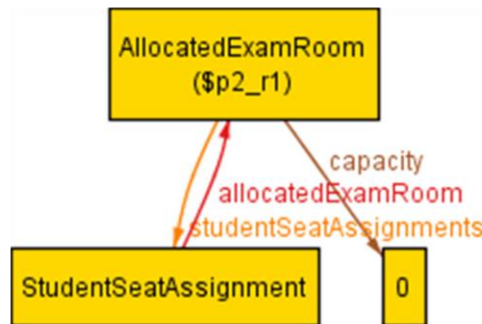


FIGURE 71. COUNTEREXAMPLE PRODUCED AS A CONSEQUENCE OF P2

The result shows that the current specification does not enforce this constraint. In fact, a counterexample (i.e., Figure 71) was generated when the assertion was checked indicating a violation of the expected behavior. The counterexample shows an "AllocatedExamRoom" with one "StudentSeatAssignment" but its capacity being "0".

P3: The relevant signatures to verify the satisfaction of P3 includes: "AllocatedExamRoom", "Exam" and "Student". A counterexample was generated when we verified the property – this indicates a violation of the property. The model allows AllocatedExamRoom without students since the multiplicity between Exam and Student is many (including zero).

8.3 Conclusion

In this chapter, we presented our approach to verify and validate our work. The results obtained are insightful. They answer some important questions that facilitate our understanding regarding some modeling concepts. In particular, we could have concluded wrongly if decisions were to be made by intuition (or *conventional wisdom*). For example, it is logically reasonable to conclude

that a more abstract approach is more efficient in comparison with a less abstract approach for an SSUA. An empirical study we conducted proved this assertion wrong.

Particularly, we discuss test-driven development methodology to ensure that for every Umple construct considered, expected code in target language is generated. To ensure that this step is replicable, we have presented a unified architecture describing the overall process. Similarly, we discussed our simulation strategy to avoid problems of over- and under-specification. We presented an architecture that can be followed in any similar project (or *task*).

We have validated various systems for static and dynamic properties and conducted relevant empirical studies. One of our studies evaluates performance of various alternative modeling solutions to and-cross transition. Another study asserts the equivalence of a hierarchical system to its non-hierarchical version. It also studies the impact of abstract (i.e., hierarchical and non-hierarchical) on performances during model checking. We also validated an SSUA with the goal of certifying that it is free from the issue of non-determinism.

9 Related Work

In this chapter, we present a discussion of related work encountered in the literature. We compared the various analysis engines found in the literature based on a set of parameters developed for this research. However, the major selection criterion for these analysis engines is based on their capabilities to facilitate full automated reasoning (i.e. no user-guidance).

Similarly, we compared varieties of implementation for designing and verifying software systems with “Our Work”. To achieve this, we developed a set of criteria to facilitate the comparison, explain the contributions, and summarize the solutions at a glance.

9.1 Literature on Analysis Tools

We focus this study on research to integrate formal methods with model-driven engineering. In particular, we are interested in research effort with the goal of analyzing UML state machines or class models with model checking, static analysis and theorem proving approaches.

Research Questions - The aim of this survey is to identify relevant literature that is closely related to our work. This will allow us relevant questions to guide our research. In view of this, we have designed the following set of questions (see Table 21) as the primary focus of this investigation.

TABLE 21. RESEARCH QUESTION FOR THE SURVEY

RQ-1	*What problem is being solved by the solution?*
RQ-2	*What aspect of software system is being addressed?*
RQ-3	*What approach is proposed or adopted to solving the problem?*
RQ-4	*What contribution was made to science by the work?*
RQ-5	*What are the strengths and limitations of the solution offered by the work?*
RQ-6	*How does the approach compare to our work?*
RQ-7	*What motivates the author(s) to study the problem?*

Data Sources - We consider various databases presented in Table 22 for the purpose of this literature search. To the best of our knowledge, the union of the resulting set of articles from these databases are an extensive list of relevant papers.

TABLE 22. SELECTED SOURCES FOR LITERATURE SEARCH

S/N	Database Name	URL
1	Scopus	https://www.scopus.com/
2	ACM Digital Library (ACM)	http://dl.acm.org/
3	Google Scholar	https://scholar.google.com/
4	Science Direct	http://www.sciencedirect.com/

Query Formulation - To formulate the search queries, we first categorized the terms into two broad groups with members of each group having similar attributes. We present these are given in Table 23.

TABLE 23. CATEGORIES OF KEYWORDS

Group A	Statechart, State Diagram, State Machine, Finite Automata, FSM, Class Diagram.
Group B	Model checking, Formal Method, Theorem Proving, Abstract Interpretation, Static Analysis.

For formulation purposes, within each group, there is logical-or (OR) operator and among groups there are logical-and (AND) operator. For example, our query is of the following form:

> (("state machine" OR "statechart" OR "finite automata" OR "FSM" OR "Class Diagram") AND ("model checking" OR "formal method" OR "Theorem proving" OR "abstract interpretation" OR "Static analysis"))

Inclusion Criteria - Similarly, to narrow the focus on specific papers that are closely related to our work, we have designed and presented a set of relevant inclusion criteria in Table 24. In furthering our refinement, a recursive exploration of various related works obtained based on our

inclusion criteria was conducted. By "recursive exploration", we mean the study of related works referenced by the article under discussion.

TABLE 24. INCLUSION CRITERIA FOR THIS SURVEY

IC-1	*Any solution addressing static or dynamic aspects of software systems.*
IC-2	*Any solution with rigorous mathematical basis.*
IC-3	*Any solution based on theorem proving, model checking, or abstract interpretation.*
IC-4	*Recursive exploration of referenced works obtained from IC-1, IC-2 and IC-3.*

Readers should note that whenever possible, we limit the search to:

Metadata: Title and Abstract.
Field: Computer Science, Mathematics and Engineering.
Document Type: Conference Paper, Article and Article in Press.

Exclusion Criteria - Our exclusion criteria are a negation of all the inclusion criteria including those highlighted in Table 24.

9.1.1 Static Analysis Tools

This section presents related works with focus on static analysis. These solutions compute systems properties without execution. Readers should note that solutions based on static analysis are numerous; we have limited the survey to those with clear relationship with our work.

9.1.1.1 Astrée

Astrée [110], [111], a commercial static program analyzer with the goal of ensuring programs written in C are free of run-time errors. It targets the analysis of programs with complex memory usage, but without dynamic memory allocation and recursion. These encompass safety-critical systems in medicine, avionics, and oceanography. It achieves a strong mathematical underpinning by relying on the notion of abstract interpretation. To be specific, the soundness of its abstraction depends on Galois connections [24], [112], [113]. However, whenever the best approximation cannot be produced, the concrete system is used for analysis as opposed to the approximated system. Its analysis is iterative, structural (by induction on the program syntax), inter-procedural,

context-sensitive for procedures, and precise for memory. Astrée's general-purpose abstractions are either non-relational or weakly relational with uniform interfaces. It has gained significant adoption for the certification of correctness of various avionics systems (see [114]–[116]).

9.1.1.2 FiniteSatUSE

FiniteSatUSE [117], [118] is a model-driven engineering tool developed for the purpose of analyzing UML class diagram to discover inconsistencies, contradictions, and finite satisfiability. The problem of finite satisfiability is a result of the presence of infinity-causing association cycles in the model. The approach implemented (i.e. FiniteSat algorithm) reduces the problem of finite satisfiability to a system of linear equations. The algorithm extends the linear inequalities-based method of Lenzerini and Nobili [119] to apply to diagrams with class hierarchy constraints. Its basic requirement for soundness and completeness is limited to cycles of class hierarchies without disjoint or complete constraints. Otherwise, the approach is sound but incomplete.

The similarity of this work to our work is based on the fact that the solution we chose (Alloy analyzer) addresses the problems solved by this solution. However, it differs significantly in some respects. First, while the approach computes properties from the model (static analysis), our work checks the model against a user-defined specification. Similarly, our work can solve modeling problems involving disjoint constraint since our solution relies on Alloy; unlike FiniteSATUSE. Alloy provides adequate syntax and semantics for specifying and analyzing the notion of disjoint elements.

9.1.2 Theorem Proving Approaches

In this section, we present a brief review of current solutions with focus on theorem provers (e.g. automated or user-guided). To be specific, the works adopts a theorem proving method for the analysis of software systems.

9.1.2.1 UML+Z

UML+Z [120]–[122] presented a template based approach to formally certify correctness of UML-based software systems (e.g. class diagrams, state diagrams, and object diagrams). The solution is based on model transformation from UML to Z and the analysis is assisted by Z/Eves [123]

theorem prover. A set of templates for some object-oriented design patterns (based on [83]) and meta-theorems were developed to facilitate automated analysis.

The key contribution of the work is a formal template language for the representation of structural patterns and meta-theorems. Our work intersects with this in some ways. For instance, both approaches are pattern-based and engineered based on model transformation techniques. However, our approach extends the list of structural patterns described in [83] with UML qualified pattern (plus some other patterns beyond associations and multiplicities) and the generation of patterns and constraints (equivalent of meta-theorems) are fully automatic. Another difference with our work is the approach to validate user requirements. UML+Z adopted a snapshot-based approach to validate the model; thus, analysts are required to be creative in designing a suitable set of test cases but our work inherits the bounded verification techniques of Alloy analyzer. While our approach is fully automatic for the purpose of analysis, UML+Z is semi-automatic. For example, the instantiation of and reasoning with templates are done manually by the analyst.

9.1.2.2 OhCircus + Z

In [124], an approach based on model transformation of UML class diagrams to OhCircus was proposed. OhCircus (an extension of Circus [125]) is a formal specification language that unifies Z [123] (a model-based language), CSP [126] (the process algebra), the refinement calculus of Morgan and object-oriented theories. The solution proposes concepts of class model to capture associations and global constraints (i.e. design patterns for class model). The solution relies on Z/Eves for the purpose of analysis since OhCircus has a refinement theory of Z language.

This approach is similar to our work because it proposes a pattern-based approach to UML class model analysis; though the solution supports the notion of global constraints and associations, the details of support provided is unclear. Another limitation of the approach compared to ours is that it provides no support for the analysis of behavior diagrams (e.g. state machine). Although, the level of user guidance required is not discussed, we believe that the analysis is not fully automatic.

9.1.2.3 ROZ

RoZ [127] is a tool for analyzing UML class diagrams with the formalism of Z. The tool automatically generates Z specifications (i.e. elementary operations and proof obligations) semantically equivalent to the input UML class diagrams annotated with relevant constraints

(expressed in Z-latex style). For the purpose of analysis, the solution relies on Z/Eves theorem prover. It is integrated within Rational Rose [128].

This approach is similar to our work because its goal is to automate the generation of formal specification of UML class models and associated constraints. However, our solution is better than this in a number of ways. Its integration within commercial tool (i.e. IBM Rational Rose) may hinder its adoption in academic context, because its setup will be expensive. The tool requires analysts to specify relevant constraint in Z. This may be a daunting exercise for novice users of Z. Above all, model validation is based on theorem provers like Z/Eves, despite the automatic generation of proof obligations, will demand user-guidance.

9.1.2.4 LAMBDES-DP

Zhu *et. al* [129], presented a tool known as *LAMBDES-DP*. It adopted a pattern-based approach to the formalization of UML diagrams, based on descriptive semantics. LAMBDES system translates UML diagrams into a first order logic language (developed in [130]) and invokes SPASS [131] (a theorem prover) to determine conformance to design patterns presented in [83]. It also allows analysts to specify conjectures to be proved to facilitate a wide range of logical analysis on models and consistency of models. This solution is similar to our work since it is pattern-based approach to the analysis of UML diagrams. However, while our work focuses on reasoning about correctness of models, this solution focuses on certifying the conformance of design patterns used in the model under analysis (MUA) with the standard design patterns implemented in LAMBDES-DP repository.

9.1.2.5 ABS + KeY

In [132], a comparison of a runtime assertion checkers is conducted with a theorem prover to certify correctness of concurrent and distributed systems using object-orientation, asynchronous method calls and futures [133]. The solution presented a variant of the ABS modeling language [134]. ABS is an executable modeling language designed to facilitate formal analysis and code generation for concurrent and distributed systems. Analysis of models relies on the KEY theorem prover [23] for its high degree of automation and support for ABS constructs. The main finding of the work reveals that type checking is similar to testing on the basis that it can only identify the presence of errors but not their absence; while formal verification is able to prove correctness of a

program with respect to any given specification. This is similar to our work from the perspective of model analysis; but its focus on concurrent and distributed systems makes it suitable for the analysis of dynamic aspects but ignoring static aspects. Our work cannot only analyze the dynamic aspects but also the static aspects. Besides, the goal of the work is a comparative study of type checking and formal verification.

9.1.2.6 ESC/Java

Extended Static checker (ESC) [135] is an experimental program checker with the goal of tracking inconsistencies (between code and its specifications) and run-time errors for software systems written in Java. The technology features a powerful verification-condition generator and automated theorem proving techniques. ESC gives static warnings of programming and design errors such as: null dereferences, array bound errors, type cast errors, race conditions, deadlocks, etc. It also allows programmers to annotate programs with design decisions that must be satisfied at run-time. The solution achieves greater scalability by adopting the principle of modular checking. For the verification of routines, it invokes Simplify [136], an automatic theorem prover.

However, it differs with the fact that we tackled the problem of guaranteeing program correctness from the modeling perspective; while their work addresses the same problem from the programming point of view. Additionally, our work uses compositional approach to manage scalability but ESC-Java uses a modular approach.

9.1.3 Model Checking Approaches

In this section, we will review several relevant model checking approaches and compare them to our work.

9.1.3.1 UML2Alloy

In [54], [137], [138], a prototype implementation tool (i.e., UML2Alloy) for the transformation of well-formed UML class and state diagrams to the equivalent Alloy constructs was presented. The tool adopted the Alloy analyzer for the analysis of both static and dynamic aspects of software systems. For the purpose of transformation, the model-driven architecture (i.e. MDA) methodology was adopted. UML2Alloy generates the meta-object facility (MOF) compliant meta-

model from the EBNF representation of Alloy. A set of transformation rules were defined to map constructs of UML to the corresponding Alloy constructs.

The similarities of our work with UML2Alloy include the (a) analysis of static and dynamic aspects; (b) focus on automated analysis; (c) model checking approach for software analysis; (d) adoption of Alloy analyzer for as a back-end analysis engine. However, our work differs from UML2Alloy in some other ways. UML2Alloy accepts only KMF-compliant XMI representation of UML models. Hence, users are prematurely committed to tools (e.g. ArgoUML [9], MagicDraw [139], etc.) generating KMF-compliant model representation. Umple facilitates not only the representation of models in code but also allow users to import EMF-compliant XMI representation of UML models. Many tools generates EMF-compliant models; thus facilitating integration with our work. Alloy is based on first-order logic, thus suitable for the representation of simple behavioral properties. We adopted nuXmv, a dedicated analysis tool for finite and infinite state systems verifications instead of Alloy. Given this approach, our work cannot only analyze simple behavioural properties but also complex ones. In addition, the verification of unbounded infinite domains (e.g. integers and float) is impossible with Alloy. nuXmv was engineered for this purpose.

9.1.3.2 TABU

In [140], a tool for the active behavior of UML (known as TABU) was proposed. TABU (without user-intervention) translates UML behavioral models (expressed as UML state and activity diagrams) to SMV for the purpose of formal analysis. The input to the tool is the XMI representation of the model under analysis, since XMI is largely tool-independent.

System properties are expressible in natural language. The natural language representation of the properties is fed into a property writing wizard (a pattern-based approach) so as to transform it to semantically equivalent temporal logic (i.e. LTL or CTL) properties. A violation of any of the properties produces a counterexample trace.

The solution handles activities (e.g. do-activities) as a special type of state diagram whose states are states of the activity and transitions often result in termination. The activation of an activity occurs whenever control takes the system to the host state and deactivated whenever control takes the system out of the host state.

In comparison with our work, the solution goes beyond our work in that it has a property-oriented wizard for automatic generation of properties; but the quality of properties being generated is not discussed. On the other hand, we consider our work to be stronger than this in two major aspects. Firstly, the analysis of infinite domains (e.g. integer or real type) in TABU is confined to a specified range. Secondly, the traditional approach (i.e. flattening) is used in TABU for the representation of UML systems. The flattening approach adopted is a huge limitation in terms of scalability; but the insight provided for handling activities gives us a clue as to how to handle activities expressible in Umple.

9.1.3.3 Zurowska

Zurowska and Dingel [141], [142] present a technique for analyzing state machines expressed in UML-RT symbolically. Its purpose is to bridge semantic gaps between the language of the model checker and the language of a model for UML state machines. The approach introduces treatments of action code (i.e. problem) as modular entities. It facilitates support for diverse action languages by cleanly separating the symbolic execution of a state machine from the symbolic execution of its action code. The separation is based on the representation of results of the symbolic execution of the action code as functions. State machine and their action code are translated to a representation called functional finite state machine (FFSMs). A symbolic execution tree is generated from the FFSMs which are used to perform reachability analysis, invariant checking, output analysis and test case generation.

There are similarities to our work based on its approach to representing state machines (i.e. symbolic approach). However, its reliance on Choco solver limits its capability to solve only trivial constraints as compared to solvers integrated by nuXmv.

9.1.3.4 JPF

Mehlitz [143] proposed an approach that systematically translates UML state charts (embedded with code in guards and actions) into Java code for formal analysis by Java Pathfinder (JPF) [144] analysis engine. Detailed knowledge on mapping (i.e. source to target) for each construct can be obtained from [143]. The resulting Java program is further compiled to its bytecode (i.e. encoding format for JPF) representation.

This approach is related to our work because it is based on symbolic model checking. However, it differs from our work in the choice of analysis engine. We observed that no report was given concerning their approach to the analysis of infinite domains (e.g. integer and real types). This makes the solution inferior to our work, because our goal is to enable analysis involving these domains by relying of the nuXmv analysis engine.

9.1.3.5 Remenska

Remenska *et. al* [145] proposed an approach based on the transformation of UML systems to process algebra for the purpose of formal analysis. Given a UML system with many diagrams, the authors proposed the use of sequence diagrams in describing the behavioral aspects (as opposed to state machines) of the system and activity diagrams to represent concurrency information necessary for deriving OS-level processes in a distributed system setup. The solution relies on back-end analysis engine called mCRL2. mCRL2 [146] is a model-checking infrastructure designed to facilitate reasoning about correctness of distributed and concurrent systems. The authors adopted mCRL2 because of its ability to deal with abstract data types and user-defined functions for data transformation.

This is related to our work because this approach is based on model checking. However, since its main focus is the domain of distributed system design, we considered it as a complement to our work. Specifically, the solution provides a deep insight to handling analysis involving varieties of modeling diagrams.

9.1.3.6 Bandera

Bandera [147] is an integrated collection of program analysis and transformation components for automatic extraction of safe, compact finite-state models from source code. It takes as input Java source code and generates a program model in the input languages of SMV [148], SPIN [149], and SAL [150].

The goal of Bandera is to overcome obstacles to finite-state verification of software by using a component-based tool architecture for model extraction based on the following criteria: (a) reuse of existing technologies; (b) automated support for the abstractions used by experienced model designers; and (c) synergistic integration with existing testing and debugging techniques. Bandera leverages on an internal representation (i.e. Jimple) provided by the Soot framework [151]. Hence,

a Java program is first translated to Jimple; refined (by slicing, abstraction, and translation) to produce Bandera internal representation (BIR) and then translated to the target input language (mainly those supported by Bandera). With Bandera, properties are expressible in variants of temporal logic languages.

This work is similar to our work based on the model checking approach to establishing correctness of software systems. However, it differs from our work in many ways. First, it only targets the analysis of dynamic aspects of systems from source code; while we tackled the analysis of both static and dynamic aspects of UML models. As a mechanism to deal with scalability, the solution proposes abstraction by abstract interpretation; we dealt with this issue by adopting a scalable encoding based on the principles of separation of concerns and attribute access. In particular, our approach adopted the principles of compositional reasoning to encode hierarchical state machines.

9.1.3.7 C2BP

C2BP [152] (a component of the SLAM toolkit [153]) is a tool that automatically computes the predicate abstraction of C programs, given a set of predicates (i.e. pure C Boolean expressions without function calls) describing analysts' desired temporal properties. It combines predicate abstraction, model checking, symbolic reasoning, and iterative refinement to adequately deal with scalability issues and certify correctness of temporal properties. The resulting Boolean program shares the same control-structure as the input program, but contains only variables related to the set of input predicates. This is fed into BEBOP [154], a model checker dedicated to the analysis of inter-procedural dataflow with the aid of binary decision diagrams (BDDs).

Our work differs from this since we are dealing with model analysis but not source code. However, the solution shares techniques like model checking and symbolic reasoning with our work.

9.1.3.8 Autofocus

AutoFocus [155], proposes model transformation of UMLsec [156] (a security extension of UML) to Promela for the purpose of analysis of security requirements by the Spin model checker. UMLsec integrates basic security requirements (e.g. secrecy, integrity, etc.), threat scenarios, common security concepts (e.g. tamper-resistant hardware), and cryptographic primitives in its language and formalizes these notions. AutoFocus automatically generates semantically equivalent LTL specifications of constraints expressed as stereotypes (e.g. secrecy, etc.) on the

model under analysis. The model and the generated requirement are fed into the Spin analysis engine for verification purposes. A violation of the requirement against the model gives an attack trace.

Our work differs from this in many forms. First, the reliance of this solution on an explicit-state model checker raises a big question about the scalability of the approach; our approach based on symbolic representation deals with this systematically. Similarly, properties are only expressible as LTL statements. LTL and CTL are not mutually exclusive in terms of expressive powers. The reliance of our work on nuXmv engine enables us to express properties in both notations.

9.1.3.9 SHADOWS

SHADOWS [157], a collection of technologies for the analysis of model-driven self-healing systems. It allows analysts to certify systems against functional, concurrency, and performance issues using a model checking technology known as Gear [158]. Gear is a game-based model checking tool with capabilities of handling full modal μ-calculus and CTL. SHADOWS proposed a graphical language for specifying temporal properties and counterexamples for the model under analysis as a means of enhancing usability of formal methods.

We argue that to prematurely commit analysts to graphical notations for property specifications introduces great usability issues. Another limiting factor is that the backend analysis engine, Gear, is not currently available as an open-source tool.

9.1.4 Summary on Analysis Tools

IN

Table 25, we present a summary of existing tools discussed and compared them with "Our Work". The set of criteria used for comparison was developed for the purpose of this study.

TABLE 25. COMPARISON OF SOFTWARE VERIFICATION TOOLS WITH "OUR WORK"

Formal Method	Open Source	Specification Language	Analysis Engines							Target Artefact		Software Aspects		
			Other	Alloy	nuXm	Magic	SAL	SM	Spin	Program	Model	Static	Dynamic	Tools
MC	+	OCL		+							+	+	+	UML2Alloy [138]
MC		LTL							+	+			+	FeaVer [159]
MC	+	OCL									+	+	+	UMLtoCSP [160]
MC		SLIC								+			+	SLAM [161]
SA	+										+	+		FiniteSATUSE [117]
MC	+									+			+	JPF [162]
MC		LTL, CTL					+	+	+	+			+	Bandera [163]
MC		CCL				+				+			+	ComFort [164]
SA	+									+		+	+	CBMC [165]
SA										+		+	+	Astrée [110]
TP			+								+	+	+	UML+Z[120]–[122]
TP			+								+	+		OhCircus+Z[124]
TP		Z-latex	+								+	+		ROZ [127]
TP			+								+	+	+	LAMBDES-DP [129]
TP			+								+		+	ABS+KEY [132]
MC		LTL, CTL						+			+		+	TABU [140]
MC		LTL	+							+			+	C2BP [152]
MC		LTL							+		+		+	AutoFocus [155]
MC			+								+		+	Shadows [157]
MC			+								+		+	Zurowska [110] [111]
TP		JML	+							+		+	+	ESC-Java [135]
MC			+								+	+	+	Remenska [145]
MC	+		+					+		+			+	SATABS [166]
MC					+						+		+	BSML2SMV [86]
MC	+	Alloy, CTL, LTL		+	+						+	+	+	*Umple (Our Work)

Key: Formal Method: SA, MC, TP = Static Analysis, Model Checking, Theorem PROVING; SPECIFICATION Language: OCL, LTL, SLIC, JML, Umple-CL = Object Constraint Language, Linear Temporal Logic, Specification Language for Interface Checking, Java Modeling Language; Umple Constraint Language; Other Notation: + = Supported; LoMC = Languages of Model Checker

9.2 Literature on And-Cross Transitions

In the literature, a number of papers (such as [60], [1], [94]) have focused on and-cross transitions in various dimensions for the purpose of modeling behavior of software systems. We review these works from three different angles. First, we are interested in relevant literature that supports and-cross transitions. Similarly, we are also interested in those articles where and-cross transitions were suggested to be removed. Finally, we are interested on various strategies that have been adopted to manage complexities of state machine systems.

9.2.1 Support for And-Cross Transitions

Although the notion of and-cross transitions has been excluded from the current version of OMG's specifications for UML state machines (version 2.5 [85]), there is relevant literature where the notion is discussed.

Faghih and Day [86] proposed a parameter-based algorithm to translate a family of big-step modeling languages (BSMLs) to SMV. The translation involves representing each simple state with a Boolean variable. A macro represents a composite state which is a logical disjunction of variables of its simple states. Thus, a composite state is active whenever at least one of its simple states is active. A Boolean variable is also defined for each event and transition of the SSUA. Other details on the transformation may be obtained from [86]. The method proposed allows the representation of andstate cross (i.e., and-cross) transitions. However, they report various inconsistencies that might result from its use as a modeling element most especially for arena-orthogonal option. Consequently, they disallowed unusual transition (a special case of and-crossing).

Harel and Kugler [1] define the semantics of and-cross transitions as a subset of the general semantics of statecharts. To simplify the definition, they categorize this form of transition as a *compound transition* (CT). By CT, they meant transitions with multiple segments. The family of transitions in this category include: *fork*, *join* and *and-cross transitions*. In our work, a *fork* is synonymous to a transition into an orthogonal composite state; a high-level transition has a semantics synonymous to a *join*; and an and-*cross* transition implies both and-cross and unusual transitions (see Faghih and Day [86] also). According to their work, executing an and-cross

transition will result in a change of configuration whereby all descendants of the source state of the transitions are exited and the proper ancestors and descendants of its destination state are re-entered. In this paper, we followed this semantics closely (as outlined in Section 6.4) to formally specify a set of non-conflicting transitions to enable and disable relevant states and sub-state machines of an SSUA. However, while their work provides an informal semantics for the notion of and-cross transition, our work formally defines how the semantics can be realized for analysis purposes.

Eshuis and Van Gorp [167] proposed a method to transform an activity diagram to its state machine equivalent. However, they observed a semantic gap between an activity diagram with cross-synchronization and its state machine equivalent. In particular, the transformation maps an activity diagram with cross-synchronization to a state machine with and-cross transition. Therefore, they described semantics of and-cross transitions and analyzed its implication on the semantics of the resulting state machine diagram. The semantics of and-cross transitions results in an undesired configuration as opposed to the intended semantics in activity diagram modeling. To resolve this difference, the authors propose to adapt the resulting state machine semantics (i.e., with and-cross transitions) by generating relevant internal events and constructing necessary guard statements so that the desired semantics is realized. An important difference between our work and theirs lies in the purpose of the research. In particular, their goal is to transform activity diagrams to their state machines equivalent without jeopardizing the semantics. Similarly, they identify semantic differences between activity diagrams and state machines. Instead, we demonstrated how semantics of and-cross transitions can be realized using alternative approaches within the context of state machine modeling. In particular, we neither focus on the transformation between activity diagram and state machine nor focus on the preservation of cross-synchronization semantics of activity diagram in state machine context.

In [60], we discuss and-cross transitions in the context of the detection of non-determinism in state diagrams. We represent state machines in Umple, and we automatically transform them to SMV. We then rely on our match-making algorithm to systematically compute a minimal set of potentially conflicting pairs of transitions. We formally specify invariants based on these transitions, and analyze the resulting system to identify false positives (i.e. conflict-free pairs) and generate execution traces leading to nondeterminism (i.e. conflicting pairs). In [60], we also take

into account two categories of non-determinism, namely: same-source and region-cross transitions. In this work, we enhance our algorithm to report non-deterministic cases resulting from multiple and-cross transitions in the same enclosing state. We also discuss the usefulness of and-cross transitions.

9.2.2 On the rejection of And-Cross Transitions

We acknowledge that and-cross transitions can be complex to manage. Thus, we explore the literature as well as modeling tools to study various arguments supporting its rejection.

According to Mueller [93], and-cross transitions (which they refer to as *transitions across region borders*) should be disallowed while composing state machines from regions since their underlying complexities they perceive to be large. The argument they present is that allowing at least two and-cross transitions with the same trigger from different regions of the same enclosing state may lead to inconsistency. Thus, they conclude that to manage the complexities, transitions across region borders must be disallowed from state machine's constructs. We respect the view and thus, do not allow such cases in our work. In particular, we consider these cases as non-deterministic. Therefore, as described earlier, we enhanced our algorithm to label such pairs as potentially conflicting. These can further be analyzed dynamically so that *false positives* are preserved but *true positives* are avoided.

To the best of our knowledge, the most widely used modeling tools such as ArgoUML [168], Magic Draw [169], Visual Paradigm [170], UModel [171], Astah [172], MetaUML [173] and BOUML [174] do not support and-cross transitions. Even though their documentation does not disclose the rationale behind this choice, we assume that it is due to the alignment with the OMG's specifications for UML state machines (version 2.5 [85]) that no longer supports and-cross transitions.

9.2.3 Managing complexities of state diagrams

In the literature, various research efforts have focused on methods (e.g., [67]) to improve formalization of state machine systems or manage its underlying complexities (e.g.,

Badreddin et al. [67], proposed an enhanced approach to formalize composite state machines for the purposes of code generation and minimizing the state-space explosion problem. The authors proposed an algorithm that systematically transforms state machine systems to an internal representation without jeopardizing the structure, as opposed to traditional flattening where the structure is lost during transformation. This internal representation is then used for code generation. Our work is largely influenced by their method. However, there are clear differences between ours and theirs. First, the notion of and-cross transitions (i.e., a major focus of this work) was neither formalized nor discussed. Similarly, while we focus on the generation of SMV representation of the state machine systems, their work focuses on the generation of Java.

Zhang and Holzl [175] presented a set of metrics for the purpose of understanding complexity arising from the notion of *non-locality*. By non-locality, they meant the way information for the current behavior of the system influences modeling elements in the global state-space. Their goal was to provide an understanding of the complexity underlying UML state machines. There are close similarities with our work. A formal description of cases resulting in non-determinism was described as we did. Additionally, cross region cases and their complexity were discussed. However, no particular emphasis was made with respect to and-cross transitions in their work. Similarly, the discovery of non-determinism is done statically in their work; while we further the step by discovering these cases dynamically. While they provide metrics to compute complexity of state machines, we propose a method to manage the underlying complexity.

9.3 Literature on Discovering Nondeterminism

In the literature, there is a variety of research about manage non-determinism. For ease of discussion, we have categorized these into three aspects. These include: algorithmic solutions (e.g., [176]); adoption of SPIN model checker (e.g., [177]–[179]); tool-based solutions.

9.3.1 Algorithmic Solutions

Zulkernine and Seviora [176] propose an assume-guarantee approach to deal with the state-space explosion problem in analyzing dynamic aspect of software specification for failure detection purposes. A particular fault addressed by the authors is non-determinism.

The approach adopts a communicating finite state machine formalism for the description of the SSUA. For implementation purposes, a hybrid of UML and SDL (i.e., System Description Language) was adopted. The failure detector proposed is a collection of algorithms each of which is responsible for task at each step of analysis. To analyze SSUA, the fault detector is attached to either a partial or complete representation of the SSUA. Fault detection involves observing the system at every stable configuration of the SSUA given the external I/O. First, the fault detector generates assumptions and guarantees of each process with respect to the input, output and stable states of the SSUA. A failure is reported whenever: a) assumptions or guarantees cannot be generated; b) assumptions generated cannot be discharged. Otherwise, the cycle is repeated for the next global configuration until the system is completely explored.

9.3.2 Solutions adopting SPIN Nondeterministic Selector

In [177]–[179], cases of nondeterminism in state machines are managed by engaging non-deterministic selections of executable paths in the SPIN model checker [180]. For example, Latella et al. [177] specifies the SSUA in UML statechart then systematically translates the system to PROMELA, an input language of SPIN for execution purposes. Chikatoshi [178] proposes to encode SSUAs using Simulink stateflow then systematically translate the system to PROMELA for nondeterministic selection of execution path. In the same vein, Arcaini et al. [179] combines runtime verification and model-based testing to manage the presence of nondeterminism in an SSUA. The authors delegated the capability of SPIN to manage path selection after successful translation of the SSUA (Java program) to PROMELA.

9.3.3 Tool-Based Solutions

Other approaches to analyze state diagrams for nondeterminism include BSML [86], RSML [181], STATEMATE [182], SCR [183] and SMUML[34].

BSML [86] detects conflicting transitions based on Esmaeisabzali's semantics framework with model checking. Transitions $t_{i,j}$ can execute simultaneously whenever they are consistent with small-step consistency and pre-emption structural aspects. These are: any two simple transitions originating from a source; simple transitions with transitions in its source parts; Or-state cross transitions with the same source; and-cross transitions within a scope. Our work differs by flagging

false positives where necessary. Similarly, a sub-state machine is handled as a black box, hence minimizing the total number of comparisons.

RSML [181] discovers nondeterminism by statically analyzing the SUD based on functional composition, whose key concepts include union, serial and parallel composition. The union approach constrains the system so no two transitions of a simple state execute simultaneously; serial approach constrains SUD such that a transition generating an event executes before the transition consuming the event executes; the parallel approach focuses on simultaneous execution of pair wise orthogonal transitions. Guard conditions are evaluated using and/or tables. The limitations of this include: inability to track and-cross cases of nondeterminism; state-space explosion for complex conditions and infinite state domains; and inability to flag false positives.

SMUML [34] facilitates the discovery of nondeterminism but the detail of its approach is not given. SCR [183] assumes that the SSUA is deterministic, hence does not discover nor manage nondeterminism. STATEMATE [182] adopts case statement of the SMV [184] to resolve nondeterminism.

9.4 Summary on And-Cross and Nondeterminism

For our summary on related works we overviewed on tool support, interested readers should consult Section 9.1.4.

In Table 26, we summarize related work we reviewed on the notion of and-cross transitions and the discovery of nondeterminism. We compare the solutions with our work to allow readers identify the contributions. To simplify comparative study, we adopted the following criteria:

And-Cross Support -This implies whether the solution offered by the tool allow the representation or analysis of and-cross transitions.

Static Discovery – Nondeterminism – By “static discovery”, we mean analysis of nondeterministic cases are discovered and managed without executing the system. A limitation of tools in this category will be inability to flag false positives.

Dynamic Discovery – Nondeterminism – By "dynamic discovery", we mean analysis nondeterministic cases are discovered by executing the SSUA. The strength of tools in this category will be able to flag false positives.

Support for Unbounded Domains – This involves the analysis of SSUAs with unbounded integer and real types for the purpose of discovering nondeterminism.

Reliance on SPIN for Nondeterminism – This refers to tools relying on SPIN model checking capability to handle nondeterministic cases. Tools in this category dynamically execute the SSUA thus, having the capability of flagging false positives.

Reliance on NuXMV for Nondeterminism – This refers to tools relying on NuXMV model checking capability to discover nondeterministic cases. Tools in this category dynamically execute the SSUA thus, having the capability of flagging false positives and analyzing unbounded domains.

The outcome of the evaluation is described as keys for Table 26, depending on what criteria being facilitated and how well the criteria is satisfied.

TABLE 26. SUMMARY OF TOOL ON SUPPORT FOR AND-CROSS TRANSITIONS AND NONDETERMINISM

	Pure And-Cross Support	Unusual Transition	Static Discovery - Nondeterminism	Dynamic Discovery - Nondeterminism	Support for Unbounded Domains	Degree of Automation	Reliance on SPIN for Nondeterminism	Reliance on NuXMV for Nondeterminism
BSML2SMV [86]	S	N	S	N	N	P	N	N
STATEMATE [182]	U	U	N	S	N	N	N	N
RSML [181]	U	U	S	N	N	N	N	N
SCR [183]	U	U	N	N	N	N	N	N
SMUML [34]	U	U	N	N	N	N	N	N
Latella et al. [177]	U	U	N	S	N	P	S	N
Chikatoshi [178]	U	U	N	S	N	P	S	N
Arcaini et al. [179]	U	U	N	S	N	P	S	N
Zulkernine + Seviora [176]	U	U	S	N	N	P	N	N
Mueller [93]	N	N	U	U	N	U	U	N
*Our Work	S	S	N	S	S	F	N	S

KEY: P – PARTIALLY SUPPORTED; F – FULLY SUPPORTED; S – SUPPORTED; N – NOT APPLICABLE; U - UNKNOWN

10 Conclusion and Future Work

In this thesis, we present our approach to integrate formal methods with Umple, a variant of UML for the specification of static and dynamic aspects of software systems. We focus on state machines as dynamic aspect and class model for static aspect. Besides, we have provided a tool that automate the transformation from Umple to the choice analysis languages (i.e., Alloy – Static Aspect and SMV – Dynamic Aspect).

In this Chapter, we summarize answers to our top-level research question; scientific contributions of our research; and suggest various directions to improve this research and facilitate its adoption.

10.1 Summary with Respect to Research Question

The following indicate how we have answered our original research questions.

> **RQ 1 -** How can we improve the adoption of formal methods, both in industry and in teaching, by overcoming some of the complexity, while maintaining analysis capabilities and enhancing scalability?
>
> **Answering RQ1**
>
> a.) **Improving adoption of formal methods** - In **Chapter 1**, we illustrate diagrammatically (see Figure 1) the various usages of formal methods for the process of developing software systems. Consequently, maintaining consistency between model and its source code, as well as model and its formal specification(s) is the challenge we address. In **Chapter 2**, we adopted Umple, Alloy and SMV to prove the concept and improve adoption of formal methods. First, Umple allows us to maintain consistency between model and the code while Alloy and NuSMV (or nuXmv) provide back-end analysis engines to facilitate formal analysis of static and dynamic aspects of software systems respectively. In **Chapter 3**, we adopted meta-modeling, templating and model

transformation as means to engineer a tool to automate the transformations (see **Chapters 4** & **6**) from model to formal specifications. This allows us develop a tool that allow modelers to maintain consistency between model and its formal specifications; though in a unidirectional manner.

b.) **Overcoming Underlying Complexities** – The process of formalizing software abstractions is a daunting exercise. Therefore, there is a huge amount of complexity underlying it. We have addressed some of these complexities in this thesis.

i. And-Cross Transitions - The notion of and-cross transition has been removed from UML (v 2.5) due its underlying complexity, limited use-in-practice; availability of alternative solutions among others. In **Chapter 5**, we address the formalization of state machines even in the presence of and-cross transitions as we discover it provides relevant abstraction needed for modeling real-world problems. In **Chapter 8**, we presented a real-world automotive example where and-cross is a relevant solution and explored various alternative solutions that can replace and-cross transition. Results show that and-cross transition reduces the number of modeling elements to be analyzed.

ii. Discovering Nondeterminism and Unreachability Issues - Nondeterminism has been a relevant tool for requirement engineers to specify a system. But it is undesirable in the actual implementation of the system. Therefore, it must not be shipped in the actual implementation. While existing solutions lack the capability to differentiate between false and true positives (i.e., static discovery), discovering non-determinism in a large system is impractical manually. Thus, we focus **Chapter 6** on quality assurance to discuss our methods and algorithms to analyze and discover these issues in SSUAs. We validated our approach by deploying our algorithms on real-world case study and report a bug that might have been shipped into implementation unnoticed in **Chapter 8**.

c.) Maintaining Analysis Capabilities and Scalability – In **Chapter 8**, we deployed our tool to translate static and dynamic aspects of software systems for the purpose of analysis. Particularly, we deployed our tool on moderately complex systems to generate formal specifications and analyse the systems by the target model checkers. Results have shown that our approach can uncover bugs both in industrial and academic contexts.

10.2 Summary on Scientific Contributions

We present a summary of scientific contributions of this work to software engineering and science at large.

a.) A unified approach to formally specify, analyze and generate a complete system from the same model.

Existing approaches for integrating formal methods with model-driven engineering do not rely on one model for formal analysis and the generation of complete systems. Due to this, software systems are susceptible to errors arising from inconsistencies between a model and its code as well as model and its formal specification. Hence, in this thesis we address this challenge by presenting an approach that facilitates formal analysis by having the developer model the complete system in a simple modeling language (in our case, Umple), and systematically generating formal methods from this for verification purposes, while at the same time generating the final system from the same model, thus preventing the need for re-implementation.

b.) A novel approach to encode state machine systems even in the presence of and-cross transitions for symbolic model verification (see [59]**).**

In this thesis, we present a novel approach to formally encode state machine models (including those expressed in Umple) for symbolic verification. It extends Badreddin et. al's [55], [67] work in various ways to allow symbolic verification of state machine diagrams (both simple and hierarchical). We enable states (i.e., simple and composite) and disable sub-state machines

(parallel and non-parallel) by means of transitions. The novelty of this thesis includes: a) the formulation of sound methods to compute a set of non-conflicting transitions to enable and disable states and sub-state machines even in the presence of and-cross transitions; and b) an explorative study of how the solution offered by Badreddin et. al's [55], [67] work can be adapted for symbolic verification.

c.) A fully automated approach to certify an SSUA to be free of non-determinism even in the presence of unbounded variables and multiple and-cross transitions in the same enclosing state (see [60]**).**

In this thesis, we present a fully automated technique to detect non-determinism in state diagrams. Although nondeterminism is a tool often adopted by requirement engineers for specification of an SUD, it is normally undesirable in actual implementation. Discovering nondeterminism manually is infeasible for industrial-sized systems. Solutions in the literature [34], [86], [181]–[183] lack the capability to analyze infinite-state systems. We leverage the nuXmv model checker to analyze unbounded domains and implement an algorithm that systematically computes a minimal set of comparable transitions for the SUD yet eliminates false positives by model checking. The novelty of this work includes: a) an algorithm to compute a set of potentially conflicting transitions; and b) a method to eliminate false positives from the resulting set even in the presence of unbounded integer and real types.

d.) A comparative study of and-cross transitions and various alternative approaches that can substitute and-crossing for modeling state machine diagrams (see [49], [61]**).**

Abstraction is known as an effective mechanism for the description of complex systems. Abstraction with adequate encoding can also help with the state-space explosion problem by reducing the total number of elements to be analyzed. A tactic that can significantly reduce the number of transitions required for modeling certain systems is the use of and-cross transition. And-cross

transitions are those whose source and destination states are in distinct parallel regions of the same enclosing orthogonal state. The removal of and-cross notation from the Object Management Group's (OMG) specifications for the Unified Modeling Language (UML), version 2.5 [85] warranted a comparative study of various alternative solutions that may replace and-cross transition. Hence, the novelty includes: a) a variety of case studies to demonstrate usefulness of and-cross transitions; and b) a performance evaluation of various alternative approaches and and-cross transition using parameters like memory usage, execution times, and the number of required BDD nodes.

e.) An empirical study of the impact of abstraction on some performance parameters (e.g., execution time, memory usage and the number of Binary Decision Diagrams - BDDs).

By conventional wisdom, system analysts may erroneously conclude that abstraction implies better performance. To assert this, we investigate abstractions offered by and-cross transition and state hierarchy. We designed an experiment to compare hierarchical and flattened versions of an SSUA (i.e., Xholon watch) and verify some requirements. We designed another experiment to compare various versions of a system with a focus on abstraction by and-cross transition and verify some requirements. We recorded execution times, memory usage and number of BDDs in both cases. For model checking tasks, we observed that abstraction is inversely proportional to resource usages for analysis purposes. However, for non-model checking tasks (e.g., computing the set of potentially conflicting transitions) resource usage is directly proportional to the degree of abstraction.

f.) Transformation tools from Umple (and hence from UML) to SMV and Alloy.

We have created a tool that automatically generates formal representation of static class models and dynamic state machine models of software systems and their requirements. It seems reasonable to conclude that this should facilitate a

significant reduction in workload required for the specification of formal models of systems. A challenge that has required considerable effort is to ensure the semantics of the source language (i.e. Umple) is preserved in the target language (e.g. Alloy or SMV) through sound model transformations. We have formulated meta-models and an adequate set of constraints for Alloy and SMV languages to facilitate a semantics-preserving transformation. Similarly, we have designed a set of templates (expressed in Umple) to facilitate the generation of the generated formal language code. This has been evaluated through our case studies.

10.3 Future Directions

A key long-term goal is to develop a tool that will facilitate the teaching of formal methods as an integral part of undergraduate software engineering education. We also want to improve its adoption in industry and minimizing failure rates of software projects. Although, we have developed a tool to automate the process of generating formal specifications of software systems and validated it on various case studies with promising results, the tool will need to be continuously improved.

Consequently, we recommend improvements of the following kinds soon to facilitate its adoption and improve its analysis capabilities:

Validation – Future work should apply our approach on a variety of software systems, spanning diverse domains to uncover and address relevant issues necessary to formally analyze real-world system in these domains. Doing this will further demonstrate how scalable our approach is and identify any gap that must be addressed.

Improving Our SMV Generator – An important limitation of this work is its inability to automatically generate formal specifications corresponding to action code in Umple models (embedded code in native programming languages). This is a consequence of the fact that Umple does not currently parse action code. Therefore, there is possibility of capturing these aspects by

our code generator. At this time, action code is added manually to the SMV code generated. We consider this error-prone and not scalable. Future work should formalize the representation of action codes in Umple and automate the process of generating its equivalent formal specifications for the purpose of analysis.

Maintaining Consistency between Formal Specification and Umple Models - At the moment, our approach maintains consistency between model and code in a unidirectional manner. However, it might be more effective to ensure model and its formal specification can be consistent in a bidirectional manner. In particular, we envision a tool that automate forward and backward transformations between model and its formal specifications. This would be analogous to Umple's ability to either edit the diagram or the Umple textual code.

Specification of Properties – We acknowledge that requesting developers to specify properties that the SUD must conform to is a limitation. The auto-generation of a comprehensive set of domain-dependent properties is not realistic as properties vary from one domain to another. Consequently, we have defined some domain-independent properties to be autogenerated from the SUD. These include nondeterminism and reachability analysis of state machine models. Other properties of interest include cycle detection, consistency and completeness checking, etc. We defer the auto-detection of these properties to the future. Some properties, such as simply cycle detection do not require formal methods generation and are already handled natively by the Umple compiler.

Analysis of Timing Properties – In real-world applications (e.g., automotive and healthcare), systems are often required to fulfill some timing requirements. To ensure systems satisfy these requirements, it would be necessary to improve our analysis capabilities to deal with timing.

Exploring Other Analysis Strategies – Having understood the limitations of analysis engines currently adopted for this research, we consider it important to explore more tactics that are sound and complete. This will help address different challenges arising from different domains.

Backward Transformation to Umple – To facilitate adoption of our work, particularly for teaching purposes, it is imperative to ease the readability of model checking results. In particular, the lower-level model checking results should be presented in a more abstract manner such as being, translated back into Umple, so they can be presented in Umple user interfaces, hiding the formal methods technology. This is necessary because this category of users (i.e., students) may lack expertise to understand and interpret results. Due to the underlying complexities in defining the rules and visualizing the results, we defer this work to the future.

References

[1] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)," *Integr. Softw. Specif. Tech. Appl. Eng.*, vol. 3147, pp. 325–354, 2004.

[2] J. Schauffele, T. Zurawka, and S. Germany, *Automotive Software Engineering*. 2005.

[3] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer (Long. Beach. Calif).*, vol. 26, no. 7, pp. 18–41, 1993.

[4] P. Codd, "Top 10 Software Failures Of 2011." [Online]. Available: http://www.businesscomputingworld.co.uk/top-10-software-failures-of-2011/. [Accessed: 02-Apr-2015].

[5] C. Baier and J.-P. Katoen, *Principles Of Model Checking*. Cambridge, MA: MIT Press, 2008.

[6] D. C. Schmidt, "Model-Driven Engineering," *IEEE Comput. Soc.*, pp. 25–31, 2006.

[7] J. Rambaugh, I. Jacobson, and G. Booch, *Advanced Praise for The Unified Modeling Language Reference Manual , Second Edition*, 2nd ed. Toronto: Addison-Wesley, 2004.

[8] OMG, "OMG Unified Modeling Language: Version 2.5," *Object Management Group*, 2013. [Online]. Available: http://www.omg.org/spec/UML/2.5/PDF.

[9] A. Ramirez *et al.*, "ArgoUML User Manual A tutorial and reference description," *ArgoUML Community*, 2011. [Online]. Available: https://people.cs.pitt.edu/~chang/1635/ArgoUMLman.pdf.

[10] J. Offutt, "Programmers Ain't Mathematicians, and Neither Are Testers," *Form. Methods Softw. Eng. 10th*, p. 5256, 2008.

[11] M. Ouimet and K. Lundqvist, "Formal software verification: Model checking and theorem proving," in *Technical Report - ESL-TIK-00214, Massachusetts Institute of Technology (MIT)*, 2007, pp. 1–12.

[12] T. Coe, T. Mathisen, C. Moler, and V. Pratt, "Computational aspects of the Pentium affair," *IEEE Comput. Sci. Eng.*, vol. 2, no. 1, pp. 18–31, 1995.

[13] P. Baumgartner, "Model Evolution-Based Theorem Proving," *IEEE Intell. Syst.*, vol. 29, no. 1, pp. 4–10, 2014.

[14] S. Schiffel and M. Thielscher, "Automated theorem proving for general game playing," in *IJCAI International Joint Conference on Artificial Intelligence*, 2009, pp. 911–916.

[15] C. E. Brown, "Reducing higher-order theorem proving to a sequence of SAT problems," in *Journal of Automated Reasoning*, 2013, vol. 51, no. 1, pp. 57–77.

[16] M. J. C. Gordon, "HOL: A Proof Generating System for Higher-Order Logic," in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Boston, MA: Springer US, 1988, pp. 73–128.

[17] Y. Bertot and P. Castran, *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[18] M. Kaufmann and J. S. Moore, "An ACL2 Tutorial," in *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008.*, O. A. Mohamed, C. Muñoz, and S. Tahar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 17–21.

[19] S. Owre, N. Shankar, M. Park, S. Owre, M. Park, and N. Shankar, "A Tutorial on Using PVS for Hardware Verification," in *Theorem Provers in Circuit Design: Theory, Practice and Experience Second International Conference*, 1994, no. February, pp. 258--279.

[20] P. Lammich and A. Lochbihler, "The Isabelle Collections Framework," in *Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, M. Kaufmann and L. C. Paulson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 339–354.

[21] A. Bove, P. Dybjer, and U. Norell, "A Brief Overview of Agda -- A Functional Language with Dependent Types," in *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78.

[22] H. Li, "Automated Theorem Proving," in *Geometric Algebra with Applications in Science and Engineering*, E. B. Corrochano and G. Sobczyk, Eds. Boston, MA: Birkh{ä}user Boston, 2001, pp. 110–119.

[23] W. Ahrendt *et al.*, "The KeY platform for verification and analysis of Java programs," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8471, pp. 55–71.

[24] P. Cousot and R. Cousot, "Abstract interpretation: A unified latice model," in *POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[25] B. Blanchet *et al.*, "A Static Analyzer for Large Safety-Critical Software," in *PLDI '03 Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2007, pp. 196–207.

[26] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo, "Automatic inference of necessary preconditions," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7737 LNCS, pp. 128–148, 2013.

[27] A. Miné, "The octagon abstract domain," *Higher-Order Symb. Comput.*, vol. 19, pp. 31–100, 2006.

[28] P. M. Benoy, "Polyhedral Domains for Abstract Interpretation in Logic Programming," University of Kent, 2002.

[29] B. Y. A. L. Bessey and K. Block, "A few Billion Lines of code Later using static Analysis to find Bugs in the Real World," 2000.

[30] Ö. Eğecioğlu and T. F. Gonzalez, "A Computationally Intractable Problem on Simplicial Complexes," *Comput. Geom.*, vol. 6, no. 2, pp. 85–98, 1996.

[31] S. P. Miller, M. W. Whalen, and D. D. Cofer, “Software model checking takes off,” *Commun. ACM*, vol. 53, no. 2, pp. 58–64, 2010.

[32] A. Cimatti, E. Clarke, and E. Giunchiglia, “Nusmv 2: An opensource tool for symbolic model checking,” in *International Conference on Computer Aided VerificationComputer Aided Verification*, 2002, vol. 2404, pp. 359–364.

[33] R. Eshuis, D. N. Jansen, and R. Wieringa, “Requirements-Level Semantics and Model Checking of Object-Oriented Statecharts,” *Requir. Eng.*, vol. 7, pp. 243–263, 2002.

[34] J. Dubrovin and T. Junttila, “Symbolic model checking of hierarchical UML state machines,” in *8th International Conference on Application of Concurrency to System Design*, 2008, pp. 108–117.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Progress on the state explosion problem in model checking,” in *Informatics*, 2001, pp. 176–194.

[36] L. Z. Markosian and J. J. Penix, “Program Model Checking : A Practitioner ’ s Guide,” no. March, 2008.

[37] E. Clarke and A. Emerson, “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic,” *Log. Programs*, pp. 52–71, 1981.

[38] A. Pnueli, “The temporal logic of programs,” in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.

[39] D. Jackson, “Alloy: a lightweight object modelling notation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 256–290, 2002.

[40] R. E. Bryant and C. Meinel, “Ordered Binary Decision Diagrams,” *Log. Synth. Verif.*, vol. 654, no. July, pp. 285–307, 2000.

[41] R. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6174 LNCS, pp. 24–40, 2010.

[42] A. Mishchenko, N. Een, R. Brayton, and S. Jang, “Magic: An industrial-strength logic optimization, technology mapping, and formal verification tool,” in *Proc. IWLS’10*, 2010, pp. 5–8.

[43] W. Chan *et al.*, “Model Checking Large Software Specifications,” *IEEE Trans. Softw. Eng.*, vol. 24, no. 7, pp. 498–520, 1998.

[44] J. Klein, “Model-Driven Engineering : Automatic Code Generation and Beyond,” 2015.

[45] R. Cavada *et al.*, “The NUXMV Symbolic Model Checker,” in *26th International Conference on Computer Aided Verification*, 2014, pp. 334–342.

[46] L. De Moura and N. Bjørner, “Satisfiability modulo theories: An appetizer,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5902 LNCS, pp. 23–36, 2009.

[47] A. Cimatti *et al.*, “Formal verification and validation of ERTMS industrial railway train spacing system,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7358 LNCS, pp. 378–393, 2012.

[48] a. Chiappini *et al.*, “Formalization and validation of a subset of the European Train Control

System," *2010 ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 2, pp. 109–118, 2010.

[49] M. Bozzano and A. Villafiorita, "The FSAP/NuSMV-SA safety analysis platform," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 1, pp. 5–24, 2007.

[50] A. Cimatti, S. Mover, and S. Tonetta, "SMT-based scenario verification for hybrid systems," *Form. Methods Syst. Des.*, vol. 42, no. 1, pp. 46–66, 2013.

[51] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 modulo theories via implicit predicate abstraction," in *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture No., vol. 8413, K. Abraham, Erika and Havelund, Ed. Springer Berlin Heidelberg, 2014, pp. 46–61.

[52] D. Jackson, *Software Abstractions*. Massachussets: The MIT Press, 2012.

[53] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002.

[54] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A challenging model transformation," in *10th International Conference on Model Driven Engineering Languages and Systems*, 2007, pp. 1–15.

[55] O. Badreddin, "A manifestation of model-code duality: facilitating the representation of state machines in the umple model-oriented programming language," University of Ottawa, 2012.

[56] CRUiSE, "UmpleOnline: Generate Java, C++, PHP, or Ruby from Umple code," *UmpleOnline: Generate Java, C++, PHP, or Ruby from Umple code*, 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umpleonline/. [Accessed: 18-Jan-2016].

[57] T. C. Lethbridge, "Teaching modeling using Umple: Principles for the development of an effective tool," in *2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE&T)*, 2014, pp. 23–28.

[58] T. C. Lethbridge, V. Abdelzad, M. H. Orabi, A. H. Orabi, and O. O. Adesina, "Merging Modeling and Programming Using Umple," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, vol. 17, T. Margaria and B. Steffen, Eds. Corfu: Springer International Publishing, 2016, pp. 187–197.

[59] O. O. Adesina, T. C. Lethbridge, S. S. Somé, V. Abdelzad, and A. B. Belle, "Improving Formal Analysis of State Machines with Particular Emphasis on And-Cross Transitions," *Under Consid. Publ. Comput. Syst. Lang. Struct.*, pp. 1–31.

[60] O. O. Adesina, T. C. Lethbridge, and S. S. Somé, "A Fully Automated Approach to Discovering Nondeterminism in State Machine Diagrams," in *10th International Conference on the Quality of Information and Communications Technology*, 2016, pp. 73–78.

[61] O. O. Adesina, S. S. Somé, and T. C. Lethbridge, "Modeling state diagrams with and-cross transitions," in *CEUR Workshop Proceedings - 13th Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa)*, 2016, vol. 1713, pp. 46–53.

[62] CRUiSE, "Umple User Manual," *Umple User Manual*, 2015. [Online]. Available: http://cruise.eecs.uottawa.ca/umple/GettingStarted.html. [Accessed: 06-May-2015].

[63] A. Forward, "The Convergence of Modeling and Programming: Facilitating the

Representation of Attributes and Associations in the Umple Model-Oriented Programming Language," *Doctor of Philosophy (PhD) Thesis, School of Electrical Engineering and Computer Science, University of Ottawa*, 2010. [Online]. Available: http://dx.doi.org/10.20381/ruor-13300. [Accessed: 13-Apr-2017].

[64] D. Brestovansky, "Exploring Textual Modeling using the Umple Language," *MCS Thesis, School of Electrical Engineering and Computer Science, University of Ottawa*, 2008. [Online]. Available: http://dx.doi.org/10.20381/ruor-12237.

[65] R. E. Gansner, E. Koutsofios, and S. North, "Drawing graphs with dot," in *dot User's Manual (910904-59113-08TM)*, 2015, pp. 1–40.

[66] Y. Lu, J. M. Atlee, N. A. Day, and J. Niu, "Model Checking Template-Semantics Specifications," in *Technical Report - CS-2004-20, University of Waterloo*, 2004, pp. 1–62.

[67] O. Badreddin, T. C. Lethbridge, A. Forward, M. Elasaar, and H. Aljamaan, "Enhanced Code Generation from UML Composite State Machines," in *Modelsward 2014*, 2014.

[68] O. Mahmoud, "Facilitating the Representation of Composite Structure, Active objects, Code Generation, and Software Component Descriptions for AUTOSAR in the Umple Model-Oriented Programming Language," *PhD Thesis (To be submitted), School of Electrical Engineering and Computer Science, University of Ottawa*, 2017. [Online]. Available: To be provided later.

[69] M. Bozzano *et al.*, "nuXmv 1.1.1 User Manual," 2016. [Online]. Available: https://es-static.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf. [Accessed: 13-Apr-2017].

[70] L. de Moura, S. Owre, and N. Shankar, "The SAL language manual," in *Technical Report - SRI-CSL-01-02 , Computer Science Laboratory, Standford University*, 2003, vol. 2, pp. 1–39.

[71] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained, The Model-Driven Architecture: Practice and Promise*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[72] M. Garzon, "Reverse Engineering Object-Oriented Systems into Umple : An Incremental and Rule-Based Approach by," *Doctor of Philosophy (PhD) Thesis, School of Electrical Engineering and Computer Science, University of Ottawa*, 2015. [Online]. Available: http://dx.doi.org/10.20381/ruor-4206. [Accessed: 13-Apr-2017].

[73] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, no. 1–2, pp. 125–142, 2006.

[74] A. Kleppe, "MCC: A model transformation environment," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2006, vol. 4066 LNCS, pp. 173–187.

[75] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained, The Model-Driven Architecture: Practice and Promise*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[76] T. Mens, K. Czarnecki, and P. Van Gorp, "04101 Discussion -- A Taxonomy of Model Transformations," in *Language Engineering for ModelDriven Software Development*, 2005, pp. 1–10.

[77] M. a. Garzon, H. Aljamaan, and T. C. Lethbridge, "Umple: A framework for Model Driven Development of Object-Oriented Systems," in *2015 IEEE 22nd International Conference*

on Software Analysis, Evolution, and Reengineering (SANER), 2015, no. September, pp. 494–498.

[78] CRUiSE, "Umple Metamodel," 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umple/umple-compiler-classDiagram.shtml. [Accessed: 31-Aug-2016].

[79] CRUiSE, "Umple Metamodel," 2016. .

[80] D. Jackson, *Software Abstractions: Logic, Language and Analysis*. 2012.

[81] CRUiSE, "Umple's Source Code," 2016. [Online]. Available: https://github.com/umple/umple/tree/master/cruise.umple/src. [Accessed: 20-Sep-2016].

[82] T. C. Lethbridge and R. Laganiere, *Object-Oriented Software Engineering : Practical Software Development Using Uml and Java*, 2nd Editio. New York: McGraw-Hill Companies, 2004.

[83] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented*. Toronto: Addison-Wesley Publishing Company, 1995.

[84] M. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, "NuSMV: a new symbolic model checker," in *11th International Conference on Computer Aided Verification*, 1999, pp. 495–499.

[85] OMG, *Unified Modeling Language (UML)*, vol. 5, no. March 2015. 2015, pp. 1–786.

[86] F. Faghih and N. A. Day, "Mapping Big-Step Modeling Languages to SMV," in *Technical Report - CS-2011-29, University of Waterloo*, 2011, pp. 1–57.

[87] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[88] O. Adesina, "Integrating formal methods with model-driven engineering," in *18th International Conference on Model Driven Engineering and Languages and Systems (MODELS'15)*, 2015, pp. 6–10.

[89] E. Posse and J. Dingel, "An executable formal semantics for UML-RT," *Softw. Syst. Model.*, vol. 15, no. 1, pp. 179–217, 2016.

[90] S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu, "Deconstructing the semantics of big-step modelling languages," *Requir. Eng.*, vol. 15, no. 2, pp. 235–265, 2010.

[91] A. Vakili and N. A. Day, "Verifying CTL-live Properties of Infinite State Models Using an SMT Solver," *Proc. 22Nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, no. Version 1, pp. 213–223, 2014.

[92] L. De Moura and N. Bj??rner, "Z3: An efficient SMT Solver," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 4963 LNCS, pp. 337–340, 2008.

[93] P. Mueller, "Sinelabore RT User Manual," *SinelaboreRT*, 2015. [Online]. Available: https://www.sinelabore.com/lib/exe/fetch.php?media=wiki:downloads:sinelaborert.pdf. [Accessed: 13-Apr-2017].

[94] O. O. Adesina, S. S. Somé, and T. C. Lethbridge, "Modeling State Diagrams with And-Cross Transitions," in *13th Workshop on Model Driven Engineering, Verification and*

Validation, 2016.

[95] H. Aljamaan, T. Lethbridge, M. Garzón, and A. Forward, "UmpleRun: A dynamic analysis tool for textually modeled state machines using umple," in *CEUR Workshop Proceedings*, 2015, vol. 1560, pp. 16–20.

[96] CRUiSE, "The home heating system case study." [Online]. Available: try.umple.org/?example=HomeHeater&diagramtype=state. [Accessed: 24-Jun-2016].

[97] CRUiSE, "Test suite for code generation from Umple to SMV," *Umple Source Codes - GitHub*, 2016. [Online]. Available: https://github.com/umple/umple/tree/master/cruise.umple/test/cruise/umple/implementation/nusmv. [Accessed: 30-Dec-2016].

[98] CRUiSE, "Test Suite for Alloy Code Generation," 2016. [Online]. Available: https://github.com/umple/umple/tree/master/cruise.umple/test/cruise/umple/implementation/alloy. [Accessed: 31-Dec-2016].

[99] CRUiSE, "SMV Generator's Test Oracle," 2016. [Online]. Available: https://github.com/umple/umple/blob/master/cruise.umple/test/cruise/umple/implementation/nusmv/NuSMVTemplateTest.java. [Accessed: 30-Dec-2016].

[100] CRUiSE, "Test Oracle for Alloy Code Generator," 2016. [Online]. Available: https://github.com/umple/umple/blob/master/cruise.umple/test/cruise/umple/implementation/alloy/AlloyTemplateTest.java. [Accessed: 31-Dec-2016].

[101] A. Abdulkhaleq and S. Wagner, "Integrating state machine analysis with system-theoretic process analysis," in *Proceedings - Series of the Gesellschaft fur Informatik (GI)*, 2013, vol. P-215, pp. 501–514.

[102] CRUiSE, "The Collision Avoidance System without And-Cross Transition," 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umpleonline/?example=CollisionAvoidance&diagramtype=state. [Accessed: 04-Jan-2017].

[103] CRUiSE, "The Collision Avoidance System (Alternative Solution 1)," 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umpleonline/?example=CollisionAvoidanceA1&diagramtype=state. [Accessed: 04-Jan-2017].

[104] CRUiSE, "The Collision Avoidance System (Alternative Solution 2)," 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umpleonline/?example=CollisionAvoidanceA2&diagramtype=state. [Accessed: 04-Jan-2017].

[105] CRUiSE, "The Collision Avoidance System (Alternative Solution 3)," 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umpleonline/?example=CollisionAvoidanceA3&diagramtype=state. [Accessed: 04-Jan-2017].

[106] T. Hafer and W. Thomas, "Computational Tree Logic CTL* and Path Quantifiers in the Monadic Theory of the Binary Tree," in *Automata, Languages and Programming*, 1987, vol. 267, pp. 269–279.

[107] J. a. Cruz-Lemus, M. Genero, M. E. Manso, S. Morasca, and M. Piattini, "Assessing the

understandability of UML statechart diagrams with composite states-A family of empirical studies," *Empir. Softw. Eng.*, vol. 14, no. 2009, pp. 685–719, 2009.

[108] J. a. Cruz-Lemus, M. Genero, M. E. Manso, and M. Piattini, "Evaluating the effect of composite states on the understandability of UML statechart diagrams," in *Model Driven Engineering Languages and Systems*, 2005, vol. 3713 LNCS, pp. 113–125.

[109] F. Xie, G. Yang, and X. Song, "Component-based hardware/software co-verification for building trustworthy embedded systems," *J. Syst. Softw.*, vol. 80, no. 5, pp. 643–654, 2007.

[110] P. Cousot, R. Cousot, J. Feret, A. Miné, and X. Rival, "The Astrée Static Analyzer," 2015. [Online]. Available: http://www.astree.ens.fr/. [Accessed: 06-Jun-2015].

[111] L. Mauborgne, "Astrée : Verification of Absence of Run-Time Error ∗," in *IFIP International Federation for Information Processing*, 2004, pp. 385–392.

[112] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks," in *POPL '79 Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1979, pp. 269–282.

[113] P. Cousot and R. Cousot, "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation," in *Proceedings of the 4th International Workshop Programming Language Implementation and Logic Programming, PLILP'92,* 1992, vol. 631, pp. 269–295.

[114] D. Delmas and J. Souyris, "ASTRÉE: from Research to Industry," in *Proc 14th International Static Analysis Symposium SAS 2007*, 2007, vol. 4634, pp. 437–451.

[115] J. Souyris and D. Delmas, "Experimental assessment of Astree on safety-critical avionics software," in *Proceedings of the 26th international conference on Computer Safety, Reliability, and Security*, 2007, pp. 479–490.

[116] O. Bouissou, E. Conquet, and P. Cousot, "Space software validation using abstract interpretation," in *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, 2009, no. 1, pp. 1–7.

[117] M. Balaban and A. Maraee, "Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, p. 24:1-24:42, 2013.

[118] M. Balaban, A. Maraee, A. Sturm, and P. Jelnov, "A pattern-based approach for improving model quality," *Softw. Syst. Model.*, pp. 1–29, 2014.

[119] M. Lenzerini, "On the satisfiability of dependency constraints in entity-relationship schemata," *Inf. Syst.*, vol. 15, no. 4, pp. 453–461, 1990.

[120] N. Amálio, F. Polack, and S. Stepney, "Frameworks Based on Templates for Rigorous Model-driven Development," *Electron. Notes Theor. Comput. Sci.*, vol. 191, no. 1, pp. 3–23, 2007.

[121] N. Amálio, F. Polack, and S. Stepney, "UML+Z: Augmenting UML with Z," *Softw. Specif. Methods*, pp. 81–102, 2010.

[122] J. R. Williams and F. a C. Polack, "Automated formalisation for verification of diagrammatic models," *Electron. Notes Theor. Comput. Sci.*, vol. 263, no. 191, pp. 211–226, 2010.

[123] M. Saaltink and R. Road, "The Z/EVES System," in *ZUM '97: The Z Formal Specification Notation*, 1997, pp. 72–85.

[124] R. M. Borges and A. C. Mota, "Integrating UML and Formal Methods," *Electron. Notes Theor. Comput. Sci.*, vol. 184, no. SPEC. ISS., pp. 97–112, 2007.

[125] A. Feliachi, M. Gaudel, and B. Wolff, "Isabelle/Circus : a Process Specification and Verification Environment," *Verif. Softw. Theor. Tools, Exp.*, vol. 7152, pp. 243–260, 2012.

[126] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8. pp. 666–677, 1978.

[127] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud, "An Overview of RoZ : a Tool for Integrating UML and Z Specifications," in *12th International Conference on Advanced information Systems Engineering-CAiSE*, 2000.

[128] G. Cernosek, "Next-generation model-driven development," *IBM Software Group*, vol. 1. pp. 1–16, 2004.

[129] H. Z. H. Zhu, I. Bayley, L. S. L. Shan, and R. Amphlett, "Tool Support for Design Pattern Recognition at Model Level," in *33rd Annual IEEE International Computer Software and Applications Conference*, 2009, vol. 1, pp. 1–6.

[130] I. Bayley and H. Zhu, "Specifying behavioural features of design patterns in first order logic," in *Proceedings - International Computer Software and Applications Conference*, 2008, pp. 203–210.

[131] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski, "SPASS version 3.5," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5663 LNAI, no. Section 5, pp. 140–145, 2009.

[132] C. C. Din, O. Owe, and R. Bubel, "Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems," in *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2014*, 2014, pp. 480–487.

[133] B. Liskov and L. Shrira, "Promises: linguistic support for efficient asynchronous procedure calls in distributed systems," *ACM SIGPLAN Not.*, vol. 23, no. 7, pp. 260–267, 1988.

[134] D. Calrke *et al.*, "Highly Adaptable and Trustworthy Software using Formal Models," in *7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme*, 2011, pp. 1–199.

[135] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," *ACM SIGPLAN Not.*, vol. 37, no. 5, p. 234, 2002.

[136] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[137] B. Bordbar and K. Anastasakis, "UML2ALLOY: A tool for lightweight modelling of discrete event systems.," *Guimarães, N., Isaías, P. IADIS Int. Conf. Appl. Comput. 2005*, no. 1999, pp. 209–216, 2005.

[138] K. Anastasakis, "UML2Alloy Reference Manual," *UML2Alloy, School of Computer Science, University of Birmingham*, 2009. [Online]. Available: http://www.cs.bham.ac.uk/~bxb/UML2Alloy/files/uml2alloy_manual.pdf. [Accessed: 13-Apr-2017].

[139] M. Inc., "Magic Draw: User Manual," *No Magic, Inc.*, 2015. [Online]. Available: http://www.nomagic.com/files/manuals/MagicDraw UserManual.pdf. [Accessed: 13-Apr-2017].

[140] M. E. Beato, M. Barrio-Solórzano, C. E. Cuesta, and P. De La Fuente, "UML automatic verification tool with formal methods," *Electron. Notes Theor. Comput. Sci.*, vol. 127, no. 4, pp. 3–16, 2005.

[141] K. Zurowska and J. Dingel, "Symbolic execution of UML-RT State Machines," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, 2012, p. 1292.

[142] K. Zurowska, "Domain Specific Analysis of Statemachine Models of Reactive Systems," in *Proceedings of 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, 2013, pp. 81–86.

[143] P. C. Mehlitz, "Trust your model - Verifying aerospace system models with Java Pathfinder," in *IEEE Aerospace Conference Proceedings*, 2008, pp. 1–11.

[144] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Autom. Softw. Eng.*, vol. 10, pp. 203–232, 2003.

[145] D. Remenska *et al.*, "From UML to process algebra and back: An automated approach to model-checking software design artifacts of concurrent systems," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7871 LNCS, pp. 244–260, 2013.

[146] S. Cranen *et al.*, "An overview of the mCRL2 toolset and its recent advances," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 7795 LNCS, pp. 199–213.

[147] J. C. Corbett *et al.*, "Bandera: extracting finite-state models from Java source code," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, 2000, pp. 439–448.

[148] K. L. McMillan, "Symbolic Model Checking," in *Symbolic Model Checking*, Boston, MA: Springer US, 1993, pp. 25–60.

[149] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[150] S. Bensalem *et al.*, "An Overview of SAL," in *5th NASA Langley Formal Methods Workshop*, 2000, pp. 187–196.

[151] R. Vallée-Rai, P. Co, É. M. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java Bytecode Optimization Framework," in *Proceedings of the 2010 Centre for Advanced Studies Conference (CASCON'10)*, 2010, pp. 214–224.

[152] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, vol. 36, no. 5, pp. 203–213.

[153] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Computer aided verification*, 2001, pp. 260–264.

[154] T. Ball, T. Ball, S. Rajamani, and S. Rajamani, "Bebop: A Symbolic Model Checker for

Boolean Programs," in *SPIN Model Checking and Software Verification*, 2000, pp. 113–130.

[155] J. Jürjens and P. Shabalin, "Tools for secure systems development with UML," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5–6, pp. 527–544, 2007.

[156] J. Jurjens, "UMLsec: Extending UML for secure systems development," in *Proceedings of the 5th International Conference on The Unified Modeling Language*, 2002, pp. 412–425.

[157] M. Bakera, T. Margaria, C. D. Renner, and B. Steffen, "Tool-supported enhancement of diagnosis in model-driven verification," *Innov. Syst. Softw. Eng.*, vol. 5, no. 3, pp. 211–228, 2009.

[158] M. Bakera and C. Renner., "GEAR – A Game-based Model Checking Tool - jABC," *Game-based, Easy And Reverse model-checking*, 2015. [Online]. Available: http://jabc.cs.tu-dortmund.de/modelchecking/. [Accessed: 11-Jun-2015].

[159] G. J. Holzmann and M. H. Smith, "Automating software feature verification," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 72–87, 2000.

[160] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, 2007, p. 547.

[161] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Commun. ACM*, vol. 54, pp. 68–76, 2011.

[162] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *Int. J. Softw. Tools Technol. Transf.*, vol. 2, pp. 366–381, 2000.

[163] J. C. Corbett *et al.*, "Bandera: extracting finite-state models from Java source code," *Proc. 2000 Int. Conf. Softw. Eng. ICSE 2000 New Millenn.*, pp. 439–448, 2000.

[164] J. Ivers and N. Sharygnia, "Overview of ComFoRT : A Model Checking Reasoning Framework," in *Technical Report - CMU/SEI-2004-TN-018 , Carnegie Mellon University (CMU)*, 2004, no. April, pp. 1–55.

[165] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, vol. 2988, pp. 168–176.

[166] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based Predicate Abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, 2005, vol. 3440, pp. 570–574.

[167] R. Eshuis and P. Van Gorp, "Synthesizing object life cycles from business process models," *Softw. Syst. Model.*, vol. 15, no. 1, pp. 281–302, 2016.

[168] "ArgoUML." [Online]. Available: http://argouml.tigris.org/. [Accessed: 20-Feb-2017].

[169] "MagicDraw." [Online]. Available: https://www.nomagic.com/products/magicdraw. [Accessed: 20-Feb-2017].

[170] "Visual Paradigm." [Online]. Available: https://www.visual-paradigm.com/. [Accessed: 20-Feb-2017].

[171] "Altova (UModel)." [Online]. Available: https://www.altova.com/umodel.html.

[172] "Astah." [Online]. Available: http://astah.net/. [Accessed: 20-Feb-2017].

[173] "MetaUML." [Online]. Available: https://github.com/ogheorghies/MetaUML. [Accessed: 20-Feb-2017].

[174] "BOUML." [Online]. Available: http://www.bouml.fr. [Accessed: 20-Feb-2017].

[175] G. Zhang and M. Holzl, "A Set of Metrics of Non-locality Complexity in UML State Machines," in *Revised Selected Papers of the International Workshops on Behavior Modeling -- Foundations and Applications*, 2015, vol. 6368, pp. 59–81.

[176] M. Zulkernine and R. E. Seviora, "Assume-Guarantee Algorithms for Automatic Detection of Software Failures," in *3rd International Conference on Integrated Formal Methods (IFM)*, 2002, pp. 89–108.

[177] D. Latella, I. Majzik, and M. Massink, "of Computing Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN," *Form. Asp. Comput.*, vol. 11, no. 6, pp. 637–664, 1999.

[178] C. Yamada and D. M. Miller, "Using SPIN to Check Nondeterministic Simulink Stateflow Models," *Proc. Int. Symp. Mult. Log.*, vol. 2015–Septe, pp. 145–151, 2015.

[179] P. Arcaini, A. Gargantini, and E. Riccobene, "Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism," *Proc. - IEEE 6th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2013*, pp. 178–187, 2013.

[180] G. J. Holzmann, "The Spin Model Checker," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[181] M. P. E. Heimdahl and N. G. Leveson, "Completeness and consistency in hierarchical state-based requirements," *IEEE Trans. Softw. Eng.*, vol. 22, no. 6, pp. 363–377, 1996.

[182] E. M. Clarke and W. Heinle, "Modular translation of statecharts to SMV," in *Technical Report - CMU-CS-00-XXX, Carnegie Mellon University, Pittsburgh (Contract no: 98-0164/65 - General Motors)*, 2000, pp. 1–40.

[183] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, pp. 231–261, 1996.

[184] A. A. Mir, S. Balakrishnan, and S. Tahar, "Modeling and Verification of Embedded Systems Using Cadence SMV," in *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492)*, 2000, vol. 1, pp. 179–183.

[185] S. Schwoon, "Model-Checking Pushdown Systems," Universitat Munchen, 2002.

[186] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie, "Zing: A Model Checker for Concurrent Software," in *Computer Aided Verification*, 2004, vol. 3114, pp. 28–32.

[187] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3 - A modern refinement checker for CSP," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, vol. 8413 LNCS, pp. 187–201.

Appendix

Appendix 1 – Home Heating System - SMV

The following describes the SMV model for the home heating system. We refer to it as a supplement because we manually add codes representing actions in the corresponding Umple system. For the sake of brevity, we skipped some parts (i.e., "…") since the can be generated from via Umple2SMV [56]. The Umple version can be obtained from [96].

```
-- This file is generated from newHeatingSys.ump --

-- PLEASE DO NOT EDIT THIS CODE --
-- This code was generated using the UMPLE 1.24.0-7aed471 modeling language! --


-- This defines a NuSMV module for HeatControlSystemSm --
MODULE HeatControlSystemSm
(
 _smHouseRoom, _smHouseRoomRoom, _smHouseRoomRoomNoHeatReq, _smHouseRoomRoomHeatReq, _smHouseController,
 _smHouseControllerController, _smHouseControllerControllerControllerOn, _smFurnaceFurnaceFurnaceNormal,
 _smHouseControllerControllerControllerOnHeaterActive, _smHouse, _smFurnaceFurnace
)

  VAR
    state : { Sm_heatingSystem , null };
    event : { ev_heatSwitchOn, ev_heatSwitchOff, ev___autotransition__, ev_userReset,
      ev_furnaceFault, ev_null };
    furnaceStartUpTime : 0..5; -- range(0..5)
    waitedForWarm : 0..5; -- range(0..5)
    valvePos : 0..2; -- range(0..2)
    waitedForCool : 0..5; -- range(0..5)
    furnaceRunning : boolean;
    activate : boolean;
    deactivate : boolean;
    requestHeat : boolean;
    furnaceReset : boolean;

  IVAR
    setTemp : 16..24; -- range(16..24)
    actualTemp : -45..30; -- range(-45..30)

  DEFINE
    sm_stable :=  !( event = ev_heatSwitchOff | event = ev_userReset
     | event = ev_heatSwitchOn | event = ev___autotransition__
     | event = ev_furnaceFault );
    t1 := event = ev___autotransition__
      & _smHouseRoomRoomNoHeatReq.state = SmHouseRoomRoomNoHeatReq_idleNotHeat & g1;
    t2 := event = ev___autotransition__
      & _smHouseRoomRoomNoHeatReq.state = _SmHouseRoomRoomNoHeatReq_waitForHeat & g2;
    t3 := event = ev___autotransition__
      & _smHouseRoomRoomNoHeatReq.state = SmHouseRoomRoomNoHeatReq_waitForHeat & g3;
    t4 := event = ev___autotransition__
      & _smHouseRoomRoomNoHeatReq.state = SmHouseRoomRoomNoHeatReq_waitForHeat & g4;
```

```
  t5 := event = ev___autotransition__
    & _smHouseRoomRoomNoHeatReq.state = SmHouseRoomRoomNoHeatReq_waitForHeat & g5;
  t6 := event = ev___autotransition__
    & _smHouseRoomRoomHeatReq.state = SmHouseRoomRoomHeatReq_idleHeat & g6;
  t7 := event = ev___autotransition__
    & _smHouseRoomRoomHeatReq.state = SmHouseRoomRoomHeatReq_waitForCool & g7;
  t8 := event = ev___autotransition__
     & _smHouseRoomRoomHeatReq.state = SmHouseRoomRoomHeatReq_waitForCool & g8;
  t9 := event = ev___autotransition__
     & _smHouseRoomRoomHeatReq.state = SmHouseRoomRoomHeatReq_waitForCool & g9;
  t10 := event = ev___autotransition__
     & _smHouseRoomRoomHeatReq.state = SmHouseRoomRoomHeatReq_waitForCool & g10;
  t11 := event = ev_heatSwitchOn & _smHouseControllerController.state = SmHouseControllerController_off;
  t12 := event = ev_heatSwitchOff
     & _smHouseControllerController.state = SmHouseControllerController_controllerOn;
  t13 := event = ev_furnaceFault
     & _smHouseControllerController.state = SmHouseControllerController_controllerOn;
  t14 := event = ev_userReset & _smHouseControllerController.state = SmHouseControllerController_error;
  t15 := event = ev___autotransition__
     & _smHouseControllerControllerControllerOn.state = SmHouseControllerControllerControllerOn_idle & g11;
  t16 := event = ev___autotransition__  & g12 &
     _smHouseControllerControllerControllerOn.state = SmHouseControllerControllerControllerOn_heaterActive;
  t17 := event = ev___autotransition__  & g13 & _smHouseControllerControllerControllerOnHeaterActive.state
     = SmHouseControllerControllerControllerOnHeaterActive_actHeater;
  t18 := event = ev_furnaceFault & _smFurnaceFurnace.state = SmFurnaceFurnace_furnaceNormal;
  t19 := event = ev___autotransition__ & _smFurnaceFurnace.state = SmFurnaceFurnace_furnaceErr & g14;
  t20 := event = ev___autotransition__
     & _smFurnaceFurnaceFurnaceNormal.state = SmFurnaceFurnaceFurnaceNormal_furnaceOff & g15;
  t21 := event = ev___autotransition__
     & _smFurnaceFurnaceFurnaceNormal.state = SmFurnaceFurnaceFurnaceNormal_furnaceAct & g16;
  t22 := event = ev___autotransition__
      & _smFurnaceFurnaceFurnaceNormal.state = SmFurnaceFurnaceFurnaceNormal_furnaceAct & g17;
  t23 := event = ev___autotransition__
      & _smFurnaceFurnaceFurnaceNormal.state = SmFurnaceFurnaceFurnaceNormal_furnaceAct & g18;
  t24 := event = ev___autotransition__
      & _smFurnaceFurnaceFurnaceNormal.state = SmFurnaceFurnaceFurnaceNormal_furnaceRun & g16;
  g1 := (setTemp - actualTemp) > 2;
  g2 := waitedForWarm < WARMUPTIMER;
  g3 := (valvePos != 2) & (waitedForWarm = WARMUPTIMER);
  g4 := ((setTemp - actualTemp) <= 2);
  g5 := (waitedForWarm = warmUpTimer)&(valvePos = 2)&((setTemp - actualTemp) > 2);
  g6 := (actualTemp - setTemp) > 2;
  g7 := (valvePos != 0) & (COOLDOWNTIMER = waitedForCool);
  g8 := (valvePos = 0)&( COOLDOWNTIMER = waitedForCool)&((actualTemp - setTemp)> 2);
  g9 := waitedForCool < COOLDOWNTIMER;
  g10 := ((actualTemp - setTemp) <= 2);
  g11 := requestHeat = TRUE;
  g12 := requestHeat = FALSE;
  g13 := furnaceRunning = TRUE;
  g14 := furnaceReset = TRUE;
  g15 := activate = TRUE;
  g16 := deactivate = TRUE;
  g17 := furnaceStartUpTime < FURNACETIMER;
  g18 := furnaceStartUpTime = FURNACETIMER;
  FURNACETIMER := 5; -- constant integer variable assigned value 5
  WARMUPTIMER := 5; -- constant integer variable assigned value 5
  coolDownTimer := 5; -- constant integer variable assigned value 5

ASSIGN
  init( furnaceStartUpTime ) := 0;
  next( furnaceStartUpTime ) := case
            furnaceStartUpTime + 1 = 6 : 5;
            t20 : 0;
            t22 : furnaceStartUpTime + 1;
            TRUE : furnaceStartUpTime;
  esac;

ASSIGN
```

```
  init( waitedForWarm ) := 0;
  next( waitedForWarm ) := case
           waitedForWarm + 1 = 6 : 5;
           waitedForWarm - 1 = -1 : 0;
           t1 | t3 : 0;
           t2 : waitedForWarm + 1;
           TRUE : waitedForWarm;
  esac;

 ASSIGN
  init( valvePos ) := 0;
  next( valvePos ) := case
           valvePos + 1 = 3 : 2;
           valvePos - 1 = -1 : 0;
           t1 | t3 : valvePos + 1;
           t6 | t7 : valvePos - 1;
           TRUE : valvePos;
  esac;

 ASSIGN
  init( waitedForCool ) := 0;
  next( waitedForCool ) := case
           waitedForCool + 1 = 6 : 5;
           waitedForCool - 1 = -1 : 0;
           t5 | t7 : 0;
           t9 : waitedForCool + 1;
           TRUE : waitedForCool;
  esac;

 ASSIGN
  init( furnaceRunning ) := FALSE;
  next( furnaceRunning ) := case
           t23 : TRUE;
           TRUE : furnaceRunning;
  esac;

 ASSIGN
  init( activate ) := FALSE;
  next( activate ) := case
           t15 : TRUE;
           TRUE : activate;
  esac;

 ASSIGN
  init( deactivate ) := FALSE;
  next( deactivate ) := case
            t12 | t16 : TRUE;
            TRUE : deactivate;
           esac;

 ASSIGN
  init( requestHeat ) := FALSE;
  next( requestHeat ) := case
           t5 | t8 : FALSE;
           TRUE : requestHeat;
  esac;

 ASSIGN
  init( furnaceReset ) := FALSE;
  next( furnaceReset ) := case
           t14 : TRUE;
           TRUE : furnaceReset;
  esac;

MODULE HeatControlSystemSmHouseRoom ( _sm )
 VAR
  state : { SmHouseRoom_room , null };
 ASSIGN
```

```
    init( state ) := null;
    next( state ) := case
      _sm.t8 | _sm.t1 | _sm.t3 | _sm.t10 | _sm.t7 | _sm.t4 | _sm.t2 | _sm.t5 | _sm.t6
        | _sm.t9 : SmHouseRoom_room;
      _sm.state = Sm_heatingSystem & state = null : SmHouseRoom_room;
      TRUE : state;
    esac;

MODULE HeatControlSystemSmHouseRoomRoom ( _sm , _smHouse )
  VAR
    state : { SmHouseRoomRoom_noHeatReq , SmHouseRoomRoom_heatReq , null };
  ASSIGN
    init( state ) := null;
    next( state ) := case
      _sm.t4 | _sm.t2 | _sm.t8 | _sm.t1 | _sm.t3 : SmHouseRoomRoom_noHeatReq;
      _sm.t10 | _sm.t7 | _sm.t5 | _sm.t6 | _sm.t9 : SmHouseRoomRoom_heatReq;
      _smHouse.state = SmHouse_house & state = null : SmHouseRoomRoom_noHeatReq;
      TRUE : state;
    esac;

MODULE HeatControlSystemSmHouseRoomRoomNoHeatReq ( _sm , _smHouseRoomRoom )
  VAR
    state : { SmHouseRoomRoomNoHeatReq_idleNotHeat,
      SmHouseRoomRoomNoHeatReq_waitForHeat , null };
  ASSIGN
    init( state ) := null;
    next( state ) := case
      _sm.t5 | _sm.t7 | _sm.t9 | _sm.t6 | _sm.t8 | _sm.t10 : null;
      _sm.t4 : SmHouseRoomRoomNoHeatReq_idleNotHeat;
      _sm.t2 | _sm.t1 | _sm.t3 : SmHouseRoomRoomNoHeatReq_waitForHeat;
      _smHouseRoomRoom.state = SmHouseRoomRoom_noHeatReq
        & state = null : SmHouseRoomRoomNoHeatReq_idleNotHeat;
      TRUE : state;
    esac;

MODULE HeatControlSystemSmHouseRoomRoomHeatReq ( _sm , _smHouseRoomRoom )
  VAR
    state : { SmHouseRoomRoomHeatReq_idleHeat, SmHouseRoomRoomHeatReq_waitForCool,
      null };
  ASSIGN
    init( state ) := null;
    next( state ) := case
      _sm.t1 | _sm.t3 | _sm.t5 | _sm.t2 | _sm.t4 | _sm.t8 : null;
      _sm.t10 : SmHouseRoomRoomHeatReq_idleHeat;
      _sm.t7 | _sm.t6 | _sm.t9 : SmHouseRoomRoomHeatReq_waitForCool;
      _smHouseRoomRoom.state = SmHouseRoomRoom_heatReq
       & state = null : SmHouseRoomRoomHeatReq_idleHeat;
      TRUE : state;
    esac;

MODULE HeatControlSystemSmHouseController ( _sm )
  VAR
    state : { SmHouseController_controller , null };
  ASSIGN
    init( state ) := null;
    next( state ) := case
      _sm.t14 | _sm.t16 | _sm.t17 | _sm.t12 | _sm.t11 | _sm.t15
        | sm.t13 : SmHouseController_controller;
      _sm.state = Sm_heatingSystem & state = null : SmHouseController_controller;
      TRUE : state;
    esac;

MODULE HeatControlSystemSmHouseControllerController ( _sm , _smHouse )
  VAR
    state : { SmHouseControllerController_off,
      SmHouseControllerController_controllerOn, SmHouseControllerController_error,
      null };
  ASSIGN
```

```
  init( state ) := null;
  next( state ) := case
   sm.t12 | _sm.t14 : SmHouseControllerController_off;
   sm.t11 | _sm.t15 | _sm.t16
    | sm.t17 : SmHouseControllerController_controllerOn;
   sm.t13 : SmHouseControllerController_error;
   smHouse.state = SmHouse_house &
     state = null : SmHouseControllerController_off;
   TRUE : state;
  esac;

MODULE HeatControlSystemSmHouseControllerControllerControllerOn ( _sm, _smHouseControllerController )
 VAR
  state : { SmHouseControllerControllerControllerOn_idle,
   SmHouseControllerControllerControllerOn_heaterActive , null };
 ASSIGN
  init( state ) := null;
  next( state ) := case
   sm.t12 | _sm.t14 | _sm.t11 | _sm.t13 | _sm.t15 : null;
   sm.t16 : SmHouseControllerControllerControllerOn_idle;
   sm.t15 | _sm.t17 : SmHouseControllerControllerControllerOn_heaterActive;
   smHouseControllerController.state = SmHouseControllerController_controllerOn &
   state = null : SmHouseControllerControllerControllerOn_idle;
   TRUE : state;
  esac;

MODULE HeatControlSystemSmHouseControllerControllerControllerOnHeaterActive ( _sm,
   _smHouseControllerControllerControllerOn )
 VAR
  state : { SmHouseControllerControllerControllerOnHeaterActive_actHeater,
   SmHouseControllerControllerControllerOnHeaterActive_heaterRun, null };
 ASSIGN
  init( state ) := null;
  next( state ) := case
   _sm.t15 | _sm.t16 : null;
   _sm.t17 : SmHouseControllerControllerControllerOnHeaterActive_heaterRun;
   _smHouseControllerControllerControllerOn.state
     = SmHouseControllerControllerControllerOn_heaterActive & state = null :
     SmHouseControllerControllerControllerOnHeaterActive_actHeater;
   TRUE : state;
  esac;

MODULE HeatControlSystemSmHouse ( _sm )
 VAR
  state : { SmHouse_house , null };
 ASSIGN
  init( state ) := null;
  next( state ) := case
   _sm.t4 | _sm.t2 | _sm.t5 | _sm.t6 | _sm.t9 | _sm.t14 | _sm.t16 | _sm.t17
    | sm.t8 | _sm.t1 | _sm.t3 | _sm.t10 | _sm.t7 | _sm.t12 | _sm.t11 | _sm.t15
    | sm.t13 : SmHouse_house;
   _sm.state = Sm_heatingSystem & state = null : SmHouse_house;
   TRUE : state;
  esac;

MODULE HeatControlSystemSmFurnaceFurnace ( _sm )
 VAR
  state : { SmFurnaceFurnace_furnaceNormal, SmFurnaceFurnace_furnaceErr, null };
 ASSIGN
  init( state ) := null;
  next( state ) := case
   _sm.t19 | _sm.t24 | _sm.t22 | _sm.t21 | _sm.t20
    | _sm.t23 : SmFurnaceFurnace_furnaceNormal;
   _sm.t18 : SmFurnaceFurnace_furnaceErr;
   _sm.state = Sm_heatingSystem & state = null : SmFurnaceFurnace_furnaceNormal;
   TRUE : state;
  esac;
```

```
MODULE HeatControlSystemSmFurnaceFurnaceFurnaceNormal ( _sm , _smFurnaceFurnace )
  VAR
   state : { SmFurnaceFurnaceFurnaceNormal_furnaceOff,
     SmFurnaceFurnaceFurnaceNormal_furnaceAct,
     SmFurnaceFurnaceFurnaceNormal_furnaceRun, null };
  ASSIGN
   init( state ) := null;
   next( state ) := case
     _sm.t18 | _sm.t19 : null;
     _sm.t21 | _sm.t24 : SmFurnaceFurnaceFurnaceNormal_furnaceOff;
     _sm.t20 | _sm.t22 : SmFurnaceFurnaceFurnaceNormal_furnaceAct;
     _sm.t23 : SmFurnaceFurnaceFurnaceNormal_furnaceRun;
     _smFurnaceFurnace.state = SmFurnaceFurnace_furnaceNormal &
       state = null : SmFurnaceFurnaceFurnaceNormal_furnaceOff;
     TRUE : state;
   esac;


MODULE HeatControlSystemSm_Machine

  -- This part declares state variables for the given NuSMV module --
  VAR
   heatControlSystemSm : HeatControlSystemSm( heatControlSystemSmHouseRoom,
          heatControlSystemSmHouseRoomRoom, heatControlSystemSmHouseRoomRoomNoHeatReq,
          heatControlSystemSmHouseRoomRoomHeatReq, heatControlSystemSmHouseController,
          heatControlSystemSmHouseControllerController,
          heatControlSystemSmHouseControllerControllerControllerOn,
          heatControlSystemSmHouseControllerControllerControllerOnHeaterActive, heatControlSystemSmHouse,
   heatControlSystemSmFurnaceFurnace, heatControlSystemSmFurnaceFurnaceFurnaceNormal);
   heatControlSystemSmHouseRoom : HeatControlSystemSmHouseRoom( heatControlSystemSm );
   heatControlSystemSmHouseRoomRoom : HeatControlSystemSmHouseRoomRoom( heatControlSystemSm,
     heatControlSystemSmHouse );
   heatControlSystemSmHouseRoomRoomNoHeatReq : HeatControlSystemSmHouseRoomRoomNoHeatReq(
     heatControlSystemSm, heatControlSystemSmHouseRoomRoom );
   heatControlSystemSmHouseRoomRoomHeatReq : HeatControlSystemSmHouseRoomRoomHeatReq( heatControlSystemSm,
          heatControlSystemSmHouseRoomRoom );
   heatControlSystemSmHouseController : HeatControlSystemSmHouseController( heatControlSystemSm );
   heatControlSystemSmHouseControllerController : HeatControlSystemSmHouseControllerController(
          heatControlSystemSm , heatControlSystemSmHouse );
   heatControlSystemSmHouseControllerControllerControllerOn :
     HeatControlSystemSmHouseControllerControllerControllerOn( heatControlSystemSm ,
     heatControlSystemSmHouseControllerController );
   heatControlSystemSmHouseControllerControllerControllerOnHeaterActive :
     HeatControlSystemSmHouseControllerControllerControllerOnHeaterActive( heatControlSystemSm ,
     heatControlSystemSmHouseControllerControllerControllerOn );
   heatControlSystemSmHouse : HeatControlSystemSmHouse( heatControlSystemSm );
   heatControlSystemSmFurnaceFurnace : HeatControlSystemSmFurnaceFurnace( heatControlSystemSm );
   heatControlSystemSmFurnaceFurnaceFurnaceNormal : HeatControlSystemSmFurnaceFurnaceFurnaceNormal(
     heatControlSystemSm, heatControlSystemSmFurnaceFurnace );

MODULE main
  VAR
   heatControlSystemSm_Machine : HeatControlSystemSm_Machine;


INVARSPEC ( heatControlSystemSm_Machine.heatControlSystemSm.t2 &
          heatControlSystemSm_Machine.heatControlSystemSm.t4 -> next(
          heatControlSystemSm_Machine.heatControlSystemSmHouseRoomRoomNoHeatReq.state =
          SmHouseRoomRoomNoHeatReq_waitForHeat &
          heatControlSystemSm_Machine.heatControlSystemSmHouseRoomRoomNoHeatReq.state =
          SmHouseRoomRoomNoHeatReq_idleNotHeat ) )
…
```